universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Comparative Analysis of Procedural Planet Generators

verfasst von | submitted by

Manuel Zechmann BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Univ.-Prof. Dr. Helmut Hlavacs, for his invaluable support, guidance and encouragement throughout this project. His feedback and expertise were essential for the completion of this thesis.

I am also very grateful to my friends and family for their constant motivation and belief in me. Your support has been crucial!

A special thanks to those who took the time to play the games and participate in the questionnaire. Your feedback was critical in advancing this research and I appreciate the effort everyone made to contribute to this thesis.

In addition, I want to acknowledge ChatGPT, an AI language model, which has been used to refine the clarity and language of this text.

Abstract

Procedural generation techniques are widely used in game development these days. They shine in creating dynamic and diverse environments, especially in open-world scenarios and planetary simulations. This thesis explores how different levels of environmental detail and procedural generation methods affect user perception. To test this, two planets were created in Godot, each of them with a different height generation algorithm. Afterwards, a study was conducted which compares the generated worlds with two other procedural planet projects. The study asked 15 people to assess each of the planets using 12 questions with the following topics: “Immersion and Presence,” “Exploration and Fun,” “Terrain Features and Realism,” and “Movement and Controls.” The study found that the environments created in this project resulted in higher ratings for each of the categories compared to the other projects, which received lower scores. This means that the techniques used in this thesis can potentially help increase immersion levels in procedural planet generation.

Kurzfassung

Prozedurale Generierungstechniken werden heutzutage häufig in der Spieleentwicklung benutzt. Sie sind vor allem für die Erstellung von dynamischen und vielfältigen Umgebungen, beispielsweise Open-World Szenarien und Planetensimulationen, gut geeignet. Diese Arbeit untersucht, wie verschiedene Detailstufen von Umgebungen und prozeduralen Generierungstechniken die Wahrnehmung der Benutzer beeinflussen. Um dies zu testen, wurden zwei Planeten in Godot erstellt, die jeweils einen anderen Algorithmus zur Höhengenerierung verwenden. Danach wurde eine Studie durchgeführt, in der diese beiden Planeten mit zwei weiteren Planetenprojekten verglichen wurden. In der Studie wurden 15 Personen gebeten, jeweils 12 Fragen pro Planeten zu beantworten. Die Fragen behandelten die Themen "Immersion und Präsenz," "Erkundung und Spaß," "Geländemerkmale und Realismus" sowie "Bewegung und Steuerung." Die Studie zeigte, dass die für dieses Projekt erstellten Planeten in fast allen Kategorien höhere Werte aufwiesen als die anderen Projekte. Dies deutet darauf hin, dass die in dieser Arbeit verwendeten Techniken das Immersionsniveau in der prozeduralen Planetengeneration steigern können und zu einer noch realistischeren und fesselnden Spielerfahrung führen können.

Contents

1. Introduction

1.1. Background and Motivation

It is safe to assume that everyone who has played games like "No Man's Sky" or "Minecraft" was at least partially in awe of how large and diverse the world is. According to their homepage [No a], there are 18 quintillion unique planets in "No Man's Sky," which is an 18 followed by 18 zeroes, a number not imaginable by the human mind. On the other hand, a normal world in "Minecraft" has a size of 60 million blocks by 60 million blocks [Sca]. Again, no human would ever be able to accurately grasp the dimension of these numbers. While both of these games have an enormous amount of content to visit, the file sizes are quite small with an average "Minecraft"-world being as low as 4 KB to multiple GBs in size depending on how much has been explored, all of "Minecraft" having around 1 GB [Gam] and the entirety of "No Man's Sky" fitting on 15 GB of data [Ste]. How can such big games fit on so little space?

The answer lies in procedural generation. Instead of manually placing each block or designing each of the planets individually, the data is generated algorithmically. Procedural generation is not only used in these games, but it is in fact a cornerstone of many modern games. Its ability to create vast, intricate worlds without requiring immense storage capacities or time makes it an essential tool for creating dynamic, interactive environments. Despite its advantages, procedural generation comes with significant challenges. For one, there is quite the difficulty in ensuring that the generated content is realistic and diverse, not just repetitive or generic. Achieving natural variation, maintaining performance, and controlling generation algorithms in a way that balances randomness and realism are non-trivial tasks. Especially in terrain generation, when trying to generate mountains, rivers or other geographic features, a well-made algorithm has to be developed, in order to make the user immersed into the experience [TCL+13].

Planet generation, in particular, introduces further complexity. While flat terrains are easier to handle, the spherical surface of planets requires algorithms that account for curvature and seamless transitions across the whole planet. Additionally, the program must maintain performance while rendering high-detail terrains near the player without slowing down or even lagging [MK18].

This thesis focuses on generating realistic and intriguing looking planets that are fun to explore. These planets will be created using procedural generation techniques, including Perlin Noise. Furthermore, managing detail becomes crucial. A key challenge is implementing a Level of Detail system that ensures smooth performance by dynamically adjusting the complexity of the surface of the planet at different regions as the camera approaches or moves away. This is particularly useful for large planets, where rendering

everything at high resolution would be impossible.

All in all, this thesis aims to not only create a real time procedural planet generator, but also tries to advance the algorithms commonly used in these types of scenarios by adjusting them accordingly.

1.2. Objectives

The main objective of this thesis lies in the development of a procedural planet generator, that is capable of quickly creating interesting terrain with a high degree of quality. This main objective can be split up into multiple sub objectives. For one, an algorithm for a planet structure with a good distribution of vertices has to be found. Next, a good working height calculation algorithm has to be developed. Another objective is the addition of a system that allows the planet to increase the resolution in certain areas of the planet, while keeping the resolution at other locations the same. The final objective is to further increase the look of the planet and the environment in order to create a more immersive experience.

1.3. Research Question

The primary research question of this thesis is how varying levels of environmental detail and procedural generation techniques affect user perceptions. To evaluate this, a study has been conducted which asked the participants a multitude of questions about the two planets generated in this project and two similar projects for reference. These questions touch on the topics of "Immersion and Presence," "Exploration and Fun," "Terrain Features and Realism," and "Movement and Controls."

1.4. Structure of the Thesis

This thesis is divided into six chapters, beginning with this introduction. The second chapter, "Related Work," reviews existing approaches to procedural planet generation, focusing on noise functions, algorithmic foundations, terrain generation, and Level of Detail systems. It also examines the use of procedural generation in video games, simulations, and performance optimization techniques. The third chapter, "Methodology," outlines the key design decisions and approaches used in the project. It explores various options for essential sections such as the development environment, planet mesh generation, height calculation, level of detail management, and the addition of environmental features like atmospheric effects, biomes, and shaders. The thought process behind selecting specific techniques is also discussed. The fourth chapter, "Technical Implementation," provides a detailed description of the implementation process. This includes the setup of the project and game engine, planetary sphere generation, height calculations, surface realism enhancements, and biome assignment. Additionally, it covers features such as tree spawning, atmospheric effects, user interface elements, and the final steps of exporting

the project into a standalone executable. In the fifth chapter, "Evaluation," the planets generated in this project are compared with two other procedural planet generators. This chapter describes the study design, including the questionnaire and data collection process. The results are analyzed in detail, highlighting differences between the projects and identifying key insights, while also addressing limitations of the study. Finally, the sixth chapter, "Conclusion," summarizes the main findings of the thesis and discusses potential areas for future work to further improve procedural planet generation and its applications.

2. Related Work

In this chapter, state-of-the-art methods and foundational concepts, that form the basis of this research, are reviewed. This thesis focuses on essential topics within procedural generation, including noise functions, terrain generation techniques as well as level-of-detail (LOD) systems for enhancing the performance of simulations and video games. By analyzing these core areas, this review establishes a solid foundation for identifying and implementing effective solutions for the common problems found in a procedural planet generation project.

2.1. Noise Functions and Algorithmic Foundations

Procedural generation relies heavily on noise functions, which are mathematical functions that take a position in space as input and return a random value, which in turn can then be used for many purposes, for example to determine parameters of a planet like terrain height or texture features. While there are many different noise functions, it is crucial to find a fitting one for the specific use-case.

White Noise (Figure 2.1) is a purely random noise that assigns random values independently at each point in space. This is very useful when continuity is not needed, but in applications like textures or surfaces, which need smooth transitions, this is not ideal.

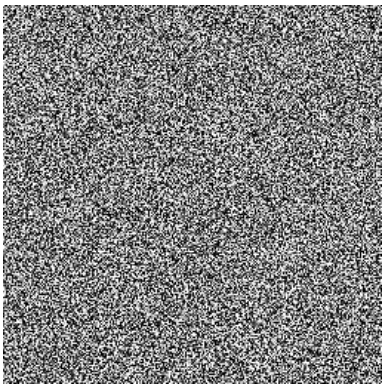

Figure 2.1.: Example of White Noise [Ado23b]

One of the most commonly used types of noises are Lattice Gradient Noises, which are able to generate values at certain positions in space (for example at every positive x-y integer pair). For any point between these positions, the gradients are interpolated to create a smooth and continuous texture. Notable examples for this noise type are Perlin

Noise and Simplex Noise. Perlin Noise (Figure 2.2), which was developed by Ken Perlin in 1985, is one of the most common gradient noises. Values are interpolated based on gradient vectors, which causes the function to produce smooth and continuous patterns. This noise function is widely used in all kinds of applications, especially when there is a need for clean transitions between the areas. Simplex Noise(Figure 2.3) on the other hand is an extension of Perlin Noise, which is designed to be more computationally efficient in higher dimensions. Instead of using an integer grid (basically squares), Simplex Noise uses a simplex grid (a generalization of a triangle in n dimensions, which in 2D is simply a triangle). This lowers the computational cost and additionally helps in eliminating undesired directional artifacts.

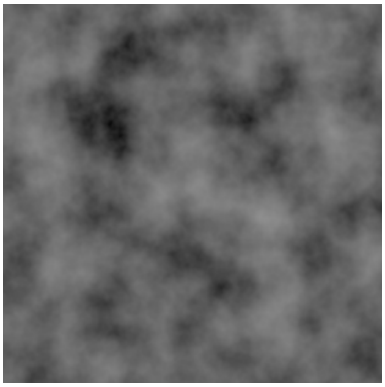

Figure 2.2.: Example of Perlin Noise using Godot

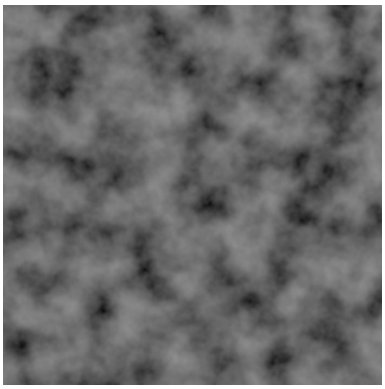

Figure 2.3.: Example of Simplex Noise using Godot

Explicit Noises are another group of noise besides Lattice Gradient Noises, which instead of having a lattice and using interpolation to get the necessary values, are based on mathematical functions, that allow for finding the value at an exact position in space. They often have a more intricate and detailed pattern. Wavelet Noise (Figure 2.4) is one of the most known explicit noises and uses wavelet basis functions to create smooth

textures. It fixes an issue with Perlin Noise, which is only weakly band limited and is therefore prone to aliasing and detail loss. As Wavelet Noise is almost perfectly band limited, these issues are mostly solved with this approach. Anisotropic Noise (Figure 2.5) is another common explicit noise. It supports anisotropic filtering, which mitigates the trade-off between aliasing artifacts and detail loss. Additionally, it allows for high quality and directionally tuned textures and is therefore often used in applications that need detailed surface textures with minimized artifacts.

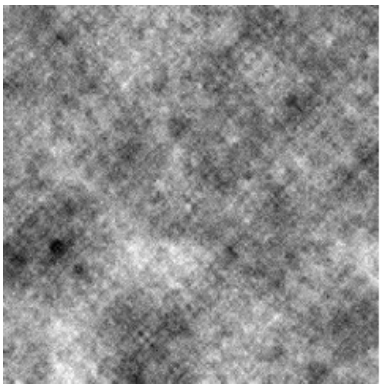

Figure 2.4.: Example of Wavelet Noise [AHI09]

Figure 2.5.: Example of Anisotropic Noise [Ado23a]

To decide which noise type should be used, one has to consider multiple trade-offs:

- **Computational Efficiency**: Anisotropic Noise is generally faster due to efficient filtering, while Perlin Noise can slow down in higher dimensions. Simplex Noise is efficient across dimensions, and Wavelet Noise is typically more computationally demanding.
- **Dimensional Scalability**: Simplex Noise scales well to higher dimensions, making it suitable for complex tasks. Perlin Noise performs less effectively in 3D and above,

whereas Wavelet Noise adapts well across dimensions. Anisotropic Noise is often precomputed at a single scale, requiring real-time adjustments.

- **Visual Quality**: Perlin and Simplex Noise generate smooth gradients suitable for terrains, though Perlin Noise can suffer from aliasing artifacts. Wavelet Noise offers high-quality textures with smooth transitions, while anisotropic Noise provides orientation-specific detail.
- **Memory Requirements**: Wavelet and Anisotropic Noises typically use more memory due to their complex structures. Simplex Noise is efficient, while Perlin Noise can have a high memory overhead, potentially leading to artifacts.

In conclusion, various noise functions serve distinct purposes. "A Survey of Procedural Noise Functions" by Lague et al [LLC+10] provides an in-depth analysis of the most common ones, highlighting their advantages and disadvantages concerning computational efficiency, dimensional scalability, visual quality and memory requirements.

2.2. Uses of Noise and Procedural Generation in Video Game and Simulation Contexts

Having understood the foundational concept of a noise function, one can now look into the uses of noise functions in video games and simulations. Noise functions are essential tools in procedural generation, allowing for significantly improving the richness and variability of environments and other topics. In this section, various ways in which this concept is used will be looked at.

Procedural content generation has been established as a key solution for efficiently creating vast and rich worlds for content creation in topics like video games or movies. As games grow larger and there is an increasing expectation to it providing immersive and detailed worlds, the necessary effort is also growing. In order to decrease the time needed, procedural generation can significantly reduce the workload for design teams, as they can focus on central gameplay elements instead of building the world by hand. Procedural Generation especially shines in creating large, peripheral areas that the player will not interact too much with, but still contribute to the sense of scale and realism of a game. [TCL+13]

In "A Survey of Procedural Content Generation Techniques Suitable to Game Development," Daniel Michelon De Carli et al. [CBPd11] dive into various techniques used in procedural content generation, organized by the content type such as methods for terrains, rivers, roads and cities. Furthermore, these techniques are also classified by the level of human involvement they need. There are assisted techniques, which require significant human input to achieve the desired results, as well as non-assisted techniques, which work without or with minimal human guidance. This paper helps guiding developers in deciding which procedural content generation techniques are suitable for their needs.

One of the most iconic examples of noise-based procedural generation in video games is the terrain generation algorithm in "Minecraft." Alan Zucconi's article, "The World

Generation of Minecraft," [Zuc22] provides an in-depth explanation of the algorithm used to create the unlimited world of the game. "Minecraft" uses Perlin Noise to generate vast landscapes filled with various biomes, terrain features and a cave system. Building onto these basics, the terrain generation of "Minecraft" has evolved over time to introduce more diverse biomes, temperature gradients, and realistic river and ocean systems. Updates, such as those in version 1.18, brought even greater complexity, including new cave types and refined ore distribution that adapts to the environment. These enhancements keep the landscapes of "Minecraft" both immersive and varied, proving the adaptability and power of noise-based procedural generation for creating vast, dynamic game worlds. Further details on the world-generation system of "Minecraft" are documented in the "Minecraft"-Wiki [Min24], which provides exact values, noise functions, and calculations used to determine terrain type and structure in the game. These resources are invaluable for developers trying to understand the practical application of noise functions in creating a procedurally generated world.

Another game that uses procedural content generation is "The Binding of Isaac," as explained by BorisTheBrave in his blog post "Dungeon Generation in Binding of Isaac." [Bor20] The game generates interconnected square rooms on a grid, where the number of rooms increases with each level. Rooms are placed based on specific rules, ensuring an unique layout every run. Special rooms, such as boss rooms and secret rooms are strategically positioned to enhance gameplay. This algorithm allows for high replayability, as each dungeon offers different challenges and surprises.

These examples demonstrate the versatility of procedural content generation across different genres, illustrating how noise functions can not only improve the development process, but also enhance player experience through varied and immersive environments.

2.3. Terrain Generation

Terrain generation is a crucial aspect of creating immersive and expansive planetary environments in video games and simulations. The complexity and realism of these terrains significantly enhance player experience and engagement. This section will look at various procedural generation methods used to create planetary terrains, focusing on noise-based algorithms, fractal approaches and other techniques for automating terrain generation.

In their ICCCI 2020 paper "Comparison of Procedural Noise-Based Environment Generation Methods," Marek Kopel and Grzegorz Maciejewski [KM20] examine various algorithms for generating environments. Their study focuses on identifying effective algorithms for two primary purposes: teleological and oncogenetical. Teleological has the aim of creating terrains that realistically mimic natural processes, while oncogenetical methods prioritize an appearance of naturalness, even if the processes are not reflective of reality. The paper evaluates the performance of four popular noise types, revealing that Godot implementations outperformed those in Unity, with Perlin noise demonstrating the fastest execution time, but with higher memory requirements.

An innovative approach for automatic terrain generation has been proposed by Li

Kui-Ru and Lei Xiao-Feng in their paper "An Accumulated Method Based on Fractal for Automatic Terrain Generation" [LL10]. This method utilizes fractal theory, specifically through the Fault Formation algorithm, to create an initial terrain model. The authors are able to achieve diverse and realistic terrain contours by controlling the iteration and the amplitude of this process. Following this, they use a Middle Point Displacement method to enhance the initial model with additional terrain details, which results in a rich and detailed landscape. This approach not only automates the terrain generation process, as the user simply needs to set parameters, but it also provides the flexibility to produce varying levels of detail, which leads to it being usable in terrain simulation.

Qing-Zhong Li and Xiu-Rong Gao present an innovative method for generating controllable and predictable natural terrains using Fractal Brownian Motion (FBM) in their paper "Generation and Visualization of 3-D Realistic Natural Terrain Based on FBM" [LG07]. Fractal Brownian Motion is a stochastic process that is especially good at describing many phenomena in nature and, according to the paper, is said to be the best stochastic process for modeling natural terrains at the time this paper was written. The authors begin by dividing the base plane of the terrain into several triangles using coordinates from known characteristic points. They then apply the modified random midpoint displacement method to recursively generate three-dimensional data for the terrain within each triangle. This approach allows for the creation of a diverse range of terrain types, including multi-peak mountains and flat surfaces, which traditional FBM methods struggle to produce in a controlled manner. By adjusting parameters during the generation process, users can achieve both predictability and control over the features of the terrain, addressing the limitations of previous FBM models that often resulted in unpredictable terrain. The simulation results demonstrate that this novel algorithm not only enhances visual realism but also provides an efficient and effective solution for generating various terrains in a controllable way.

Another approach for automatic terrain generation is presented by Kenichi Sugihara and Takahiro Murase in their paper "Automatic Generation of a 3D Terrain Model From Key Contours" [SM17]. They utilize straight skeleton computation to automatically create 3D terrain models from key contour polygons. By assigning specific elevations to these contours, the system constructs the faces that connect them, resulting in a realistic 3D terrain representation. This method integrates Geographic Information Systems with computer graphics, streamlining the traditionally labor-intensive process of generating terrain models and building sites. Furthermore, the paper introduces a mass granular flow simulation technique that employs Newtonian mechanics to predict natural disasters, such as avalanches or debris flows, on the generated terrains and 3D town models. This combination not only enhances terrain realism but also provides valuable insights into how natural hazards alter the created environments.

In their paper "Procedural Generation of 3D Planetary-Scale Terrains," Ryan J. Vitacion and Li Liu [VL19] investigate the complexities of generating large-scale virtual terrains using procedural methods. They emphasize the inefficiency of manual modeling for expansive environments, such as entire planets, which can consume considerable time and resources. The authors explore various noise algorithms for calculating terrain height,

assessing their performance in terms of generation time, memory usage, and quality. Their tests reveal that Perlin and Simplex noise exhibit good generation times; however, the performance of Simplex Noise declines at higher resolutions. Similarly, while the Diamond Square Algorithm has comparable generation times, it requires more memory than the other options. Furthermore, algorithms like Perlin and Simplex noise provide significant geometric variation and fine detail while minimizing directional artifacts, especially when compared to alternatives such as the Cubic Algorithm or the Diamond Square Algorithm.

"Procedural Terrain Generation Using Generative Adversarial Networks" by Voulgaris et al. [VMP21] addresses the challenge of generating realistic synthetic terrain for virtual reality applications through a novel approach using Generative Adversarial Networks (GANs). Unlike traditional procedural algorithms, which often require significant manual effort, the proposed method, known as GAN-terrain, simplifies this process by allowing users to create terrain images from rough 2D scatter plots of desired Points of Interest. The GAN is trained on these Points of Interest and their corresponding satellite images, enabling rapid production of photorealistic terrain images. By reducing the need for complex input data, such as height maps, GAN-terrain requires minimal manual supervision during inference, making it user-friendly and efficient. Creating a 3D terrain from such an image can then be easily achieved by interpreting the colors to derive height information.

In "Terrain Generation Using Procedural Models Based on Hydrology," Genevaux et al. [GGG+13] present a framework for quick and intuitive terrain modeling inspired by hydrological concepts. The proposed method allows terrain generation from a simple initial sketch, controlled by a few parameters, using an analytic and continuous representation stored in a novel construction tree. This tree structure combines operations at internal nodes with terrain features at the leaves. The framework utilizes rivers as primary modeling elements, first creating a hierarchical drainage network represented as a geometric graph. The network is then analyzed to construct watersheds and characterize river types and trajectories. The final terrain is generated by blending procedural terrain and river patches through carving operations. This method enhances controllability compared to traditional procedural and physics-based techniques while producing geologically reliable outputs. It allows for the representation of large terrain models with complex river networks and geomorphological consistency. The main contributions include an intuitive framework for procedural terrain generation, a hydrology-inspired generation technique, and an efficient hybrid terrain data representation for editing and visualization.

In the paper "AutoBiomes: Procedural Generation of Multi-Biome Landscapes," Roland Fischer et al. [FDWZ20] address the challenges of creating large, realistic virtual landscapes, particularly those composed of multiple biomes. They introduce a system called AutoBiomes, which combines various procedural terrain generation techniques, digital elevation models, and simplified climate simulations to efficiently create expansive terrains with diverse biome distributions. The system also features an intuitive asset placement component for complex distributions of multiple assets on the terrain. By focusing on usability, the authors demonstrate that AutoBiomes enables the rapid creation of realistic terrains, balancing realism, performance, and flexibility in the process.

In conclusion, the need for terrain generation, especially for huge landscapes or planets,

is reflected in the various innovative techniques developed, for example noise-based algorithms, fractal generation, and hydrology-inspired frameworks. Each approach brings its own unique strengths and weaknesses, addressing challenges related to realism, scalability, and user control. As the demand for expansive virtual worlds continues to grow, the number of procedural terrain generation algorithms used will increase even more, offering a large variety of options for enhancing user experience and engagement in video games and simulations.

2.4. Procedural Planet Generation

Now that a foundation has been established for generating planetary terrain, the next step is to explore the creation of immersive procedural planets. This process extends beyond just terrain height to add a variety of elements that contribute to realism, much like those seen on Earth. Key components such as the sky, a day-night cycle, vegetation, and various environmental features significantly enhance immersion in virtual worlds. In this chapter, two notable tutorials on procedural planet generation will be examined, alongside a discussion of the acclaimed game "No Man's Sky," which shows successful implementation of these techniques. Furthermore, research on planetary biomes will be looked into, how they function and how they are generated procedurally, as well as the atmospheric effects, including weather patterns.

One key resource for this thesis is Sebastian Lague's Unity tutorial series [Lag18], which provides a thorough approach to creating a procedurally generated planet. The tutorial starts with constructing a spherical mesh, then progressively adds features: customizable settings, layered noise using Fractal Brownian Motion for complex terrain, noise filters, and a shader for the surfaces. Lague also introduces basic biome generation and adds an ocean floor to integrate land and water seamlessly.

Another invaluable resource for this project is the YouTube series of SimonDev on 3D world generation [Sim21], which dives into performance and scalability aspects critical for procedural planet generation. Unlike the YouTube series of Lague, the tutorials of SimonDev emphasize using a quadtree structure and Level of Detail (LOD) system to optimize terrain generation. The series starts by creating a height map with Perlin Noise, implementing the quadtree, and adapting the model into a planetary shape. He then covers advanced techniques like triplanar texturing, atmospheric scattering, and mesh stitching to prevent gaps between quads with different levels of details. Additionally, he addresses threading for performance, numerical precision, and biome generation. Together, these techniques provide an essential foundation for generating expansive, immersive worlds and offer effective solutions to many common issues encountered in procedural planet generation and LOD systems.

In "No Man's Sky" [Sky24], players begin their journey on a randomly generated planet in a star system within the Euclid galaxy, experiencing diverse planetary characteristics. The game features over 18 quintillion possible planets, each with unique biomes, weather conditions, and flora and fauna frequencies, which are revealed upon player exploration. For instance, planets can have distinctive features based on their biome type, such as the

exotic biome, which has large hexagonal formations that are not found elsewhere. A key aspect of "No Man's Sky" is its use of procedural generation to create these environments. This includes algorithms for generating terrain, atmospheric conditions, and ecosystem interactions, allowing for dynamic and immersive exploration. Planets are also tidally locked, resulting in a consistent day-night cycle without the planets rotating on their axes. Each Sol, representing a day, lasts about 15 real-time minutes, allowing the sun to rotate around the planet once. Additionally, planets can host moons and can also contain various Points of Interest for players to discover. The procedural generation of the planets significantly enhance the exploratory aspect of the game. These concepts and techniques from "No Man's Sky" can be very useful for the project of this thesis, providing insights into effective methods for implementing procedural generation as well as giving ideas on how to create a immersive experience for the player.

In summary, the exploration of procedural planet generation reveals an abundance of techniques and concepts that can significantly enhance the immersive experience of virtual worlds. By examining valuable resources such as the tutorials of Sebastian Lague and SimonDev, one gains practical insights into building complex terrains and optimizing performance for planetary environments. Furthermore, the case study of "No Man's Sky" gives an amazing example for successful application of procedural generation in creating diverse and dynamic planetary features, including unique biomes and day-night cycles. The lessons learned from these resources not only provide a solid foundation for this thesis project but also inspire innovative approaches to crafting immersive planetary experiences.

2.5. Level-of-Detail (LOD) Systems and Performance Optimization

After defining the planetary terrain, the next step is to ensure efficient rendering by adjusting the detail level based on proximity to the player. Closer regions need high detail, while distant ones can be simplified, optimizing both the rendering speed and resource use. This section explores approaches like quad trees, adaptive LOD systems, and techniques for large-scale terrain optimization to manage the complexity of procedural landscapes. Additionally, performance optimization strategies, including GPU-based and parallel processing techniques, are discussed to maintain a seamless experience across large terrains.

Liang, Zhou, and Liu's work in "Research on Large Scale 3D Terrain Generation" [LLZ14] explores essential performance strategies for large-scale terrain generation, which faces limitations in graphics processing power. Given the high demands of large terrains in geographic information systems, virtual reality, and simulations, they emphasize Level of Detail models as a core optimization strategy to allow rendering expansive terrains with limited processing capabilities. They discuss two common LOD model types: the triangular irregular network (TIN) and the regular square grid (RSG), noting that RSG offers a faster build rate and is typically favored in real-time applications. However, in large-scale terrain, LOD alone does not fully address performance needs, prompting

the authors to propose additional optimization methods. By introducing an improved quad-tree-based LOD algorithm and a view frustum clipping technique, they show how subdividing terrain with a tile pyramid model and selectively rendering terrain based on proximity to the viewpoint can effectively balance performance and detail. Their experimental results demonstrate that these methods reduce memory use and loading times, important considerations for creating real-time, large-scale terrains without sacrificing visual quality.

In "Learning Based Infinite Terrain Generation with Level of Detailing" Aryamaan Jain et al. [JSR24], bring up a novel approach on how an infinite terrain could be created. Current approaches based on procedural generation often have the problem, that their realism is not at an acceptable level. Therefore, their idea is to use a quad tree and learning based generative network. This approach works seamlessly with most quad tree implementations and it also minimizes errors occurring because of the quad tree structure, like the visible gaps in the mesh caused by differences in mesh height between two quads with different LOD levels. After creating the algorithm, they extensively test the approach, proving that the generated terrain is highly realistic and has effective level of detailing.

In "GPU-Accelerated LOD Generation for Point Clouds," Schütz et al. [SKKW23] propose a GPU-accelerated approach for generating Level of Detail structures for point clouds, which are essential for handling large-scale terrains. Their method combines voxel and point representations within an octree structure, optimizing lower LODs through a hybrid of layered point clouds and voxelization. This strategy improves performance significantly, achieving construction rates of up to 4 billion points per second on a high-end GPU, compared to CPU-based methods which cap at around 12 million points per second. The GPU-optimized pipeline not only accelerates LOD generation but also enhances visual quality at lower resolutions through color filtering. Although it primarily addresses point cloud data, the principles of octree-based data partitioning and GPU acceleration could be adapted to improve performance in procedural landscape generation, especially for high-resolution terrain models.

In conclusion, optimizing level-of-detail (LOD) systems is crucial for efficient and realistic procedural planet generation. The strategies discussed, such as quad trees, adaptive LOD systems, and GPU acceleration, show the importance of managing the complexities of large-scale terrains to improve visual quality while conserving computational resources. By using advanced data structures and performance techniques, significant gains in rendering capabilities can be achieved. The research of Schütz et al. also shows how GPU technology can enhance LOD generation, addressing the needs of expansive virtual environments. Together, these methods enable the creation of immersive worlds that balance detail and performance in procedural generation.

2.6. Conclusion

In this chapter, the fundamentals of procedural generation have been explored, beginning with foundational noise functions that serve as the basis for creating diverse and realistic

environments, emphasizing the trade-offs between them. The application of these noise functions across various contexts, particularly in video games and simulations, is examined through examples like "Minecraft." With this foundational knowledge established, terrain generation for planets is researched, focusing on different approaches taken to create rich and engaging terrains. After understanding how flat terrain can be generated, the next logical step is the creation of spherical terrain that reflects planetary surfaces.

To address the complexities beyond mesh generation, two tutorials are analyzed, along with "No Man's Sky," which serves as the primary inspiration for the project. The final step in crafting an immersive planet involves optimizing performance to maintain interactive frame rates. Several strategies are considered, including the use of a quadtree structure or GPU-accelerated mesh generation.

3. Methodology

The following chapter describes key considerations made in developing the project, their advantages and disadvantages, as well as an explanation on the thought process behind the selection of the solution. First, the development environment used for working on the project is discussed. Next, several methods for generating a spherical planet are examined, followed by evaluations on the height generation algorithm, which adds the topographical variety. Another crucial step is the Level of Detail (LOD) system: As the player explores the planet, high resolution detail is necessary for terrain close to the player, while more distant areas on the planet do not need this resolution, because the difference is not noticeable. The final steps involve adding interesting details to the planet to enhance realism. While many features could be added, the focus was on adding the most impactful options. This includes a space background, a sun, atmosphere and a halo, clouds, trees, high resolution textures for the planet and a fitting player model. The chapter concludes with a discussion of additional features that could be added in further iterations.

3.1. Development Environment

For the development environment, there were many options to choose from. One of these would be to use an already established game engine like Unreal Engine [Unr] or Unity [Uni]. Using one of these has the advantage of working with cutting edge technology, because they are the main engines that are used by many game studios. There is also strong community support for these, whether in forums, on Reddit, or through YouTube tutorials. However, these powerful engines also have the drawback that they require more resources to run the editor and game, which can lead to longer loading times. Additionally, the code base is much larger than in smaller engines, which makes it harder to tailor the engine to specific needs.

Another option would be to use a less established tool like Godot [God]. One disadvantage of choosing this route is the lower amount of support and tutorials it receives, as fewer people use it. Additionally, since these alternatives are not as fleshed out as the big contenders, some important features might be missing in the current versions. Not only that, but game performance might also suffer due to missing optimizations that have yet to be implemented. However, there are also upsides to using such a framework, specifically Godot. For one, it is open source, making it easier to modify for those who want to adjust its behavior. Additionally, many developers contribute to the project out of passion, continuously improving it into a polished and capable tool. Another advantage of Godot is its scene system and intuitive scripting language, which allow for very rapid prototyping.

The third option would be to create a custom game engine specifically for the project. This is by far the most time consuming and complex option, as all the necessary features must be implemented from the ground up. But the big advantage with this approach is that one has complete control over what they add. Therefore, the game engine can be tailored precisely to the needs of the project, potentially making it extremely performant. Additionally, this is a great opportunity to learn computer graphics and game engine related concepts. [Tyl]

The project in this thesis was created using Godot, not only because of the fast prototyping but also because of the possibility to learn more about the engine. The features of the engine were sufficient for the planet generation project and some quality of life changes got introduced in recent versions, which made this option even more viable. Additionally, the process of creating an executable file for the user to play the game is very simple.

3.2. Planet Mesh Generation

Several factors had to be taken into account in order to create a suitable mesh for the planet. In order to accurately represent a planet, this mesh has to start out sphere-like, to make sure that it can later receive deforms that turn it into a planet. Additionally, this mesh must be able to be representable in several resolutions. By ensuring this, the base mesh is able to further subdivide in later steps, when a higher level of detail is necessary. Another important factor is the distribution of the vertices. There are sphere generation methods, that will lead to good vertex distribution around the equator but a very poor and stretched distribution around the poles. This is not desirable, as the poles also need similar resolution as the rest of the planet. Several common methods are used for generating sphere meshes, each with distinct advantages and disadvantages depending on the intended application.

The UV Sphere (Figure 3.1) is one of the simplest and most widely used approaches. This method tessellates a sphere along its meridians and parallels, resulting in a mesh composed of normalized quads at the equator, which gradually become triangles toward the poles. Its simplicity makes it easy to create and map textures onto, making it a popular choice for basic applications. However, the UV Sphere suffers from poor vertex distribution near the poles in form of these triangles, leading to visual artifacts and uneven geometry, which can be problematic.

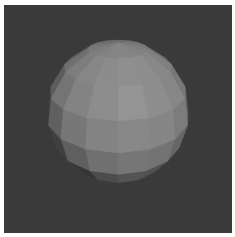

Figure 3.1.: Example of an UV Sphere

The Icosphere (Figure 3.2) on the other hand is created by subdividing an icosahedron, a polyhedron with 20 triangular faces, to generate a mesh composed entirely of triangles. This method provides more uniform vertex distribution than the UV Sphere, making it particularly suitable for applications where even surface geometry is critical. It is also highly scalable, allowing for multiple levels of detail. However, the Icosphere has a slightly higher computational cost due to its subdivision process, and its UV mapping can be more complex to implement.

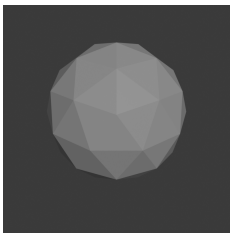

Figure 3.2.: Example of an Icosphere

Another commonly used technique for sphere generation, the quadsphere (Figure 3.3), uses quads as a base. It starts out with a hexahedron (a cube) that is subdivided on each side and then projects each vertex outward to lie on the surface of a sphere. This method is particularly effective in environments that require grid structures, such as planetary surfaces, and is compatible with quadtree-based level-of-detail systems. Nevertheless, achieving uniformity in vertex distribution is challenging with this approach, which can lead to inconsistencies in the appearance of the sphere.

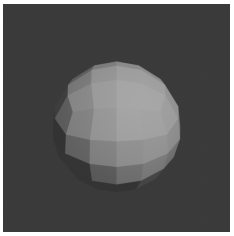

Figure 3.3.: Example of a Quadsphere

Finally, the Goldberg Polyhedron (Figure 3.4) represents a more complex method of sphere generation. This approach involves constructing a mesh with hexagonal and pentagonal faces, resulting in a shape reminiscent of a football. The Goldberg Polyhedron provides exceptionally even vertex distribution, making it ideal for high-precision applications. However, its implementation is considerably more complex, and real-time rendering can be computationally intensive, making it less practical for many scenarios.

In conclusion, there are many possible options for building a sphere mesh. With the requirements for a procedural planet to be able to be subdivided on the go, the quad sphere is one of the most promising approaches as it works very well with a quadtree

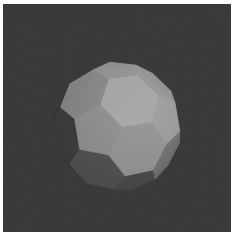

Figure 3.4.: Example of a Goldberg Polyhedron

and has a good distribution of vertices, albeit not as good as something like a Goldberg Polyhedron [Sie21].

3.3. Height Calculation

The next step in the process of creating a procedural planet is the height calculation. For each of the vertices, a height has to be calculated, which translates the vertex away from the start in order to create terrain. A good terrain is one, that convincingly resembles a planetary landscape. Part of this involves the ability to generate features like mountains and beaches, as they contribute to a visually rich environment. In order for this to work in real time, the height calculation method must be efficient, ensuring that the terrain generation does not significantly decrease performance. Additionally, the system should support an increase in quality and detail as the mesh is refined and more vertices are included. This scalability allows for the representation of visually impressive features and variation in elevation, adding to an overall immersion and realism of the generated planet.

After evaluating several methods, two distinct height calculation approaches were chosen for their strengths at the respective planet generation algorithm. The first planet generator uses Fractal Brownian Motion with Perlin Noise layers. The basic idea is to calculate the height by adding together multiple layers of Perlin Noise, each with a different frequency and amplitude. This idea is called Fractal Brownian Motion. This leads to a detailed height map, allowing for simulating a variety of features such as hills and mountains. The smooth transitions between these features contribute to a more natural appearance, making the Fractal Brownian Motion approach a great choice for generating procedural planets.

Inspired by the procedural height generation algorithm technique used in “Minecraft”, the second planet generator utilizes three values generated by Perlin Noise per vertex to determine its elevation, with an additional temperature value to influence biome characteristics. This method allows for the generation of distinct features and supports many terrain types. To further enhance control, spline points are used to fine tune the rise in elevation. Figure 3.5 describes this idea for one of the 3 height values. If the value generated by the Perlin Noise is -1, then the terrain height is 50. If the noise value is between 0.3 and 0.4, then the terrain height will be linearly interpolated between 50 and 100. In a similar fashion, this continues for the rest of the example. These two ideas allow

for a very detailed and customizable planetary terrain, which might have a very positive effect on immersion.

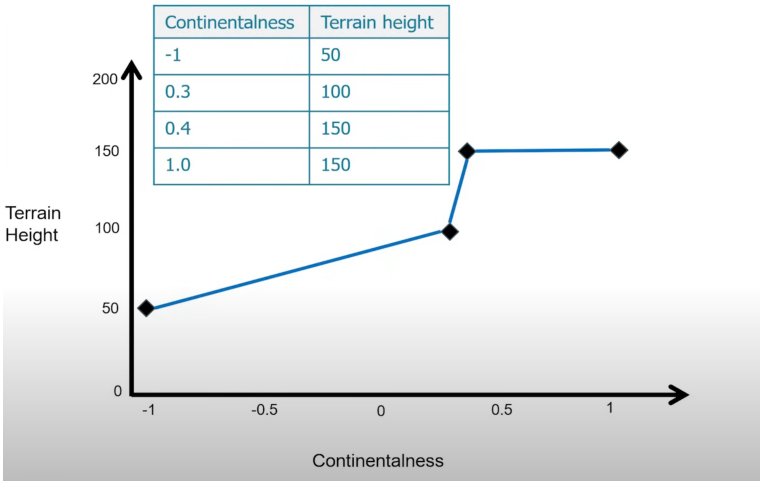

Figure 3.5.: Spline point example from Henrik Kniberg [Hen22]

3.4. Level of Detail

To achieve a visually rich and performance-optimized procedural planet, it is usually not possible to create the full planet at maximum resolution, as this would cause too many vertices and assets to be loaded which slow down the game. Instead, a Level of Detail (LOD) system has to be created, that allows for dynamical adjustments of the complexity of parts of the terrain mesh based on the proximity of the player to it. Close-up views require a high level of detail for the user to be immersed, while parts further away can be represented with simpler geometry to reduce load and render times. The LOD system used in this project must meet several requirements:

- Real time adaptability: The LOD system must be able to operate in real time, with levels adjusting smoothly as the player moves to or away from the planet without causing noticeable visual artifacts or lag. Each of the quads should be doing this on their own.
- Efficiency: CPU, GPU and memory usage should be as low as possible while still being able to render a planet at a high level of detail.
- Smooth transitions between levels: There should be no visible gaps between two quads with different resolution.

With these needs in mind, a quadtree-based LOD system was selected to handle the varying detail levels across the planet's terrain. Quadtrees are hierarchical structures,

that efficiently manage two-dimensional spatial data by recursively subdividing quads into four smaller quads. This approach was chosen, because it enables flexible detail adjustments based on the player's distance from the terrain. Additionally, quadtrees are lightweight and well-suited for real-time rendering of a large-scale, dynamic planet mesh.

However, achieving perfectly smooth transitions between LOD levels in a quadtree structure is challenging, especially when a quad with higher resolution is next to a quad with lower resolution. This leads to visible gaps, which can luckily be mitigated in several ways. One idea is to extend the sides of the mesh downwards and color them in a similar fashion as the other parts of the planet. This approach helps maintain visual continuity, improving the experience of the player as they explore the planet (see Section 4.3 for a detailed discussion on this issue and the solution used).

3.5. Adding more Details to the Universe

While a solid working planet mesh is already good, additional objects and fitting textures add greatly to the immersion. This is an area, where an infinite amount of detail can be added. Therefore it was important to find a good balance between increasing immersion and time.

One of the best additions to increase the atmosphere of a planetary simulation is the presence of a sun and with it a light source, allowing for shadows. Theoretically, it makes the most sense for the sun to be a point light, meaning that the light starts at the center of the sun and then illuminates the whole scene from there. Unfortunately, long distances and a large amount of objects make this approach inefficient, especially with increasing object count. Another option is to use a directional light, with the direction going from the center of the sun to the center of the earth. While this is by no means exact, the calculation of shadows is much faster and scales way better with object count, therefore this approach is chosen over the other one. Just like the sun, the moon is an essential object for increasing realism of the sky in a scene. While the moon does not directly send out light, it still rotates around the earth but with a smaller radius and in a faster manner. Furthermore, the moon should be shined on by the sun, illuminating one side and leaving the other side dark, which leads to visible moon phases on the planet. The final step to have a good working moon is to make sure, that the moon is mostly visible at the night side of the planet, while it should be almost invisible or not visible at all during day time. In this project, all of these factors are respected and implemented in order to have a good and dynamic sky.

One of the more complex parts of a procedural planet generator lies in the creation of a convincing atmosphere and halo for the planet. This can be split up into a few main goals. First, when being on the bright side of the planet, the sky should be completely blue. Next, when the player is on the dark side of the planet, the blue sky should be invisible, showing the universe as well as the moon. Third, when flying from the bright side to the dark side, the blue color should slowly fade into an orange tint, before it disappears at some point. Conversely, traveling into the other direction should start with a night sky, slowly turn orange and then fade into the blue sky at day. The last

subject that needs to be addressed is the halo of the atmosphere, which is visible when flying around outside of the planet's atmosphere and looking at the planet. This halo has a similar color as the blue sky on the planet and should be the strongest, where the sun shines at the planet. For parts, that deviate from the perfect spot, the blue color should slowly fade into invisibility (Figure 3.6 shows, how this looks on earth). To implement the planet's atmosphere, a simple solution is to add a sphere around the planet that displays different colors depending on the lighting conditions: blue, orange, or transparent as necessary. The correct color can be determined by calculating the angle between the planet center-to-player vector and the planet center-to-sun vector. The halo around the earth, which is called atmospheric scattering, on the other hand is extremely complex. Calculating it accurately is a computationally intensive task and therefore most computer graphics models implementing this phenomenon are using a simplified version. Additionally, many models expect the user to always be on earth or at least close to it, which is not the case in this project. Luckily, with some adaptions, an algorithm can be created, that allows for a nice atmospheric scattering effect at interactive framerates, while being able to move around in space. "GPU Gems 2," a book written by NVIDIA [NVI04] in depth shows how this can be done by first setting up the necessary scattering equations, namely Rayleigh Scattering and Mie Scattering. In the next step these equations are solved and the whole concept is translated into a shader language, so the GPU can accelerate the process. While this concept is extremely interesting, a good enough result can be achieved with a way simpler approach. This is done by using another sphere with a size slightly bigger than the planet and the maximum height of objects on the planet like mountains and clouds, which aids as the halo sphere. With this setup, the dot product between the light direction and direction of the vertex can be used to create a good looking atmospheric scattering. In this project, either the atmospheric scattering or the atmosphere is used, depending on the current position of the player, resulting in a convincing representation of how these effects work on earth while still maintaining simplicity.

A naturally occurring phenomenon on earth is the presence of clouds, which float across the sky. Adding these to the planet increases the immersion. Creating clouds is straightforward, as they can be represented as a billboard with a cloud texture on them. Billboards are simple quads that always face the camera, such that it appears to the player as a three-dimensional object. The final step is to place them in the scene in a logical and visually appealing pattern.

The next step is creating a compelling background for the universe. The most logical solution is to have something similar to how the universe looks like from earth. There are some possibilities on how to accomplish this. One option is to use the inbuilt functions of Godot, where a sky with a sky material can be added and altered [God24c]. Even though this works quite well, there are other options that work even better. Another solution would be to use a sphere, color it black and place random stars around the universe. Unfortunately, the big amount of stars that are needed to make this look reasonably good, make this not feasible. An even easier and by far the best working option is to use a skybox. The basic idea is to use six quads with a texture on each of them, one for the

Figure 3.6.: Image of the Earth showcasing the Halo [NAS16]

top part of the skybox, four for the side parts and a final one for the bottom part. Godot offers a feature to add these textures into the scene, making it easy to use. The harder part is to create or find these textures, as they need to have a seamless transition or otherwise it will not look convincing. Luckily, there is a free tool, "Spacescape," [Ale24] which offers an easy way of generating a space sky by placing stars or adding nebulas. When done, it is simple to export the file to create the necessary skybox, which can then be used in the engine.

Trees are crucial for enhancing immersion in environments like grass landscapes, forests, or beaches. When it comes to procedural planet generation, there are mainly two concerns: Generating the trees and placing them on the planet. For the tree generation there are at least two good possibilities. The first one would be to have a billboard of a tree, similar to how the clouds are done, which can be rendered very fast in large amounts. The second option would be to create a model of a tree in a modeling software like Blender and then add it to Godot. As the first approach looks great but not at a low distance and the second option does look good at any distance but is much more costly for the performance, many games use a compromise between the two ideas. They render the trees as billboards when the player is far away and as soon as the player gets close enough, they swap out the billboard with the tree model. When done correctly, this works well and leads to very convincing trees [CCH05]. Unfortunately, due to the size of the planet and the level of detail system of this project, it does not make sense to spawn the trees immediately. For one, this would cause the engine to have to spawn thousands and thousands of trees. Secondly, the trees would change drastically in position, since increasing or decreasing the level of detail could also update the terrain height by a lot. This leads to the trees

jumping around, decreasing immersion. Therefore the simplest and best idea is to just spawn tree models at a certain level of detail and remove them again, if the level of detail is lower than that. The second problem that has to be solved is how to procedurally place the tree on the planet as well as how many. While there are numerous options for this, a very simple procedure would be as follows. Depending on the biome or height of the center of a certain quad, the number of trees is set. For example, trees could be spawned at a normal rate on some biomes and spawn at higher rates in forest biomes. A simple solution for the placement of the trees can be to place them randomly on the quad it belongs to, making sure that the height is correct.

One effective way to enhance the appearance of a planet is by adding colors. There are at least three effective methods to transform an uncolored planet into a visually compelling one.The first method is to simply select a color and apply it to parts of the planet using the shader. A second option is to use textures instead of a uniform color. Although setting up textures requires more work, the results can be highly realistic. This process begins with creating or sourcing suitable textures, which are then imported into the engine. In the engine, the textures are sent to the shader, where they are sampled and applied to the surface of the planet. An advantage of this method is that additional textures, such as normal maps, can be layered to increase surface realism by adding detail and variation. The third approach is to use the shader to generate textures procedurally. These textures are created through mathematical functions and color variations rather than images, which is especially effective for dynamic elements like water that require wave-like movement. For this project, textures were chosen to enhance the appearance of the planet at close distances. At greater distances, the detail becomes less visible making textures unnecessary. As a result, textures are only used once the planet reaches a certain level of detail on any of its quads, transitioning from solid colors to detailed textures as needed. The water and lava textures on the planet are generated in the shader and at lower detail levels also revert to simple colors. The imported textures include not only an albedo map (color), but also a displacement map (to vary vertex height), a normal map (to adjust surface normals for added detail), and a roughness map (to control texture shininess). One challenge with textures is texture repetition, where the same texture repeats across the surface, creating visible patterns. Although fully eliminating this repetition is difficult, several methods can reduce its visibility. A discussion of these methods is beyond the scope of this thesis.

In most games, designers must decide between a first-person or third-person perspective for the player. In a first-person view, the camera represents the eyes of the player, offering a direct line of sight. In contrast, a third-person view usually places the camera behind the player model, providing a broader view of the environment. Both approaches are valid for a planetary exploration game. While a first-person perspective can slightly increase immersion, a third-person perspective offers a clearer view of the surroundings and increases the ability to navigate in three-dimensional space. Additionally, when the player model is a spaceship rather than a human character, a third-person camera is more intuitive. Considering these factors, this project uses a third-person camera and a rocket as the player model.

One of the primary challenges in this project is selecting features that significantly enhance immersion while avoiding those with minimal measurable impact. The main features implemented were detailed in the previous sections, but many other possibilities could also increase the immersion of the project. While it is impossible to explore every potential enhancement in this thesis, the following are some notable examples. Adding dynamic wildlife, such as birds, land animals, or fish, would make the ecosystem feel more alive by introducing other living creatures alongside the player. An explorable ocean, populated with fish and marine plants, could further enhance the sense of immersion. To adjust for underwater exploration, the player model could even transform into a submarine when diving into the ocean. While the current plains already appear acceptable, incorporating grass clips or additional vegetation could make the planet's surface more convincing. Techniques such as compute shaders on the GPU could generate large-scale grass fields without significant performance costs. Introducing human-made structures, like houses, cities, or ancient landmarks such as pyramids, would create points of familiarity and intrigue for the player. Expanding the diversity of biomes and textures would further enrich the world, with a specific look at "Minecraft", which has already established a well-evaluated system of biomes usable for this project. Besides this, texture repetition is a problem that will likely break immersion. Addressing this would require implementing advanced techniques to reduce noticeable patterns, but completely eliminating them is challenging. In addition to these features, a more realistic atmospheric rendering with enhanced halos could contribute to the visual realism of the planet. While physically accurate simulations may be too computationally expensive for such a project, simplified approximations could achieve a more convincing effect with acceptable performance. Finally, the transitions between biomes are currently abrupt. Smoother, more gradual transitions would improve the cohesiveness and realism of the environment. Given that projects like this can be expanded almost indefinitely, this thesis focuses only on the ideas described before to balance quality with practical time constraints. There are many additional features and approaches not covered here that could further increase immersion, offering exciting opportunities for future exploration.

4. Technical Implementation

This chapter discusses the development of the Godot project and the necessary steps for replication. The first topic of the technical implementation focuses on the engine, covering how to download and compile it, as well as how to set up a basic project. Next, the implementation of a rocket-shaped player character is described, which facilitates debugging and testing in the game. Then, a simple cube is created and gradually improved, resulting in a uniquely shaped planet with detailed textures and engaging terrain. Finally, this thesis explores additional features to enhance immersion, such as a sun, clouds, and other elements.

4.1. Creation of the Project and Engine Setup

The engine utilized for this project is Godot, an open-source game engine that supports the development of both 2D and 3D games across various platforms and hardware configurations [God24b]. Godot is particularly appealing for its flexibility and its open-source nature, allowing developers to easily modify the engine to suit their specific needs.

One of the major challenges encountered in this project was the scale of the planets, which made it necessary to have a more accurate handling of spatial coordinates than the standard precision offered by Godot. By default, the engine uses 32-bit floating-point numbers to represent positions. While this format provides a sufficient range for many applications, it introduces significant inaccuracies when dealing with very large values, such as those required for planetary scales. As the vertex values increase, the number of available bits for fractional representation decreases, leading to noticeable errors in positioning. A visual example of this issue is illustrated in Figure 4.1.

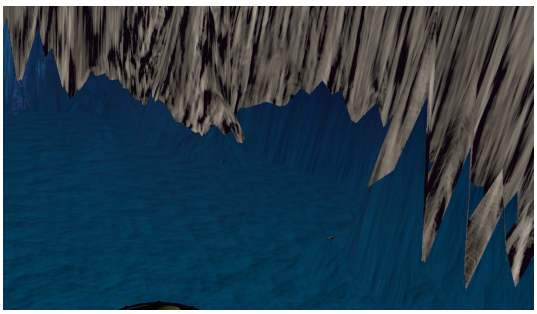

Figure 4.1.: Precision issues at large scales with 32-bit floating-point numbers

To mitigate this issue, the Godot engine can be compiled from source with the `precision=double` flag, enabling the use of 64-bit double precision values internally. This modification allows for greater accuracy when working with large planetary scales, as it preserves the necessary precision for position calculations without sacrificing the range. To compile Godot with this setting on Windows, these steps can be followed:

1. Install Visual Studio Community, Python 3.6+ and SCons 3.1.2+.
2. Open the terminal and run the following command:
```
python -m pip install scons
```
3. Download the latest Godot source code from the official GitHub repository.
4. Navigate to the Godot source folder in the terminal.
5. Use the following command to compile Godot with double precision:
```
scons platform=windows target=release precision=double
```

Further details on the process are available in the official documentation [God24a]. After compilation, the engine will support double precision and can be used to create a new project.

4.2. Rocket Setup and Movement

In this project, the player controls a rocket, which serves as the primary vehicle for navigating the scene. To implement this in Godot, a 3D rocket model was combined with components and scripts for movement, collision, and camera control. First, a high-quality 3D rocket model was selected and imported into the project. Next, a dedicated scene was created for the rocket, containing the camera and movement scripts. The imported rocket model was attached to the main node of this scene in order for it to be displayed. A capsule-shaped collision object was added to the main node, ensuring smooth and accurate collisions while keeping the setup simple. To implement a 3D camera that follows the orientation of the rocket, it was necessary to add several nodes in a nested hierarchy. Firstly, "CameraRoot" is added as the main node for the camera component. It dictates the distance of the third-person camera to the player character. Next, a "TwistPivot" is attached to this node, which handles the horizontal rotation upwards and downwards. For handling vertical rotation to the left and right, the "PitchPivot" is attached to the twist pivot. Finally, the actual camera is added to this pitch pivot. With this hierarchy and tweaking of the positions and rotations of the pivots, a third person camera for the rocket has been created. It easily allows for rotations in all directions, while keeping the distance to the player the same. Custom controls were set up in Godot's project settings to handle movement and camera focus. The primary movement controls allow the rocket to move in all six directions: `W` moves forward, `A` moves left, `S` moves backward, `D` moves right, `Q` moves upward, and `E` moves downward. Holding `Shift` increases movement

speed, allowing for quicker exploration of the universe, especially useful for reaching the planet. Furthermore, pressing `F1` enables "Capture Mouse Movement" mode, allowing the mouse to control the camera, while pressing `Escape` disables it. To achieve smooth and responsive camera rotation, an additional script handles horizontal and vertical rotation. By capturing mouse motion and applying it to the camera pivots, the camera view can be adjusted seamlessly with the orientation of the rocket. With this setup, the player can navigate the scene smoothly with responsive controls and an intuitive camera system.

4.3. Planetary Sphere

This section provides a detailed explanation of the approach used to generate the planet meshes. First, the basic mesh generation is explored, ensuring an even distribution of vertices across the planet. In this step, a major focus is ensuring the mesh can later use a quadtree structure for Level of Detail purposes. The quadtree introduces some challenges, which are identified and addressed. Next, collision handling is explored, followed by height generation, which is the main difference between the two planets.

The basic approach to generating the planetary mesh involves building a subdivided cube and transforming it into a sphere by moving each vertex a unit distance outward. This method allows for efficient handling of quads, which will later support a quadtree structure that is capable of adjusting rendering detail as needed. The first step of the mesh generation is about initializing the face normals and setting the resolution of the planet. Face normals define the orientation of each side of the cube. Next, for each side of the cube, a grid of quads based on the chosen resolution is created and each quad is aligned with its corresponding face normal. The first vertices that are created are the corner points of each quad, which are calculated using the base normal of the face and two axis vectors (`axis_a` and `axis_b`) that define the orientation of the face. Next, the inner vertices of the quads have to be generated. This is done by distributing vertices evenly within each quad, where a higher resolution would lead to them being closer together. The cube is transformed into a sphere by normalizing each vertex position and scaling it to the desired radius, forming a quad sphere. Each quad must store and propagate its information to child nodes, enabling dynamic LOD-based subdivision and merging. Each grid point is added to a list storing the quad's vertices, normals, and UV coordinates. The normal is computed by normalizing the vector from the sphere's center to the vertex and the UV coordinate is calculated by splitting the base UV coordinates of the quad into smaller pieces in a similar fashion to the creation of the inner vertices. Each quad is triangulated into two triangles, with vertex indices stored in an index array. Using Godot's SurfaceTool, the mesh is assembled by specifying the vertices, indices, normals and UV coordinates. It then efficiently compiles these attributes into a final mesh, which is then added to the planet node for rendering in the scene. Through these steps, a high-quality spherical mesh that forms the basic structure of the planet is achieved. This mesh will later support additional details such as height variation and quadtree-based Level of Detail adjustments.

To enable efficient rendering and smooth detail scaling as the player approaches or

moves away from planetary surfaces, a quadtree-based approach is used. In this system, each quad on the surface of the planet is treated as a node in a quadtree structure, allowing for dynamic Level of Detail (LOD) adjustments as necessary. Each quad in the mesh can subdivide into four smaller quads. This process is regulated by threshold distances, which determine at which point a quad should split or combine based on the proximity of the player. Additionally, it is dependent on the `max_split_depth`, which limits how many times a quad can split, controlling the maximum detail level for performance optimization. To decide whether a quad should split, the distance from the spaceship to the quad's center is first calculated. If this distance falls below a specified threshold, the quad will split if it has not reached the maximum depth. When a quad is ready to split, four smaller quads are created, each of them representing a quarter of the area of the original quad, effectively doubling the resolution. The most important part is the transfer of data from the original quad to the smaller ones, as some parameters have to be adjusted. To ensure smooth rendering across all LOD levels, each new quad created during a split requires updated coordinates for positioning within the larger texture and mesh, a new resolution, which doubles the original resolution for increased detail and adjusted UV values for each sub-quad to maintain texture alignment. When the player moves away and the distance exceeds the threshold, each child quad checks whether it can be merged. If all of the child quads of a parent quad want to merge, they will remove themselves from the world and the parent quad is recreated. This reduces the level of detail in that area and also frees up memory and resources. This approach dynamically adjusts the resolution based on the distance of the player from the quad, enabling detailed surfaces up close while reducing unnecessary detail for distant areas. This quadtree setup provides a smooth gameplay experience with scalable detail levels while keeping requirements low.

One significant challenge in the procedural generation of planetary meshes using a quadtree structure is the appearance of visible gaps between quads of different LOD levels as can be seen in figure 4.2. These gaps arise, because bigger quads sample only half as many vertices as its child quads. This causes misalignments along the edges, resulting in visual seams between the higher and lower-resolution quads. This issue is especially noticeable at lower LODs, where the additional vertices of the higher LOD quads can deviate significantly from the linear surface defined by the lower LOD quad, resulting in noticeable height differences.

To address the issue of visible seams on the surface of the planet, a solution was implemented in which the edges of each quad are extended downward toward the center of the planet. These extended quads are then textured in alignment with the main planet surface, making the visual seams effectively invisible after some adjustments. To ensure sufficient coverage, these quads must extend at least a minimum distance toward the center. Otherwise, the seam-filling effect would be incomplete. Each of these quads is a 1-by-resolution quad, with its corner vertices constructed by taking two corners of the original mesh quad and multiplying these by a factor of 0.95, leading to a quad parallel to the center. Similar to the main planet mesh, these quads are subdivided into smaller quads to allow for accurate texturing, and additionally allowing for continuity with the main surface of the quad. The normals for these quads need to be updated in order to

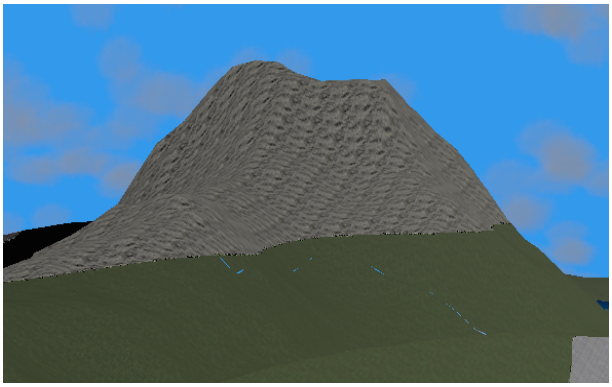

Figure 4.2.: Visible Gaps which are caused by the quadtree structure

ensure proper lighting and shading. Each normal vector is computed as the normalized vector from the center of the planet to each vertex. While the solution of extending quads downward reduces visible gaps, there are still potential issues with texture mapping. Since the normals of the extended quads point directly toward the center of the planet, UV mapping may become stretched or compressed along certain edges, creating visual artifacts. To address this, the U-coordinate in the UV mapping is normalized by a factor of $\frac{1.0}{200.0}$, which visually eliminates this distortion. The final improvement is achieved by the switch from a 32-bit to a 64-bit float engine, which completely removes any remaining UV distortions caused by rounding errors at high levels of detail. With these optimizations, the mesh now appears visually seamless, with no gaps or distortions in either texture or structure.

4.4. Height Generation

Currently, the planet looks rather boring with a simple round surface, that is able to increase and decrease resolution of parts of the mesh but with a uniform height. In this section, the main goal is to describe the height generation processes, which have been used for the two planets created in this project. Each of the planets has its own algorithm, in which a height value is generated for each vertex using its position. The final solution should be able to generate height values that allow the planet to have interesting terrain features and natural looking topography.

4.4.1. Approach 1: Fractal Brownian Motion using Perlin Noise

The first approach to terrain height generation uses Fractal Brownian Motion (FBM) with Perlin Noise to simulate realistic and varied terrain features. This technique is based on the tutorial of SimonDev on procedural planet generation [Sim21]. FBM combines multiple layers of Perlin Noise to produce natural-looking, fractal landscapes. For each vertex in the mesh, a height value is computed using FBM. This value, ranging

from 0 to 1, is scaled by a `base_height` factor and added to the initial height of the vertex, which allows for values between `base_height` and twice that height. Additionally, an ocean level can be set to create a minimum height. Any height below this level is adjusted upward to create a consistent ocean surface, which is then colored with a blue ocean texture. The FBM function generates multi-octave noise by combining Perlin noise values across several octaves. The main parameters of this algorithm, `persistence`, `lacunarity`, `exponentiation`, `start_octave` and `number_of_octaves`, control the frequency, amplitude, and overall shape of the terrain and allow for fine-tuned adjustments to its appearance. These parameters have the following purpose:

- **Persistence**: Controls how much each octave contributes to the height, gradually decreasing or increasing amplitude over octaves.
- **Lacunarity**: Controls how the frequency decreases or increases over octaves, allowing for more or less rugged terrain.
- **Exponentiation**: Used at the end of the algorithm, allowing for more or less smooth height levels. This gives options to create more cliff-like features or more gentle slopes, depending on the value.
- **start_octave**: Determines with which octave the FBM calculation starts from, allowing for more control over the smoothness.
- **number_of_octaves**: Determines how many octaves should be used.

The implementation of the FBM function with potential parameters is shown in listing 4.1.

```
var persistence = 0.5
var lacunarity = 1.5
var exponentiation = 1.7
var start_octave = 0
var number_of_octaves = 5
var noise_map_center = Vec3(0, 0, 0)
var size_of_noise_map = 2000
var noise_function = ...

func compute_fractional_brownian_motion(position_normalized):
    var amplitude = 1.0
    var frequency = 1.0
    amplitude *= pow(persistence, start_octave)
    frequency *= pow(lacunarity, start_octave)

    var normalization = 0
    var total = 0
    var position_in_noisemap = (position_normalized * noise_map_center)
                               * size_of_noise_map
    for o in range(number_of_octaves):
        var noiseValue = noise_function.get_noise_3dv(
   position_in_noisemap
```

```
                        * frequency) * 0.5 + 0.5
        total += noiseValue * amplitude
        normalization += amplitude
        amplitude *= persistence
        frequency *= lacunarity
    total /= normalization
    return pow(total, exponentiation)
```

Listing 4.1: Fractional Brownian Motion Algorithm

The final step of this algorithm is to check if any of the vertices are below a certain height, the ocean height. If that is the case, they are adjusted to that level, which creates a flat ocean surface. With this approach, the terrain generation algorithm is highly customizable, allowing for many different parameters. By gradually adjusting them and examining at the results, good settings can be found which generally produce interesting and varied terrain with many features across different regions of the planet.

4.4.2. Approach 2: Minecraft-Inspired Algorithm

The second height generation algorithm takes heavy inspiration from the terrain generation algorithm of "Minecraft" [Min24], offering more parameters and finer control over terrain features, which leads to easier creation of biomes. This approach uses 4 different parameters per vertex, namely `Continentalness`, `Erosion`, `Peaks & Valleys` and `Temperature` with the following ideas:

- **Continentalness**: Defines the base elevation across different landmasses.
- **Erosion**: Determines how sharply terrain features erode, adding variation.
- **Peaks & Valleys**: Adds high and low points to simulate mountains and valleys.
- **Temperature**: Gives vertices a certain temperature to allow for biome differences based on that.

Each of these values is generated using a Perlin Noise generator, which supports many different noise fractal types and parameter configurations to use, leading to a highly customizable feature. With these parameters set, this generator uses a 3D-position and returns a normalized Perlin Noise-generated value back. While this approach works well for creating varied terrain, it does not offer enough flexibility to produce terrain that smoothly transitions between gentle slopes and steep cliffs at specific elevations. To address this, a spline-based modification has been added to the algorithm. This allows for more control over terrain features by modifying the noise values according to spline points, enabling smoother or steeper terrain at different height levels. Using spline points, each noise generator can now be customized to adjust terrain shape and slope based on elevation. For instance, with carefully chosen spline points, terrain near coastlines can appear smoother, while higher elevations can feature steeper gradients, enhancing the visual realism. Listing 4.2 shows how spline points can be defined for a "continentalness"-noise generator.

```
continentalness.add_spline_point(0.0, 0.0)
continentalness.add_spline_point(0.4, 0.3)
continentalness.add_spline_point(0.6, 0.4)
continentalness.add_spline_point(0.7, 0.6)
continentalness.add_spline_point(1.0, 1.0)
```

Listing 4.2: Spline Point Example

In this configuration, the spline points guide the terrain generation as follows: underwater terrain (values between 0.0 and 0.4) is kept steep, while beach-level terrain (0.4 to 0.6) is smoothed out to create gentler slopes. Beyond this, the height increases more sharply again at mid-to-high elevations (0.6 to 0.7), with the highest peaks (0.7 to 1.0) remaining steep. The interpolation between these points allows the noise function to create a natural transition across different terrains, while enabling targeted adjustments in smoothness and gradient at each height level. This approach improves the realism and variety of the terrain by allowing for both smooth and steep features at different points on the planet's surface. The height of each vertex is determined by combining noise values from `continentalness`, `erosion`, and `peaks & valleys`. These values influence the position of the vertex, which is adjusted and scaled using the `max_height_difference` parameter to define the overall elevation. The formula to calculate the height of a vertex is given by equation 4.1.

$$v = v + ((c + p) \cdot (1 - e)) \cdot m \cdot v_{\text{norm}} \tag{4.1}$$

- v: Value of the vertex
- c: Continentalness value from noise
- p: Peaks & Valleys value from noise
- e: Erosion value from noise
- m: Difference between maximum and minimum height of planet
- v_{norm}: Normalized value of the vertex

To account for the ocean, the algorithm also checks, if the height is below ocean level and adjusts it if that is the case. Similar to the other algorithm, this leads to a nice ocean around the planet.

In this model, the temperature values used for later steps are influenced by two factors. First, the `temperature_random`, calculated using Perlin Noise, provides a base variability. Second, `temperature_sun_dependent` uses the angle of each vertex relative to the equator, simulating the natural gradient where regions near the equator are warmer and polar regions are colder. This latitude-based temperature factor, which works in a similar fashion on earth, is given by equation 4.2.

$$T_{\text{sun}} = 1 - |v_{\text{norm}} \cdot n_{\text{eq}}| \tag{4.2}$$

- T_{sun}: Sun-dependent temperature factor
- v_{norm}: Normalized vertex position
- n_{eq}: Normal vector of the equatorial plane

This approach gives each vertex a realistic temperature based on both a random value and its latitude, enabling further biome customization. The `Continentalness`, `Erosion`, and `Peaks & Valleys` values are not only used for the terrain height but combined with the temperature also help in the decision on which terrain should be generated. To implement this, the noise values are passed to the shader via the custom parameter of the SurfaceTool, enabling per-vertex data transfer (Listing 4.3).

```
var surface_tool = SurfaceTool.new()
surface_tool.begin(Mesh.PRIMITIVE_TRIANGLES)
# Setting format of the custom data to send (RGBA floats in this case)
surface_tool.set_custom_format(0, SurfaceTool.CUSTOM_RGBA_FLOAT)

for i in range(vertices.size()):
    surface_tool.set_normal(normals[i])
    surface_tool.set_uv(uvs[i])
    surface_tool.set_custom(0, Color(continentalnessArray[i].x,
    erosionArray[i].y, pvArray[i].z, temperatureArray[i].w))
    surface_tool.add_vertex(vertices[i])
```

Listing 4.3: Surface tool usage

With this framework, the developer can easily select a biome based on the values retrieved from `Continentalness`, `Erosion`, `Peaks & Valleys`, and `Temperature` to model the surface of the planet (Listing 4.4).

```
if (world_height > ocean_level) {
    if (continentalness < 0.37) {
        if (temperature < 0.2) {
            //cold beach
        } else if (temperature < 0.35) {
            // normal beach
        } else {
            // desert
        }
        ...
```

Listing 4.4: Biome Shader

This approach provides a robust, multi-layered height generation model, enabling precise terrain control through noise parameters and spline adjustments. The use of custom RGBA values allows for an easy addition of more information into the shader and thus easy calculation of biome types partially independent of the height of the planet.

4.5. Enhancing Surface Realism

Enhancing the surface realism of the planet is essential for creating an immersive experience. This section discusses the techniques used to improve visual authenticity, focusing on

using colors versus textures, and using custom shaders for water and lava surfaces. To maintain visual continuity across different Levels of Detail while not having to render textures when they are too distant, the project employs a strategic combination of colors and texture usage that dynamically adapts based on the player's distance from the terrain. At high altitudes, the terrain is rendered using simplified colors to represent distant landscapes. As the player approaches the surface, detailed textures are used, enhancing realism and immersion. To ensure smooth transitions between colors and textures, the base color was selected to closely match the albedo of the textures used at close range. This minimizes color shifts and maintains visual consistency when switching between plain-colored and textured surfaces. Each texture includes three primary components: `Albedo`, `Roughness`, and a `Normal Map`.

- **Albedo**: Provides the base color of the terrain.
- **Roughness**: Adds surface detail by controlling reflectivity and making surfaces appear more or less glossy.
- **Normal Map**: Simulates small-scale surface irregularities, enhancing textures like rocky ground or sand without adding complexity to the mesh geometry.

Together, these textures create a realistic terrain appearance that smoothly shifts between color and detail as the player moves from or to the planet

4.5.1. Biome Assignment for Simple Planet

In the Fractal Brownian Motion-based terrain, the biome is determined solely by the height of each vertex. The ocean biome is applied to the lowest terrain levels, representing deep water regions. Above the ocean level, the beach biome appears, giving a transitional shoreline effect. Lowland areas are covered by the grass biome, simulating grassland biomes. The forest biome is assigned to mid-elevation regions, providing a lush, forested look. At high elevations, the mountain biome appears, simulating rocky mountain terrain. Finally, the snow biome covers the highest peaks, representing snowy regions above the snow line. This height-dependent biome assignment is easy to implement and still results in a visually appealing planet.

4.5.2. Biome Assignment for Minecraft Planet

In the Minecraft-inspired method, biome assignment is calculated using four noise values: `Continentalness`, `Erosion`, `Peaks & Valleys`, and `Temperature`. As discussed previously, temperature is higher near the equator and lower toward the poles, creating climate zones. The resulting biomes vary based on combinations of elevation, erosion, and temperature, leading to more complex terrain types. Beach areas are found in regions where the continentalness is low, characterized by flat, sandy terrain. These include cold beaches, located in cooler climates featuring snowy sand; beaches in moderate climates with darker sand typical of temperate beach environments; and warm beaches, found in

hot climates and characterized by very fine, golden sand. Grass covers low to midland regions with moderate temperatures. Mountainous areas are applied to high-altitude regions, where the temperature and erosion influence the biome type. These include snow at high-elevation areas with low temperatures, simulating icy peaks; basalt in high-altitude warm regions, giving volcanic mountains a unique look; and lava at the peaks of very high mountains in warm climates, simulating active volcanoes. While this is just a simple version of biome selection, the framework allows for very detailed and intricate worlds on levels similar to Minecraft if needed. Furthermore, the approach is easy to use for designers and can be calculated quickly by the computer.

4.5.3. Water & Lava Shader

The lowest points in altitude of the planet is for the ocean biome, which is defined by the ocean level parameter defined earlier. Similar to other surfaces, the water also has a predefined color at a lower Level of Details, but instead of a texture, the higher LOD levels use a water shader. This shader works by combining several parameters to create a highly adjustable and dynamic wave texture, giving the surface of the water a sense of depth and flow. The properties defined for the shader are as follows: Colors (`water_albedo` and `water_albedo2`), which represent different shades of water, with the first being the primary color when looking straight down at the water surface and the second color being visible at shallower viewing angles; Metallic and Roughness (`water_metallic` and `water_roughness`), which control how reflective and smooth the water surface appears, with higher metallic values making the water more reflective and higher roughness making reflections appear sharper; Normal Maps (`water_texture_normal` and `water_texture_normal2`), where two normal maps create wave patterns, each associated with a specific wave direction (`water_wave_direction` and `water_wave_direction2`), adding complexity and realism to the wave motion; and Wave Speed (`water_time_scale`), which controls the animation speed of the waves by scaling the time value applied to the wave directions. To create more interesting wave patterns, the direction values should not have the same direction or same magnitude and should be reasonably large. The calculation of the color to use depends on the viewing angle to the specific location on the surface, factoring in the normal of the fragment, the view vector of the player, and a strength value "amount" (Listing 4.5). With these values, the water color at a certain fragment can be calculated as can be seen in Listing 4.6.

```
1 float fresnel(float amount, vec3 normal, vec3 view) {
2     return pow(
3         (1.0 - clamp(dot(normalize(normal), normalize(view)), 0.0, 1.0)),
      amount);
4 }
```

Listing 4.5: Fresnel

```
1 vec2 time = (TIME * water_wave_direction) * water_time_scale;
2 vec2 time2 = (TIME * water_wave_direction2) * water_time_scale;
3
4 vec3 normal_blend = mix(texture(water_texture_normal, UV + time).rgb,
```

```
                        texture(water_texture_normal2, UV + time2).rgb,
    0.5);

float fresnel = fresnel(5.0, NORMAL, VIEW);

waterColor = mix(water_albedo, water_albedo2, fresnel);
METALLIC = water_metallic;
ROUGHNESS = water_roughness;
NORMAL_MAP = normal_blend;
```

Listing 4.6: Water Shader

Using this setup, the water appears realistic, with a well-defined sun reflection when viewed from an optimal angle. An additional feature is the blend between water and terrain texture at the shoreline. This transition is achieved by interpolating between the water and terrain albedo textures based on height. With the code in listing 4.7, this effect is only affecting the area slightly above ocean level, shown in figure 4.3. With these values, the water color at a certain fragment can be calculated as can be seen in Listing 4.6.

```
float blendFactor = smoothstep(ocean_level, ocean_level +
    ocean_blend_height_distance, world_height);
ALBEDO = mix(waterColor, surfaceColor, blendFactor);
```

Listing 4.7: Ocean Water Blend

Figure 4.3.: The water shader in effect

With all of these features added, the water looks realistic now, moving in an interesting pattern with convincing visuals.

The lava shader is used exclusively for high volcanic biomes within the “Minecraft”-style planet. Functionally, it shares a lot of code with the water shader but with a red-ish color for the albedos and higher wave speeds to create the impression of a more dynamic and volatile liquid surface.

4.6. Multithreading

To optimize mesh generation and improve performance, the project uses multithreading. Each of the six sides of the planet is assigned to a separate thread, which allows concurrent updates and minimizes the time needed to generate or refresh the terrain. Threads are utilized to both generate and update each side, significantly reducing the load on the main thread. Additionally, this makes sure that the main thread is not blocked and that there are no frame drops due to creation of new terrain. The threading process involves several steps. First, each side of the mesh is assigned to an individual thread, which performs all necessary calculations and updates for that side independently. Every frame, the system checks whether the thread for a given side has completed its update. If the thread has finished, it waits briefly to ensure all updates are complete before initiating the next update cycle for that side. If a thread for a side is still active, the system skips that side for the current frame, allowing it to complete before restarting. The code structure ensures each side operates in parallel, using available CPU cores for faster mesh generation. This process is managed by checking the status of each thread (Listing 4.8).

```
var i = 0
for side_id in quad_tree_nodes.keys():
    if threads[i] == null or !threads[i].is_alive():
        if threads[i] != null:
            # Ensure the previous thread is completely finished
            threads[i].wait_to_finish()
        threads[i] = Thread.new()
        var side_nodes = quad_tree_nodes[side_id]
        threads[i].start(update_side.bind(side_nodes, camera_pos))
    i += 1
```

Listing 4.8: Threading Example

While this multithreading approach offers consistently high frame rates, it introduces a potential issue: the player can continue moving while terrain construction is in progress, which may cause them to occasionally "glitch" or clip into the partially updated mesh, especially when moving quickly. However, the benefits in terms of smooth and uninterrupted performance make this a preferable solution for enabling an overall responsive and visually consistent experience.

4.7. Collision detection

Collision detection is a critical component in ensuring proper interaction between the player model and the terrain. To achieve this, each terrain mesh quad is assigned a `ConcavePolygonShape3D`, which is then linked to a `StaticBody3D` for collision handling. Since the collision mesh is generated per quad, it closely matches the terrain, ensuring precise collision detection while keeping computational overhead low. As previously mentioned, the player model has its own collision shape, preventing unintended movement through the terrain with minimal performance impact.

4.8. Spawning Trees

Spawning and rendering all possible trees on the planet simultaneously is not a feasible solution because of performance reasons, especially when keeping in mind the size of a planet and the usual tree density. Instead, the current Level of Detail of a certain quad decides whether trees for that area should be spawned or not. If a player has a low enough distance to a part of the terrain and therefore the LOD is high enough, it will allow trees to be spawned, while areas that are further away from the player, resulting in a lower Level of Detail for that area, will not have any trees. These trees are added as children of the terrain mesh, allowing for easy removal when the LOD changes, because they can simply be deleted when the quad is deleted. To further increase immersion, the project uses two types of trees. Beach biomes spawn palm trees, while every other biome will spawn the normal tree. These trees are simple 3D meshes, making them easy to modify or replace them with different assets as needed. Additionally, the realism of the environment is increased by adjusting the density and placement based on height. On the lower altitudes like the beach and grass biome, the tree spawn rate is lower than at higher altitudes in the forest biome, which has a three times higher tree density. At a certain point in elevation, there are no trees spawned anymore, simulating mountainous and snowy mountain biomes. To add variation and avoid repetitive patterns, the size and rotation of each of the trees is randomized, leading to a more natural and unique appearance across the landscape. While the quadtree-based level-of-detail system brings many advantages, it also poses a challenge for spawning the trees in the correct position. The spawn position of the trees is calculated by using the edges of the quad it needs to be spawned on and then choosing a random spot on this plane. The problem arises, when the Level of Detail at this position is increased, as the vertices will move and the trees will possibly not be exactly on the quad anymore. While moving the tree to its new location could be a feasible solution, this would cause a lot of visible movement on the planet and therefore this project opted for a different and simpler solution. As the difference between the height at the later LODs is low, the trees are not moved at all after they have been spawned but instead already have their trunks in the model itself extended. This allows the trees to always touch the floor, and therefore does not need recalculation or respawning and additionally helps with bringing in variety, as the tree height is now also randomized.

4.9. Realistic Objects in the Sky

Creating a realistic sky is crucial for immersion in a planetary simulation. Several ideas have been included into the project, some based on more physically accurate approaches, some based on simpler ideas. This section focuses on the objects, that can be seen in the sky.

The sun is represented as a distant yellow sphere that orbits around the planet, accompanied by a directional light that is updated every frame to match the direction of the sun to the earth. While a more realistic approach could involve a spotlight - which

emits light rays from a specific point in the scene - this approach is generally too costly, particularly for the shadow calculation, resulting in jittery shadows that decrease the visual quality. Because of that and the fact that a directional light visually looks similar to a spotlight at a large distance, the directional light has been chosen to represent the illumination created by the sun. With this approach, the directional light would project the shadow of the sun mesh onto the earth, therefore the last step is to disable the shadow creation of this mesh.

The moon also orbits around the planet, much like the sun, but is smaller, rotates faster, and is positioned closer to the planet. Represented as a white sphere, the moon utilizes the directional light of the sun to display a realistic depiction of lunar phases. Additionally, it is possible that the moon moves between sun and earth and creates a solar eclipse.

The cloud generation process is relatively straightforward. Clouds are rendered as billboards with textures applied to them. During the creation of the planet, a designated number of cloud spawner objects is added to the scene. For each cloud spawner, a random position on a unit sphere is calculated and adjusted based on a predefined distance from the center of the planet. The orientation of each spawner is aligned to ensure that the clouds remain parallel to the ground beneath them. The spawners are 3D nodes containing a GPUParticles3D child, which is responsible for generating the cloud particles in the scene. Random constraints are established for the number of clouds to spawn, and additionally the X, Y, and Z limits that define the bounding box for all clouds, as well as the size parameters for each cloud, ensuring a diverse and varied cloudscape.

4.10. Atmosphere and Halo

To enhance the appearance of the planet from different perspectives, the system implements three key visual effects: a halo around the planet when viewed from space, a blue atmospheric tint that obscures the stars on the sunny side while highlighting the sun, and a color transition at the day-night boundary that shifts from blue to a reddish hue before revealing the starry night.

When the player is exploring the universe from outside the planet, a halo effect appears around its edge. This phenomenon, similar to the faint blue halo seen around Earth from space due to Rayleigh scattering, is created using the shader in Listing 4.9. In this shader, the `diffuse` variable corresponds to the color of the halo, which is dependent on the dot product between the normal of the current fragment of the sphere and the light direction of the sun. The `ALPHA` variable defines the maximum transparency of the halo. This leads to an effect where looking at the planet from the position of the sun leads to no visible halo, while on the other side, the halo is very prominent as seen in Figure 4.4.

```
1 shader_type spatial;
2
3 uniform vec3 color_day : source_color = vec3(0.2, 0.6, 0.922);
4
5 void fragment() {
6     ALPHA = 0.5;
```

```
}

void light() {
    vec3 light_direction = normalize(-LIGHT);
    float NdotL = max(dot(NORMAL, light_direction), 0.0);
    vec3 diffuse = color_day * NdotL;
    DIFFUSE_LIGHT = diffuse;
}
```

Listing 4.9: Halo Effect

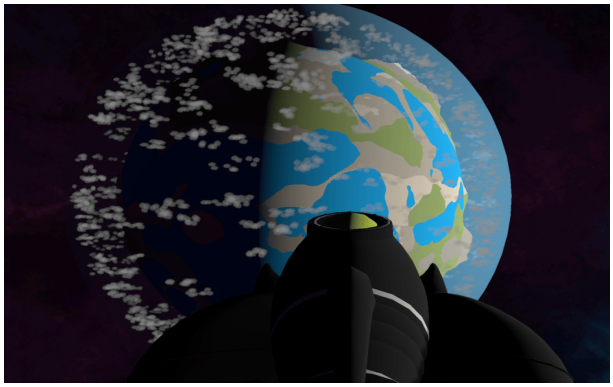

Figure 4.4.: Halo of the planet

When viewed from the sunny side of the planet, the atmosphere obscures the universe and stars, creating a vibrant blue color that highlights the sun. Conversely, when looking to the sky at the dark side of the planet, the moon and other celestial bodies are visible. The transition from the sunny side to the dark side involves a shift in atmospheric color from a bright blue to a warm reddish hue, resembling the evening and morning skies, eventually fading to invisibility. This effect is achieved with the shader code in listing 4.10. This shader colors a sphere with a radius greater than the distances to the sun and moon, allowing both celestial bodies to remain visible within the atmosphere. Since the star background is essentially at an infinite distance, it becomes visible only when the alpha value of the atmosphere is sufficiently low. In this shader, the atmosphere color and the alpha value is calculated depending on the directions from the center of the planet to the sun and the center of the planet to the player, which leads to the effect described earlier as Figure 4.5 shows.

```
void fragment() {
    vec3 to_sun = normalize(sun_center);
    vec3 to_player = normalize(playerPosition);
    float angle_cos = dot(to_sun, to_player);

    vec3 atmosphere_color = color_day;

    float color_transition_start = 0.2f;
    float color_transition_end = -0.4f;
```

```
    if (angle_cos > color_transition_start) {
        atmosphere_color = color_day;
    } else if (angle_cos < color_transition_end) {
        atmosphere_color = color_sunset;
    } else {
        float mix_strength = smoothstep(color_transition_start,
    color_transition_end, angle_cos);
        atmosphere_color = mix(color_day, color_sunset, mix_strength);
    }

    float alpha = 0.0f;
    float alpha_transition_start = 0.1f;
    float alpha_transition_end = -0.2f;
    if (angle_cos > alpha_transition_start) {
        alpha = 1.0f;
    } else if (angle_cos < alpha_transition_end) {
        alpha = 0.0f;
    } else {
        alpha = smoothstep(alpha_transition_end, alpha_transition_start,
    angle_cos);
    }

    ALBEDO = atmosphere_color;
    ALPHA = alpha;
}
```

Listing 4.10: Atmosphere Calculation

Figure 4.5.: Atmosphere on the planet

The halo effect is rendered using an external shader, while the atmospheric effects are managed within an internal shader. The system simply switches between these shaders based on if the player is inside the halo or outside, transitioning between the two main effects.

4.11. User Interface

The user interface serves both as a tool for real-time parameter adjustments and as a debugging aid, providing insights into performance metrics and in-game variables.

For this purpose, Godot offers a simple approach where a new scene is created with a user interface node as the main node. Canvas layers are used to manage rendering order. Layers with higher values appear on top, while lower values render behind, potentially causing partial or full occlusion. The main node of the scene is set to layer 0, while a UI canvas layer is added at layer 1 to ensure it renders in front of the game viewport. A VBoxContainer is then added as a child of the UI, stacking its elements vertically. Although UI elements could be more spread out for visual appeal, stacking them on the left side was sufficient for this project. Adding new UI components is as simple as attaching them to the VBoxContainer. While there are many different options for the user interface components, option buttons, a single normal button, and labels were sufficient for all purposes. Clicking an option button presents a list of selectable items, triggering an event upon selection. A regular button functions similarly but directly triggers an event when clicked. Labels display text on screen and can be updated each frame. Over the course of the project, multiple variables and buttons were added. One of the labels prints the frames per second. As the engine provides a built-in function for this purpose, a script updates this label each frame with this information. Furthermore, especially for debugging, multiple positions are also printed here, namely the position of the player, the position of the sun and the position of the planet. The distance to the center of the planet is also displayed to aid in debugging the Level of Detail system and quadtree adjustments. The button of the UI adds the functionality to create a new planet of the currently selected planet type. Pressing this button causes an event to be sent to the script of the universe, which then calls a function to replace the old planet with a new one. Lastly, there are four option buttons, which allow the user to change certain aspects of the game during play. One option lets the player switch from directional sunlight to ambient lighting, which uniformly illuminates the scene. Next, there is an option to enable and disable shadows. While this does not brighten the dark side of the planet, it disables shadows cast by trees and the rocket. The next option is for switching between the two types of planets included in this game, the “Simple Planet” and the “Minecraft Planet”. The final option allows the user to let the sun move around the planet or stop it from doing that. All these buttons function similarly, sending an event to the universe script on click, which then executes the corresponding code with the specified parameters.

4.12. Configuration files

As this project contains numerous adjustable variables spread across multiple files, it was necessary to make them editable in a single location. This way it is straightforward for anyone to modify any variable without having to know the whole code base. For simplicity, there are three different configuration files, one for global project settings and two others, each dedicated to a specific planet. The base configuration file contains

variables that specify the planet type, light mode, shadow settings, and whether the sun moves at game start. Additionally, it includes variables for sun and moon movement, rocket speed in both modes, and the cooldown for the 'Randomize Planet' button. The other two files contain all the important variables for each of the planets, which have been described in earlier sections. Using the configuration files is straightforward — load the file and retrieve variables using the built-in function.

4.13. Exporting the Project

Godot offers a simple way of creating an executable file of the game, which can then run on any device for the target platform. As this game is supposed to run on Windows, this section will specifically focus on building an executable for that operating system. Opening the project menu of Godot gives the user several options, including an export option. Clicking it opens a window which provides tools for the developer to streamline the exporting process. To generate an executable file of the game, a new export profile must be created to configure all necessary variables and data. In the case of this project, several things had to be changed. First, the executable flag must be enabled to ensure the game runs as a standalone program. Next, to have one singular file for the game, the pck file needs to be embedded, which must also be enabled. Additionally, the icon, name, and author information of the game can be customized here. Additionally, all required resources must be added in the Resources tab. As the .ini files are not included by default, they have to be explicitly whitelisted here. The final step before exporting is setting up and configuring the export templates. There are two options for this. The first option is to use the built-in manager of Godot, which automatically downloads and installs these files for the version of the game engine. However, this method does not work with modified engine versions, such as those with the `precision=double` flag enabled. In such cases, the engine must generate custom export templates instead, as described in the documentation. The paths to these can then be selected in the export window. Once everything is set up, the executable can be generated with just a few clicks [God24a].

5. Evaluation

This chapter outlines the setup of the experiment, aimed at evaluating how different terrain generation approaches affect immersion and other factors. Additionally, this chapter presents the results of the study, providing an analysis on how different planet generation approaches impacted the immersion and exploration experiences of the participants. Statistical tests are used to assess the differences between the four planets, focusing on the four question categories "Immersion and Presence," "Exploration and Fun," "Terrain Features and Realism" and "Movement and Controls." Following the presentation of the results, a discussion of the findings is provided. This discussion interprets the statistical outcomes and addresses the implications of the findings. Furthermore, the limitations of the study, including potential biases and factors that might have influenced the results, are considered.

5.1. Additional Planet Generators for Comparison

In order to have a good comparison, two more projects, that aim to generate procedurally generated planets, were chosen. Thus there are four planets, which are explored by the participants.

"Poor Man's Sky" (Figure 5.1) is a very simple game that generates an explorable procedural planet. It was created by two people and released on itch.io by the account jfc3 [jfc]. The title as well as the idea is a reference to the well known game "No Man's Sky" [No b], which brought procedural planet generation to the next level. "Poor Man's Sky" features a first person camera and a universe containing a planet and a star skybox. This project does not use any directional light, nor does it contain objects like trees or collision. The primary focus is solely on terrain generation itself. The source code for the project is unfortunately not published, therefore the inner workings of the terrain generation algorithm are unknown. The controls for this game are simple but different to the other games used in this experiment. Pressing the `W`-Button on the keyboard will orbit the player around the planet upwards and `S` is doing the opposite. `A` and `D` are used to orbit the player around the planet to the left or right. To ascend, the `Space` button is used, while the `Shift` button will cause the player to descend. It is also possible to generate a new planet using `R` or regenerate the current planet using `T`. With `G`, the user can also switch the planet type from an earth-like planet to one that resembles mars and back. To allow the participants of the survey to play the game, they simply need to open the website provided.

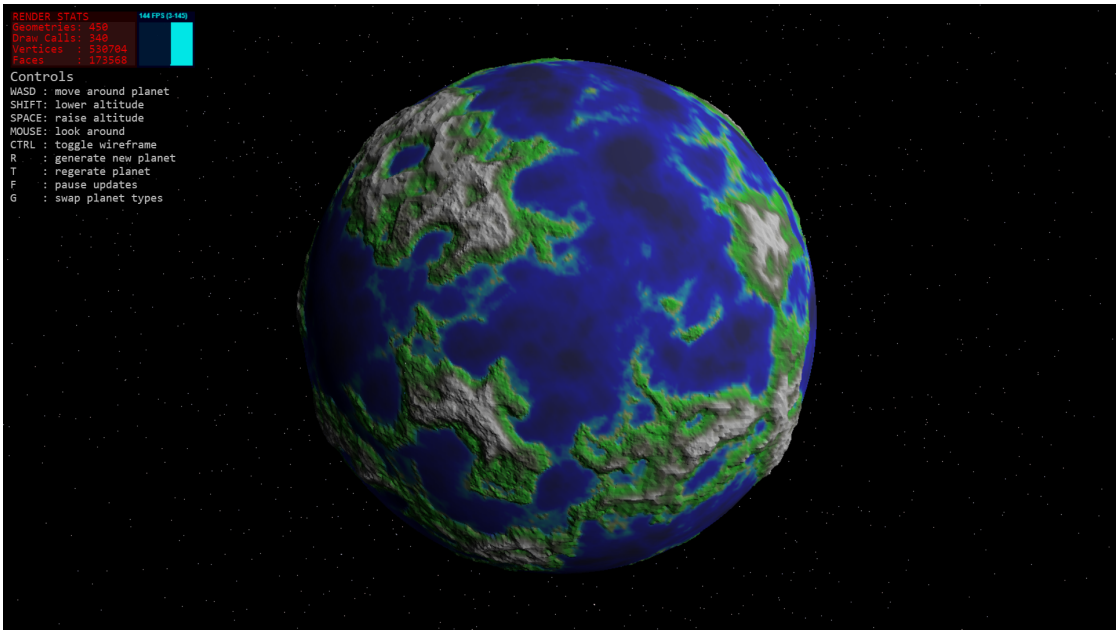

Figure 5.1.: Poor Man's Sky

The other project, "ProceduralPlanetGodot" (Figure 5.2), was released on GitHub [Ath24b] by Athillion. A video showing the project has also been published on YouTube, with the goal of showcasing an open source procedural planet generator in Godot. [Ath24a]. Similar to the other ones in this thesis, it draws heavy inspiration from the Procedural Planet Generation YouTube series of Sebastian Lague [Seb18]. "ProceduralPlanetGodot" features a simple universe, containing a planet, a sun as the directional light that is orbiting around the planet and a star skybox in the background. The player is in first person and can move freely in the world. As the player approaches the planet, windy sounds and a shaking camera are used to increase the feeling of entering an atmosphere. To increase the visual look, the project tries to incorporate a halo to both the sun and the planet. Furthermore, the planet has an ocean level, with a visually pleasing ocean shader texture and additional foam on the area between water and land. In contrast to the projects created for this thesis, the height of the terrain is not clamped to an ocean level but instead, the height is kept natural and a second sphere with the radius of the ocean height and a water texture is added. This makes it so the player is actually able to explore the terrain in the ocean as well. While the player can dive deep into the abyss, there is no collision for the planet mesh. As explained in his video, the planet uses the same approach as the planet generation series of Sebastian Lague, which is partially similar to what the first planet of this thesis uses. But due to the fact that there are so many possible parameters to adjust, the similarity between the terrain features and actual feel of this version and the "Simple Planet" is very low. The controls had to be

adjusted as the version on GitHub did not work properly. With these adaptions, the player can now move forward with `W`, backward with `S`, to the left with `A`, to the right with `D`, up with `Q` and down with `E`. To move the camera in the desired direction, the player must click onto the screen with the right mouse button, which causes the game to use a captured mouse mode, meaning that the mouse is invisible and at the center of the screen. Movements of the mouse in this mode are used as camera input. Exporting the project was simple as it was the same process as described in the export section of the technical implementation. The result is again an easily shareable executable file.

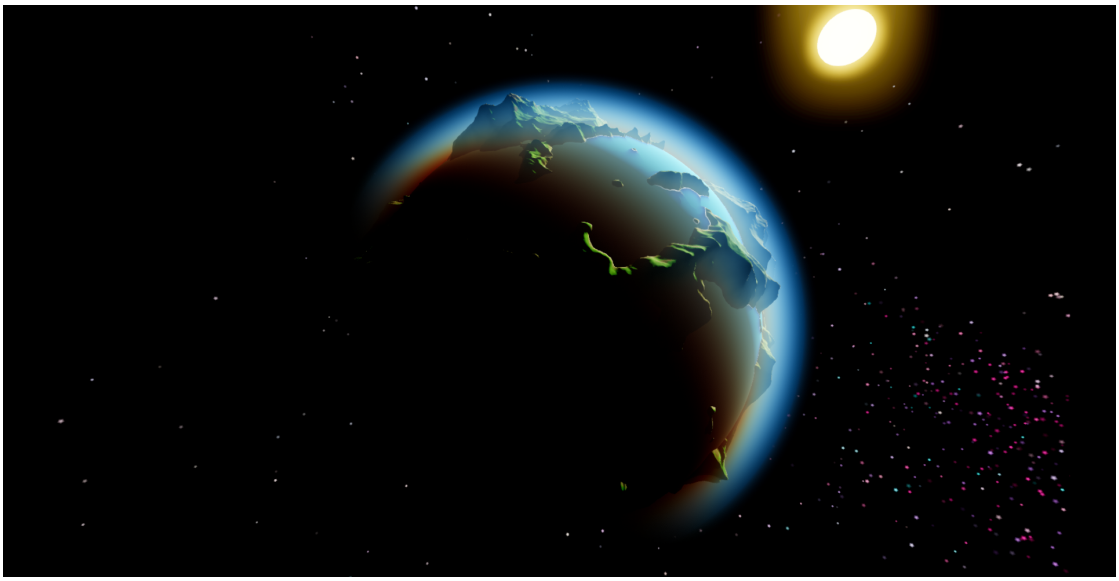

Figure 5.2.: ProceduralPlanetGodot

5.2. Experiment

To compare the four projects, a questionnaire was developed with a focus on measuring immersion and the experiences of participants during world exploration. The questions chosen are inspired by the "Igroup Presence Questionnaire" [igr] as well as the "Presence Questionnaire" by Witmer and Singer [UQO04]. They can be grouped into four categories: "Immersion and Presence", "Exploration and Fun," "Terrain Features and Realism," and "Movement and Controls." The questions are as follows:

- I felt that the virtual world surrounded me.
- I felt like I was just perceiving pictures.
- I had a sense of acting in the virtual space, rather than operating something from outside.

- I felt as if I were physically there in the virtual space.
- The planet was fun to explore.
- I was completely captivated by the virtual world.
- The terrain featured a diverse range of features and landscapes.
- The mechanism which controlled movement through the environment felt natural.
- I was able to examine objects closely.
- I was able to examine objects from multiple viewpoints.
- I felt like there was delay between my actions and the expected outcomes.
- The terrain looked realistic.

Participants rated each of these 12 questions for each of the four planets on a 5-point Likert scale, ranging from "Strongly Disagree" to "Strongly Agree." As it is impossible to grade a planet, when you do not know the other planets, the order of the projects shown to the participants of the study was randomly selected using a simple python script, which just prints the numbers 1 to 4 in a random order. These numbers are noted at the last page of the questionnaire in order to remember this for later use. The number 1 corresponds to the first approach created for this thesis, the "Simple Planet". Number 2 is the "Minecraft"-inspired approach, number 3 is "Poor Man's Sky" and finally number 4 corresponds to "ProceduralPlanetGodot". To gather a diverse dataset, 15 participants were selected, vastly ranging in age, gender and video game experience.

The following list should illustrate, how such an experiment might look like.

1. Explain the general process of the experiment to the participant.
2. Run the python script to get a random order of the numbers from 1-4.
3. Write these numbers down on the last page, so it is known to the researcher in which order the participant rated the planets.
4. Let the participant fill out the demographic questions on the first page.
5. If for example the order of the numbers was 4-2-3-1, then:
   a) Start with the 4th Planet, "ProceduralPlanetGodot". Explain how to move around in the world and let the player explore the planets. Make sure, that the player does not miss an important feature of the world.
   b) Answer the 12 Questions for Planet 1 on the questionnaire.
   c) Let the participant explore the 2nd Planet, the "Minecraft"-inspired Planet.
   d) Answer the 12 Questions for Planet 2 on the questionnaire.

e) Continue with the 3rd Planet, “Poor Man’s Sky,” and fill out the Questions for Planet 3 on the Questionnaire.

f) Do the same for Planet 1, the “Simple Planet,” and fill out the Questions for Planet 4 on the Questionnaire.

Once the participants have completed the exploration of all four planets and filled out the questionnaire, the collected data is analyzed to evaluate the impact of each terrain generation approach on immersion and other factors.

Data from participant questionnaires was systematically organized in an Excel sheet, with each row representing a participant. The demographic data, such as Respondent-ID, Name, Age, Gender and Experience Level, was recorded alongside responses to the 12 questions for each of the four procedural planet projects. To ensure consistency between all questions, the responses for the negatively framed questions were marked, so that they can later be inverted, as a score of 1 should be treated as 5 there. A custom Python script was then developed to process and analyze the data using the “pandas” and “matplotlib” libraries. This script automated the fetching of the data from the Excel sheet, the calculation of means, standard deviations, and visualization tasks, producing box plots, bar charts, and ANOVA statistical tests to assess significant differences between the planets.

The ages of the participants ranged from 13 to 58 years with a median age of 26. This broad age range allows for a variety of perspectives, people that grew up with technology and other people that did not. The graph in figure 5.3 shows the distribution of this.

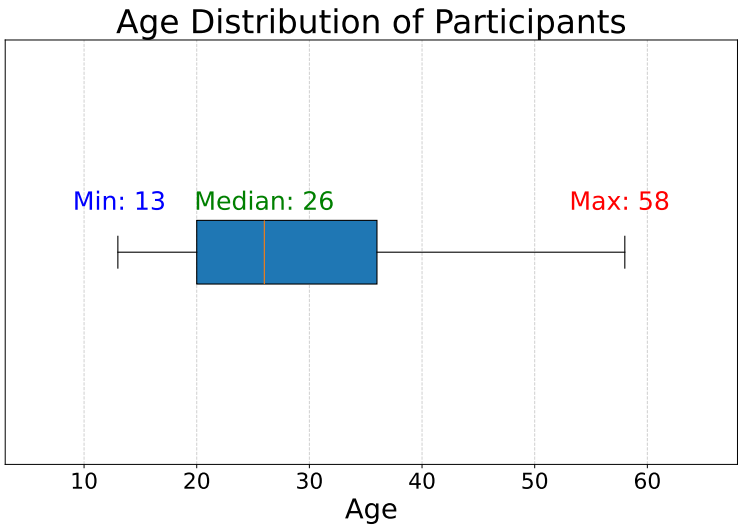

Figure 5.3.: Boxplot of the ages of the participants

Participants were also asked to rate their level of experience with video games, using a scale from 1 to 4 where a 1 means that the participant plays video games every day, 2 meaning the participant plays video games a few times a week, 3 should be used if the

participant plays video games only a few times a month and 4 if it is less than a few times a month. This scale was designed to capture the frequency of engagement of the participants with digital environments, a factor that might influence their interaction with procedurally generated terrains. The results of this section can be found in figure 5.4.

Figure 5.4.: Video game experience distribution of participants

The gender distribution of participants showed a slight male dominance. However, the sample still includes a balanced range of participants. The gender breakdown is displayed in figure 5.5.

With this understanding of the participant demographics, the following section will dive into the experimental results, examining how the participants responded to the questionnaires.

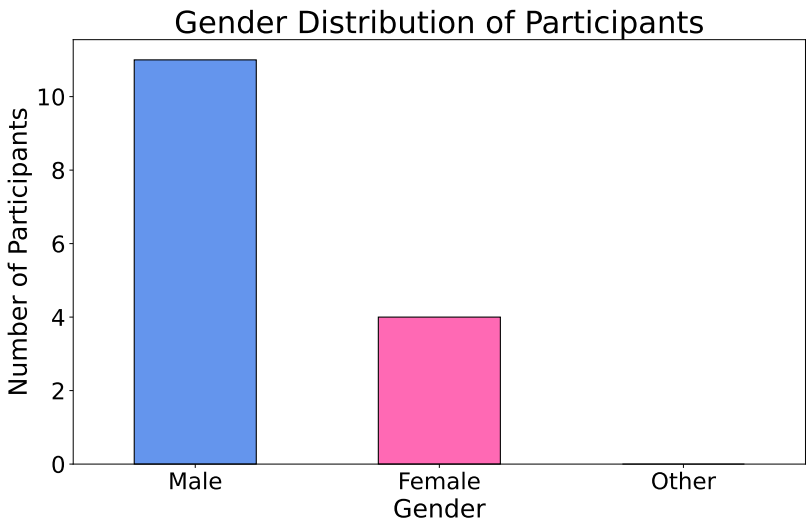

Figure 5.5.: Diagram of the gender distribution of the participants

5.3. Results

In this section, the results of the experiment are presented, including descriptive statistics in the form of means and standard derivations, a comparison of means for each planet based on question categories and the results of ANOVA tests performed to assess statistical significance between planet experiences.

Descriptive statistics in form of means and standard deviations for each of the 12 questions were calculated separately for each planet. These statistics provide an overview of how participants responded to each question and give an overall experience with each planet. To better illustrate these results, the figures 5.6, 5.7, 5.8 and 5.9 display these statistics for each question on each planet.

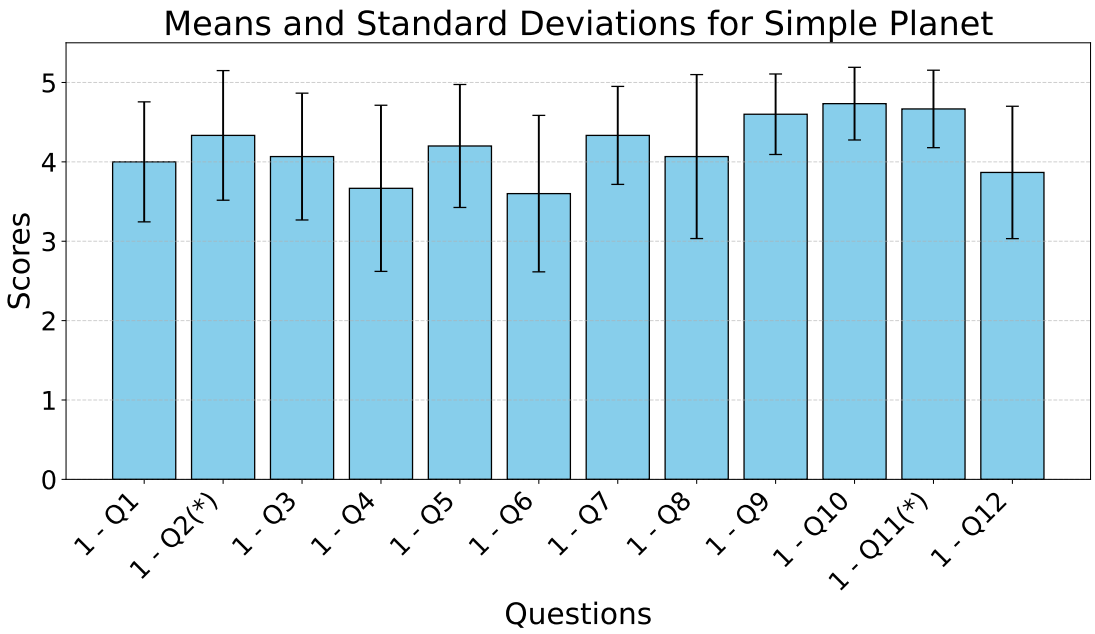

Figure 5.6.: Mean and standard derivation of the questions of "Simple Planet".

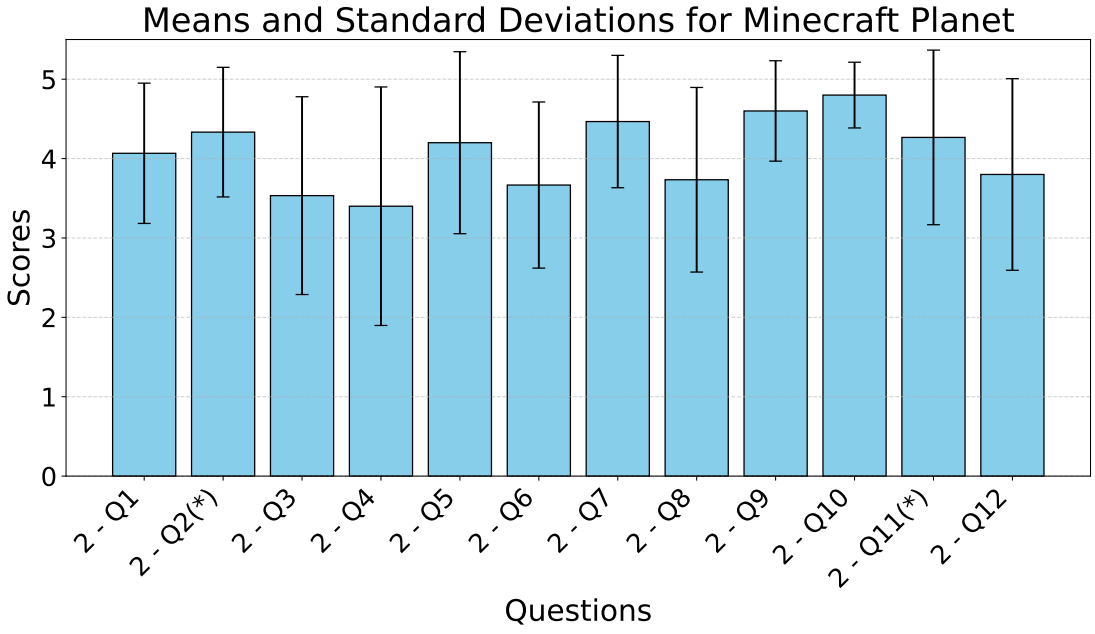

Figure 5.7.: Mean and standard derivation of the questions of "Minecraft Planet".

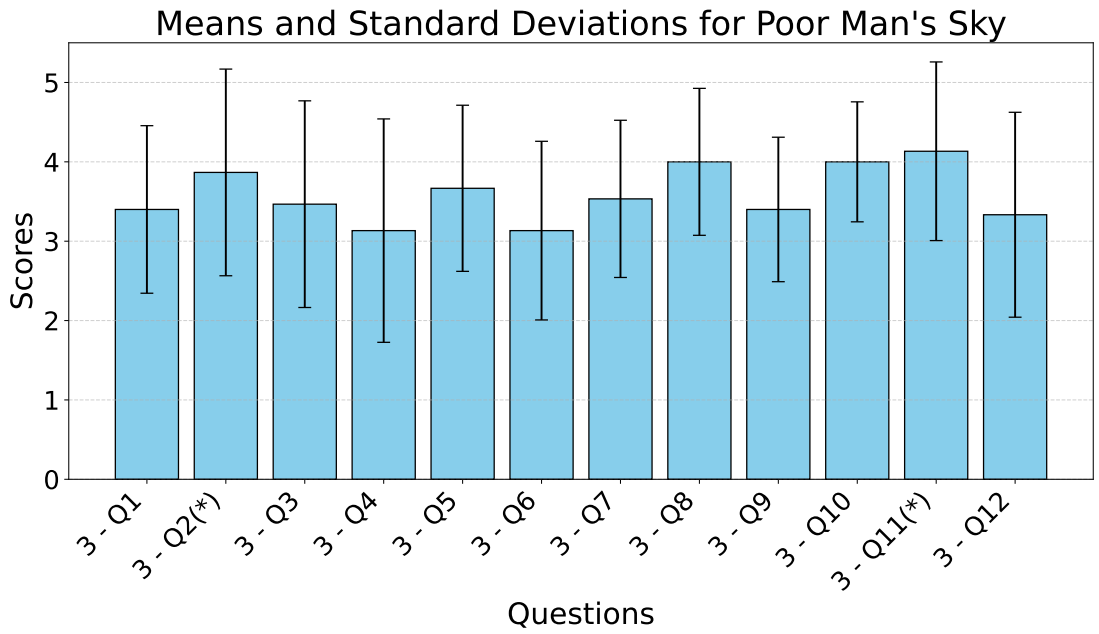

Figure 5.8.: Mean and standard derivation of the questions of "Poor Man's Sky".

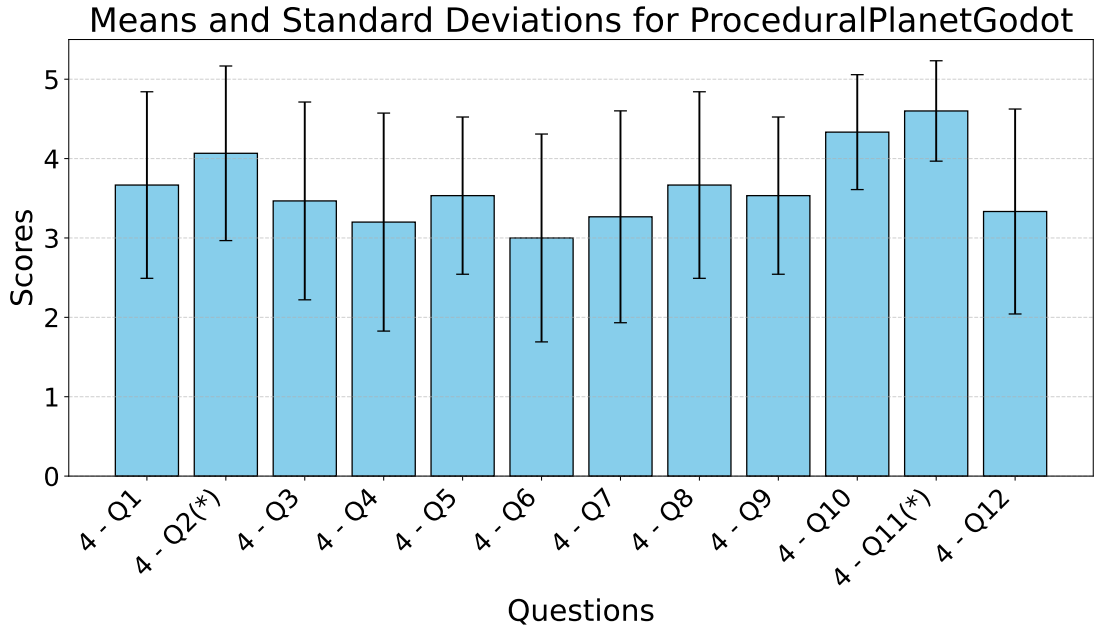

Figure 5.9.: Mean and standard derivation of the questions of "ProceduralPlanetGodot".

These graphs show both the consistency of participant ratings via standard deviations and the average response of each question for each planet via the means. Generally, high means and low standard deviations may indicate positive, consistently strong experiences, while high standard derivations may indicate mixed responses to particular questions.

In addition to the individual questions, the planets were grouped into the following categories: "Immersion and Presence," "Exploration and Fun," "Terrain Features and Realism" and "Movement and Controls". This grouping allows for a more focused comparison of experiences of participants across the different planets in these central aspects. The figure 5.10 presents the means and standard deviations for each category across all four planets, providing a better view of overall trends compared to the single-question results.

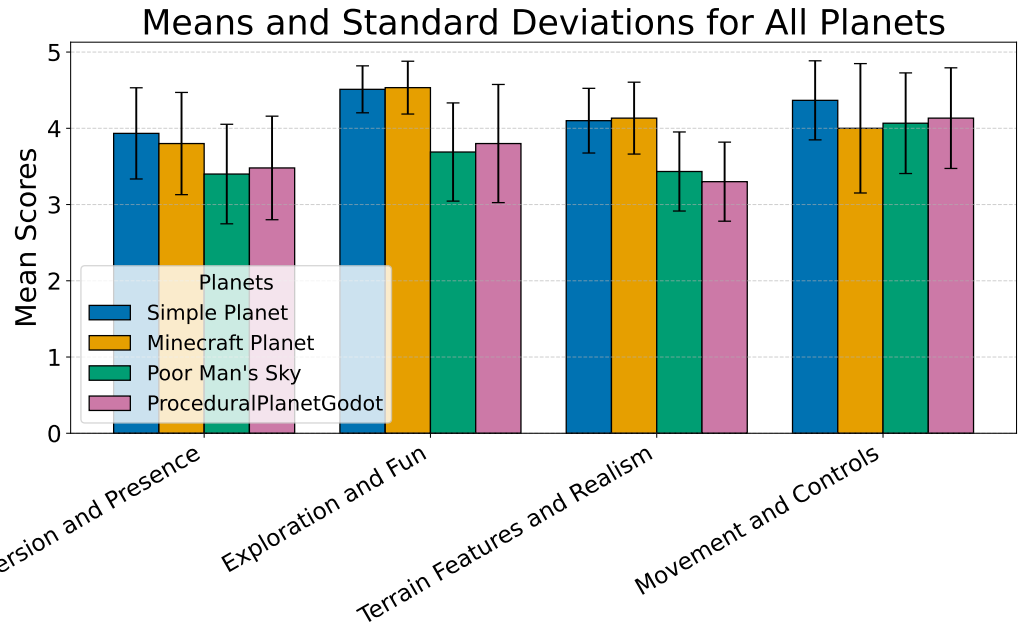

Figure 5.10.: Comparison of the Categories for the Planets

This graph illustrates differences in participant experiences across planets for each category. As an example, variations in "Immersion and Presence" may indicate how different terrains and environments contributed to the overall sense of being immersed. Similar to this, a shift in "Exploration and Fun" shows, how much fun the participants had while exploring the worlds. "Terrain Features and Realism" tells more about how the perception of realism of each planet was, while differences in "Movement and Controls" reflect how intuitive and free of lag the navigation systems worked.

To assess whether there were significant differences between the ratings of the four planets, ANOVA tests were conducted for each of the grouped categories. Table 5.1 shows the results of these tests.

Category	F-statistic	P-value
Immersion and Presence	5.4482	0.0013
Exploration and Fun	4.5460	0.0043
Terrain Features and Realism	2.8680	0.0380
Movement and Controls	3.3878	0.0194

Table 5.1.: ANOVA Results for Each Category

The p-values for all the categories were below the significance threshold of 0.05, indicating that there were statistically significant differences between the ratings of the planets in each of the categories. This suggests, that the terrain generation approaches had distinct effects on perceptions of participants in immersion, fun, realism, and movement.

5.4. Discussion of the Results

The "Immersion and Presence" category showed a significant difference between the planets. The "Simple Planet" (Planet 1) scored the highest, followed closely by the "Minecraft Planet" (Planet 2). The "ProceduralPlanetGodot" (Planet 4) scored lower, and "Poor Man's Sky" (Planet 3) had the lowest ratings. Several factors could explain these differences. The first two planets were specifically designed to enhance immersion, with features that likely contributed to their higher scores. In contrast, the other two planets had different focuses, which may have led to their lower scores in this category. Another potential explanation is the lack of a player model in the first-person perspective used in the third and fourth planets. Some participants mentioned that they would have preferred to experience the world from inside a rocket for a first-person perspective. On the other hand they figured that a rocket model works well for third-person views. This discrepancy could have influenced immersion ratings of the participants.

In the "Exploration and Fun" category, the "Minecraft Planet" (Planet 2) scored the highest, though only slightly above the "Simple Planet" (Planet 1). The other two planets, "Poor Man's Sky" (Planet 3) and "ProceduralPlanetGodot" (Planet 4), scored considerably lower. The higher score for the "Minecraft Planet" may be due to its more dynamic terrain generation compared to the "Simple Planet". Both of these planets offered engaging exploration experiences, which explains their high scores. On the other hand, the other two planets were criticized for having limited content. "Poor Man's Sky" primarily featured terrain without much variation, which led to quicker boredom, according to participant feedback. "ProceduralPlanetGodot", while visually striking at first, also lacked depth in exploration, leading to mixed opinions and one of the highest standard deviations in scores. This suggests that while some participants found the initial visuals exciting, they quickly became less engaged with the planets.

The "Minecraft Planet" (Planet 2) scored the highest in "Terrain Features and Realism", as expected, given its focus on realistic and detailed terrain. The "Simple Planet" (Planet 1) followed closely behind, likely due to its similarly detailed terrain and realistic textures. In

contrast, the other two planets, "Poor Man's Sky" (Planet 3) and "ProceduralPlanetGodot" (Planet 4), scored lower, likely because they lacked the same level of detail. These two planets mainly featured colored terrain without additional features like trees or textured surfaces, which contributed to their lower realism scores. The addition of more complex features in the first two planets, such as textures and trees, as well as a vibrant blue atmosphere, likely contributed to their higher ratings in this category.

In the Movement and Controls category, the "Simple Planet" (Planet 1) received the highest score, with "Minecraft Planet" (Planet 2) scoring notably lower. The "Minecraft Planet" also displayed the largest variability in scores, as indicated by the high standard deviations. This discrepancy could be explained by differences in how participants understood the question, "I felt like there was delay between my actions and the expected outcomes". Some participants associated the delay with the smooth movement of the rocket ship, while others mentioned the loading times for terrain and tree generation, which could have contributed to lower ratings for Planet 2. The "Poor Man's Sky" (Planet 3) and "ProceduralPlanetGodot" (Planet 4) both performed better than the "Minecraft Planet" in this category, with "ProceduralPlanetGodot" scoring slightly higher. Participants reported that the "Movement and Controls" for all planets were generally smooth, besides the issues mentioned for Planet 2.

Another interesting takeaway is that Planet 1 (Simple Planet) and Planet 2 (Minecraft Planet) performed quite similarly across many categories, which makes sense as they are visually similar, despite having different terrain generation algorithms. This raises the question of whether the presentation of the planets had a significant impact on the overall user experience. Both planets offered visually rich and engaging terrain that may have contributed to participants feeling similarly immersed, which is why the scores for "Immersion and Presence" were close across both planets.

A second possible explanation for their similar performance could be the loading times of the terrain generation and in-world features of Planet 2. As it had more complex terrain that required additional time to load, this could have disrupted the overall immersion for some participants. This could explain why, despite being a more complex world, its scores were closer to Planet 1 than expected. If loading times were reduced or eliminated, it is possible that Planet 2 would have received even higher scores, particularly in "Immersion and Presence" and "Movement and Controls".

One noteworthy pattern observed across the results is the strong correlation between ratings in the different categories. For example, participants who rated "Immersion and Presence" highly often also rated "Exploration and Fun" and "Terrain Features and Realism" higher. This suggests that, for most participants, the overall quality of the terrain and sense of immersion were important factors in determining their enjoyment of exploration and the perceived realism of the world. Conversely, planets with lower ratings in one category often scored poorly in other areas as well. This suggests that improvements in one area (e.g., better terrain features or smoother movement) might also boost ratings in other areas, such as immersion and exploration. These findings show that it is important to not only consider individual elements alone but also how they interact with the complete scene.

5.5. Limitations

While this study provides valuable insights into how different levels of environmental detail and procedural generation techniques affect user perceptions, there are several limitations to it.

One potential limitation of this study is participant bias. Some participants may have known which planets were developed by the researcher. This awareness might have influenced their feedback, making them more likely to provide positive responses or less inclined to offer critical remarks, potentially skewing the ratings - especially if the participants felt the need to be encouraging. To mitigate this in future studies, steps could be taken that prevent the participants from knowing more about the projects until after the experiment has been finished. Another limitation of the study is the lack of sufficient content in "Poor Man's Sky" and "ProceduralPlanetGodot". These planets were intentionally designed to be more simple, focusing especially on the terrain generation. However this lack of additional interactive elements may have contributed to lower user ratings. One of the problems of Planet 2, which has already been mentioned, is the performance issue with loading the planet. It would be very interesting to know how the opinion on the planet would change, if this was not the case. As Planet 2 already performed very well, it could be possible, that the ratings would increase even more and the level of immersion would be the highest by far. The four planets used in this study represent only a small subset of possible procedural generation techniques and environmental designs. Future studies could benefit from testing a wider range of planets with different algorithms and environmental features, which would provide a more comprehensive understanding of how procedural generation affects user perception.

6. Conclusion

In this thesis, two unique procedural planets were created using Godot. The first planet, referred to as the "Simple Planet", uses a basic Fractal Brownian Motion approach applied per vertex to generate terrain features. This approach is simpler, but still produces varied and natural-looking results. The second planet, the "Minecraft Planet", employs a more advanced method, utilizing four different Perlin Noise values per vertex, similar to how terrain is generated in the game "Minecraft". This method adds more complexity and variation to the terrain, making it feel more dynamic. To start with, both planets were created on a sphere with a well-distributed mesh, ensuring that the terrain is smoothly applied to the spherical surface. Various additional features, such as textures, trees and a sun, were added to enhance the visual appearance and realism of the planets. After generating the planets, executable versions were created for user testing.

For the comparative part of the study, two additional procedural planet generation projects were chosen as reference. One was based on a popular tutorial, offering a visually pleasing planet with basic terrain features. The other was a straightforward planet with less detailed procedural generation, which can nonetheless be explored at close distance. These served as benchmarks to compare the effectiveness of the techniques used in this thesis. The study involved 15 participants, each asked to evaluate all four worlds. The participants answered 12 questions for each planet, divided into four categories: "Immersion and Presence", "Exploration and Fun", "Terrain Features and Realism", and "Movement and controls". The results of the questionnaire showed that the self-created worlds ("Simple Planet" and "Minecraft Planet") consistently performed better than the comparison examples. In particular, the Minecraft-based planet scored slightly higher than the Simple Planet, likely due to its more complex terrain generation, though this planet also had noticeable performance issues. These issues, such as delays in terrain and tree generation, may have impacted the user experience and decreased the overall rating in certain areas. However, without these performance drawbacks, the Minecraft Planet could potentially have been rated even higher. In contrast, the comparison planets performed slightly worse overall.

Future research could explore a broader range of procedural generation techniques to create even more diverse and complex environments. Further testing with a larger sample size and different levels of user experience could provide additional insights into how users with varying backgrounds perceive these worlds. In addition, improving performance, reducing load times and adding more features could help create even more immersive and engaging experiences.

Bibliography

[Ado23a] Adobe. Anisotropic noise | substance 3d designer, 2023. Accessed: 2024-11-12.

[Ado23b] Adobe. White noise | substance 3d designer, 2023. Accessed: 2024-11-12.

[AHI09] Bradley Atcheson, Wolfgang Heidrich, and Ivo Ihrke. An evaluation of optical flow algorithms for background oriented schlieren imaging. *Experiments in Fluids*, 46:467–476, 03 2009. Accessed: 2024-11-12.

[Ale24] Alex Peterson. Spacescape, 2024. Accessed: 2024-09-26.

[Ath24a] Athillion. I made sebastian lague's procedural planet in godot 4, 2024. Accessed: 2024-09-23.

[Ath24b] Athillion. Proceduralplanetgodot, 2024. Accessed: 2024-09-23.

[Bor20] Boris. Dungeon generation in binding of isaac. *BorisTheBrave.Com*, 2020. Accessed: 2024-10-30.

[CBPd11] Daniel Carli, Fernando Bevilacqua, Cesar Pozzer, and Marcos d'Ornellas. A survey of procedural content generation techniques suitable to game development. In *A Survey of Procedural Content Generation Techniques Suitable to Game Development*, pages 26–35, 11 2011. Accessed: 2024-09-26.

[CCH05] Alberto Candussi, Nicola Candussi, and Tobias Höllerer. Rendering realistic trees and forests in real time. *Proc. Eurographics 2005*, 01 2005. Accessed: 2024-09-26.

[FDWZ20] Roland Fischer, Philipp Dittmann, René Weller, and Gabriel Zachmann. Autobiomes: procedural generation of multi-biome landscapes. *Vis. Comput.*, 36(10–12):2263–2272, October 2020. Accessed: 2024-10-10.

[Gam] Game.Guide. How much space does minecraft take up? Accessed: 2025-01-11.

[GGG+13] Jean-David Génevaux, Eric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics*, 32:13, 07 2013. Accessed: 2024-10-10.

[God] Godot. Accessed: 2025-03-29.

[God24a] Godot Engine. Exporting projects - godot engine, 2024. Accessed: 2024-09-26.

[God24b] Godot Engine. Features - godot engine, 2024. Accessed: 2024-07-29.

[God24c] Godot Engine. Sky - godot engine, 2024. Accessed: 2024-07-29.

[Hen22] HenrikKniberg. Minecraft terrain generation in a nutshell, 2022. Accessed: 2024-12-24.

[igr] igroup. igroup presence questionnaire (ipq) overview. Accessed: 2024-09-23.

[jfc] jfc3. Poor man's sky. Accessed: 2024-09-23.

[JSR24] Aryamaan Jain, Avinash Sharma, and K S Rajan. Learning based infinite terrain generation with level of detailing. In *2024 International Conference on 3D Vision (3DV)*, pages 1048–1058, 2024. Accessed: 2024-10-10.

[KM20] Marek Kopel and Grzegorz Maciejewski. Comparison of procedural noise-based environment generation methods. In Ngoc Thanh Nguyen, Bao Hung Hoang, Cong Phap Huynh, Dosam Hwang, Bogdan Trawiński, and Gottfried Vossen, editors, *Computational Collective Intelligence*, pages 878–887, Cham, 2020. Springer International Publishing. Accessed: 2024-09-26.

[Lag18] Sebastian Lague. Procedural planet generation. YouTube Playlist, 2018. Accessed: 2024-09-29.

[LG07] Qing-Zhong Li and Xiu-Rong Gao. Generation and visualization of 3-d realistic natural terrain based on fbm. In *Fourth International Conference on Image and Graphics (ICIG 2007)*, pages 915–919, 2007. Accessed: 2024-09-26.

[LL10] Kui-Ru Li and Xiao-Feng Lei. An accumulated method based on fractal for automatic terrain generation. In *2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT)*, volume 1, pages 426–430, 2010. Accessed: 2024-10-10.

[LLC+10] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D.S. Ebert, J.P. Lewis, K. Perlin, and M. Zwicker. A Survey of Procedural Noise Functions. *Computer Graphics Forum*, 2010. Accessed: 2024-10-10.

[LLZ14] Zhongxing Liang, Dan Liu, and Ming Zhou. Research on large scale 3d terrain generation. In *2014 IEEE 17th International Conference on Computational Science and Engineering*, pages 1827–1832, 2014. Accessed: 2024-09-26.

[Min24] Minecraft. World generation, 2024. Accessed: 2024-09-29.

[MK18] F Michelic and M Kenzel. Real-time rendering of procedurally generated planets. *CESCG*, 2018. Accessed: 2025-01-11.

[NAS16] NASA Johnson. Earth observations "islands in the sky", 2016. Accessed: 2025-3-9.

[No a] No Man's Sky. No man's sky. Accessed: 2025-01-11.

[No b] No Man's Sky. No man's sky. Accessed: 2024-09-23.

[NVI04] NVIDIA. Chapter 16. accurate atmospheric scattering, 2004. Accessed: 2024-09-26.

[Sca] Scale Of Universe. How big is a minecraft world? Accessed: 2025-01-11.

[Seb18] Sebastian Lague. Procedural planet generation, 2018. Accessed: 2024-09-23.

[Sie21] Daniel Sieger. Generating meshes of a sphere, 2021. Accessed: 2024-10-30.

[Sim21] SimonDev. 3d world generation (javascript). YouTube Playlist, 2021. Accessed: 2024-09-29.

[SKKW23] Markus Schütz, Bernhard Kerbl, Philip Klaus, and Michael Wimmer. Gpu-accelerated lod generation for point clouds, 2023. Accessed: 2024-10-10.

[Sky24] No Mans Sky. Planet, 2024. Accessed: 2024-10-30.

[SM17] Kenichi Sugihara and Takahiro Murase. Automatic generation of a 3d terrain model from key contours. In *2017 International Conference on Cyberworlds (CW)*, pages 111–117, 2017. Accessed: 2024-10-10.

[Ste] Steam. No man's sky. Accessed: 2025-01-11.

[TCL+13] J. Togelius, A.J. Champandard, Pier Luca Lanzi, M. Mateas, A. Paiva, Mike Preuss, and K.O. Stanley. Procedural content generation: goals, challenges and actionable steps. *Dagstuhl Follow-Ups*, 6:61–75, 01 2013. Accessed: 2025-01-11.

[Tyl] Tyler Glaiel. How to make your own game engine (and why). Accessed: 2025-03-29.

[Uni] Unity. Accessed: 2025-03-29.

[Unr] Unreal Engine. Accessed: 2025-03-29.

[UQO04] UQO Cyberpsychology Lab, Witmer & Singer. Presence questionnaire, 2004. Accessed: 2024-09-23, Revised by the UQO Cyberpsychology Lab (2004).

[VL19] Ryan J. Vitacion and Li Liu. Procedural generation of 3d planetary-scale terrains. In *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, pages 70–77, 2019. Accessed: 2024-09-23.

[VMP21] Georgios Voulgaris, Ioannis Mademlis, and Ioannis Pitas. Procedural terrain generation using generative adversarial networks. In *Procedural Terrain Generation Using Generative Adversarial Networks*, pages 686–690, 08 2021. Accessed: 2024-09-26.

[Zuc22] Alan Zucconi. The world generation of minecraft. *Alan Zucconi Blog*, 2022. Accessed: 2024-09-24.

List of Tables

List of Figures

Listings

A. Appendix

A.1. Detailed Interview Questions & Results

Questions:

- I felt that the virtual world surrounded me.
- I felt like I was just perceiving pictures. (Negative question, marked with *)
- I had a sense of acting in the virtual space, rather than operating something from outside.
- I felt as if I were physically there in the virtual space.
- The planet was fun to explore.
- I was completely captivated by the virtual world.
- The terrain featured a diverse range of features and landscapes.
- The mechanism which controlled movement through the environment felt natural.
- I was able to examine objects closely.
- I was able to examine objects from multiple viewpoints.
- I felt like there was delay between my actions and the expected outcomes. (Negative question, marked with *)
- The terrain looked realistic.

Answers:

Table A.1.: Participant Data Simple Planet

ID	Age	Gender	Experience	Planet 1 ID	Q1	Q2(*)	Q3	Q4	Q5
1	23	Male	1	4	3	1	5	5	3
2	58	Male	4	3	4	2	3	4	3
3	15	Male	1	3	4	1	3	3	5
4	47	Female	4	1	4	2	4	4	4
5	27	Male	1	1	4	2	3	2	4
6	27	Male	2	1	4	2	4	3	4
7	23	Male	1	3	4	4	4	4	3
8	27	Female	4	2	4	2	4	4	4
9	24	Male	2	4	4	2	3	4	4
10	47	Male	2	1	4	1	4	4	5
11	45	Female	4	2	4	1	5	4	5
12	13	Male	1	3	5	1	5	4	5
13	15	Male	1	2	2	2	4	1	4
14	26	Male	4	4	5	1	5	5	5
15	17	Female	3	2	5	1	5	4	5

ID	Q6	Q7	Q8	Q9	Q10	Q11(*)	Q12
1	2	3	5	5	5	2	4
2	3	4	3	4	4	2	3
3	3	5	5	5	5	1	5
4	4	4	3	4	4	2	4
5	4	4	2	4	5	1	2
6	3	4	3	4	5	1	3
7	3	4	4	5	5	1	4
8	4	4	4	4	4	2	3
9	3	4	4	4	4	2	4
10	4	5	5	5	5	1	4
11	4	5	3	5	5	1	4
12	5	5	5	5	5	1	5
13	2	4	5	5	5	1	4
14	5	5	5	5	5	1	5
15	5	5	5	5	5	1	4

Table A.2.: Participant Data Minecraft Planet

ID	Planet 2 ID	Q1	Q2(*)	Q3	Q4	Q5
1	2	4	1	5	5	5
2	2	4	2	3	4	3
3	4	4	1	3	3	5
4	2	4	2	5	5	5
5	2	4	2	3	2	4
6	3	5	2	2	3	4
7	1	2	2	2	1	4
8	1	4	2	3	4	3
9	1	3	4	2	1	1
10	4	5	1	5	5	4
11	4	4	1	4	4	5
12	4	5	1	5	4	5
13	4	3	2	2	1	5
14	1	5	1	4	4	5
15	4	5	1	5	5	5

ID	Q6	Q7	Q8	Q9	Q10	Q11(*)	Q12
1	3	4	5	5	5	3	4
2	3	4	3	4	4	2	3
3	3	5	5	5	5	1	5
4	4	5	4	5	5	4	5
5	4	5	2	4	5	1	2
6	4	4	4	5	5	2	3
7	3	4	2	5	5	1	4
8	4	4	3	3	4	1	3
9	2	2	2	4	4	4	1
10	5	5	4	4	5	1	4
11	3	5	4	5	5	1	4
12	5	5	5	5	5	1	5
13	2	5	5	5	5	1	4
14	5	5	3	5	5	1	5
15	5	5	5	5	5	2	5

Table A.3.: Participant Data Poor Man's Sky

ID	Planet 3 ID	Q1	Q2(*)	Q3	Q4	Q5
1	1	2	2	1	2	4
2	1	1	5	1	1	3
3	1	3	2	2	2	3
4	3	3	2	3	2	2
5	3	4	1	4	2	5
6	4	4	2	4	5	2
7	4	3	5	3	4	3
8	3	4	2	4	3	3
9	3	4	2	4	4	5
10	2	5	1	5	4	5
11	1	3	2	3	3	4
12	1	3	1	4	4	5
13	3	3	3	4	1	4
14	3	5	1	5	5	4
15	1	4	1	5	5	3

ID	Q6	Q7	Q8	Q9	Q10	Q11(*)	Q12
1	2	2	4	3	4	3	2
2	1	3	3	4	4	2	1
3	2	3	4	4	4	1	2
4	2	4	4	2	3	2	4
5	4	4	2	4	5	1	4
6	3	2	3	2	4	2	3
7	4	4	5	4	2	1	5
8	3	3	4	3	4	2	4
9	4	4	4	4	4	3	4
10	4	4	5	3	5	1	5
11	3	4	4	4	4	2	3
12	4	5	5	5	5	1	4
13	2	4	5	2	4	1	2
14	5	5	5	3	4	5	5
15	4	2	3	4	4	1	2

Table A.4.: Participant Data Procedural Planet Godot

ID	Planet 4 ID	Q1	Q2(*)	Q3	Q4	Q5
1	3	3	1	1	2	3
2	4	3	3	2	2	2
3	2	3	2	2	2	2
4	4	3	4	2	3	3
5	4	4	1	4	2	4
6	2	5	2	4	4	4
7	2	2	4	3	2	3
8	4	5	1	5	5	5
9	2	4	2	4	3	3
10	3	5	1	4	5	4
11	3	4	2	4	3	4
12	2	4	1	5	4	5
13	1	1	3	3	1	3
14	2	5	1	5	5	5
15	3	4	1	4	5	3

ID	Q6	Q7	Q8	Q9	Q10	Q11(*)	Q12
1	1	1	3	4	4	1	2
2	2	3	2	4	3	2	1
3	2	2	5	3	4	1	2
4	2	3	4	4	4	2	2
5	4	5	2	2	5	1	4
6	3	2	3	2	5	1	3
7	2	2	3	3	4	3	3
8	5	5	5	5	5	1	5
9	3	4	3	4	4	2	3
10	5	5	4	4	5	1	5
11	2	4	2	4	4	2	4
12	4	5	5	5	5	1	5
13	2	3	5	4	5	1	3
14	5	3	4	3	3	1	3
15	3	2	5	2	5	1	5