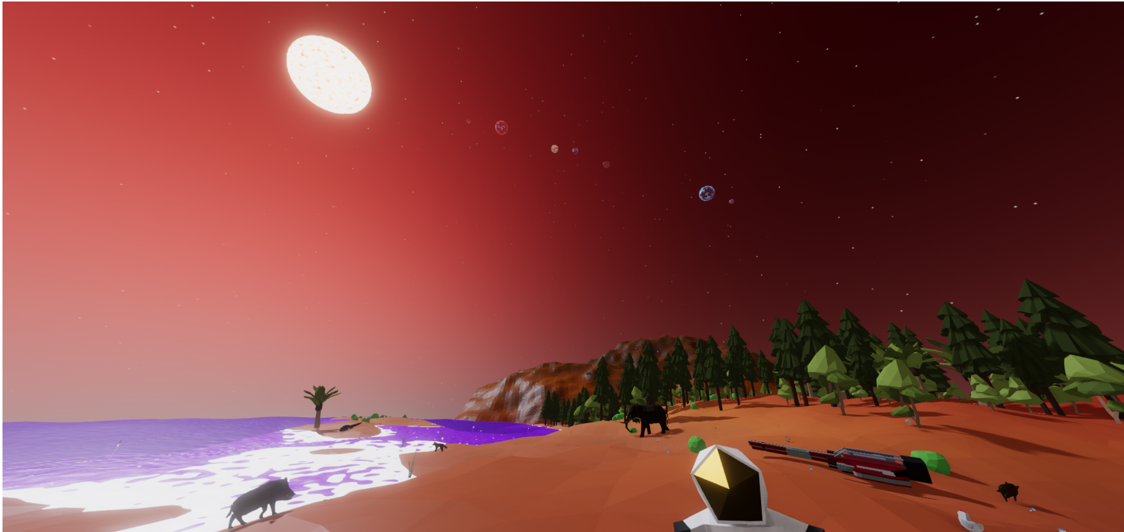
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Exo Explorer: A procedurally generated solar system

An application for interactive exploration of a procedurally generated solar system containing diverse ecosystems

Bachelor's thesis in Computer science and engineering

Joel Båtsman Hilmersson
Elin Forsberg
Isak Gustafsson
Isak Hansson
Manfred Hästmark
Daniel Persson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Bachelor's thesis 2023

Exo Explorer: A procedurally generated solar system

An application for interactive exploration of a procedurally generated solar system containing diverse ecosystems

Joel Båtsman Hilmersson
Elin Forsberg
Isak Gustafsson
Isak Hansson
Manfred Hästmark
Daniel Persson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Exo Explorer: A procedurally generated solar system
An application for interactive exploration of a procedurally generated solar system containing diverse ecosystems
Joel Båtsman Hilmersson Elin Forsberg Isak Gustafsson Isak Hansson Manfred Hästmark Daniel Persson

© Joel Båtsman Hilmersson, Elin Forsberg, Isak Gustafsson, Isak Hansson, Manfred Hästmark, Daniel Persson 2023.

Supervisor: Staffan Björk, Department of Computer Science and Engineering
Graded by teacher: Morten Fjeld, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Bachelor's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A procedurally generated solar system from the perspective of one of the planets.
Link to project repository: https://github.com/Danilll01/Kandidatarbete2023/

Typeset in LaTeX
Gothenburg, Sweden 2023

Exo Explorer: A procedurally generated solar system
An application for interactive exploration of a procedurally generated solar system containing diverse ecosystems
Joel Båtsman Hilmersson, Elin Forsberg, Isak Gustafsson, Isak Hansson, Manfred Hästmark, Daniel Persson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This project aims to develop a solar system in the Unity game engine, focusing on procedural content generation (PCG) for planets and their ecosystems. PCG, which utilizes algorithms and randomness to generate content automatically, offers a promising approach to create a vast and immersive solar system. This report will present an application of PCG to generate a solar system for users to explore and interact with. Challenges faced during development, such as adapting common 2D programming techniques to 3D spheres, are also discussed. The resulting system includes forests, lakes and various creatures on each planet, with climates adding to the environment's complexity. The creatures have hunger, thirst, and reproduction capabilities, which contribute to a dynamic ecosystem. This project establishes a strong foundation for researchers and developers interested in PCG and its applications in creating vast and immersive environments.

Sammandrag

Målet med det här projektet är att utveckla ett solsystem i spelmotorn Unity, med fokus på procedurell innehållsgenerering (PCG) för planeter och dess ekosystem. PCG, som använder sig av algoritmer och slumpmässighet för att generera innehåll automatiskt, erbjuder ett lovande tilllvägagångssätt för att skapa ett stort och inlevelserikt solsystem. Den här rapporten kommer presentera en applikation av PCG för att generera solsystem för användare att utforska och interagera med. Utmaningar bemötta under utvecklingen, som att anpassa vanliga 2D programmerings tekniker till 3D sfärer, kommer också att diskuteras. Det slutgiltiga systemet består av skogar, sjöar och olika varelser på varje planet med klimat som ökar miljöernas komplexitet. Varelserna har hunger-, törst- och fortplantningsförmågor, vilket bidrar till ett dynamiskt ekosystem. Detta projekt ger en stabil grund för forskare och utvecklare intresserade av PCG och dess tillämpningar i skapandet av stora och inlevelserika miljöer.

Keywords: PCG, simulation, solar system, generation, planet, Marching Cubes, creature simulation, noise, Unity, ecosystem

Acknowledgements

We want to thank our supervisor Staffan Björk for his fantastic engagement in our project. Staffan has been a source of many interesting ideas and was the inspiration behind many parts of the project. As a mentor, he has always been incredibly supportive, providing answers to our difficult questions and been a constant source of encouragement.

Joel Båtsman Hilmersson, Elin Forsberg, Isak Gustafsson, Isak Hansson, Manfred Hästmark, Daniel Persson, Gothenburg, May 2023

Contents

List of Figures **xii**

List of Tables **xiii**

Glossary **xiv**

1 Introduction **1**
- 1.1 Purpose . . . 1
- 1.2 Project limitations . . . 2
- 1.3 Contribution . . . 2
- 1.4 Thesis structure . . . 2

2 Background **3**
- 2.1 Procedural content generation . . . 3
- 2.2 Noise algorithms . . . 3
- 2.3 Terrain generation techniques . . . 4
- 2.4 Simulation . . . 5
- 2.5 Ecosystems . . . 5
- 2.6 GPU computing . . . 6
- 2.7 Chunks . . . 6
- 2.8 Related works . . . 6

3 Method and planning **8**
- 3.1 Tools . . . 8
- 3.2 Workflow . . . 9
- 3.3 Milestones . . . 9
- 3.4 Planning . . . 10
  - 3.4.1 Time plan . . . 11
  - 3.4.2 Solar system . . . 12
  - 3.4.3 Planet generation . . . 12
  - 3.4.4 Ecosystem . . . 12
  - 3.4.5 Player system . . . 13
- 3.5 Societal and ethical aspects . . . 13
  - 3.5.1 Education . . . 13
  - 3.5.2 Automation of content creation . . . 13

4 Process **14**

4.1 Solar system 14
4.1.1 Planet orbits 14
4.1.2 Directional sunlight 15
4.1.3 Ambient light 16
4.2 Planet Generation 17
4.2.1 Biomes 17
4.2.2 Terrain generation 18
4.2.3 Terrain coloring 21
4.2.4 Planetary atmospheres 23
4.3 Ecosystem 25
4.3.1 Creatures 25
4.3.2 Foliage 27
4.3.3 Water 29
4.4 Player system 31
4.4.1 Player movement 31
4.4.2 Player model and animations 32
4.4.3 Spaceship 32
4.5 Miscellaneous 34
4.5.1 Deterministic randomness 34
4.5.2 User interface 35

5 Results **36**
5.1 Overview of the final system 36
5.2 Solar system 36
5.2.1 Planet orbits and rotation around their axis 36
5.2.2 Light system 37
5.3 Planet Generation 37
5.3.1 Biomes 38
5.3.2 Terrain generation 38
5.3.3 Terrain coloring 38
5.3.4 Planetary atmospheres 38
5.4 Ecosystem 39
5.4.1 Creatures 39
5.4.2 Foliage 40
5.4.3 Water 40
5.5 The player system 41
5.6 Miscellaneous 41
5.6.1 Deterministic randomness 41
5.6.2 User interface 41
5.6.3 Music and sound effects 42

6 Discussion **43**
6.1 Societal and ethical aspects 43
6.2 Project overview 43
6.2.1 Method reflection 45
6.2.2 Process reflection 45
6.3 Solar system 46

6.3.1 Lighting 46
6.4 Planet generation 46
6.4.1 Biomes 47
6.4.2 Terrain generation 47
6.4.3 Planetary atmospheres 47
6.5 Ecosystem 47
6.5.1 Creatures 48
6.5.2 Foliage 48
6.6 Player system 48
6.7 Miscellaneous 49
6.7.1 Deterministic randomness 49
6.7.2 User interface 49
6.7.3 Chat-GPT 49
6.8 Future work 49

7 Conclusion **50**

Bibliography **51**

A An additional technique **I**
A.1 Using heighmap-techniques on Marching Cubes I

B Additional images **II**

List of Figures

2.1 Samples from Perlin, Simplex and Worley noise. 4
2.2 Terrain generation using heightmaps and Marching Cubes. 5
2.3 Example of games using PCG. 7
2.4 Example of games using ecosystem simulation. 7

3.1 Time plan for the project. 11

4.1 Early stage of the solar system with displayed orbits. 14
4.2 The two different main lightning ideas considered and used. 15
4.3 The final sunlight solution shown in two colors based on the sun's temperature. 16
4.4 A comparison between not having and having ambient light when the application is running. 17
4.5 The different distances the ambient system uses to calculate the light intensity. 17
4.6 Sphere texture and the resulting body generated by the Marching Cubes algorithm. 18
4.7 First and second attempt of terrain generation. 19
4.8 Planet chunks and chunk culling. 19
4.9 Terrain improvements and optimizations. 20
4.10 An image of the mountains generated with the new mountain function. 21
4.11 The flat-looking terrain shader before assigning a normal angle. . . . 22
4.12 The first complete version of the terrain shader. 22
4.13 Part of the terrain shader implementation amplifying noise using "power" and "smoothstep" functions. The amplification is done by a power node that uses the formula $value = value^2$ for each value. The smoothstep node filters the output from the power node to make it suitable for linear interpolation. These functions make the aspect more prominent. 23
4.14 Two versions of an atmosphere shader. 23
4.15 Difference between light intensities of the atmospheres. 24
4.16 Three possible different looking planet atmospheres. 25
4.17 Pack spawning. 26
4.18 Different creatures eating. 26
4.19 First and second implementation of foliage side by side. 28
4.20 Final implementation of foliage and the circle that represents the area which is used for spawning foliage. 28

4.21 Two figures illustrating forests and biome based spawning. 29
4.22 Final implementation of foliage after the coloring was added. 29
4.23 First and second implementation of water side by side. 30
4.24 Final version of water from two perspectives of the same planet. . . . 31
4.25 The different player models used under development from placeholder to final version. 32
4.26 The two different spaceship movement options and their effect on the final ship rotation. 33
4.27 The chain of seeds where the generators X, Y, and Z's seed is unique while still being deterministic. U is the universal seed. 34

5.1 The different orbiting states of the solar system. 37
5.2 The light system at three points during one day cycle. 37
5.3 Different types of colors for each biome area. 38
5.4 Every creature species present in the simulation. 39
5.5 The final foliage and water systems. 40
5.6 The final start menu, loading menu, pause menu and player GUI. . . 42

B.1 Different angles taken from the ship. II

List of Tables

3.1 Must have features in the project. 10
3.2 Should have features in the project. 10
3.3 Could have features in the project. 10

6.1 Must have features in the project. 44
6.2 Should have features in the project. 44
6.3 Could have features in the project. 44
6.4 Non-planned added features. 45

Glossary

Assets - Files for use in game development. Most typically 3D models, images and sounds.
Branch - Used in GitHub to develop features in parallel
Ecosystem - An area consisting of organisms and nonliving parts working together to form a bubble of life.
GUI - Graphical User Interface is a visual interface that allows users to interact with electronic devices using icons, windows, buttons, and menus.
Multiplayer - When you play a game with multiple people, either online or locally.
MVP - Minimal viable product is a simplified version of a product with just enough features to be classified as a product.
N-body simulation - Calculates the motion of N objects over time in an N-body system with gravitational interactions.
Noise - A pseudo random continuous function used to create a procedural texture primitive.
Procedural Content Generation (PCG) - Automatic generation of content, often used in video games.
Procedural generation - An algorithm to automatically generate data or content instead of doing it manually.
Pull request - A developer's requests to merge their code into the master branch.
Raycast - A line projection in 3D space to determine intersection between objects.
Simulation - A way to imitate real-world processes and systems by modeling it in, for example, a computer program.
Single player - When you play a game by yourself.
Terraforming - The process of manually shaping terrain to one's liking.
Texture - A texture refers to the visual surface of an object. Usually a 2D image or set of images.
Two-body system - A model that consists of two objects interacting through gravitational forces.
Unity - Popular game engine and development tool.
Seed - Seed, a textual string employed to procedurally generate a game world.
Skybox - A box that makes up the sky in the Unity game engine
Raycast - A ray that is sent out in a given direction to possibly find a hit location

1 Introduction

The concept of procedural generation has been a topic of interest in the gaming industry for many years. The ability to generate complex and diverse environments in real-time has opened up new opportunities for game designers and players alike [1]. Not only does this technique greatly enhance the player experience by offering endless possibilities and unique content, but it also greatly increases the replayability of games, as no two playthroughs will be exactly the same. Moreover, procedurally generating game elements can save a considerable amount of time and resources compared to traditional design methods, where the content would be created manually [2].

Modeling and simulating a solar system in 3D software is insightful and useful in many ways. If done right, it can allow users to discover and develop a rich understanding of different astronomical phenomena and pose questions to themselves that motivate further investigation [3]. Likewise, modeling an ecosystem can theoretically have the same effect on the user, encouraging further investigation into how ecosystems work. Consequently, a solar system model with planetary ecosystems can help individuals gain interest in the field.

From an academic standpoint, the procedural generation and simulation of a solar system is an interesting area to investigate. It provides an opportunity to advance the understanding of solar systems and pushes the boundaries of what is possible in game development and computer graphics further. To accomplish this, game engines such as Unity can be utilized since they offer complex built-in rendering and simulation capabilities with relatively little work [4]. This allows developers to collaborate with researchers to advance the fields and contribute to the academic community.

1.1 Purpose

The purpose of this project is to construct a game-like application to simulate and procedurally generate a simplified solar system that a player has the option to explore. Exploration will consist of seamless walking and spaceship travel, enabling players to traverse on and between various planets within the generated system. To allow players to revisit solar systems, the generation will be deterministic, allowing the same system to be generated again. Most planets will have their own simplified ecosystem which allows for diversity throughout the generated planets. The simu-

lation must also be able to run on a moderate computer while allowing good frame rates.

1.2 Project limitations

Due to time constraints on the project, the general scope of the project had to be limited. Therefore, a decision was made to only generate one solar system every playthrough. Additionally, the focus of the project was placed on the generation, exploration and simulation of the solar system rather than developing an explicit game story as this fits the purpose of the project better.

For the same reason, it was decided not to have multiplayer or multi-user support. This decision was made to help avoid spending extra time on developing split-screen functionality or dealing with complicated network coding, allowing more focus and time on the core features of the project.

Lastly, a choice was made to develop the application specifically for the Windows platform to prevent any potential problems that could arise when working with other operating systems. By focusing on a single platform, more time and effort could be dedicated to creating different features and improving the simulation experience, ensuring the time could be utilized efficiently.

1.3 Contribution

The report aims to contribute to the research done with simulations and procedural generation techniques for creating various assets. This will be achieved by providing a comprehensive understanding of the process, highlighting areas for improvement and lessons learned.

The project itself has the possibility to pique interest in space, planets and ecosystems. It can therefore contribute to an increased excitement towards the mentioned areas and drive individuals to learn more about them.

1.4 Thesis structure

The structure of this thesis is as follows. After the introduction, a background chapter follows which discusses previous work in the field and tools that assists in development. Subsequently, a theory chapter presents current research and factual information relevant to the project. This is continued by a method and planning chapter which details the approach taken, working structure, and final desired features. The thesis then has a process chapter that in detail explains the step-by-step process of implementing the different features found in the application. At the end, a result chapter shares the final version of the project with a discussion and conclusion chapter tightly behind.

2
Background

This chapter will explore several techniques and concepts that are essential for the procedural generation of solar systems with planetary ecosystems. These include algorithms for procedural content generation and noise generation, as well as Newton's law of gravity. Considerations for performance improvements are also discussed. Lastly, related work within the areas will be brought forth.

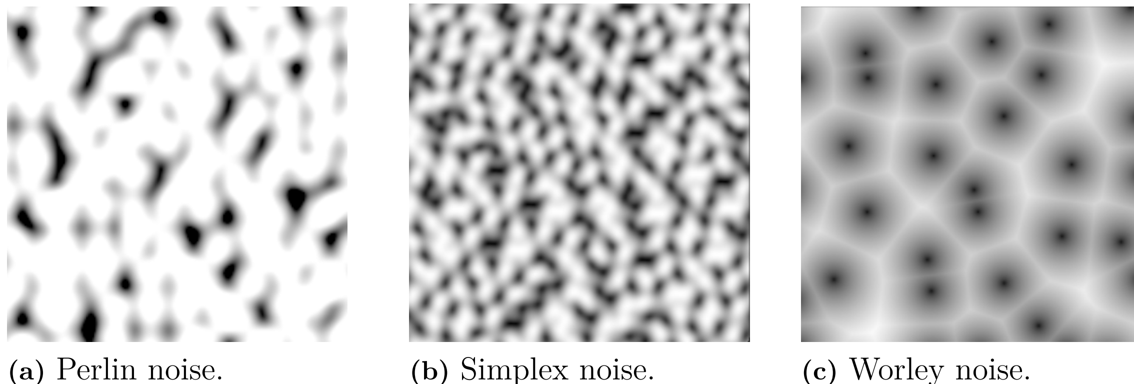
2.1 Procedural content generation

The use of Procedural content generation (PCG) has become widespread in the field of game design due to its ability to produce new and unique content with each run, leveraging algorithms and mathematical models to introduce diversity to the game [1]. PCG refers to the creation of content from algorithms, either partially as in modifying a manually created piece, or totally as in creating an entirely new piece of content from the algorithm without a human's input.

Different qualities of a PCG solution can be evaluated and used to determine the validity of a solution for a certain problem [5, pp. 6-7]. Speed, reliability, controllability, diversity, and believability can describe the quality of a PCG solution to the end user. Most of the evaluation of a solution, except for speed, is made easier by human input as experiences are often best interpreted by humans due to the complexities of testing human experience. However, there are ways to estimate the quality of the generated content programmatically and, based on this estimation, evaluate the solution [5, p. 23].

2.2 Noise algorithms

As described in section 2.1, PCG is the process of creating content using algorithms. Some of these algorithms are called procedural noise algorithms [5, pp. 57-72]. According to Ares Lagae, noise is "the random number generator of computer graphics" [6, p. 2580], and there exists many different algorithms for creating this randomness. The noise algorithms which will be covered in this section are Simplex, Perlin and Worley noise. Samples from these noises can be seen in figure 2.1.



(a) Perlin noise. (b) Simplex noise. (c) Worley noise.

Figure 2.1: Samples from Perlin, Simplex and Worley noise.

Perlin noise is a gradient noise algorithm, with the advantage of being diverse which makes it good for terrain generation or texture generation [7]. The drawbacks are that there are some directional artifacts and the computational complexity scales badly as the dimension of the noise increases [8]. To further improve the Perlin algorithm, Ken Perlin developed the simplex noise algorithm [8]. Improvements included better computational complexity and no directional artifacts.

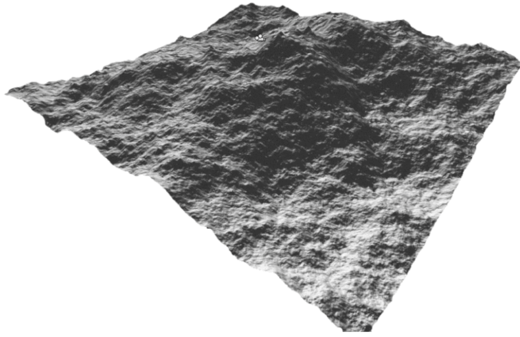
Worley noise is a visually different noise algorithm compared to Perlin and Simplex noise. It can be used to generate procedural textures such as bricks [9], and its characteristics are very different compared to the previously mentioned algorithms as seen in figure 2.1.

2.3 Terrain generation techniques

Terrain generation is an important topic when creating a procedurally generated game. This is because terrain has to be generated during runtime. Two different techniques that can be used for this are heightmaps and Marching Cubes.

Heightmaps use a grid of numbers to define how much a flat surface is raised or lowered at each point [10]. Although fast, this limits the method to only one dimension of variation for each point on the height map, thus removing the possibility of generating caves and overhangs. Furthermore, terraforming also becomes more rudimentary using this method. However, memory usage for this method scales well with increasing terrain size, making it suitable for generating large terrains. Figure 2.2a illustrates terrain generated using the height map technique.

The other technique, Marching Cubes, is an algorithm that enables a 3D representation of the terrain. It has a large feature set that allows for complex and detailed terrain generation such as caves, tunnels and overhangs [11]. While this is good for creating more advanced terrain, the memory usage scales rapidly as terrain size increases [10]. The extended feature set also enables complex modification to the terrain in real-time. Figure 2.2b illustrates the terrain's appearance.



(a) Height map technique applied to a flat surface [12]. Reproduced with permission.

(b) Terrain alteration using Marching Cubes in Astroneer.

Figure 2.2: Terrain generation using heightmaps and Marching Cubes.

2.4 Simulation

A computer simulation is a method to imitate real-world processes and systems. To simulate a system, it needs to be modeled with key characteristics or behaviors of that system [13]. Simulations are typically used to study dynamic behavior in environments that cannot be implemented in real life.

In general, if a system or process can be drawn as a flowchart of events, it can be simulated [14]. These simulations can be divided into three types: two that model the progression of space and time, and one that models the interaction between two systems. Moreover, the simulations can be used to optimize a process, test theories, train individuals and increase entertainment in video games among other things.

2.5 Ecosystems

An ecosystem is defined as a geographic area where organisms and nonliving parts work together to create a balance of life [15]. Ecosystems consist of factors that directly or indirectly depend on each other factor in the system. The interconnection of each element within the ecosystem is crucial to the sustainability of the whole system.

There are many different ecosystems, and they can be large like forests or tundra, or small like tide pools [15]. The surface of the Earth is a series of connected ecosystems. These ecosystems are also often connected in a larger biome which defines as large sections of land, sea, or atmosphere. Within each biome, there will exist several different ecosystems. Ecosystems are however threatened by human activities and numerous have been overtaken by humans [15]. The Great Plains are now used as farmland and the Amazon rainforest is also being destroyed to make room for farmland, housing and industry.

2.6 GPU computing

GPU computing refers to code running on the GPU instead of the CPU. In difference to the CPU, the GPU code is highly parallelized and works on matrices with up to more than 1000 cores active all at once [16]. To write GPU code, compute shaders can be used. It can accept any number of inputs and outputs of any type [17]. Compute shaders can be used to compute or solve any problem, just like normal code. This is different from normal shaders which are typically used for materials in game engines to tell the GPU how to draw the pixels on the screen [18].

2.7 Chunks

Chunks have many meanings in computer science but are usually about dividing data into smaller pieces that are easier to handle, such as the division of a world into discrete pieces of 3D space [19]. The simplified exclusion of certain chunks provides an easy way of increasing performance by removing entire blocks of the world from the simulation [19]. The complete or partial culling of entire chunks is made possible due to the discrete nature of chunks, with entire 3D spaces being represented with a single item that can be activated/inactivated. The culling can be done on a neighbor basis with neighbors beyond a certain degree not being included in the currently active system [19]. It may also be done by calculating the spacial distances to chunks and culling them after a certain distance.

2.8 Related works

As mentioned in section 2.1, PCG has become popular to use in various video games to create content. One modern example can be seen in the game *No man's sky* by Hello Games [20]. This popular game makes use of procedural content generation to generate a virtually infinite number of solar systems containing unique planets with their own ecosystems. A screenshot from the game can be seen in figure 2.3a.

Another famous example of a game using PCG is *Minecraft* [21], created by Mojang. To generate the world, the game uses noise to generate a height map, which combines with biome values to finalize the world generation [22]. Figure 2.3d shows a screenshot from the game.

A further application of PCG can be found in the game *Astroneer* developed by System Era [23]. In the game, planets are generated with a random seed from 32 768 unique seeds for each planet type upon the creation of a new save file. A screenshot from inside the game is shown in figure 2.3c.

There is also a well-known project that explores the procedural generation of planets and moons inside a solar system by Sebastian Lague [24]. The project generates planets and moons orbiting around a sun and can be seen in figure 2.3b.

(a) No man's sky solar system [25]. Fair use.

(b) Sebastian Lague's project [24]. MIT License.

(c) Screenshot of Astroneer.

(d) Screenshot of Minecraft.

Figure 2.3: Example of games using PCG.

There are also several games that try to simulate ecosystems. One such game is *Equilinox*. The game is about managing an ecosystem where the player can breed plants and animals to evolve new species. These creatures have specific behaviors, requirements and life cycles, while also carrying genes they can pass on to offspring with a chance of mutating. An image from the game is displayed in figure 2.4a.

The game *Dwarf Fortress* developed by Bay 12 Games is another example of ecosystem simulation [26]. It is a single-player fantasy game that generates a unique and random world with accompanying history that is home to a range of civilizations. It also generates towns, caves, and regions filled with diverse wildlife. Figure 2.4b contains an image of the world in the game.

(a) Equilinox [27]. CC-BY-SA.

(b) Screenshot of Dwarf Fortress from Youtube video by Nookrium [28].

Figure 2.4: Example of games using ecosystem simulation.

3
Method and planning

This chapter outlines the methods employed in this project, explaining the reasoning for their usage and describing the process of planning the project. The methods include the tools and workflow that were planned to be used to develop the simulation and the key features of the project.

3.1 Tools

Several tools were needed to create and manage the project. It was decided early on that the Unity game engine would be best suited for development, as the team had previous experience working with it. Unity is a widely used game engine and development tool for creating interactive digital content, primarily video games. The engine provides a suite of integrated functionalities that simplify the process of integrating essential components such as graphics, sound, physics, user interactions, and networking into games [29]. Unity also handles all data in the scene such as lights, meshes, and behaviors, and processes this information for the user [30]. This significantly reduces the workload for developers and enables them to focus on creating engaging and immersive gaming experiences [29].

The second tool planned to be used during the project was GitHub, serving as the file and version control system [31]. This online platform provides file storage and management capabilities, facilitating collaborative code maintenance among developers. It was chosen since it has great integration with Unity and is a service using Git [32] which the group is comfortable using. Trello was another tool planned to be employed in conjunction with GitHub to assist the group in version tracking [33]. Trello is a web-based project management tool that enables users to create boards, lists, and cards to organize tasks and projects. The group planned to employ it to set up activities and monitor major milestones, such as the minimum viable product (MVP).

Lastly, it was decided that Open AI's Chat Generative Pre-Trained Transformer (ChatGPT), which is a chatbot built on top of OpenAI's GPT-3.5 [34], would be used. The group planned to use ChatGPT as a peer reviewer, as it can provide an alternative perspective or correct obvious mistakes.

3.2 Workflow

A scrum-like workflow was decided as the desired workflow for the project since the feature-oriented nature of a scrum work method was perceived as more suitable for this type of project. Scrum emphasizes flexibility, collaboration, and iteration [35]. It involves regularly reassessing and adapting the project plan based on feedback and progress. Work is divided into small, manageable tasks, and progress is tracked through regular check-ins and demos. The focus of Scrum is to deliver a functional, usable product in a short time frame, rather than trying to plan and complete the entire project in advance.

It was decided that the workflow would be integrated by setting up one-week sprints. During these sprint meetings, the progress of the previous week would be assessed, and new tasks would be created and assigned. During the sprint, the plan was for developers to create a new branch in GitHub with the same ID as the task picked in Trello. Upon completion, a pull request for the branch would be created and the task would be moved to a "testing" tab in Trello. Another developer would then review the changes. If the quality standards and performance requirements were met, the task was considered finished and would be merged into the main branch.

After each sprint, the plan was to review and reflect on the work from the earlier sprint. To enhance these sessions, a Key Performance Indicator (KPI) form containing questions regarding how the developers felt about the project will be used. These forms would be completed before each evaluation session and remain anonymous, which would allow for more sensitive topics to be discussed.

3.3 Milestones

To develop the application, the plan would be to divide the project into smaller and easier-to-implement features. This is called the divide-and-conquer method [36]. In addition, it was decided to categorize these features into the levels "must have", "should have" and "could have" based on the MoSCoW model [37]. The model also has a "won't have" level, which was decided not to be used since the limitations cover this. The level "must have" focuses on implementing an MVP and thus this level has the highest priority. The next level "should have" contains non-essential but expected features for the final product. The last level "could have" contains features that have low priority and can be implemented if time allows.

Moreover, these features were planned to be subdivided into four different categories reflecting the different parts of the solar system. These are "Solar system", "Planet generation", "Ecosystem" and "Player Systems". The different goals for the different levels and categories represented can be seen in tables 3.1, 3.2 and 3.3.

Table 3.1: Must have features in the project.

Must have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Generate a sun	Procedural terrain	Creatures	Player who can walk on planets
Generate a couple of planets	Water	Creatures should move, eat, drink	Player can travel via a spaceship
Have stable orbits		Vegetation	

Table 3.2: Should have features in the project.

Should have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Moons	Stones	Predators	EVA in space
Planet rotation about its axis	Visual atmospheres	Temperature based ecosystem	Player GUI
	Temperature	Genes	
		Different creature species	
		Creatures can reproduce and die	

Table 3.3: Could have features in the project.

Could have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Asteroids	Clouds	Sea creatures	Terraforming
N-body system	Gasplanets	Creature evolution	Save system
	Caves	Procedurally generated looks for creatures	A solar system map

3.4 Planning

Prior to development, a plan for the project was determined considering the method, tools and features discussed earlier. This section highlights large decisions concerning general planning in conjunction with specific planning for individual parts of the project.

Firstly, a number of limitations were planned to restrict the scope of the project. With these in place, it was decided that further limitations could be added throughout the development process. The project then had a selection of initial limitations while remaining flexible for future directions of the project.

Secondly, regarding feature prioritization, the goal was to create a minimal viable product (MVP) and complete the majority of the must-have features in a timely manner. The philosophy was to quickly build a foundation where features could easily be added without waiting for other features to be completed. Therefore, the plan involved completing these within three sprint weeks and moving on to should-have features after the MVP is done.

Lastly, during the initial sprint planning, the decision was made to distribute work so the majority of the work could be done in parallel. This was planned to be done by creating independent tasks and providing each developer with their own workspace in Unity.

3.4.1 Time plan

To help keep track of the project's progress, a time plan was created, depicted in figure 3.1, based on the MoSCoW principle mentioned in section 3.3. It allowed for an overview over when specific parts would begin and be completed. As a general approach, the aim was to write the final report in parallel with developing the product. To structure the coding, three coding tasks and one finalizing task were planned.

The first coding task focused on producing an MVP with all necessary features present. The second coding task aimed to extend the MVP with additional features to enhance the product further. The third coding task intended to add optional features. Lastly, the finalizing task aimed to refine the product and prepare it for the final presentation.

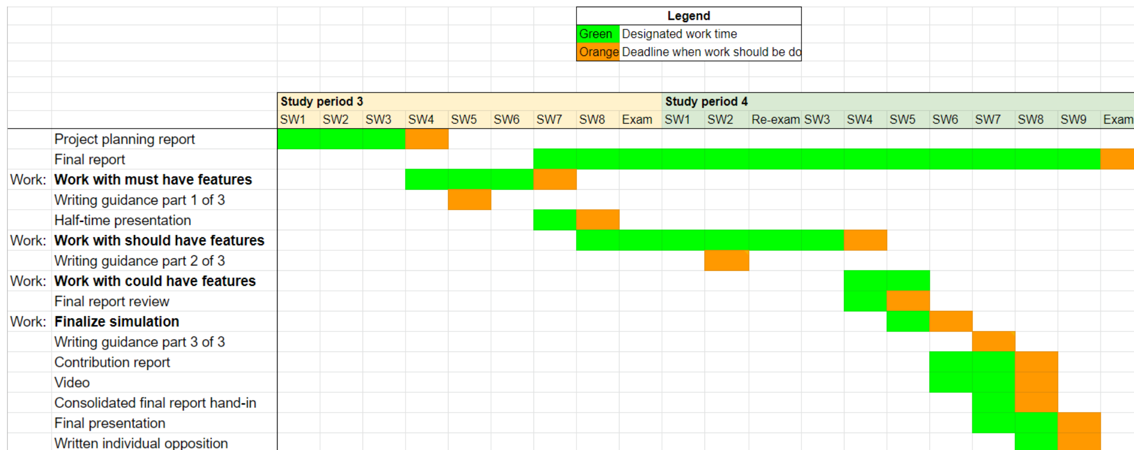


Figure 3.1: Time plan for the project.

3.4.2 Solar system

When planning a predictable solar system the choice between a 2-body or n-body system needed to be made. The argument for an n-body system is that it provides a more realistic simulation as it accounts for all gravitational forces rather than just one, as in the 2-body system [38]. This results in accurate planet simulation but becomes unstable after a long time according to the three-body problem [39]. Therefore, the focus lied on creating a 2-body simulation with less accuracy but with predictability contrary to an n-body system. Further research and implementation of the n-body system could have been done if time allowed.

Additionally, a decision was made to make the simulation reproducible by making it deterministic. This works by utilizing a controlled and seedable random generator to generate the same sequence of outputs when using the same seed.

To limit the scope of the solar system, evolution would not be simulated. This allows for more predictable and stable simulations at the expense of limiting the long-term realism of the system. Additionally, this means that the life cycles of the planets would not be simulated. This includes planetary events such as glaciation, tidal and volcanic cycles. These mechanics would be difficult to implement and hard to simulate due to the many interactions between different entities in the solar system.

3.4.3 Planet generation

The different terrain generation techniques in section 2.3 were researched and discussed. Ultimately, the Marching Cubes technique was chosen as complex 3D features such as caves would be easier to implement compared to heightmaps.

After Marching Cubes was decided, an art style had to be selected. A high-poly style gives a highly detailed terrain while a low-poly style gives a minimalistic look to the terrain. A low-poly art style was determined appropriate due to the Marching Cubes algorithm's ability to easily generate terrain in that style. The resolution needed for a high-polygon style would scale badly with the algorithm, decreasing performance drastically.

In addition to terrain, it was decided that the planets should have biomes. The plan was to divide the surface of each planet, creating different areas with various colors and climate changes similar to Earth. These biomes were planned to affect the spawning of vegetation and creatures depending on temperature or type of terrain.

3.4.4 Ecosystem

The next step involved planning how ecosystems would function and interact with the planets. The goal for this part of the simulation was to provide planets with life such as foliage and creatures without it being too complex.

To limit the ecosystems on the planets, it was decided to not implement a full-scale realistic ecosystem on the planets. This would be too difficult to create due to the large number of external factors that affect ecosystems such as temperature, climate, and pollution.

Furthermore, intelligent civilizations and civilian-made structures such as cities or road infrastructure would not be implemented. The creation of such civilizations requires thorough research and was not feasible to produce in the time frame. Instead, simple ecosystems composed of a few species would be constructed. In these ecosystems, the species' goal was to meet their hunger and water needs. Additional functionality to the ecosystem could be added if time allowed.

3.4.5 Player system

The player system was defined as all the elements a user can interact with in the simulation, enabling exploration of the solar system. The plan was to create a spaceship in which the user could control and explore the simulation by flying on and between planets. On each planet, the player would be able to walk around on the ground and jump. This would make the experience feel more immersive rather than a flying camera exploring the simulation.

3.5 Societal and ethical aspects

During the planning phase, two important societal and ethical aspects were identified to keep in mind during the project.

3.5.1 Education

An application exploring space and the life within it may convey educational information about astronomy and ecology. This offers an opportunity to educate users about astronomy and ecology. On the contrary, it may also convey misinformation to the user if the information presented is incorrect. The information a user learns by interacting with such an application will be a part of their informal education and contribute to the understanding of astronomy and ecology.

3.5.2 Automation of content creation

As mentioned in section 2.1, it was decided that the project would make use of PCG for the generation of celestial bodies and creatures. This creation would otherwise be done by an artist and brings to question if advancements in PCG will create unemployment. A similar premise has already been presented in the early 2000s as the traditional 2D animation industry made way for more 3D animated movies [40]. Jobs such as colorist and tracer were made redundant as computers could do the same job in 3D by calculating what model was seen by the camera.

4
Process

In this chapter, the development process will be covered. The arrangement follows one large system at a time with the included features inside it. The details about each feature will be described in chronological order.

4.1 Solar system

This section will describe the solar system generation. This includes orbits of planets, as well as how the sunlight and ambient light in the scene function.

4.1.1 Planet orbits

To implement orbits, placeholder planets were first placed in a circular pattern around a sun. Similarly, moons were also placed around a few planets. To have planets orbit around the sun, and moons around planets, the sourced asset *SimpleKeplerOrbits* [41] was used to make them orbit their individual attracting bodies on a common axis, thus achieving initial orbital movement. Figure 4.1 shows the early stage of the generated solar system with the planet's and moon's orbits.

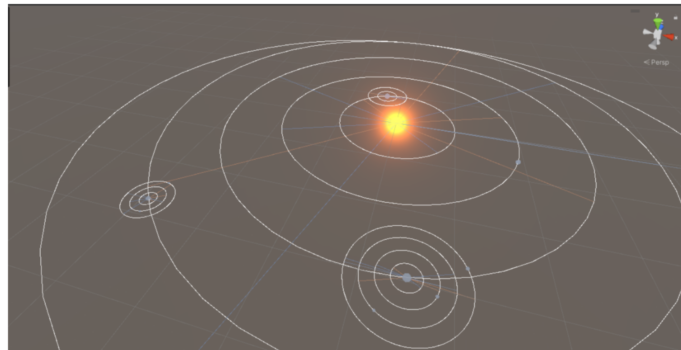


Figure 4.1: Early stage of the solar system with displayed orbits.

During the implementation of vegetation, a significant performance decrease was observed when moving a planet with large numbers of objects. To address this issue, it was decided to implement a geocentric model, resulting in the player's current planet being stationary at the center of the solar system while the sun and other planets orbit around it [42]. When the player moved away from the planet, the system would default back to the sun at the center. This approach significantly

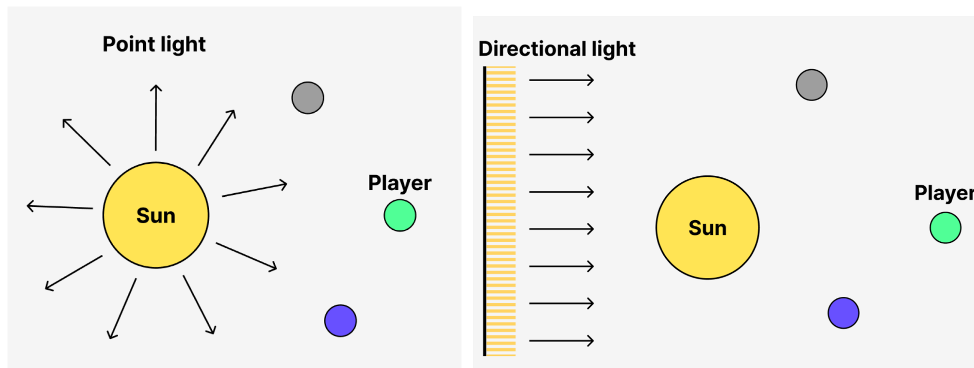
improved the performance while still looking accurate from the player's perspective, as the orbit of the current planet was faked.

Once orbits around the sun were set up, the next step was to rotate the planets around their own axis. Given that the high number of objects on the player's planet would be affected by this motion, it was necessary to fake this rotation as well. The first step was to rotate each planet except for the player's planet around their own axis, and this was done successfully. The next step was to fake the rotation of the player's planet by rotating the whole solar system and the skybox around that planet's axis.

During this step, it became clear that the *SimpleKeplerOrbits* asset had several flaws, making the implementation challenging. Despite numerous attempts to get past them, it was ultimately decided to stop using the asset and implement the orbits manually as more problems came up. Much of the existing code was however reusable, as the asset was mostly used to orbit the planets around the common axis. This resulted in a simpler implementation of orbits and rotations while faking them on the player's current planet.

4.1.2 Directional sunlight

To make the solar system feel more dynamic, the sun needed to cast sunlight on the planets. It was also decided that solar eclipses should occur when another planet moves between the player and the sun.



(a) Unused point light idea which shined light in all directions.

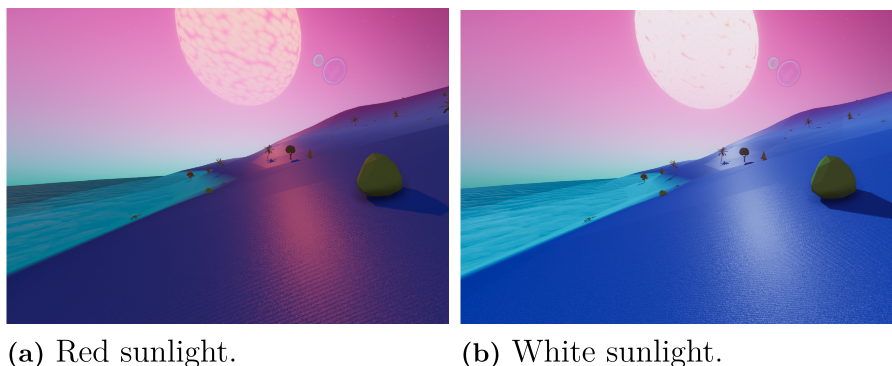
(b) Used directional light idea which shines light from one direction.

Figure 4.2: The two different main lightning ideas considered and used.

The first version was to put a point light that shines in all directions in the center of the sun. However, using the standard Unity light proved problematic as it required an extremely high light strength to illuminate the distant planets. This resulted in the closest planets being overly bright and unplayable. In trying to solve this, the light falloff distance was changed, which gave better results, but was eventually dropped in favor of a system using the directional light in Unity. These solutions

are illustrated in figure 4.2a and 4.2b.

The directional light was chosen as a substitute for the inefficiencies of the point light solution. The system fakes sunlight by rotating a directional light to point from the sun towards the player at all times. Furthermore, shadow casting from the sun was turned off, allowing directional light to shine through it. This enabled shadows and solar eclipses but had the drawback of having minor lighting issues on other planets. Figure 4.3a and 4.3b depicts the final version of sunlight with color based on sun temperature.



(a) Red sunlight. **(b)** White sunlight.

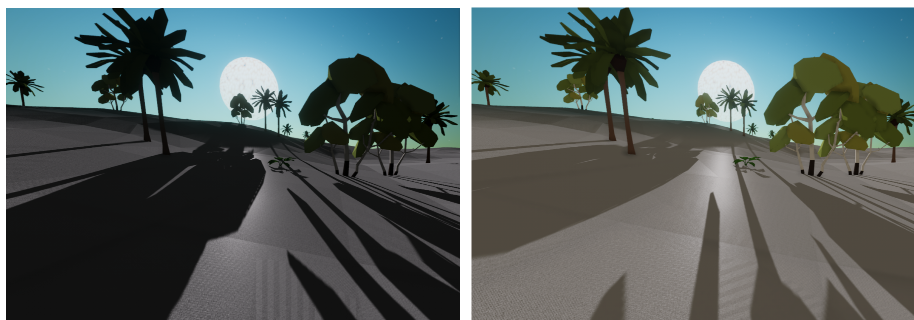
Figure 4.3: The final sunlight solution shown in two colors based on the sun's temperature.

4.1.3 Ambient light

After the sunlight had been added, it was noted that the lighting was not satisfactory since the shadows were too dark during the day, seen in figure 4.4a. This was caused by the directional light only being able to light the side facing the sun while the other side remained dark. To address this, an ambient light was added to light up the entire world uniformly. Figure 4.4b illustrates the System with the ambient light.

With the ambient light added, there was now a need to be able to control it since it became too bright during nighttime on the planets. A system to locate the player on the side towards the sun, shown in figure 4.5, was developed. The calculated position is then used to sample a color gradient which determines the brightness of the ambient light. This system allows for daylight, sunrise/sunset and night.

After the atmospheres were implemented, the ambient light system was reworked. Instead of being calculated globally using only one color gradient, each different atmosphere now got its own gradient. This enhances player immersion by simulating the atmosphere's tinting of light as it passes through it. During this rework, it was also decided to fade to the nighttime ambient light when entering space. Previously the system did not care for this which caused harsh jumps in ambient light.



(a) Game view with no ambient light.

(b) Game view with added ambient light.

Figure 4.4: A comparison between not having and having ambient light when the application is running.

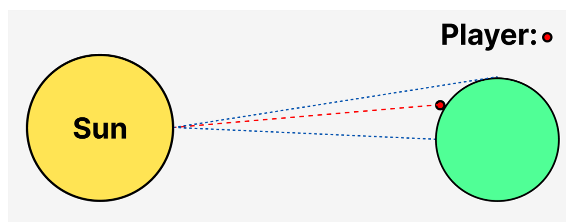


Figure 4.5: The different distances the ambient system uses to calculate the light intensity.

4.2 Planet Generation

This section explains the development process of how planets are generated. This includes planet terrain and coloring using a set of rules.

4.2.1 Biomes

Biomes were implemented with three aspects in mind, mountains, temperature and forests. The mountain aspect was created first since the mountains were determined to not depend on any of the other aspects, this was done using simplex noise discussed in section 2.2.

Secondly, the temperature aspect was implemented. The poles were made colder than the equator, and the temperature was affected by mountains and the distance to the sun. The equation 4.1 was used to determine the overall temperature of the planet depending on the distance to the sun.

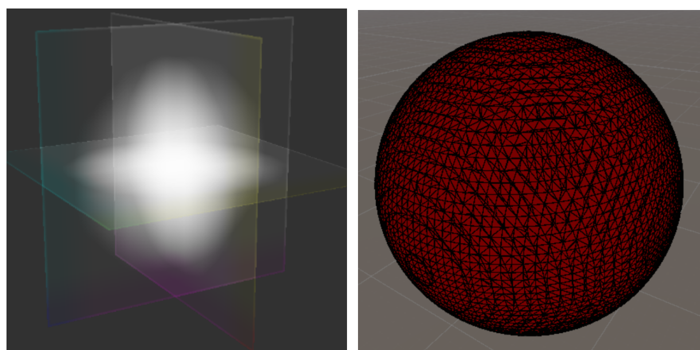
$$Temperature = \frac{1 - FarTemperature}{decay^2 * distance + 1} + FarTemperature \tag{4.1}$$

The last aspect implemented was the forests, this was required to distinguish deserts from rainforests as they would otherwise have the exact same conditions. The forest aspect was also implemented using simplex noise.

4.2.2 Terrain generation

As mentioned in section 3.4.3, the Marching Cubes algorithm was chosen as the terrain generation algorithm, thus the first step was to create this algorithm. To assist with the implementation, a sample texture was created which represented a sphere. The texture for the sphere can be seen in figure 4.6a.

After creating the texture, the implementation of the Marching Cubes algorithm began. A compute shader was developed to run the algorithm as GPU computing, mentioned in section 2.6, could be used for its implementation. The shader was inspired by a C++ implementation by Paul Bourke [43]. Moreover, a C# script to manage the compute shader was created. The results of the working algorithm can be seen in figure 4.6b.

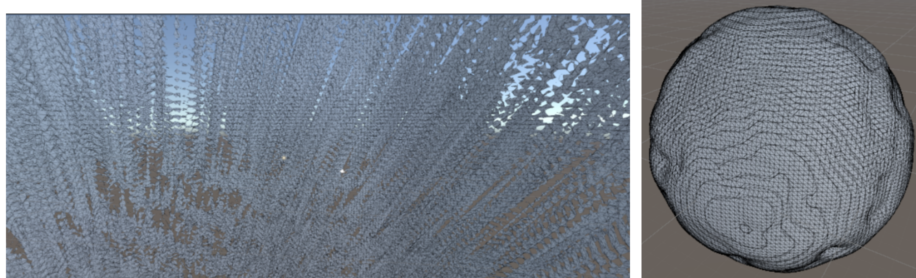


(a) Texture for sphere. **(b)** Marching Cubes generated sphere.

Figure 4.6: Sphere texture and the resulting body generated by the Marching Cubes algorithm.

The next step was to create terrain on the planets. This is commonly done using gradient noise, such as Perlin or Simplex noise, for the underlying algorithm [44]. Perlin noise, mentioned in section 2.2, was used as it was easier to implement. The first approach sampled noise at different integer values, which in turn led to a non-continuous surface. The result of this can be seen in figure 4.7a.

As can be seen, the first approach was a failure and therefore had to be reconsidered. To solve the issue, the noise was instead sampled using normalized points. The algorithm also had to generate a sphere, thus, the algorithm had to rely on the equation for a sphere in some way. To achieve this, the noise was weighted more towards the center than on the edges. The result of the second attempt can be seen in figure 4.7b.



(a) First method used to generate terrain, which did not work.

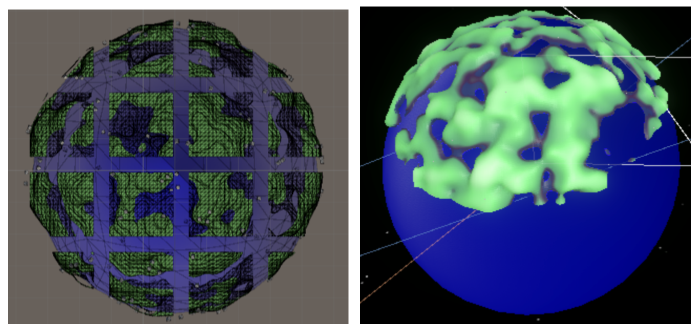
(b) Second method used which resulted in terrain.

Figure 4.7: First and second attempt of terrain generation.

Overall this method succeeded in making the terrain continuous and plausible. However, the performance of the planets was not great, since all planets were fully rendered. To improve the performance, it was decided that the chunking method mentioned in section 2.7 would be implemented.

To understand the implementation, the Marching Cubes algorithm has to be explained. The algorithm represents the planet as one large cube divided into smaller cubes, resulting in a grid of cubes. It was therefore decided that a collection of neighboring cubes would represent a chunk. The resulting chunk cubes can be seen in figure 4.8a.

Initially, the new chunks were created such that chunks below a certain point on the current planet would be disabled when not visible by the player. Furthermore, the planets also had different amounts of chunks, this was done by having smaller, but more chunks on the current planet and fewer, but larger chunks on other planets. Figure 4.8b displays the chunks below a certain point being cutaway.



(a) The individual chunks of the planets.

(b) Chunks being culled below a point.

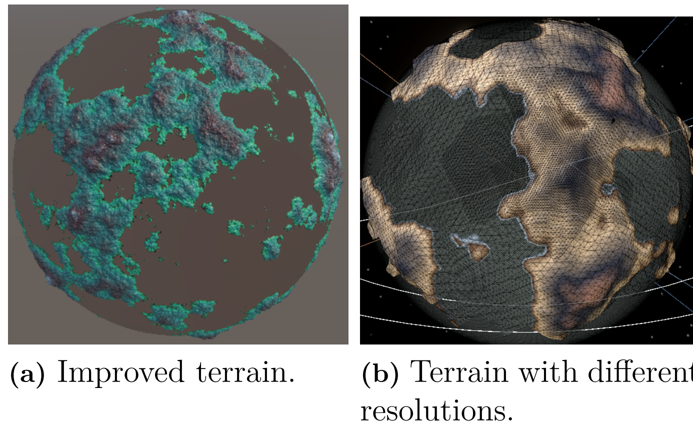
Figure 4.8: Planet chunks and chunk culling.

The first terrain algorithm described did not create interesting terrain, thus, the

next step was to improve it. As most terrain generation methods are based on the heigthmap technique mentioned in section 2.3, the algorithm mentioned in appendix section A.1 was used to use these methods with the Marching Cubes algorithm. To start only the method of adding multiple layers of noise was used [45], which resulted in more realistic looking terrain features depicted in picture 4.9a. Moreover the noise algorithm was changed from Perlin to Simplex-noise because of the improvements mentioned in section 2.2.

After improving the terrain, performance issues were addressed concerning lag spikes when the player entered and left the planet. This was caused by the chunk resolution changing and the chunks being rendered at the highest resolution. An attempted solution took inspiration from Astroneer, mentioned in section 2.8, which implements dynamic chunk resolutions. This improved the overall performance by decreasing the resolution of chunks further away from the player while keeping it higher on those closer. However, this did not solve the lag spikes, but it was still kept.

To further attempt to solve the lag spike, two other solutions were tested, one of which loaded in chunks over time when entering the planet. Unfortunately, this did not solve the lag spikes and caused visible chunk loading. The second solution was to generate all the planets in both high and low chunk resolution during loading and then disable/enable the chunks when entering/leaving the planet, which solved the lag spike issue. One drawback with this method was the higher RAM usage when running, and longer application loading times. Picture 4.9b depicts a planet with dynamic chunk resolution implemented.



(a) Improved terrain. **(b)** Terrain with different resolutions.

Figure 4.9: Terrain improvements and optimizations.

The planets currently had more realistic terrain features as mentioned earlier, however, there were no refined mountains. Therefore, research was conducted on how they could be refined. It was discovered that if the sampled noise values were raised to a power, the terrain would become peakier and thus achieve mountains [45]. However, this method lacked control over where the mountains should be placed. Thus, the function had to be modified to follow the mountain aspect of the biomes mentioned in section 4.2.1. The equation 4.2 is the modified function for the mountains.

$$mountainGround = noise^{(1+biomemap^2)*mountainPeakness} * biomemap$$

$$flatGround = noise^{smoothness}$$

$$ground = mountainGround * mountainHeigth + flatGround \tag{4.2}$$

In addition to following the mountain aspect of the biomes, the equation 4.2 both gave control over the height and peakiness of the mountains, as well as the flatness of the ground where no mountains exist. The equation in its essence creates mountains and flat ground and then blends these together. The resulting terrain can be seen in figure 4.10.

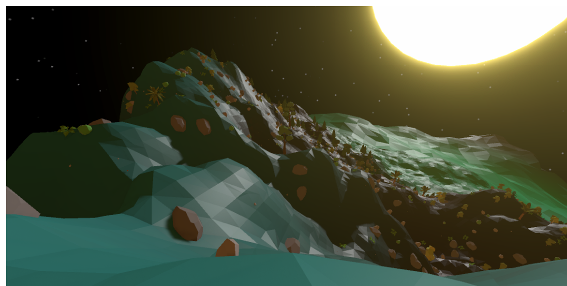


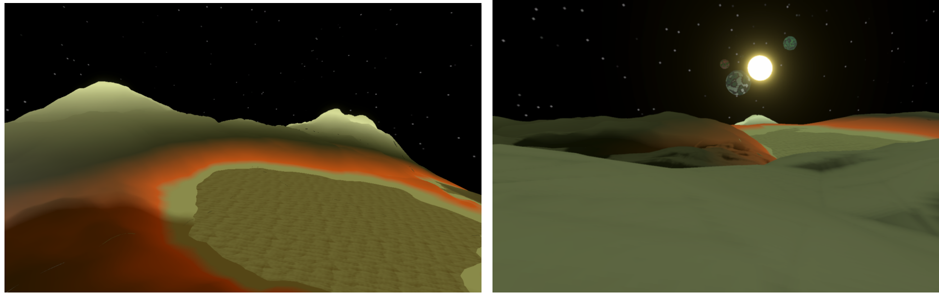
Figure 4.10: An image of the mountains generated with the new mountain function.

4.2.3 Terrain coloring

It was known that the generated planet terrain needed to have some sort of color or texture other than the standard white color seen in figure 4.7b. A Unity material needs to be applied to the terrain, which tells the program how to draw the object. To construct the terrain material, a shader, mentioned in section 2.6, had to be developed.

When it came to what aspects would decide the color of the terrain, it was determined that altitude and vertical angle would be used as starting points. These factors were chosen loosely based on reality. At higher altitudes, it often gets more mountainous with the possibility of snow compared to lower altitudes. Furthermore, steep cliffs do not let vegetation grow, causing them to only have a rock color. These factors were also deemed to be quickly doable which was another benefit in the project's beginning.

The colors worked as intended, as figure 4.11a and 4.11b illustrate. However, the result appeared flat, which was not the intended goal. After researching, it was found that to achieve the "low-poly" art style decided in section 3.4.3, the terrain needed to have an assigned normal vector to each polygon to determine how the light

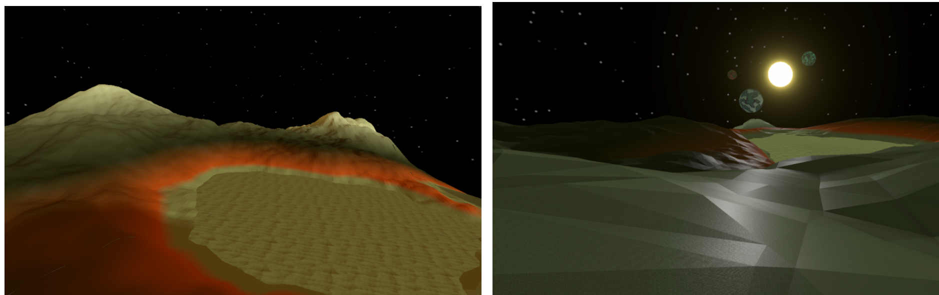


(a) Overview of distant mountains using flat terrain shader.

(b) Close-up of terrain with sunlight using flat terrain shader.

Figure 4.11: The flat-looking terrain shader before assigning a normal angle.

should bounce on the terrain. The resulting terrain shader shown in figure 4.12a and 4.12b, appeared to have the intended depth wanted, which therefore became the first version before implementing biomes.



(a) Overview of distant mountains with modified shader.

(b) Close-up of terrain with sunlight with modified shader.

Figure 4.12: The first complete version of the terrain shader.

To set different colors on planets, a script generating a gradient for each planet was used. The gradient represented the color from low to high altitude of the terrain. This allows the gradient to be interpreted by the shader to map it from the water level to the highest point on the terrain. The different colors for the gradient were at this stage simply colors generated by a palette generator found online [46].

With the addition of biomes, the terrain coloring needed an update to color each aspect of the biomes correctly. Firstly, biome aspects such as temperature, forests and mountains were extracted, and then made more prominent, which can be seen in figure 4.13. The next step was to color the planet using the aspects, this was done by blending a base color for the planet with biome-specific color gradients, such that black spots in figure 4.13 would have the biome-specific color and the white spots would have the base color.

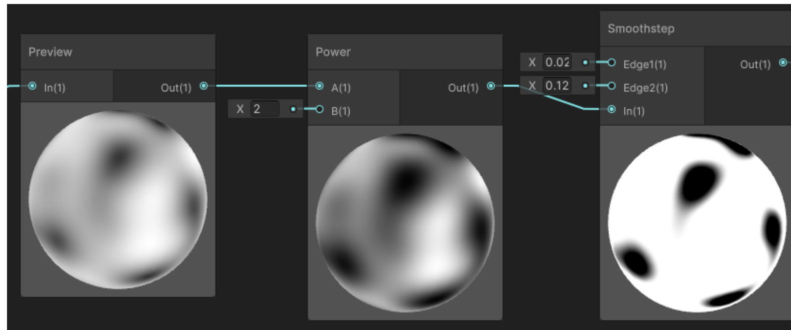
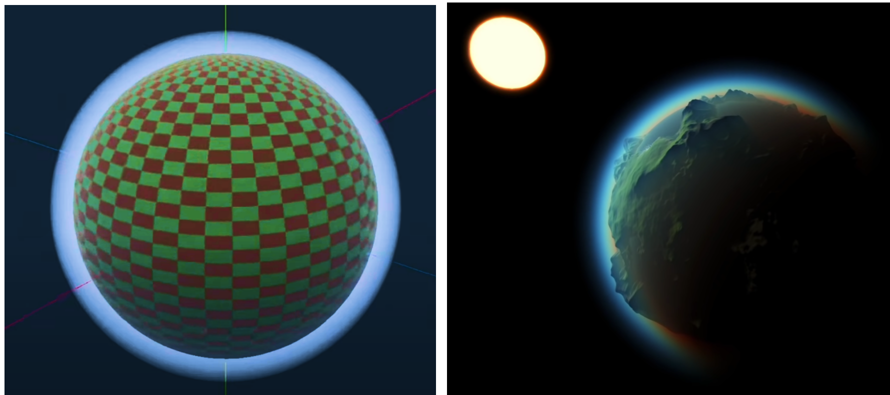


Figure 4.13: Part of the terrain shader implementation amplifying noise using "power" and "smoothstep" functions. The amplification is done by a power node that uses the formula $value = value^2$ for each value. The smoothstep node filters the output from the power node to make it suitable for linear interpolation. These functions make the aspect more prominent.

4.2.4 Planetary atmospheres

One important "should have" feature was to implement planetary atmospheres. It was decided that no air resistance would be accounted for and that atmospheres would only be visual. The atmospheres needed to have different colors to simulate different types of atmospheres. Additionally, it also had to accommodate different planet sizes, heights, and densities. After these requirements had been set up, different solutions started to be considered.



(a) Screenshot of a simple "bubble" atmosphere created by Martin Donald [47].

(b) Screenshot of an accurate atmosphere created by Sebastian Lague [48].

Figure 4.14: Two versions of an atmosphere shader.

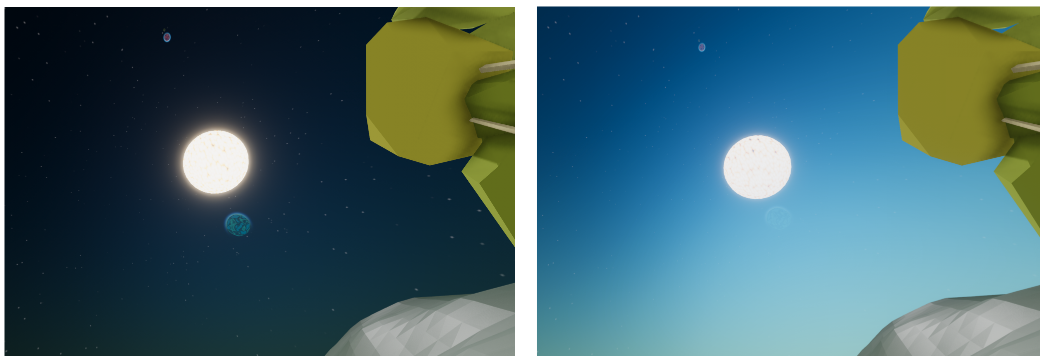
Out of these solutions, two different types of atmospheres emerged. The first was a simple sphere around the planet, shown in figure 4.14a. The second was a more accurate atmosphere, seen in figure 4.14b, which makes use of Rayleigh and Mie scattering [49]. This atmosphere, although harder to implement, was chosen to be implemented due to its larger feature set. This included control over where the light

hits the atmosphere and different light scatterings for individual wavelengths.

To implement the second method, research showed that a sphere intersect method had to be implemented for marching rays through the atmosphere [49]. These rays sample the color value at a fixed interval towards the sun and then add them together to form the final pixel color. Additionally, during the research, a few pre-existing free solutions were found. These solutions could speed up the atmosphere implementation, which outweighed the downside of not having a custom-made solution.

The first pre-made atmosphere solution tested was a post-processing effect created by Sebastian Lague for his "Geographical Adventures" game [50]. However, during testing, problems emerged as the post-processing effect did not work with the game's existing skybox, causing the starry background to disappear. The solution would require replicating Lague's star post-processing system. Moreover, this system did not support multiple planets, a requirement for the project, so it was deemed unsuitable for use.

In contrary, the second pre-made solution used an easier-to-implement shader found on GitHub gists [51]. The atmosphere shader uses the mentioned intersect sphere method and combines light during steps to form the output but does not feature optimizations found in Lague's solution. Another disadvantage compared to Lague's solution is the absence of fog and atmospheric haze on planets. Although there were features missing, it was decided that this shader would be used as it still met all initial requirements.



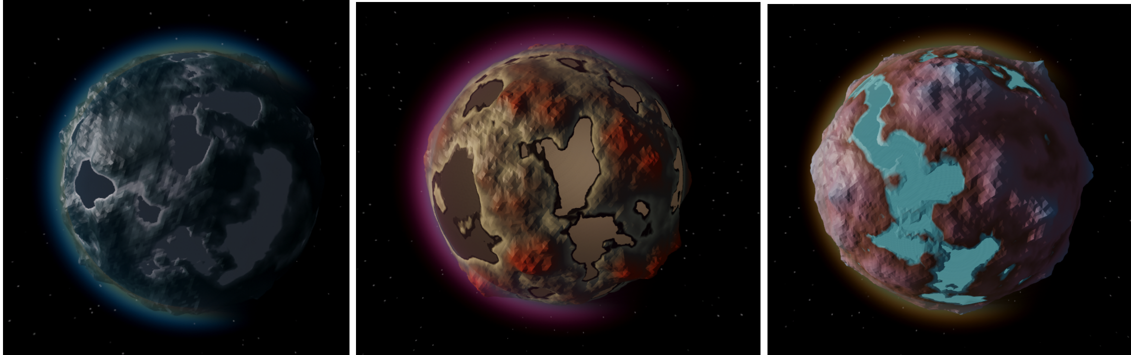
(a) Atmosphere without increased light intensity.

(b) Atmosphere with increased light intensity.

Figure 4.15: Difference between light intensities of the atmospheres.

Throughout implementation, a few unexpected problems with the shader occurred. Firstly, the strength of the atmosphere was too weak close to the ground. This resulted in the player not being able to see enough atmosphere, as shown in figure 4.15a. To solve this, the light intensity had to be increased while on the planet to simulate a thicker atmosphere and decreased when in space. This solution is depicted in figure 4.15b. Secondly, there were strange flashes coming from the at-

mospheres. This was caused by the shader returning the pixel color value Not a Number (NaN). This was fixed inside the shader code to not give out NaN values.



(a) Earth-like atmosphere. **(b)** Red-pink atmosphere. **(c)** Mars-like atmosphere.

Figure 4.16: Three possible different looking planet atmospheres.

The atmosphere's color, intensity and height are decided at random from a list of allowed values. This is then used to create 252 different possible planet atmospheres. The different color, intensity and height options were created manually by adjusting the parameters. The results can be seen in figure 4.16 where the three planets show different variations of the atmosphere.

4.3 Ecosystem

This section explains the development procedure for elements on planets, including creatures, foliage and water.

4.3.1 Creatures

The first step to developing creatures was to develop a state machine that controls the states of the creature. To do this, four states were created. The first state, "walking", enables creatures to choose a random point on a circle and walk to that point. The second state "looking for food" indicates that the creature searches and approaches food based on its diet either being herbivore or carnivore. The third state "looking for partner", represents if the creature is searching for a partner, which can happen if its other needs have been satisfied. The last state, "breeding" indicates that the creatures are currently creating offspring, the breeding state is entered if a partner is close and they are looking for a partner.

To spawn creatures, a spawning mechanism had to be implemented. The first iteration used raycasts from a random point on a sphere to the planet center. When the raycast hit the planet, it spawned a creature at that position. Raycasts were chosen due to their good performance and ease of use. Although, one downside is

that creatures can not spawn under possible overhangs or caves. This is because raycasts only return the first point they hit.

Building upon this concept, pack spawning was then implemented where multiple creatures spawn within a close radius to the first creature, seen in figure 4.17. To ensure spawning points were valid, the angle of the terrain could not be too steep and the point could not be on top of other foliage or creatures. In that case, the creature would not be spawned at that point.

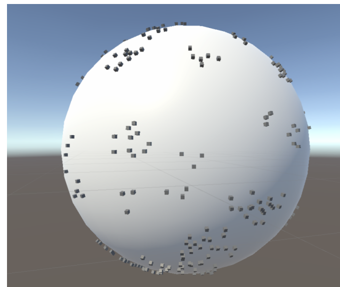


Figure 4.17: Pack spawning.

The transition to chunk-based spawning necessitated changes in creature spawning. The changes were such that chunks would now perform their own spawning when loaded. This resulted in the spawning system only running when the chunk loads the first time. The spawned creatures could then be activated/deactivated together with the chunks. This allowed already spawned creatures to be reused, improving performance.

To create diversity in ecosystems, different kinds of creatures were introduced. The first type was a herbivore and was represented by a hen. It can only eat small plants and bushes as seen in figure 4.18a. The second type was a carnivore represented by a fox, displayed in figure 4.18b. Carnivore creatures are able to eat other creatures depending on their size but they will not eat creatures of the same species. Thus preventing self-extinction by not allowing children and parents to eat each other.

(a) Hen eats a plant.

(b) Fox eats a hen.

Figure 4.18: Different creatures eating.

Reproduction was introduced to create a more dynamic ecosystem. This feature enabled creatures to have offspring when hunger and thirst levels are above a certain

threshold. When trying to bread, creatures search for an available partner of the same species and pathfind to each other. Upon arrival, the creatures have a random chance of spawning an offspring. To prevent overpopulation, there is a limit to the total number of offspring a creature can have.

To ensure that creatures met the performance criteria, a few optimizations were needed. Firstly, it was needed to optimize the algorithm for finding resources. The old algorithm worked by looping over every nearby object, which was very memory and time-consuming. This was improved by filtering out irrelevant resources like trees or stones and reducing the call frequency of this method, resulting in less memory allocation and execution time. Another change was to stop executing code and disable the physics component on non-visible creatures. This drastically improved performance as only around 20 creatures were active at the same time depending on the creature density.

The last step was to further increase creature diversity. This was done by expanding the number of unique creatures in the simulation using a creature asset pack [52] and giving each two colors options. The creatures were added to the current system with minor tweaks to their behavior and spawning rules. These include defining where each creature could spawn based on the biome aspects from section 4.2.1.

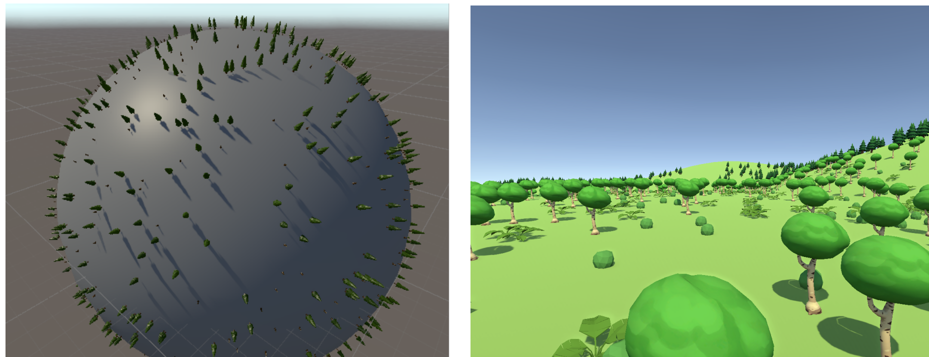
4.3.2 Foliage

Foliage is an essential component of a game's environment as it adds personality and life to the virtual world. Development began by searching for suitable assets to use as trees, bushes, and other foliage items. Careful consideration was used to find assets that fit the low-poly style decided earlier 3.4.3.

After suitable assets were found, the foliage needed to be placed on the planets. The first iteration used a built-in function to place foliage on a sphere, this can be seen in figure 4.19a. However, this method was not feasible since it did not consider the terrain of the planets. The second method therefore involved using raycasts to place foliage objects on the surface of the planet similar to the first iteration of creature spawning discussed in section 4.3.1. The success of this system led to additional rules being defined such as the density, variety and placement of foliage. The results can be seen in figure 4.19b.

Performance issues arose when placing a large number of foliage objects. To address these, several optimization techniques were tried, such as Level of Detail (LOD), static batching, occlusion culling, GPU instancing and instance staggering. Both occlusion culling and GPU instancing had too much overhead for this current application, resulting in a performance loss. However, LODs, static batching and instance staggering had large performance improvements.

When chunk spawning was introduced, the foliage spawning algorithm had to be reworked to utilize chunks and improve performance. The new algorithm needed to



(a) First iteration of foliage on a unit sphere.

(b) Second implementation of foliage using raycasts.

Figure 4.19: First and second implementation of foliage side by side.

be deterministic and operate independently within each chunk. A new method was explored during the rework which used known locations on the chunk terrain. This method had numerous advantages but did not comply with the requirement, since the positions could not be generated deterministically.

The project ultimately settled on using the previously used raycast approach. The difference this time was that the rays had to be cast within each chunk. This was achieved by spawning a circle that would cast the rays towards the chunk, illustrated in figure 4.20b. The method was chosen because it allowed for simple placement of trees without considering the orientation of the chunk. The new system was divided into a handler and a spawner component, to keep track of parameters and foliage objects. The current version of the foliage system can be seen in figure 4.20a.

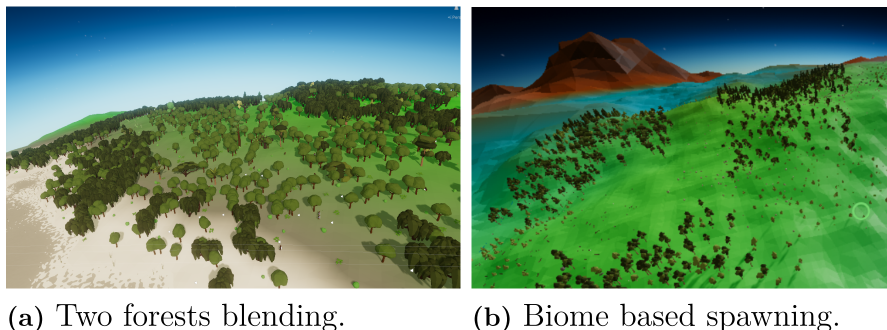
(a) Final implementation after chunks was added.

(b) Showing the circle that is used for spawning foliage.

Figure 4.20: Final implementation of foliage and the circle that represents the area which is used for spawning foliage.

The chunk overhaul also added underwater plants, forests and increased density customization. Tree types in forests were decided using Simplex noise, the result can be seen in figure 4.21a. Moreover, the overhaul allowed for better control over foliage density to ensure consistency across different planets. Prior to this, planets could vary greatly in terms of foliage density, with some lacking trees and others

being densely packed. This iteration made it considerably more adjustable.

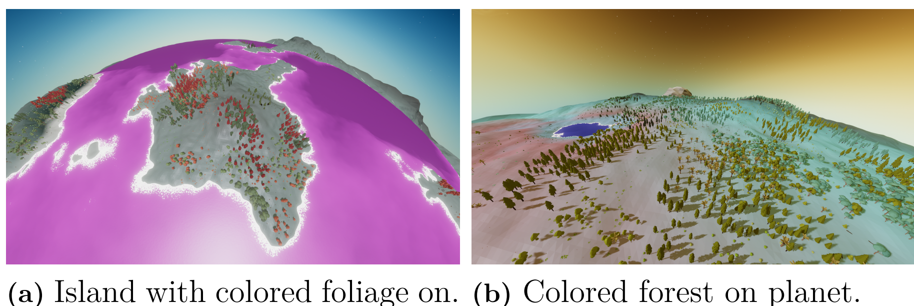


(a) Two forests blending. (b) Biome based spawning.

Figure 4.21: Two figures illustrating forests and biome based spawning.

The next step of foliage included spawning foliage based on biome aspects. This was implemented similarly to the biome-based creature spawning where each object had rules to define where it could spawn. To eliminate the need for defining rules for each foliage object, a collection was created to contain multiple foliage objects with the same spawning rules. When a foliage object then needed to be spawned, all acceptable collections were included in the pool of available foliage objects to randomly select from, the result can be seen in 4.21b.

The last step modified the color of the foliage to be based on the biome it is placed in. When the foliage is placed in a biome, it will be tinted with the color of the ground in that biome as seen in figure 4.22a and 4.22b.



(a) Island with colored foliage on. (b) Colored forest on planet.

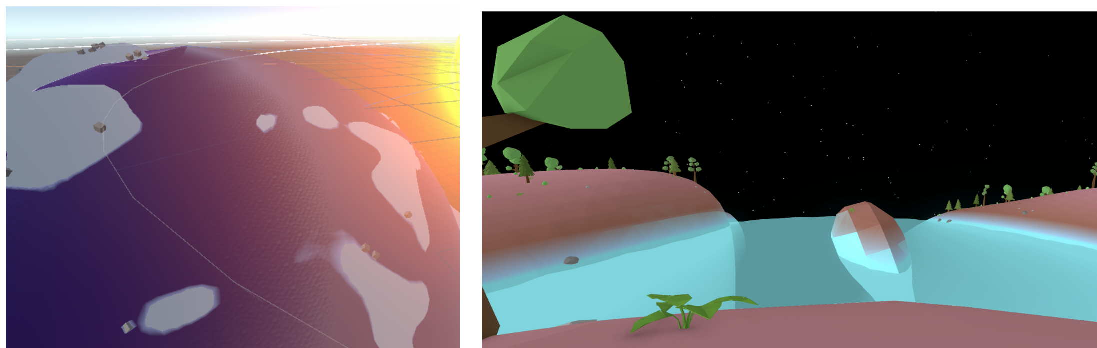
Figure 4.22: Final implementation of foliage after the coloring was added.

4.3.3 Water

Water is a key element of ecosystems as all living organisms need water to survive. The implementation of water is typically done by applying a shader to a flat plane. This would not work with planets as the spherical nature interferes with conventional water techniques. This resulted in a number of challenges, such as accurately rendering the water's surface, simulating water movement, and creating correct reflections and refractions.

Water was originally implemented using a sphere with a simple shader applied. Although simple, this implementation had some limitations. The water was invisible under the surface, resulting in the player not being able to see it along with the inability to swim in it. The implementation of this version can be seen in figure 4.23a.

As development progressed, the water feature underwent a significant overhaul which included a higher resolution sphere, physical waves, underwater visuals and swimming capabilities. The sphere was generated using a compute shader, and the results can be seen in figure 4.23b. Moreover, a normal shader was used to incorporate the physical waves and underwater visuals. It adjusted the player's view to make the water appear darker further away and modified the water's height to account for waves. Additionally, the water surface was flipped so that players could view it from underneath. The last feature added in this iteration was swimming, which necessitated extra player controls to enable swimming capabilities.



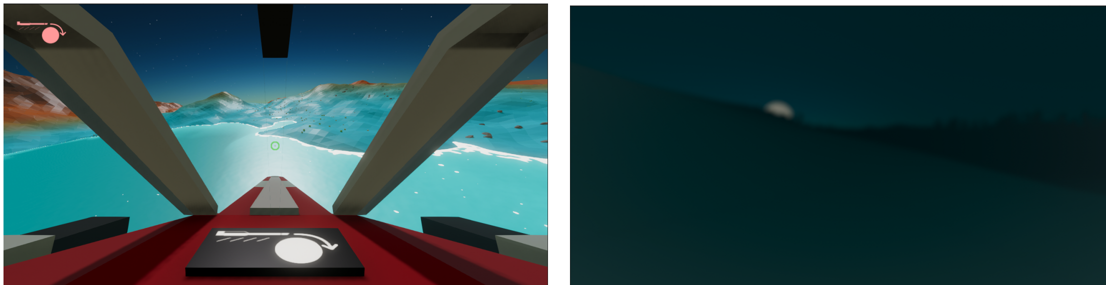
(a) First implementation of water. **(b)** First water update.

Figure 4.23: First and second implementation of water side by side.

Towards the end of the project, a rework was required to reach the standards of other features. This new update fixed major bugs that were left unresolved with the current system. The first bug that was fixed involved the normal map. After the shift to using compute shaders for water, there had been strange lighting issues due to the normal map not being recalculated after it was created. The second bug related to applying a 2D image to a 3D sphere, which led to circular patches and stretched images. This could be resolved using a technique known as Triplanar Mapping [53] but was dropped in favor of using scrolling Worley noise mentioned in section 2.2.

In addition to solving bugs, foam, specular highlights 4.24a and chunking were added to the water. This resulted in a defined water outline, light reflections and a higher water resolution combined with better performance. Specular highlights were implemented through a setting in the shader material. Water chunks were added to increase performance and fidelity of the water. Lastly, the water surface was no longer inverted as it was obstructing the sun and was exchanged for a modification of the underwater shader. The final version of the water can be seen in figure 4.24a

and 4.24b.



(a) Finalized version of the water. (b) Underwater perspective.

Figure 4.24: Final version of water from two perspectives of the same planet.

4.4 Player system

This section explains how the different player systems work in the game. Player systems includes player movement, which model and animation the player character has and the workings of the player spaceship.

4.4.1 Player movement

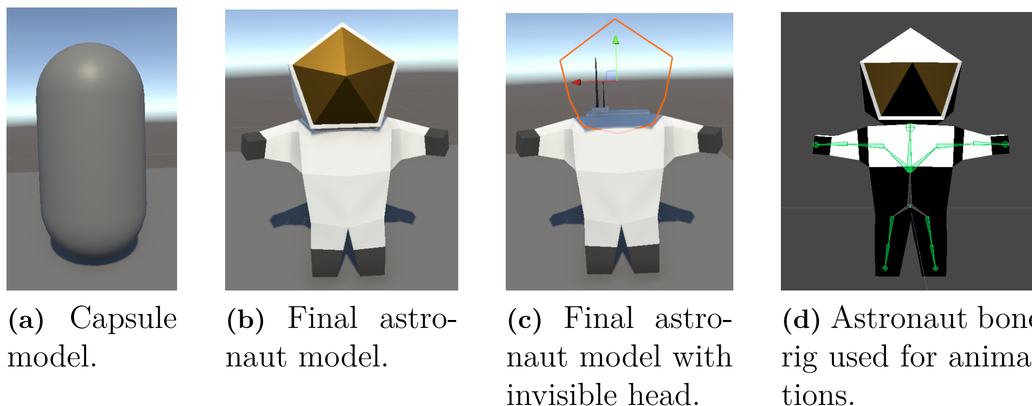
As discussed in section 3.4.5, exploration using a playable character was planned. Three types of movement were considered with different levels of complexity. The simplest version was chosen, which simply added a velocity to the player and rotated them to be upright with respect to the ground. Furthermore, the player had a collider attached to it which prevented it from going through the ground. This resulted in the player being able to levitate around the planet. This left something to be desired as the controls felt too “floaty”.

To address the movement feeling “floaty”, another state was added when grounded. This state set the player velocity to a fixed value in the movement direction instead of continuously adding it to the player. This made movement on the ground more responsive, making the user feel more in control and removing the “floaty” feeling. The old system was still retained when not grounded to offer control while airborne. This meant that jumps could be slightly corrected mid-flight by slightly adding velocity, reducing user irritation.

The next stage better merged the velocity-based movement system with the landscape’s shape. As the player moved across the surface while grounded, its velocity is rotated to be parallel with the ground. This effectively keeps the player grounded if the slope decreases as the forward velocity follows the terrain. As this was being changed multiple additions were made, including randomized spawn points, swimming, and sprinting.

4.4.2 Player model and animations

To enhance the player experience, it was decided to implement a player model. The first version was a gray capsule seen in figure 4.25a. Since the gray capsule was a placeholder, other different models from the Unity asset store [54] were considered to replace it. The astronaut model in figure 4.25b was therefore selected as the model was prepared for animations and suitable for a low poly art style.



(a) Capsule model. **(b)** Final astronaut model. **(c)** Final astronaut model with invisible head. **(d)** Astronaut bone rig used for animations.

Figure 4.25: The different player models used under development from placeholder to final version.

A problem arose from the model switch, where the helmet of the astronaut model was obstructing the player's view. To fix this, the head was separated from the body into two different components. The head component was then set to be invisible, but still allow shadow casting. The result of this can be seen in figure 4.25c.

After the new model had been assigned, the next step was to apply animations. Unity asset store was used to find animations that the model could utilize. Animations for being idle, walking, running, jumping and swimming were added. To apply the animations, bones were manually placed in the model to represent the astronaut's skeleton and then used to control animations. Figure 4.25d displays the final setup of internal character bones.

4.4.3 Spaceship

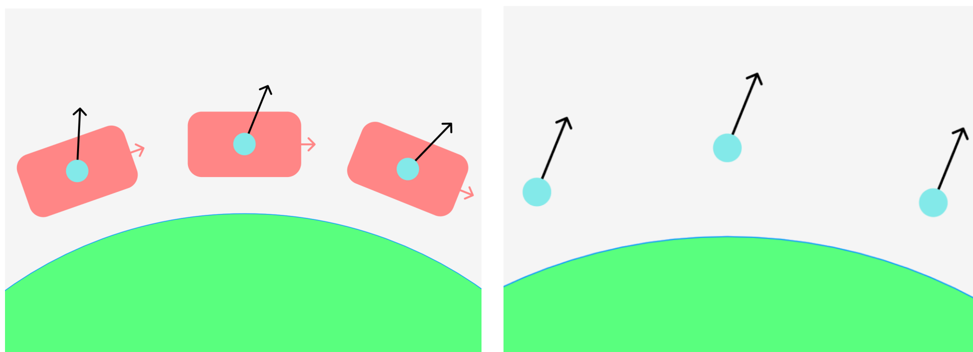
As mentioned previously in section 3.4.5, a spaceship would be created. The first version worked by adding a force to the body such that velocity would build up over time. Furthermore, the ship was also slowed down near planets. This allowed the user to travel between planets.

After this, a landing mechanism was created. The first iterations had problems with clipping through the ground and other inconsistencies. To solve this a collection of raycasts was used to create a virtual ground plane for the ship to land upon. The rotation and landing animation to align with this plane was done through rotation

and vector math together with linear interpolation to create a satisfactory landing.

At this stage, it was possible to travel to and from planets, however, the spaceship was hard to control. To solve this issue, everything except for the landing logic was discarded in favor of a newfound spaceship controller solution found online [55]. Although the solution worked, it needed to be modified to function as wanted. These modifications include changes to basic spaceship controls, new camera-angle options and additional movement modes.

The first modification focused on restoring all the original available movement options. Additionally, a boost button was added that could be used to increase the speed further. The second modification allowed the user to press a button to swap between predetermined camera positions around the ship. The last modification was conducted to address the cumbersome movement experienced in the first version of the ship movement. To overcome this, the rotational component of the spaceship was divided into two parts. One part maintains the rotation of the ship parallel to the planet's surface, while the other part stores the user-inputted rotation. By combining these two components, the spaceship could follow the planet's surface without requiring user input, a movement type therefore referred to as "orbit movement." Additionally, an alternative "straight" movement type was devised, allowing users to disable the orbit movement and rely solely on user input for rotation. These two movement options are displayed in figure 4.26a and 4.26b. These movement options worked as intended after resolving the camera-related issues.



(a) The orbit option's reliance on both the red "normal" rotation and black user rotation.

(b) The straight option's reliance on only user rotation.

Figure 4.26: The two different spaceship movement options and their effect on the final ship rotation.

The finalizing step was to gradually lower the ship to a hover after the player became inactive to introduce more grounded behavior. The lowering part was done by gradually rotating the spaceship upright and adding a force down toward the planet. To hover, virtual springs were created and put around the spaceship to simulate the ship hovering and prevent contact with the ground. The springs include dampening which results in the hover not bouncing when pushed down.

When finished, there was an issue with the ship shaking due to floating point errors at large numbers, this issue was encountered when the player traveled far away from the solar system center. To address this, it was decided to teleport the entire solar system together with the player back before the issues could be noticed. This was implemented seamlessly from the player's perspective and prevented the issue.

4.5 Miscellaneous

The project also consists of features and systems that do not belong specifically to one of the big categories mentioned above. This section will go through those features and systems.

4.5.1 Deterministic randomness

One goal which was decided at the beginning of the project was to generate the solar system deterministically. At first, this was accomplished by having a global number that all random number generators in the system would use as a seed. This was simple but all generators would generate the same sequence of numbers.

Later on, an improved scheme was implemented which consisted of a hierarchy of objects where every parent had to generate the seeds for their children. At the top of this hierarchy is the creator of the solar system, which uses the initial seed for the entire system. This creates a chain of seeds where all generators are unique while still using the same original seed. This separated objects from using the same random generator which allowed for an easier deterministic implementation. The seed chain system is shown in figure 4.27.

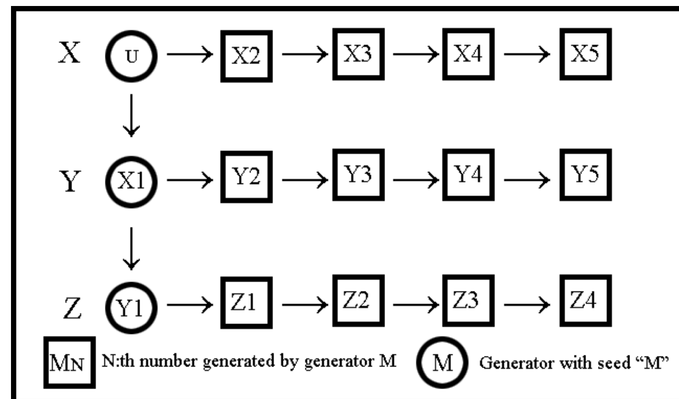


Figure 4.27: The chain of seeds where the generators X, Y, and Z's seed is unique while still being deterministic. U is the universal seed.

4.5.2 User interface

Implementing a user interface helps enhance a player's overall experience by providing control over the game and displaying useful information.Therefore, a start menu, loading screen and pause menu was added to the application. The menus were updated during the development process but retained their original functionality throughout. The menus can be seen in section 5.6.2.

Additionally, a player GUI (Graphical User Interface) was implemented in the later stages of development and included a temperature meter, spaceship speedometer and spaceship movement indicator. The GUI can also be viewed in the section 5.6.2.

5
Results

In this chapter, the results of all features present in the project will be presented. The chapter is structured similarly to the process where grouped features have their own section. In addition, an overview of the final system will be presented.

5.1 Overview of the final system

The final result of the project satisfied all “must have”, and most “should have” milestones mentioned in section 3.3. The solar system has planets with terrain and water, furthermore, they have simulated ecosystems consisting of plants and creatures. The planets also move in orbits and can be explored using the astronaut and spaceship.

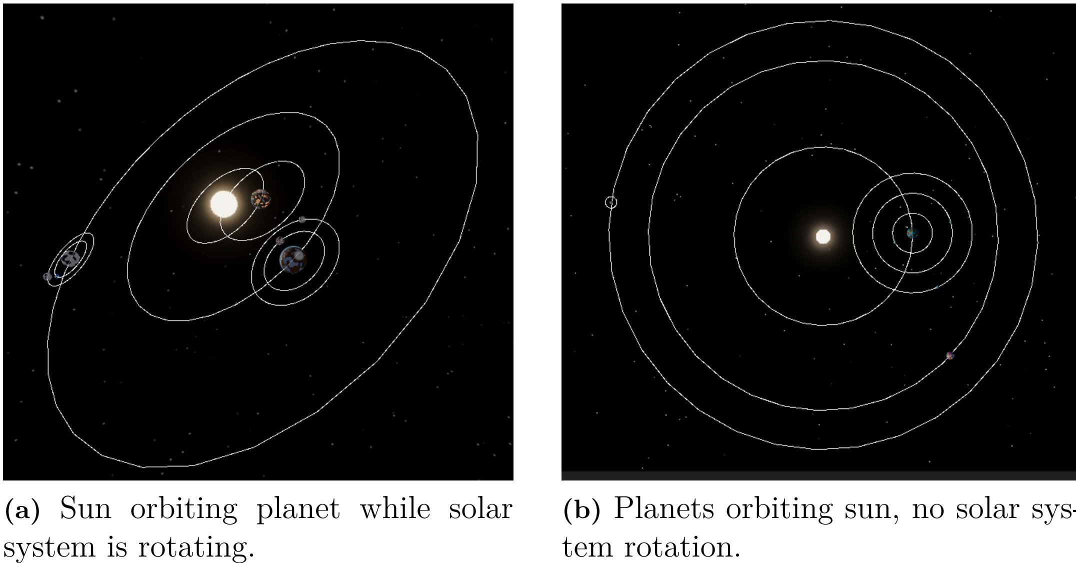
5.2 Solar system

This section will describe the resulting state of the solar system, focusing on the planet’s orbits and the world light system. These elements come together to form a solar system for the player to explore.

5.2.1 Planet orbits and rotation around their axis

The final solar system features planets and moons rotating around their own axis and orbiting their respective attracting object. Furthermore, the player’s current planet has to be stationary and non-rotating due to the reasons mentioned in section 4.1.1. This is accomplished through a geocentric model, placing the player’s planet at the center of the solar system with the sun and other planets orbiting around it. When the player is not present on any planet, the sun is at the center. From the player’s perspective, this approach fakes the current planet’s orbiting motion around the sun.

For the rotations, the player’s active planet rotation is simulated by pivoting the entire solar system and skybox around the planet’s axis. This ensures that the planet remains stationary and non-rotating. The final solar system orbits can be seen in figure 5.1.



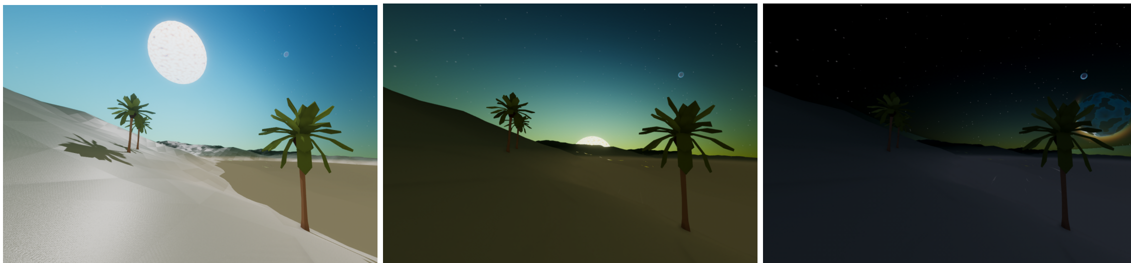
(a) Sun orbiting planet while solar system is rotating.

(b) Planets orbiting sun, no solar system rotation.

Figure 5.1: The different orbiting states of the solar system.

5.2.2 Light system

To light the world, a two-part system was created. This consists of an ambient light system and a directional light controller. The directional world light is continuously pointed from the sun towards the player to simulate real sunlight coming from the sun. It has the same color as the sun which ensures the light feels connected with the sun and the rest of the world. The ambient light bases its color and intensity on the planet position of the player. This allows the atmosphere to affect the ambient light and simulate atmosphere color tinting during the day cycle. When the player is in space, the ambient light will use a predetermined global nighttime color. Figure 5.2 shows the final lightning system during three different times of day.



(a) Day time.

(b) Sunrise / Sunset.

(c) Night time.

Figure 5.2: The light system at three points during one day cycle.

5.3 Planet Generation

The final planets have multiple features such as terrain, colors and atmospheres. This section covers the final implementation and features of the planets.

5.3.1 Biomes

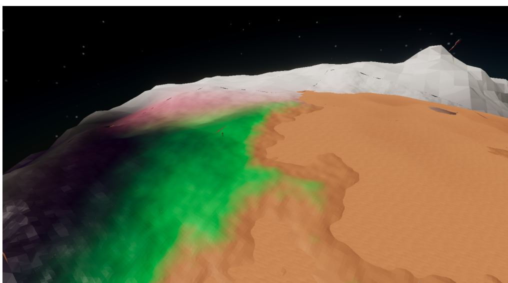
The biomes of the planets are determined through three aspects; mountains, temperature and forests. The mountain and forest aspects are the simplest and are purely normalized simplex noise. The temperature is based on the planet's distance to the sun combined with mountain noise. In addition, simplex noise is utilized to create variation in temperature, Furthermore, there are poles and warm equators as well.

5.3.2 Terrain generation

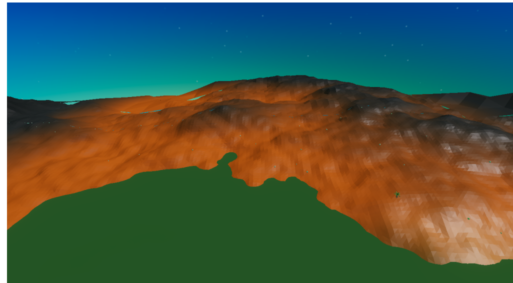
The terrain of the planets are generated using the Marching Cubes algorithm, and consists of mountains and valleys, as can be seen in figure 5.3a. These are created by extracting information on mountainous and flat regions from the mountain aspect of the biomes. The terrain is then created using layered simplex noise, a method described in section 4.2.2. The planets are also divided into chunks that have different resolutions depending on the distance to the player. The number of chunks is dynamic where more chunks are present when the player is on the planet, and fewer otherwise.

5.3.3 Terrain coloring

The terrain color is determined by blending a height-based color gradient with different biome color gradients. Colors are picked from the height-based gradient using the altitude measured from the planet's center to the spot being colored. Higher altitudes will thus have a different color compared to lower altitudes. The colors of the biomes are then added to the final color by blending them with the planet-specific ground color. An example of colored terrain can be seen in figures 5.3a and 5.3b.



(a) Mountains, polar cap and forest.



(b) Warm area around the equator.

Figure 5.3: Different types of colors for each biome area.

5.3.4 Planetary atmospheres

The final planetary atmospheres consists of a system that can generate 252 unique atmospheres for the different planets. By simulating real-world Rayleigh and Mie

scattering effects to change color, density and height of the atmosphere, it will create diverse and interesting-looking atmospheres. The atmosphere is only visual and does not include air resistance.

5.4 Ecosystem

The ecosystem consists of three parts: creatures, foliage and water which are present on most planets. This section covers the results of each part of the system.

5.4.1 Creatures

In total, the simulation has nine unique creatures seen in figure 5.4. These creatures have hunger and thirst stats which decrease over time if they do not eat or drink. Each creature has food preferences that make it a herbivore, carnivore or omnivore. Carnivores and omnivores also have the option of choosing their prey based on size. A creature can either be small, medium or large in terms of size. When a carnivore or omnivore is searching for food, an assessment is done to only allow the correct size to be consumed.

Figure 5.4: Every creature species present in the simulation.

To increase the creature population, creatures can reproduce with others of the same species. Reproduction only happens if a certain threshold of hunger and thirst is met, during reproduction, they have a probability to spawn offspring. To limit excessive population growth, the creatures have a maximum limit on how many offspring they can produce in their lifetime. When reproduction occurs, genes are transferred and mutated to the new offspring. The genes of each creature are color, speed, size, metabolism and detection radius. Mutations only occur by probability and can change the gene negatively or positively.

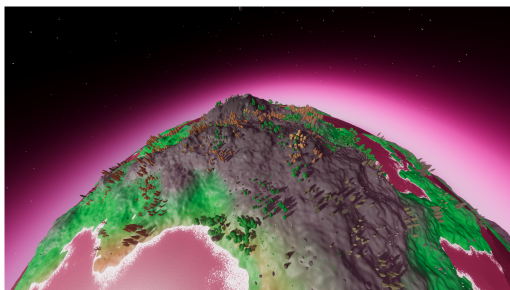
To spawn creatures, a pack spawning technique is used inside each chunk. The algorithms select random points inside the chunk and cast rays to spawn creatures. Selection of what type of creature to spawn at each location is done using the biome aspects described in section 5.3.1. All creatures are limited to certain ranges

of temperature, mountainous, and forest density which can be customized. These ranges sort out unsuitable creatures when choosing potential candidates for a spawn location and then chooses a specific creature randomly.

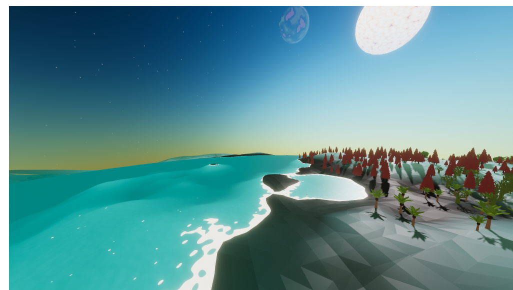
5.4.2 Foliage

Foliage is divided into five different categories, which include trees, bushes, aquatic plants, stones, and foragables. These categories consist of a total of 60 assets that come in varying sizes, shapes, and colors. To spawn these, raycasts are shot from a circle above the chunk, which can be seen in figure 4.20b. To determine which collection to spawn from, the biome aspects are used. A collection represents objects with similar biome aspects.

Moreover, a similar technique used to spawn packs mentioned in section 4.3.1 is used to create forests that contain multiple similar objects. Additionally, some plants are suitable for certain creatures to eat, and after they have been eaten, they regrow after a specific time. The foliage is also colored based on the biome they are spawned in. The final foliage system is shown in figure 5.5a



(a) Final foliage system.



(b) Final water system.

Figure 5.5: The final foliage and water systems.

5.4.3 Water

Upon starting the application, water has a chance of spawning on a planet. When spawned, it receives a random color from a preset list. Furthermore, a darker color is mixed into the shader to create depth in deep water. The water is created on the GPU in two different detail levels. To further balance out performance, the water is chunked, much like the terrain, increasing the fidelity up close and decreasing it further away. Other features such as waves and specular highlights are all generated from a shader.

The water system also allows the player to explore under the sea. A shader was applied to the camera as a filter to simulate an underwater atmosphere. This filter makes the water darker and significantly reduces the range of vision. A finalized version of the water can be seen in figure 5.5b.

5.5 The player system

The player system consists of two parts. An astronaut for exploring the surface of a planet, and a spaceship for quick transportation around and between planets. The player astronaut is animated and can move around the planet's surface by walking, running, sprinting, swimming and jumping. While disembarked, the player is pulled down by a gravitational force to keep the player on the planet or moon.

The spaceship provides convenient control over movement, allowing for full range of motion and rotation in any direction. Moreover, the spaceship contains two movement options called "orbit" and "straight" which decide if the spaceship would follow the curvature when near celestial bodies. The spaceship also features multiple camera angles, one being a third-person camera. These all can be seen in appendix B. If the player is inactive, the spaceship will gradually move down towards the surface and hover above ground. Lastly, the player can take off and land the spaceship to embark/disembark on the planet.

5.6 Miscellaneous

This section presents the result of areas that could not be categorized in any of the above categories. This includes the user interface and sound.

5.6.1 Deterministic randomness

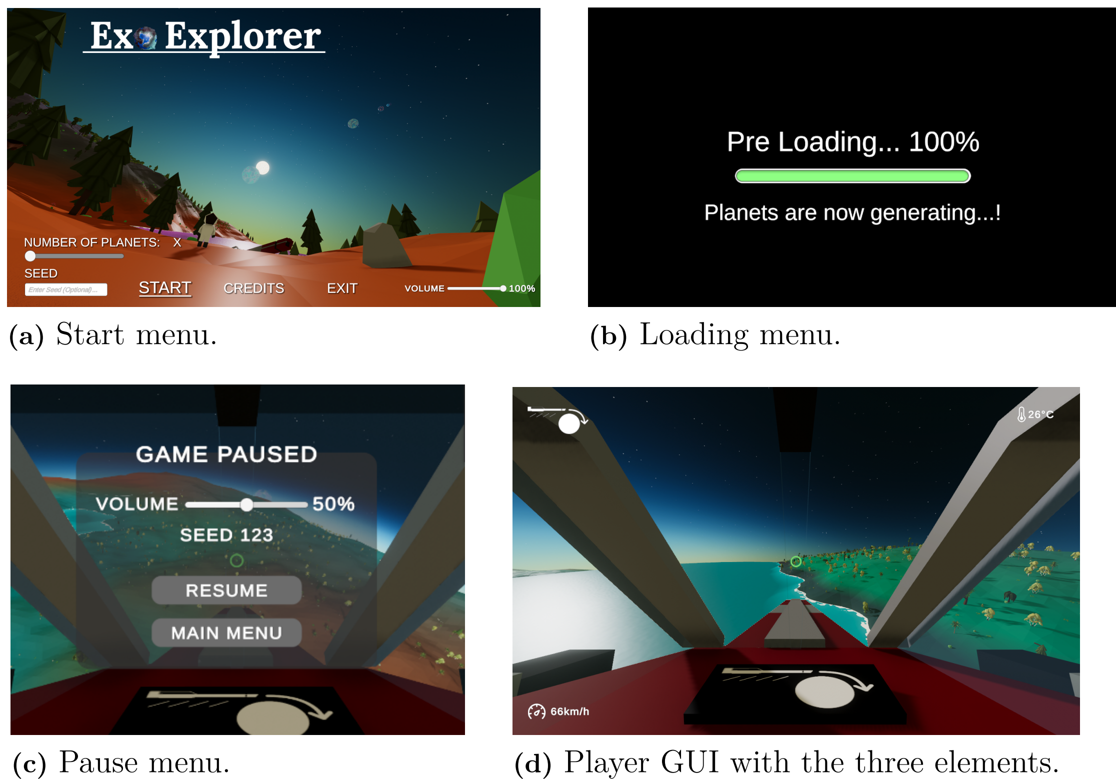
Each time the simulation is ran, a seed is used to determine asset properties. This is done deterministically where each system's randomness is isolated using chains of seeds as described in 4.5.1. This allows for a system to be regenerated using the same seed, enabling sharing of solar systems by simply sharing the seed.

5.6.2 User interface

The resulting user interface consists of a start menu, a loading menu and a pause menu. Moreover, it is combined with a GUI that displays a temperature meter, spaceship speedometer and spaceship movement indicator to create a coherent visual style throughout the application.

The start menu seen in figure 5.6a provides customizations before starting the simulation. It also includes a volume slider to select a suitable sound level throughout the application. The loading screen, displayed in figure 5.6b, gives an estimated loading duration and updated system information during loading.

Figure 5.6c shows the pause menu found in the application. It offers an option to return to the start menu and provides the user with the current seed and a functioning volume slider. The player GUI, as seen in figure 5.6d, gives the user information on temperature in the upper right corner, a speedometer in the bottom left corner and movement indicator in the top left corner. Both the speedometer and



(a) Start menu.

(b) Loading menu.

(c) Pause menu.

(d) Player GUI with the three elements.

Figure 5.6: The final start menu, loading menu, pause menu and player GUI.

movement indicator is only shown when inside the spaceship as it directly correlates to it.

5.6.3 Music and sound effects

In the game, audio can be classified into two categories: music and sound effects. Throughout the start menu, loading menu, and gameplay, different background music is consistently playing. The gameplay music features two tracks that each play correspondingly if the player is on or off the spaceship.

There are several sound effects in the application. The different user interface menus have different sounds for hovering buttons, entering text and adjusting sliders. The acceleration of the spaceship has a thrust sound effect and the running of the player has a wind sound effect. Additionally, the player has a jump and a water splash sound effect.

Moreover, different sounds are audible when the player is in proximity to a creature, such as footsteps. Faster creatures produce footstep sounds more frequently, while slower creatures have longer intervals between sounds. In addition, the game includes environmental sounds such as chicken cackles or fox howls, played at random intervals to create a more immersive experience. All sound effects are played in 3D space, providing positional audio for players with surround headsets.

6 Discussion

The following chapter discusses the working process behind the project and the finished result. Social and ethical concerns stated in section 3.5 will first be discussed, followed by a reflection of the method used in the project. Lastly, the overall project and the current state of these features will be discussed, considering potential improvements and future work.

6.1 Societal and ethical aspects

This section will review the identified ethical aspects from section 3.5. The first of these aspects is the potential misinformation this application can spread. To address this concern, thought was given to create features that functioned as close to reality as possible. Furthermore, the decision to utilize a low poly art style can help remind users that the application is a simplified representation of reality, rather than a fully accurate model. It is therefore believed that although simplifications were needed, the functionality of the features do not spread misinformation and can in some cases instead help the user differentiate between the application and the real world.

The second aspect was related to PCG's potential issue to cause unemployment. However, the implementation of PCG was more challenging than initially anticipated as it required a significant amount of work and specialized knowledge. This suggests that while PCG has the potential to automate certain aspects of content creation, it does not necessarily eliminate the need for human input. Rather, it changes the nature of the work, shifting it from traditional design towards more technical, algorithm-focused roles. This supports the idea that the move towards PCG might not lead to job losses on the scale initially feared, but instead could result in job transformation within the industry.

6.2 Project overview

The project's features were originally divided into three tables representing their priorities. This chapter adds another table representing non-planed that were added during the development process. Overall, the project was a success as all planned must-have features and most should-have features were completed. These can be seen in tables 6.1, 6.2, 6.3 and 6.4. To get a good overview of the planned features, the ones that made it into the final product are marked with a green color and the

ones that did not are marked with red. The features that were added but were unplanned are marked with blue.

Table 6.1: Must have features in the project.

Must have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Generate a sun	Procedural terrain	Creatures	Player who can walk on planets
Generate a couple of planets	Water	Creatures should move, eat, drink	Player can travel via a spaceship
Have stable orbits		Vegetation	

Table 6.2: Should have features in the project.

Should have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Moons	Stones	Predators	EVA in space
Planet rotation around its axis	Visual atmospheres	Temperature based ecosystem	Player GUI
	Temperature	Genes	
		Different creature species	
		Creatures can reproduce and die	

Table 6.3: Could have features in the project.

Could have			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Asteroids	Clouds	Sea creatures	Terraforming
N-body system	Gasplanets	Creature evolution	Save system
	Caves	Procedurally generated looks for creatures	A solar system map

Table 6.4: Non-planned added features.

Non-planned added features			
Solar system	**Planet generation**	**Ecosystem**	**Player systems**
Light System	Terrain color	Forests	Start/pause menu
	Biomes		

6.2.1 Method reflection

Generally, the method used throughout the project worked well, although, there were some advantages and disadvantages. One advantage was that it enabled parallel work via the use of independent tasks in Trello. By also setting up an MVP early on, the work on the different parts of the project did generally not interfere, and merge conflicts in GitHub were kept to a minimum. Another benefit to the way the work was carried out was that features had to be checked by other contributors before being accepted into the main software, this ensured code quality and that all group members were happy with the project.

One disadvantage of using the scrum work method was that the planning was not highly detailed prior to the development start. This meant that biomes and chunks were not well planned, and features such as foliage and creatures had to have major rewrites in order to work together with them. Had the project been done again, more time would have been spent on planning larger features beforehand.

Excluding this issue, the planning worked well. The time plan created in section 3.4.1 was followed, though it could have been more thorough to cover larger aspects such as chunks and biomes. Writing the report was originally planed to be carried out in parallel with project development. However, as work was carried out, this was not followed which resulted in many long days before the project end date.

6.2.2 Process reflection

Overall, the development process was considered to be flexible and functioned well since all major features where implemented successfully. However, general problems were encountered during the implementation regarding spherical surfaces and planning.

The problem with spherical surfaces was that many predefined Unity-features, tutorials and methods were directly tied to having a flat world. This resulted in a longer implementation time for the terrain color, water, entity gravity and player systems.

As mentioned, the lack of prior planning of chunks resulted in the foliage and creature systems having to be reworked. More specifically, the spawning of creatures and foliage stopped working and had to be rewritten. This was a mistake as the work needed to both create a complete system and afterward adapt it for the new

system increased work. A better alternative would have been to focus on creating chunks earlier, or at least mock versions, for feature systems to be built around it from the beginning.

There were also numerous customization points created during the project to adjust the simulation. To create diverse solar system, these settings needed to be changed manually. To improve this, an algorithm for setting these values could be made.

6.3 Solar system

The implementation of the features under the “Solar system” category proceeded relatively smoothly. However, the feature “planet rotation around its axis” turned out to be more complex than initially thought. It was discovered that the feature required complicated sub-features to fake it for the planet the player is on. Furthermore, the original system that was developed for the feature “Have stable orbits” using the *SimpleKeplerOrbits* asset had to be completely rewritten because of the complications with trying to incorporate the rotations. This took a considerable amount of time and was probably the reason for the last feature “Asteroids” not being implemented.

During development, the idea of implementing elliptical orbits for the planets was considered to enhance realism. However, it was discovered that this feature would be challenging to implement due to the existing orbit system and that it would not be noticed by the user. Due to these reasons, elliptical orbits were not implemented.

In the end, the solar system works well, however, improvements could have been made during the development process to ensure that less time would be spent on a single feature. With proper planning, a complete rewrite of the planetary orbits code could have been avoided and that time could have been spent elsewhere.

6.3.1 Lighting

The final solution for the light system works well, but still has some problems. Firstly, shadows from planets positioned behind the sun create visual artifacts resembling a solar eclipse, even when no eclipse is occurring. Secondly, when observing other planets, the ambient light appears to fluctuate instead of remaining constant. A continuation of the project would look for new solutions to try and fix these issues.

6.4 Planet generation

Planet generation analyses biomes, terrain generation, coloring, and planetary atmospheres, touching on the achievements and obstacles encountered throughout the development process.

6.4.1 Biomes

Initially, it was planned that the biomes would be discretely represented by for example deserts, forests and plains. However, during the implementation of biomes, it was decided that they would be derived from the biome aspects discussed in section 5.3.1. The decision was made as it would allow the biomes to blend together better and appear more dynamic.

6.4.2 Terrain generation

The advantage of the terrain system was that it had mountains and valleys that adapted to the biomes. However, the terrain generation algorithm became quite computationally expensive and affected the loading times by a significant amount. Preferably, a system that could generate all chunk resolutions in real-time would be better, as it would decrease loading times and RAM usage. Another problem was also that there were visible seams between chunks in different resolutions, this could have been covered by fog had it been implemented or an algorithm for connecting the chunks.

6.4.3 Planetary atmospheres

The planetary atmospheres diversified the planets visually as intended. Though the addition of a volumetric atmosphere to simulate fog would be more visually pleasing, it could not be implemented due to the pre-made shader used. Therefore, if the project would be developed further, a more comprehensive atmosphere solution like Sebastian Lague's discussed in section 4.2.4 would be created. This would allow a more accurate atmosphere that contains fog and volumetric sunlight, and in turn, add depth to the planet's surface while exploring.

The current atmospheres also inaccurately remain illuminated when another planet obstructs sunlight. A new atmosphere solution that could handle being partially darkened by another moon or planet would immerse the player more.

6.5 Ecosystem

As mentioned in section 6.2.2, the chunks implementation led to a major rework of the ecosystem spawning system. The new system was fast to implement and worked great. However, a disadvantage with this solution is the loss of efficiency when casting the spawning rays from the circle above the chunk since there is no current way of determining the chunk shape. Therefore, the circle had to be larger than the largest possible chunk on a given planet, which means that some raycasts miss the chunk.

6.5.1 Creatures

Currently, the ecosystem only has land-based creatures, but a desired feature was to have airborne and sea creatures. Introducing these would make increase the diversity of the ecosystem. For example, airborne creatures could interact with the foliage by eating or landing on the different bushes, trees and plants.

When adding multiple new creatures, some problems were encountered relating to animations. Some creatures had missing animations and sounds. This introduced code overhead since they could not be played for certain creatures. The solution was an adapter that checks which features are available for a specific creature.

Additionally, the current pathfinding used in creature movement is inadequate, as it only moves directly towards a point, resulting in collisions with trees, bushes and other creatures. A more advanced pathfinding algorithm called *A-star* [56] could be implemented to enhance the simulations in further development.

To improve the performance of creature movement, more efficient collision calculations could be used. Usually, these calculations are efficient when using simple colliders such as box or capsule colliders. However, the planet required a detailed collider to ensure accurate collision with the terrain, with the drawback of significantly lower performance. There are solutions to address this issue such as manually calculating the position of the object or creating local colliders beneath the creature, though this would be timely to implement and was thus not implemented.

6.5.2 Foliage

The foliage system underwent two major iterations, both of which were successful at the time they were implemented. However, with newer, large overhauls to other features, the foliage resulted in looking out of date. This was a result of bad planning, resulting in foliage falling behind in the design process. This problem could have been solved by better planning and structuring of features throughout the project. Despite this, foliage ended up having many features according to the plan.

6.6 Player system

The current solar system exploration is entirely unguided, which can be seen as liberating but also result in the user missing interesting content. Therefore, a map could be implemented to help improve the exploration system. The implementation of any map system would currently be limited to the active chunks. This is due to any possible highlights in other areas not having been generated and no way of predicting them without generating the entire area first. Furthermore, an addition of a creature scanner could entice the player to explore more and scan creatures for their stats. Lastly, the should-have feature “EVA in space” was not implemented as it was determined that it would not contribute to the purpose of the project since there is nothing to explore in space without a spaceship.

6.7 Miscellaneous

This section consists of how the deterministic randomness works, user interface and the use of Chat-GPT.

6.7.1 Deterministic randomness

The ability to recreate an entire system by using the same seed is great for world sharing as it is simpler to share a string than an entire save file. It also allows for simplified testing of systems as changing one feature does not impact the generation of other features. The design of the current deterministic randomness system was implemented halfway through the project and the rework of previously done systems was not completed immediately. This led to some systems using randomness wrong or even using shared generators, breaking the desired determinism.

6.7.2 User interface

The pause and start menu incorporated into the project were not initially planned. Even though these were not part of the original plan, they were introduced early on in the project to offer players a more customizable and pleasant experience.

6.7.3 Chat-GPT

Throughout the project, the AI language model Chat-GPT played a role in providing valuable feedback and acting as a peer reviewer for the report. It assisted in the refinement of the written content and ensured its quality. However, the feedback was not always useful and its suggestions on how to rewrite a sentence often had to be adjusted.

6.8 Future work

The section will give an overview of the future possible work that has been identified. The work includes enriching the terrain with grass and caves. Diversifying the ecosystem by introducing planet-specific color variations on foliage objects, more types of creatures, non-ray-based spawning to improve performance, and improved pathfinding for creatures. Moreover, features marked red in the tables 6.2 and 6.3 could also be worked on. Out of these, asteroids, volumetric clouds, and gas planets can be implemented to increase the realism of the solar system. Creature evolution and procedurally generated looks can add depth and variety to ecosystems. A solar system map, terraforming mechanics and a save system could provide a better exploration experience. These additions can create a more immersive, diverse, and engaging procedural generation experience for players, offering endless possibilities for exploration and interaction.

7
Conclusion

In conclusion, the application created can simulate and procedurally generate a simplified solar system, allowing users to explore and travel to celestial bodies within it. Additionally, the planets in the system can support ecosystems such as forests, lakes and diverse creatures. Each system was designed to be fully deterministic, enabling users to explore the same system using a pre-specified seed. Finally, the application is lightweight enough to run on a moderate system.

The work methodology, which involved parallel workflows and a focus on rapid progress, enabled the development of an early minimum-viable product that could then be expanded upon. Although the workflow had its strengths, a more deliberate approach to planning major application-wide features could have minimized the number of bugs and avoided the need to redo previously established systems.

Developing a large application with major features requires awareness of how performance issues can escalate quickly. Thus, a high priority was placed on optimizing performance of each feature, while also maintaining reasonable time efficiency. Priority was also put on adding more features instead of refining the already existing ones. Although refinement could have diversified used assets to provide variety, focusing on incorporating more features led to a more interesting simulation. Finding the right balance between the two was a challenging task.

This report aimed to contribute to the research done with simulations and procedural generation techniques. This was done by demonstrating the application of the Marching Cubes algorithm to procedurally generate planets. Challenges with applying common 2D techniques not typically adapted for 3D spheres were also discussed with potential solutions. By providing an overview of the process, the report has laid a foundation for future researchers and developers interested in similar techniques.

The project successfully accomplishes what it was set out to do, completing every must-have feature and most of the should-have features. The end result of the solar system, planet generation and ecosystems, combined with exploring possibilities was satisfactory. However, it could have been improved with some future work mentioned in section 6.8. Specifically, more diversity would make a large difference during exploration and enhance the simulation. This project ultimately provides great insight into the development of a huge explorable simulation.

Bibliography

[1] G. Smith, "An analog history of procedural content generation," *Northeastern University, Playable Innovative Technologies Lab*, [Online]. Available: http://www.fdg2015.org/papers/fdg2015_paper_19.pdf.

[2] H. Vuontisjärvi, "Procedural planet generation in game development," Ph.D. dissertation, Oulu University of Applied Sciences, 2014, 33 pp.

[3] S. A. Barab, K. E. Hay, M. Barnett, and T. Keating, "Virtual solar system project: Building understanding through model building," *Journal of Research in Science Teaching: The Official Journal of the National Association for Research in Science Teaching*, vol. 37, no. 7, pp. 719–756, 2000.

[4] J. Gregory, *Game Engine Architecture, Third Edition*. CRC Press, 2018, ISBN: 9781351974271. [Online]. Available: https://books.google.se/books?id=EwlpDwAAQBAJ.

[5] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games* (Computational Synthesis and Creative Systems). Cham: Springer International Publishing, 2016, ISBN: 978-3-319-42714-0 978-3-319-42716-4. DOI: 10.1007/978-3-319-42716-4.

[6] A. Lagae, S. Lefebvre, R. Cook, *et al.*, "A survey of procedural noise functions," *Computer Graphics Forum*, vol. 29, no. 8, p. 2580, 2010, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2010.01827.x.

[7] K. Perlin, "An image synthesizer," *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325247.

[8] K. Perlin and W. Sq, "Standard for perlin noise," US 6867776B2, Mar. 2005. [Online]. Available: https://patents.google.com/patent/US6867776B2/en (visited on 05/10/2023).

[9] S. Worley, "A cellular texture basis function," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 1996, pp. 291–294, ISBN: 978-0-89791-746-9. DOI: 10.1145/237170.237267.

[10] C. Leo and O. Hansen, *Spherical terrains compared: A standardized performance comparison between heightmaps and marching cubes*, 2021. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1571660/FULLTEXT02 (visited on 05/15/2023).

[11] P. Andersson and S. Johansson, *Rendering with marching cubes, looking at hybrid solutions*, 2012. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:832310/FULLTEXT01.pdf (visited on 04/26/2023).

[12] W. L. Raffe, F. Zambetta, and X. Li, "A survey of procedural terrain generation techniques using evolutionary algorithms," in *2012 IEEE Congress on Evolutionary Computation*, 2012, pp. 1–8. DOI: 10.1109/CEC.2012.6256610.

[13] (Feb. 5, 2019). "What is a computer simulation? - definition from techopedia," Techopedia.com, [Online]. Available: http://www.techopedia.com/definition/17060/computer-simulation (visited on 02/04/2023).

[14] (2023). "What is Simulation? What Does it Mean? (Definition and Examples)," [Online]. Available: https://www.twi-global.com/technical-knowledge/faqs/faq-what-is-simulation.aspx (visited on 05/15/2023).

[15] National Geographic. (May 20, 2022). "Ecosystem | national geographic society," [Online]. Available: https://education.nationalgeographic.org/resource/ecosystem (visited on 02/04/2023).

[16] Openmetal, *What is GPU Parallel Computing?* [Online]. Available: https://openmetal.io/docs/product-guides/private-cloud/gpu-parallel-computing/ (visited on 05/09/2023).

[17] OpenGL. "Compute shader - OpenGL wiki," [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader (visited on 05/09/2023).

[18] ahmadmerheb3d, *What is the Difference Between a Shader and a Material? – Artist How*, 2020. [Online]. Available: https://artisthow.com/what-is-the-difference-between-a-shader-and-a-material/ (visited on 05/09/2023).

[19] J. Barrus, R. Waters, and D. Anderson, "Locales: Supporting large multiuser virtual environments," *IEEE Computer Graphics and Applications*, vol. 16, no. 6, pp. 50–57, 1996. DOI: 10.1109/38.544072.

[20] Hello Games. (2023). "About," No Man's Sky, [Online]. Available: https://www.nomanssky.com/about/ (visited on 02/04/2023).

[21] Mojang. (2023). "What is minecraft? build, discover realms and more | minecraft," [Online]. Available: https://www.minecraft.net/en-us/about-minecraft (visited on 05/10/2023).

[22] Sponge, *World generation — sponge 5.1.0 documentation*. [Online]. Available: https://docs.spongepowered.org/5.1.0/en/plugin/wgen/index.html (visited on 05/10/2023).

[23] "Planets - official astroneer wiki," [Online]. Available: https://astroneer.wiki.gg/wiki/Planets (visited on 02/09/2023).

[24] S. Lague. (Sep. 17, 2020). "SebLague/solar-system at episode-03," [Online]. Available: https://github.com/SebLague/Solar-System/tree/Episode-03 (visited on 05/08/2023).

[25] "Start system," [Online]. Available: https://nomanssky.fandom.com/wiki/Star_system?file=NMS1dot3starsystemview.jpg (visited on 05/09/2023).

[26] B. Games. (2023). "Features," [Online]. Available: https://www.bay12games.com/dwarves/features.html (visited on 04/23/2023).

[27] (2023). "Forest," [Online]. Available: https://equilinox.fandom.com/wiki/Forest?file=Screenshot_1543392168019.png (visited on 05/09/2023).

[28] Nookrium, *Dwarf fortress: A beginners guide & tutorial [steam edition]*, Youtube, Dec. 6, 2022. [Online]. Available: https://www.youtube.com/watch?v=zEt87BikHzA (visited on 05/31/2023).

[29] J. K. Haas, "A history of the unity game engine," *Diss. Worcester Polytechnic Institute*, vol. 483, no. 2014, p. 484, 2014.
[30] "Coding in c# in unity for beginners | unity learn," [Online]. Available: https://unity.com/how-to/learning-c-sharp-unity-beginners#what-languages-can-you-use-unity--2 (visited on 02/05/2023).
[31] GitHub. (2023). "Features | github," [Online]. Available: https://github.com/features (visited on 05/10/2023).
[32] Git. (2023). "Git," [Online]. Available: https://git-scm.com/ (visited on 05/10/2023).
[33] "What is trello: Learn features, uses & more | trello," [Online]. Available: https://trello.com/tour (visited on 01/31/2023).
[34] OpenAI, *Introducing chatgpt*. [Online]. Available: https://openai.com/blog/chatgpt (visited on 05/10/2023).
[35] Atlassian. "Scrum - what it is, how it works, and why it's awesome," Atlassian, [Online]. Available: https://www.atlassian.com/agile/scrum (visited on 01/31/2023).
[36] *Divide and Conquer Algorithm*. [Online]. Available: https://www.programiz.com/dsa/divide-and-conquer (visited on 02/10/2023).
[37] "What is the MoSCoW method?" Software Quality, [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method (visited on 02/06/2023).
[38] L. A. Aguilar, "The Art of N-Body Simulations," 2006. [Online]. Available: https://www.astrosen.unam.mx/~aguilar/MySite/Teaching_files/GH06_Intro_NBody.pdf.
[39] R. Montgomery. "The three-body problem," Scientific American, (visited on 01/31/2023).
[40] puntadeleste, *How Walt Disney Cartoons are made*, Youtube, 2006. [Online]. Available: https://www.youtube.com/watch?v=mhfp6Z8z1cI (visited on 02/02/2023).
[41] Unity, *Simple Kepler Orbits | Physics | Unity Asset Store*. [Online]. Available: https://assetstore.unity.com/packages/tools/physics/simple-kepler-orbits-97048 (visited on 05/12/2023).
[42] E. Britannica. (Dec. 16, 2022). "Geocentric model," [Online]. Available: https://www.britannica.com/science/geocentric-model (visited on 04/05/2023).
[43] P. Bourke. (1994). "Polygonising a scalar field (marching cubes)," [Online]. Available: http://paulbourke.net/geometry/polygonise/ (visited on 03/26/2023).
[44] T. Archer, "Procedurally generating terrain,"
[45] A. J. Patel, "Making maps with noise," Red Blob Games, 2015. [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/ (visited on 03/26/2023).
[46] Coolors, *Coolors*. [Online]. Available: https://coolors.co/ (visited on 05/15/2023).
[47] M. Donald, *Planet atmospheres, ray-sphere intersections*, Youtube, May 15, 2020. [Online]. Available: https://www.youtube.com/watch?v=OCZTVpfMSys.

[48] S. Lague, *Coding adventure: Atmosphere*, Youtube, Aug. 22, 2020. [Online]. Available: https://www.youtube.com/watch?v=DxfEbulyFcY.

[49] S. Hillaire, "A scalable and production ready sky and atmosphere rendering technique," *Computer Graphics Forum*, vol. 39, no. 4, pp. 13–22, 2020. DOI: https://doi.org/10.1111/cgf.14050.

[50] S. Lague, *Geographical adventures*, Jun. 2022. [Online]. Available: https://github.com/SebLague/Geographical-Adventures.

[51] L. Gamage, *Atmospheric scattering shader for unity.* [Online]. Available: https://gist.github.com/T5impact/9edc753a048ea90c7a2bdbaa5c1255d8.

[52] Unity, *Low Poly Animated Animals | 3D Animals | Unity Asset Store.* [Online]. Available: https://assetstore.unity.com/packages/3d/characters/animals/low-poly-animated-animals-93089 (visited on 05/13/2023).

[53] J. Flick, *Triplanar mapping*, Apr. 29, 2018. [Online]. Available: https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/ (visited on 04/26/2023).

[54] Unity, *Unity Asset Store - The Best Assets for Game Making.* [Online]. Available: https://assetstore.unity.com/ (visited on 05/13/2023).

[55] Sharp coder, *Spaceship controller in unity 3d.* [Online]. Available: https://sharpcoderblog.com/blog/spaceship-controller-in-unity-3d.

[56] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

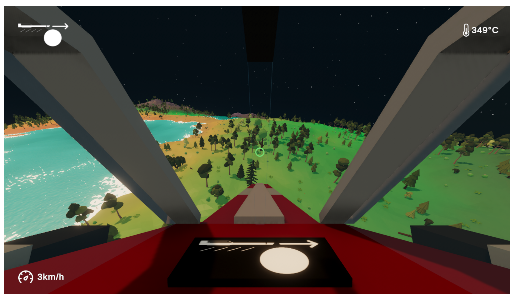
A
An additional technique

A.1 Using heighmap-techniques on Marching Cubes

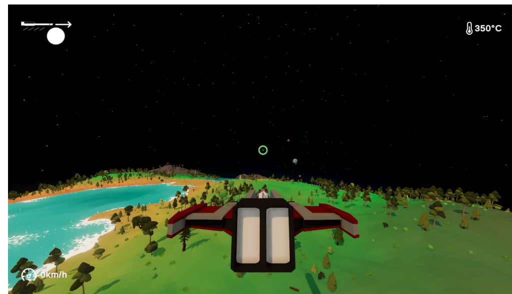
In order to use the commonly used algorithms for heightmap terrain generation with Marchin Cubes, an algorithm was created. The algorithm works by normalizing the sample point from the marching cubes, thus getting the point on a unit sphere. After this, the algorithm evaluates the terrainvalue at the point and checks if it is inside or outside of the sphere. If inside, it then interpolates the terrainvalue between 0 and the radius at that point and returns this as the terrainvalue to the marching cubes algorithm. This allows for using the common heightmap terrain techniques for the Marching Cubes algorithm.

B

Additional images

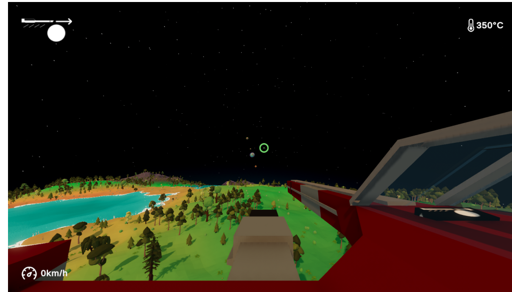


(a) Cockpit view.

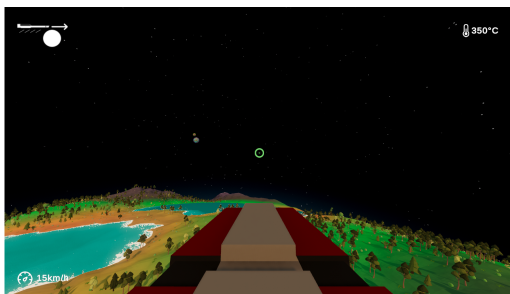


(b) Third person view.

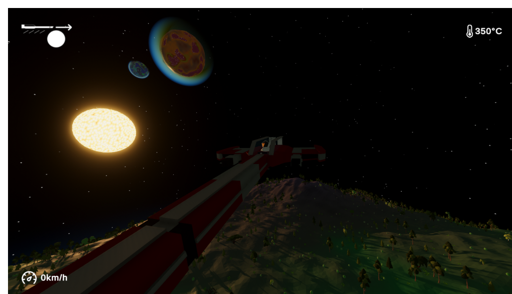
(c) Close up third person view.



(d) Wing view.



(e) Front view.



(f) Backwards view.

Figure B.1: Different angles taken from the ship.