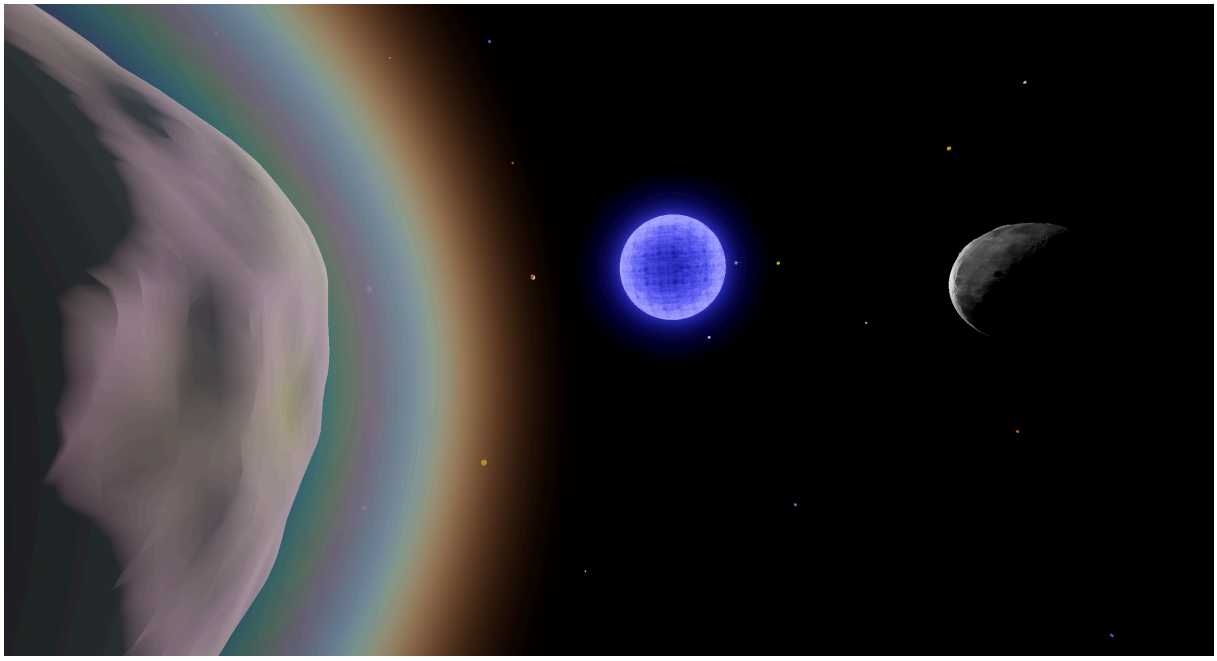


Galaxy Engine

Simulating a physics-based procedurally generated galaxy

Bachelor's thesis in Computer science and Engineering

Jacob Andersson
Erik Berglind
Anton Frejd
William Karlsson
Jonatan Lindh
Paul Soukup

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Bachelor's Thesis 2025

Galaxy Engine

Simulating a physics-based procedurally generated galaxy

Jacob Andersson
Erik Berglind
Anton Frejd
William Karlsson
Jonatan Lindh
Paul Soukup

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Galaxy Engine

Simulating a physics-based procedurally generated galaxy

Jacob Andersson, Erik Berglind, Anton Frejd, William Karlsson, Jonatan Lindh, Paul Soukup

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

Supervisor: Staffan Björk, Department of Computer Science and Engineering
Graded by teacher: Aris Alissandrakis, Department of Computer Science and Engineering
Examiners: Arne Linde and Patrik Jansson, Department of Computer Science and Engineering

Bachelor's thesis 2025
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 view of a planet (left) and its moon (right), with the solar system's star in the middle.
Link to the GitHub repository: https://github.com/JonatanLindh/kandidat

Typeset in Typst
Gothenburg, Sweden 2025

Galaxy Engine

Simulating a physics-based procedurally generated galaxy

Jacob Andersson, Erik Berglind, Anton Frejd, William Karlsson, Jonatan Lindh, Paul Soukup

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

Abstract

Procedural Content Generation (PCG) enables the algorithmic creation of vast virtual worlds, particularly relevant for space exploration simulations, yet poses significant challenges regarding computational scale, performance, and plausibility. This project addresses these challenges by developing and simulating a physics-based, procedurally generated, and explorable galaxy within the Godot game engine. The core objective was to create a deterministic and computationally efficient model. Planets were generated using noise functions (Perlin, fBm) combined with the Marching Cubes algorithm to create complex terrains, enhanced with procedurally placed oceans, vegetation, and physically-based atmospheres simulating Rayleigh scattering. Optimization was crucial, employing techniques such as multi-threading for planet generation, chunking, and Level of Detail (LOD) managed via octrees. A custom N-body physics engine, featuring a parallelized Barnes-Hut algorithm ($O(N \log N)$) built upon a Morton-code-based linear octree, simulates celestial mechanics. Solar systems and the galaxy structure utilize seeded randomization for determinism and reproducibility. Exploration spans multiple scales, from a galaxy level down to planetary surface navigation with a physics-aware controller. The project successfully yielded a real-time simulation, demonstrating efficient generation and rendering of diverse celestial bodies and stable orbital physics, prioritizing consistent frame times. This work contributes a practical implementation and analysis of techniques for large-scale procedural galaxy simulation in Godot.

Keywords: Procedural content generation, planet, solar system, galaxy, physics simulation, Barnes-Hut simulation, noise, Godot

Sammandrag

Processuell Innehållsgenerering (PCG) möjliggör algoritmiskt skapande av enorma virtuella världar, vilket är särskilt relevant för rymdutforskningssimulationer. Metoden medför dock betydande utmaningar gällande beräkningsmässig skala, prestanda och trovärdighet. Detta projekt adresserar dessa utmaningar genom att utveckla och simulera en fysikbaserad, processuellt genererad och utforskningsbar galax i spelmotorn Godot. Huvudmålet var att skapa en deterministisk och beräkningseffektiv modell. Planeter genererades med hjälp av brusfunktioner (Perlin och fBm) i kombination med Marching Cubes-algoritmen för att skapa komplex terräng. Denna terräng förbättrades sedan med processuellt placerade hav, vegetation och fysikbaserade atmosfärer som simulerar Rayleigh-spridning. Eftersom optimering var avgörande användes tekniker som flertrådning för planetgenerering, "chunking" och detaljnivåer (LOD). En specialbyggd N-kropps-fysikmotor simulerar himlakroppars mekanik. Motorn använder en parallelliserad Barnes-Hut algoritm ($O(N\ \log N)$), byggd på ett linjärt octree baserat på Morton-kod. Solsystem och galaxens struktur baseras på "seedad" slumpmässighet för att säkerställa determinism och reproducerbarhet. Utforskning är möjlig på flera skalor, från galaxnivå ner till navigering på planetens yta med fysikbaserad styrning. Projektet resulterade i en framgångsrik realtidssimulering som demonstrerar effektiv generering och rendering av varierande himlakroppar samt stabil omloppsfysik. Fokus låg på att uppnå konsekventa bildrutetider. Detta arbete utgör ett praktiskt bidrag i form av en implementation och analys av tekniker för storskalig processuell galaxsimulering i Godot.

Nyckelord: Processuell innehållsgenerering, planet, solsystem, galax, fysiksimulering, Barnes-Hut-simulering, brus, Godot

Acknowledgements

We would like to thank our supervisor Staffan Björk for his invaluable support throughout the project. From telling funny anecdotes to being a mentor, he has always been there to provide many interesting ideas and answers to even our most difficult questions.

Jacob Andersson, Erik Berglind, Anton Frejd, William Karlsson, Jonatan Lindh, Paul Soukup

May 2025

Contents

Figures

Tables

Glossary

- **Player**: User controlled element in the simulation.
- **Graphics Processing Unit (GPU)**: Computer hardware component that can effectively perform many, parallel, mathematical computations and is often used for rendering computer graphics [1].
- **Central Processing Unit (CPU)**: Central hardware component of a computer composed of the control unit, main memory, and arithmetic logic unit [2].
- **Mesh:** A collection of vertices, edges, and faces that define the shape of a 3D object [3].
  - **Vertex:** A point in 3D space, representing a corner of a triangle. Stores attributes such as position (x, y, z), normal, UV, and optionally color.
    - **Normal:** A vector that is perpendicular to a surface. Used in lighting calculations.
    - **UV coordinates:** a 2D coordinate system used to map a texture image to a 3d object surface.
  - **Edge:** A line segment connecting two vertices.
  - **Face (Triangle):** A flat surface bounded by edges (typically a triangle in real-time graphics). Triangles are used because they are always planar (all vertices lie in the same flat plane).
- **Rendering Pipeline:** The sequence of stages a GPU uses to render a 3D scene [4].
- **Shader:** A small program that runs on the GPU, primarily used for controlling how objects are rendered [5].
  - **Vertex Shader:** Processes each vertex and transforms vertex attributes, such as position [6].
  - **Fragment Shader (Pixel Shader):** Determines the final color of each fragment (roughly corresponding to a screen pixel), often by using lighting and textures [7].
  - **Compute Shader:** Used for general-purpose computation on the GPU (not directly part of the rendering pipeline) [8].
- **Procedural Content Generation (PCG):** The creation of data (models, textures, etc.) algorithmically, rather than manually. This often uses noise functions and other mathematical techniques to create varied and complex results [9].
  - **Seed:** A value used to initialize a pseudo-random number generator. Using the same seed ensures the same sequence of "random" numbers [10].
  - **Noise Function:** Algorithms for generating pseudo-random values with a smooth, continuous appearance, used for procedural generation [9], [11].
  - **Marching Cubes:** Algorithm to create triangle meshes from 3D scalar fields (e.g., noise) [12].
- **Level of Detail (LOD):** Rendering objects with varying complexity based on distance [13].
- **Performance metrics:**
  - **Frames per second (FPS):** The amount of images (frames) that a computer renders every second [14]. Higher FPS generally means smoother motion. For example, 60 FPS means the screen is updated 60 times per second. Inversely related to frame time.
  - **Frame time:** The amount of time (in milliseconds) it takes to render a single frame [15]. Inversely related to FPS.
- **Godot terminology:** [16]
  - **Node:** A fundamental building block in Godot for creating game elements. Nodes can represent various components such as images, 3D models, cameras, colliders, sounds, and more.
  - **Tree:** An arrangement of nodes organized hierarchically, where each node can have child nodes.
  - **Scene:** A collection of nodes arranged as a tree, which can be saved as a new reusable self-contained node. Scenes can be instantiated multiple times throughout an application. For example, a Player Character Scene might include nodes for the character's image, collider, and camera, all grouped together for easy reuse.

1. Introduction

Procedural content generation (PCG) offers a way to algorithmically create vast and diverse game worlds without the immense manual effort required to design every detail. Applications of PCG range from the random placement of enemies in confined dungeon spaces to the generation of entire universes comprising millions of celestial bodies. Using PCG also has the potential to increase replayability [17].

Using PCG algorithms is particularly relevant in the context of space exploration games, where the scale of the universe is inherently beyond manual creation. However, creating compelling, varied, and believable planetary systems and galaxies is a challenging problem within this domain. Key issues include the computational efficiency required to generate hundreds to millions of celestial objects, as well as the need to balance performance constraints with the goal of providing a plausible and playable experience.

This thesis addresses these challenges by developing a system for procedurally generating a galaxy composed of multiple solar systems, with an emphasis on computational efficiency and physical accuracy. The implementation is built in the Godot game engine, and requires utilization of relevant techniques, such as GPU computation, diverse terrain generation algorithms, and physical models. In addition, the project documents key development decisions to offer insight into addressing such challenges.

A related project, Exo Explorer [18], a bachelor's thesis from Chalmers University of Technology, explored similar themes. While Exo Explorer focused on a single solar system and emphasized gameplay and planetary ecosystems, this thesis places a greater focus on simulating a physically accurate model of a procedurally generated, explorable galaxy, rather than emphasizing planet-level details.

1.1. Purpose

The aim of this project is to create a physics-based simulation of a procedurally generated, explorable galaxy.

Each solar system within the galaxy will contain procedurally generated planets orbiting a central star, governed by a simplified Newtonian physics simulation. System complexity will range from a single planet to multiple planets and moons.

While procedurally generated, the galaxy will remain persistent and revisitable by ensuring deterministic generation. Different seeds allow for unique galaxy creation, while ensuring specific regions generate identically upon revisit.

1.2. Limitations

Key project limitations were established. The physics simulation was planned to be simplified, prioritizing plausible orbital mechanics for solar systems over strict adherence to real-world gravitational laws or all physical intricacies.

Furthermore, this project was going to be developed as a technical demonstration, not an actual video game. The primary focus was on terrain generation, physics simulation, and performance optimization. Consequently, gameplay-oriented features such as extensive UI for navigation or detailed planetary properties were not prioritized, as the goal was a procedurally generated galaxy model, not an engaging gameplay or narrative experience. That said, for clarity and convenience, we will still use the terms player and gameplay, even though the project is more accurately described as a simulation.

1.3. Contribution

This thesis hopes to contribute a computationally efficient, conceptually interesting, and accurate physics model for simulating celestial movements, as well as a scalable procedural generation system for populating game environments, implemented within the Godot engine.

2. Background

This section presents the foundational theoretical concepts that were needed before beginning the project, followed by a few select previous works that utilize some of these concepts.

2.1. Procedural Content Generation

Procedural Content Generation (PCG) is defined as "the algorithmic creation of game content with limited or indirect user input" [9]. While PCG is widely used in video games—typically involving the automatic generation of content such as unique levels for each gameplay session or the stochastic placement of environmental elements such as vegetation—it also has applications beyond gaming, such as in architectural design, and data generation.

2.2. Noise

In PCG, noise refers to pseudo-random, often spatially coherent, data used to introduce controlled variation and natural-looking patterns. It is typically generated by functions that produce a value for any given input point (e.g., coordinates) [9].

Noise is very commonly used in PCG, such as for generating mountains, textures and vegetation [9]. It is generated through the use of a pseudo-random function and is often stored and visualized as a texture.

Perlin noise [11] is a famous gradient noise function developed by Ken Perlin in 1985 that has been used extensively to produce procedural content in games and films.

2.3. Terrain Generation

This subsection provides an overview of height-maps and the Marching Cubes algorithm, the two techniques used to procedurally generate terrain in the project.

2.3.1. Height-maps

Height-maps serve as data structures encoding elevation information, often applied to vertices of a mesh geometry [19]. Typically, height-maps are implemented as grayscale image files, where the intensity of each pixel corresponds to the height at a specific location on the mesh. This representation is computationally efficient and widely used in terrain generation and visualization. However, height-maps have inherent limitations. Notably, they are capable of representing only a single elevation value per (x, y) coordinate. As a result, they are unsuitable for modeling complex terrain features that involve vertical structures or vertical overhangs, such as caves and arches.

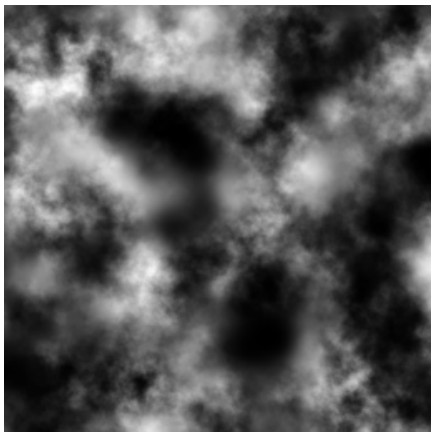


Figure 1: Heightmap texture

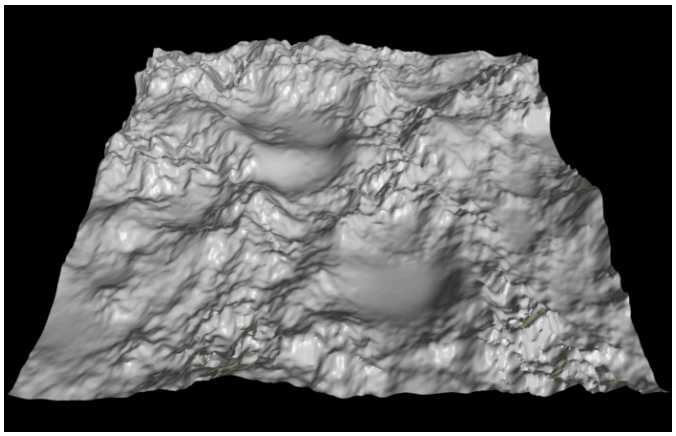


Figure 2: 3d render using the height-map texture

2.3.2. Marching Cubes

The Marching Cubes algorithm [12] is a method for extracting a polygonal mesh representation of an isosurface from a three-dimensional scalar field. An isosurface is a surface that represents points of a constant value within a volume. A scalar field, on the other hand, is a "*scalar-valued function of the points of a domain in some space, such as the temperature or the density field inside a body*" [20]. The algorithm takes a grid of scalar values as input and generates a mesh approximating the isosurface defined by a given threshold, enabling the visualization of complex structures within the data.

The procedure involves traversing the scalar field and evaluating groups of eight adjacent grid points, which form a logical cube. For each cube, the algorithm determines the polygon(s) that approximate the isosurface intersecting that region. This is achieved by classifying each vertex of the cube as either inside or outside the isosurface, based on whether its scalar value (the value at that vertex in the scalar field) is above or below the predefined iso-level (the chosen threshold value that defines the isosurface). In other words, the iso-level specifies the value that the isosurface represents, while the scalar value is the actual value at a specific vertex. The algorithm then references a precomputed lookup table to identify the appropriate triangulation for the cube's configuration.

Given that each of the eight cube vertices can exist in one of two states (on or off), there are $2^8 = 256$ possible configurations. However, due to symmetry and rotational equivalence, these reduce to 15 unique cases (see Figure 3), with all others derivable through reflection or rotation. The process is repeated throughout the entire scalar field, and the resulting polygons are combined to form the final mesh.

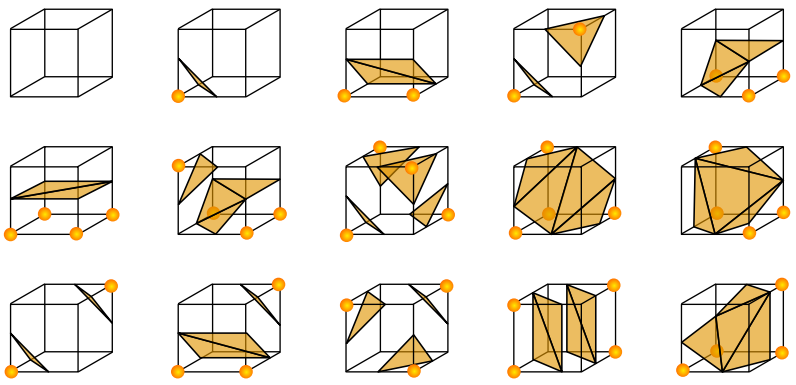


Figure 3: Marching Cubes' 15 unique polygon combinations. Ryoshoru, CC BY-SA 4.0, via Wikimedia Commons

2.4. Chunks

Chunks [21] refer to (in this context) fixed-size segments of data that are loaded, processed, and rendered independently. This approach is particularly beneficial in large or procedurally generated environments, as it allows the game engine to manage memory and computing resources efficiently by loading only chunks near the player.

2.4.1. Octrees

An octree [22] is a hierarchical data structure that recursively subdivides three-dimensional space into eight octants using a tree structure (see Figure 4). This structure is particularly effective for managing sparse or large-scale environments, as it allows for efficient spatial queries, collision detection, and level-of-detail (LOD) rendering.

Chunks are typically uniform, fixed-size sections of the game world that load and unload as needed, but octrees provide a more dynamic alternative.

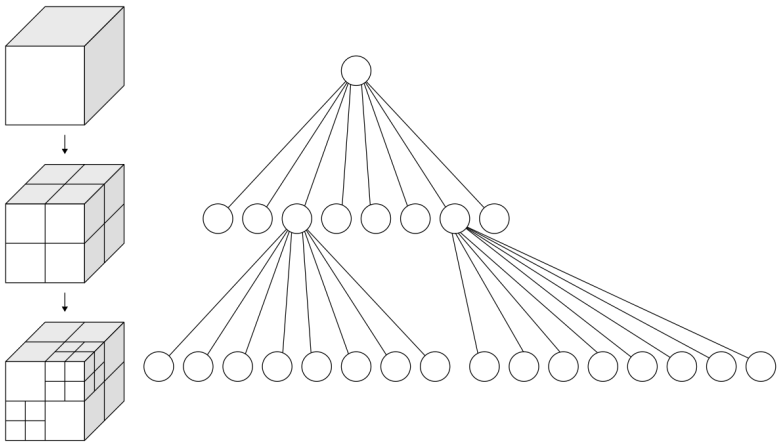


Figure 4: Visualization of an Octree structure in both a cube and tree format. WhiteTimberwolf, CC BY-SA 3.0, via Wikimedia Commons

2.5. GPU Computation

GPU computing is the process of utilizing the highly parallel nature of the GPU for running code. Since the GPU has significantly more processing units than the **CPU**, it can be utilized to write highly parallelized pieces of code to solve certain programming problems [23]. Programs that run on the GPU are often referred to as shaders [24].

2.6. Previous works

Several existing games and research projects provide a foundation for this work, demonstrating both the potential and the challenges of procedural planet and solar system generation and simulation:

2.6.1. Minecraft

While not focused on planetary systems, Minecraft [25] demonstrates the power of procedural generation in creating vast and varied landscapes using noise functions (explained in Section 2.2) and height-maps (explained in Section 2.3.1). In Minecraft, noise is used to procedurally create terrain features such as mountains, valleys, and cave systems. This approach results in endless, unique environments for the player to explore.

Figure 5: Minecraft Windswept Hills biome, CC BY-NC-SA 3.0, Mojang Studios ©

2.6.2. Outer Wilds

Outer Wilds [26] is a space exploration and adventure game that bears a small resemblance to this project. Unlike the other works mentioned, which are related through their use of procedural generation, Outer Wilds features entirely hand-crafted content. Its relevance to this project instead lies in its approach to physics. In Outer Wilds, all physics interactions are computed in real-time, with no pre-defined behaviors. For instance, planetary motion is governed by a modified version of Newton's law of gravitation, and all velocities are dynamically calculated during gameplay [27].

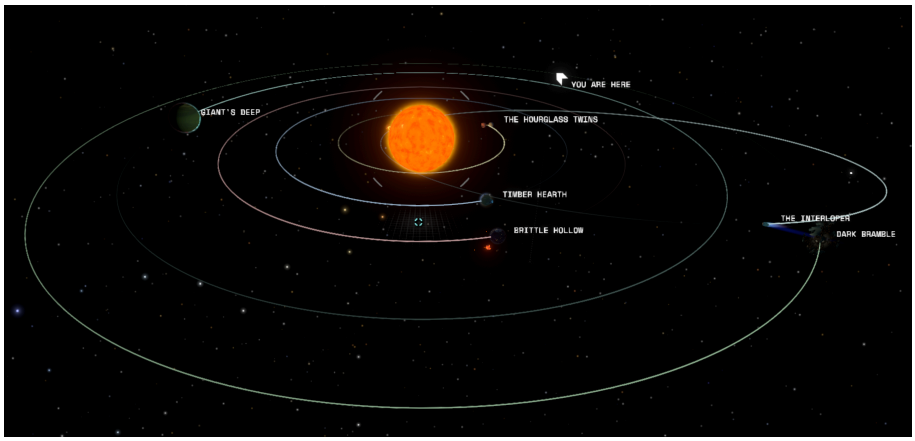


Figure 6: Map of the Outer Wilds solar system. Outer Wilds, Mobius Digital ©, 2019. Screenshot taken in-game. Used under Mobius Digital's Fan Content Policy & Guidelines

2.6.3. Exo Explorer

Exo Explorer [18] is an earlier bachelor's project, also from Chalmers, which directly addressed the challenge of procedurally generating solar systems using the Unity engine [28]. The project utilized Perlin noise [11] and the Marching Cubes algorithm [12] to create planet terrain featuring forests, lakes, and creatures with basic simulated needs (hunger, thirst, reproduction).

Exo Explorer provided valuable inspiration for this project by demonstrating techniques in procedural generation and optimization, among others. However, this project seeks to expand on that foundation. It aims to delve deeper into aspects such as the complexity of simulated physics, performance, and exploration. A key distinction is this project's greater focus on simulating multiple, simultaneously explorable solar systems.

3. Planning

This section describes the methodology and planning behind the project. It presents the chosen workflow, the selected tools and technologies, and the intended features.

3.1. Workflow

Development was planned to follow an Agile-inspired [29] workflow, meaning that the work was to be divided into week-long "sprints" with iterative task refinement. Task prioritization and addition of tasks to the backlog were done at the end of the week before the weekly supervisor meeting. Task management involved tracking various states for each task (on the Kanban board [30]), including "blocked", "to-do", "in progress", "in review", and "done".

3.2. Git

During the development process, the version control system Git [31] was planned to be utilized in conjunction with GitHub [32]. Additionally, a Kanban board [30] on GitHub was used for task management.

The project's standard workflow for Git and GitHub was planned to involve maintaining each feature within a dedicated branch [33]. GitHub's Kanban board would also allow for associating branches with specific tasks. Moreover, acceptance criteria would be established for each task on the Kanban board and, once all criteria were met, a pull request would be created to merge the changes into the main branch. Before finalizing the merge, at least one other team member would be required to review the code and the newly implemented feature. This review process would serve both as a quality assurance measure and as an opportunity to provide feedback.

3.3. Godot

The project was planned to be developed using the Godot game engine [34]. Godot is a free, open-source game engine that employs a node-based system that enables modular and reusable component design. It officially supports C# and GDScript [35]. Furthermore, community-driven extensions, such as the GDExtension [36], expand language compatibility to languages such as C++ and Rust.

3.4. Benchmarking and Performance

When developing real-time applications such as simulations, video games, or other computer applications, maintaining responsiveness and stability during runtime is essential for the user experience. A common metric to measure the performance of any such application is Frames Per Second (FPS), which is the amount of rendered images (frames) that are displayed each second. Higher and consistent FPS is desirable for a stable experience, as well as reduced visual artifacts and improved system latency from user input to its representation on the display [14].

However, average FPS alone does not always provide a complete picture of performance. Instead, examining individual frame times (the time it takes to render a frame) reveals inconsistencies that average FPS can mask. Issues such as brief momentary lag at computation-heavy moments may be overlooked in average FPS values, while being detrimental to the user experience. Metrics such as 1% lows and 0.1% lows of FPS have become common [37], to capture these worst-case scenarios. This corresponds directly to capturing the slowest (highest value) 1% highs and 0.1% highs of frame times [15], [38].

The disparities between the three values: the total frame time average, the 1% highs, and the 0.1% highs, are important. Reducing the disparities is crucial for an overall stable user experience. Gamers Nexus [39] mentions that disparities between frames of 8ms or more start to become perceptible to the user.

Overall, the project was planned to emphasize maintaining consistent frame times, rather than high average FPS, more precisely:

- Keep the disparities between the frame time average, its 1% highs, and its 0.1% highs, to a maximum of 8ms.
- Maintain an average of 60 FPS (equivalent to a frame time average of ~16.7ms), when the program is not experiencing its frame time 1%, or 0.1% highs.

Benchmarking was planned to be performed on a dedicated benchmarking computer named PC-1 (see Table A-1), but would occasionally be performed on other machines, even though their results may not directly reflect performance on PC-1. If benchmarks were performed on a different machine, it would be explicitly noted. Even so, performance comparisons before and after changes could still offer meaningful insights into a relative change on PC-1. These specifications of other machines are noted in Appendix A.

3.5. Tasks

This subsection outlines the key tasks identified during the planning phase of the project.

3.5.1. Procedural Content Generation

All content in the program was planned to be generated procedurally. To ensure persistent galaxy generation and reproducibility, all procedural content was to be generated using a fixed seed. This would allow for consistent environments across sessions and provide predictability during benchmarking and testing.

3.5.2. Planets

It was decided that the planets should primarily be generated using different noise algorithms. The planets should also be constructed using the Marching Cubes algorithm and be aesthetically distinct from one another.

3.5.3. Solar Systems

Solar systems were planned to be procedurally generated using randomized parameters such as the number of planets, celestial body orbits, and their physical attributes (e.g., mass, rotation, size). To ensure stability, key parameters such as orbital radius, mass, and velocity were designated for manual fine-tuning. This approach would allow for the possibility of generating coherent systems in real-time.

3.5.4. Physics Simulation

An accurate gravity simulation was a central requirement for achieving realistic physical behaviors. Different gravity systems were planned to be utilized for different purposes. For example, the player system could utilize the built-in physics engine to simulate gravity on planets, while the celestial dynamics utilize a manual implementation that is better suited for them.

3.5.5. Galaxy

To meet the requirement of implementing a procedurally generated, explorable galaxy, several design approaches were considered. These included generating solar systems dynamically at runtime to enable seamless exploration, as well as developing a galaxy map interface allowing players to select and explore individual solar systems via point-and-click navigation. It was also planned to apply the physics simulation to the entire galaxy.

3.5.6. Exploration

The ability to explore the generated content needed to be implemented. Multiple solutions were discussed, but since the main purpose was not to provide an engaging gameplay experience, it was

decided that a simple camera controller would be prioritized, with the option to add more complex features if time allowed for it. Additionally, functionality for inter-solar system traversal needed to be implemented to allow for full exploration possibilities of multiple solar systems.

3.5.7. Optimization

The aim was to construct a real-time application, and thus, optimization techniques were of increasing importance. Inefficient algorithms and resource management could eventually lead to performance bottlenecks. To address this, proven techniques and smart solutions were planned to be explored throughout all parts of the project.

3.6. Features

After specifying the main objectives and implementation considerations, the project's features were identified and prioritized using the MoSCoW (Must have, Should have, Could have, Won't Have) analysis method [40] (see Appendix C). This was done to better understand which features to prioritize and which to save for later.

4. Process

This section outlines the process for creating the various components that comprise the project. Each subsection represents a step in increasing scale, starting from the planet-scale, focusing on unique terrain generation and other planetary features, expanding to the system-scale organization of celestial bodies and their orbital physics, and finally reaching the galaxy-scale distribution of stars.

4.1. Planet Generation

This section describes the planet generation implementation process.

4.1.1. Height-map planets

The first planets to be constructed were the height-map planets. These planets were created by mapping a cube onto a sphere and displacing the vertices using height-map values (Section 2.3.1) to create variation in the terrain elevation. Additionally, a simple planet shader was implemented to color the planets based on their height relative to their lowest point (see Figure 7).

While height-maps were sufficient for generating simple planets, generating more complex terrain, such as overhangs or caves, required a different approach to be implemented. The Marching Cubes algorithm [12] was chosen for this purpose during the planning stage.

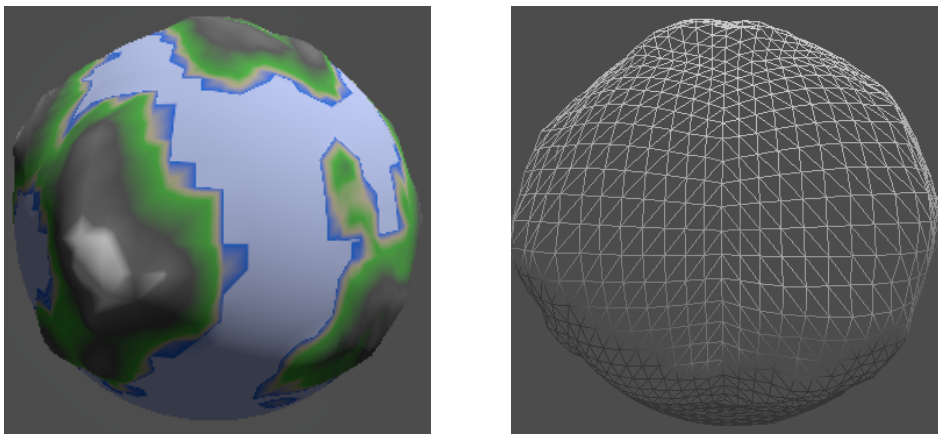


Figure 7: Height-map planet with shader (left) along with its faces (right)

4.1.2. Transitioning from height-maps to Marching Cubes

When transitioning from height-maps to Marching Cubes the method of generating the planets needed to change from a cube mapped onto a sphere to an isosurface, with each point containing a scalar values, as shown in Figure 8.

The Marching Cubes algorithm was used to implement this approach, following the method outlined in Section 2.3.2. Figure 9 shows the result of this process applied to the scalar field shown in Figure 8.

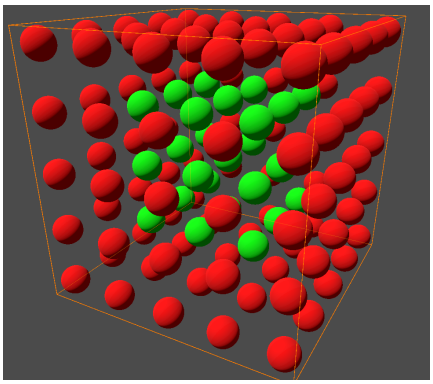


Figure 8: Scalar field input for Marching Cubes, where green represents inside the isosurface and red outside

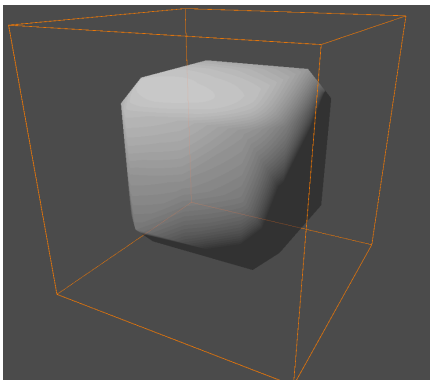


Figure 9: Marching Cube generated mesh from the scalar field

4.1.3. Noise planets

The noise planets were the first version of planets utilizing Marching Cubes. These planets were formed by generating a scalar field as shown in Figure 8. Initially, the scalar values given to each point were binary in nature, with `-1` representing that point as being "outside" the surface ("empty space") and `1` representing the point being "inside" the surface ("included in the surface geometry"). To create spherical planets, points within a defined radius from the center were assigned the value `1`, while others were set to `-1`:

```
distanceToCenter = (centerPoint - currentPosition).Length();
if (distanceToCenter < radius) {
    points[x, y, z] = 1.0f;
}
else {
    points[x, y, z] = -1.0f;
}
```

This method produced a spherical planet as shown in Figure 10. However, the generated terrain lacked surface variation, which was addressed by introducing noise (Section 2.2). Instead of assigning constant values for the points inside the planet, a 3D noise function was used.

To give variation between generated planets, a random offset was also introduced when sampling the noise to avoid generating two identical planets. The code below shows how this was implemented and the result is demonstrated in Figure 11:

```
if (distanceToCenter < radius) {
    points[x, y, z] = GetNoise3Dv(currentPosition + offset);
}
else {
    points[x, y, z] = -1.0f;
}
```

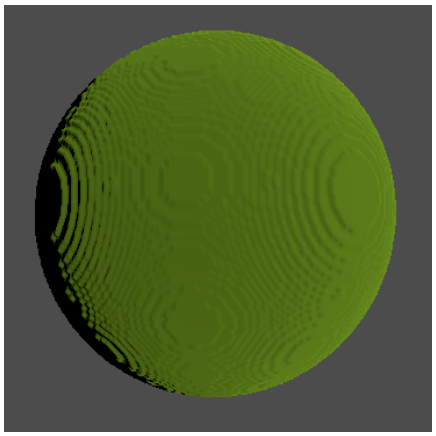


Figure 10: Spherical planet generated using constant binary values

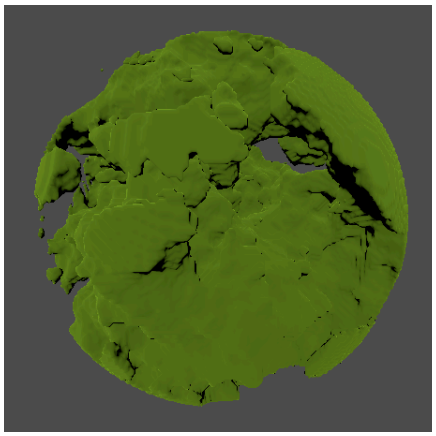


Figure 11: Spherical planet generated using 3D Perlin noise

Although using noise made the terrain more visually interesting, the planet shown in Figure 11 did not appear as visually appealing as the earlier height-map planets. This was partly due to the planets being constrained to a perfect sphere, with all detail "carved out from the surface of the planet".

4.1.4. Interpolation

The planet surfaces shown in Figure 10 and Figure 11 appeared rough, despite being spherical. This roughness was due to the lack of interpolation in both the scalar field and the Marching Cubes algorithm.

The scalar field interpolation was implemented by using the distance from the planet's surface as the scalar value, rather than binary values. This resulted in a smoother transition between points located inside and outside the planet. The implementation can be seen in the pseudo code below:

```
distanceToCenter = (centerPoint - currentPoint).Length();
points[x, y, z] = radius - distanceToCenter;
```

Interpolation in the Marching Cubes algorithm was addressed by calculating where along the edges of the cube to place each point, based on the scalar values of the two points along the edge (see Figure 12). Previously, each point was only placed in the middle of the edge (see Figure 3) which caused the mesh to become rough.

The result of both of these improvements is shown in Figure 13.

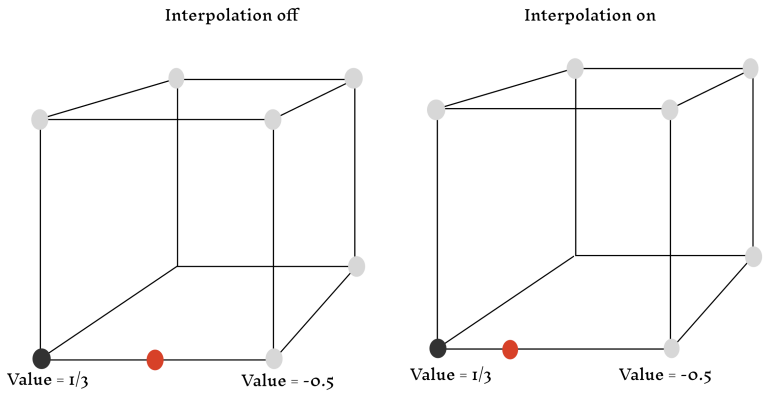


Figure 12: Visualization of internal interpolation in the Marching Cubes algorithm

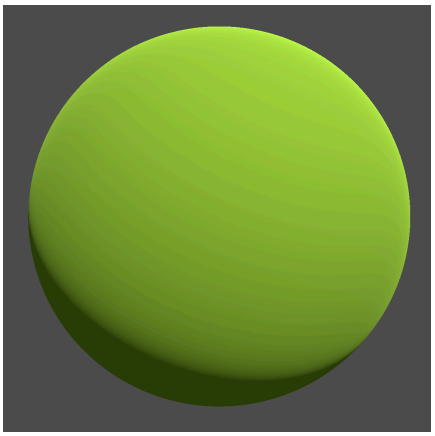


Figure 13: Smooth spherical planet with interpolation

4.1.5. fBm planets

The noise planets (Section 4.1.3) were a step in the right direction, but they lacked surface variation and resembled carved-out spheres. The initial idea to address this was to simply apply a height-map to all vertices as described in Section 4.1.1. However, this would potentially require three separate passes of the planet data: once to generate the data points, once during the Marching Cubes algorithm and finally, one more time to loop over all vertices of the newly created mesh. Furthermore, this approach would also eliminate the possibility of generating overhangs and more complex terrain features above the ground.

To address these issues, fractional Brownian motion (fBm) [41], or simply fractal Brownian motion, was introduced. It is a technique used, for example, in the computer graphics and video games industry for generating realistic-looking terrain [41], [42]. The idea is to layer several "octaves" (layers) of noise, each with increasing frequency and decreasing amplitude to produce smaller and smaller details.

Consider a sine wave: if one were to add another sine wave to the first one, the amplitude of both would be added together. If the second sine wave had a larger frequency than the first but a lower amplitude, then its "height contribution" would become smaller and would add smaller detail to the end result. Figure 14 shows the resulting graphs from the code below:

```
firstOctave = sin(x);
secondOctave = sin(x) + sin(2*x) * 0.5;
thirdOctave = sin(x) + sin(2*x) * 0.5 + sin(4*x) * 0.25;
```

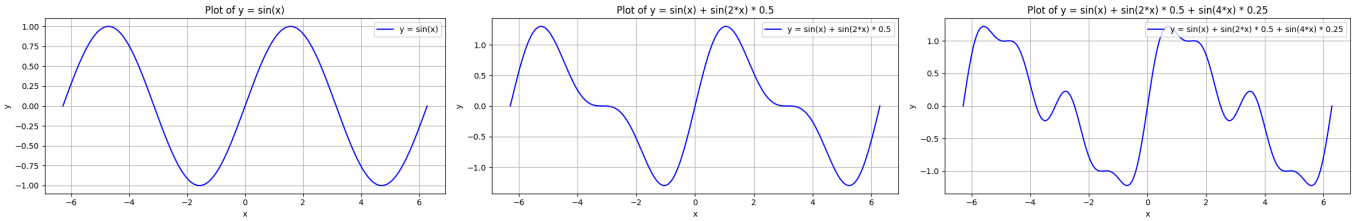


Figure 14: First octave, second octave and third octave, created using Matplotlib [43]

The lower octaves have the same appearance as the higher octaves but on a smaller scale, due to fBm using fractals and being self-similar, which means that lower octaves have the same appearance as the higher octaves at certain scales [44]. This property is useful when generating natural-looking terrain because it ensures that the detail will scale consistently and that the result will blend well together.

The implementation of fBm in this project was straightforward, and instead of using sine waves, 3D noise was used:

```
Fbm(valueToModify, pos, noise) {
  for (int i = 0; i < octaves; i++) {
      valueToModify += noise.GetNoise3Dv(frequency * pos + offset) * amplitude;
      amplitude *= persistence;
      frequency *= lacunarity;
      offset += new Vector3(random.Next(octaves));
  }
  return valueToModify;
}
```

Listing 1: Implementation of fBm

The fBm function was applied using the distance to the planet's surface as input, as this value is positive inside the planet and negative outside:

```
distanceToCenter = (centerPoint - currentPosition).Length();
distanceToBorder = radius - distanceToCenter;
points[x, y, z]  = Fbm(distanceToBorder, currentPosition, noise);
```

In the fBm function (Listing 1), there is a single for-loop which, for each octave, transforms the previously used scalar value by adding noise with increased frequency and decreased amplitude. Lacunarity is the factor by which the frequency should be multiplied each octave, and persistence is the factor by which the amplitude should be multiplied each octave. Typical values for these are `lacunarity = 2` and `persistence = 0.5` when generating natural-looking terrain [42]. To produce more varied terrain (not strictly realistic), both of these parameters were chosen randomly based on the logic described in Section 4.1.6.

Using fBm, the planets were improved further, with more varied terrain and, most importantly, variation in the elevation. The result of these changes is demonstrated in Figure 15 and Figure 16:

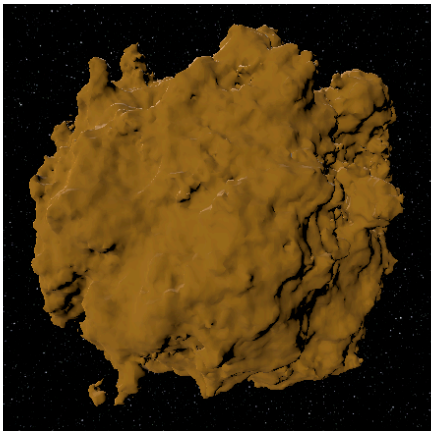


Figure 15: fBm planet

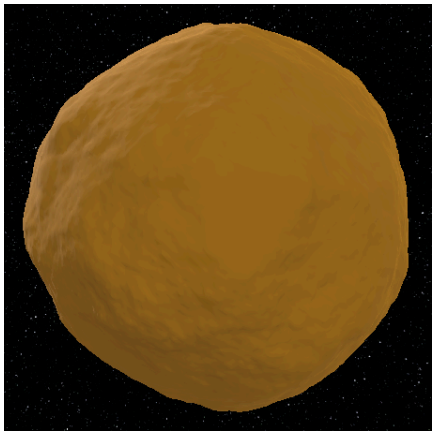


Figure 16: fBm planet with flat area

There were some issues with these planets, however, as shown in the center of Figure 16, where sometimes the scalar values at the edges got too large, cutting off areas prematurely, causing flat areas on the planet surface and occasionally square planets.

This issue was solved by introducing a fall-off parameter that reduced values closer to the edge:

```
falloffRatio = distanceToCenter / radius;
falloff = falloffRatio * falloffRatio * falloffStrength;
points[x, y, z] = Fbm(distanceToBorder, currentPosition, noise) - falloff;
```

The optimal fall-off strength had to be fine-tuned manually by balancing the frequency of flat areas with the planet's reduced size when using a large fall-off strength. Although this solution did not completely eliminate the problem, it significantly reduced both the frequency and severity of the flat regions.

Figure 17 depicts a particularly severe instance of the discussed problem and Figure 18 and Figure 19 demonstrate how the planet in Figure 17 transformed with different values of the fall-off strength. The method used to generate the planet's colors is explained later in Section 4.1.7.

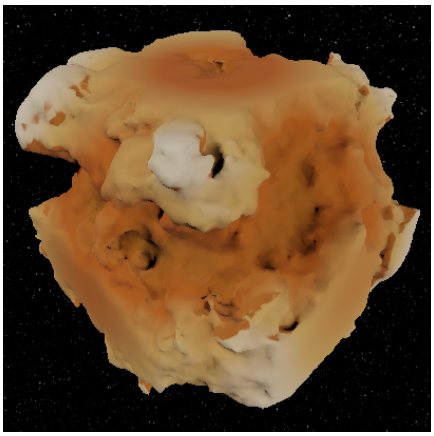


Figure 17: fBm planet without fall-off

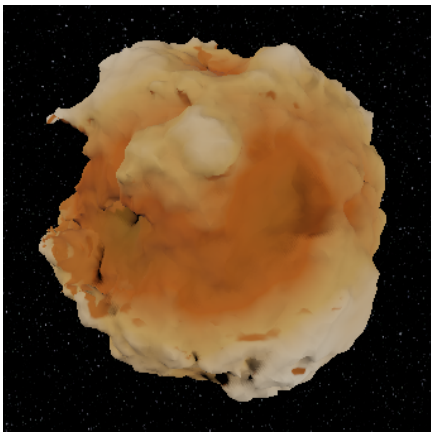


Figure 18: fBm planet with fall-off strength 8

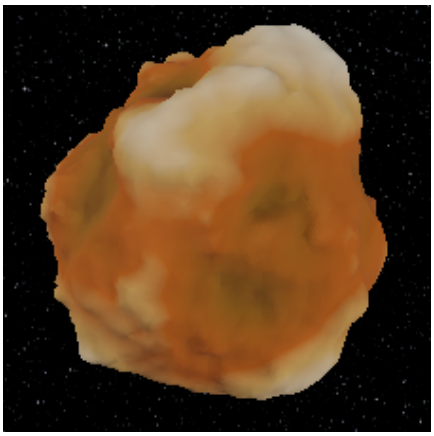


Figure 19: fBm planet with fall-off strength 32

4.1.6. Procedural planet generation

The next step after creating the planet generation was to procedurally generate the planets at runtime. This was done by manually experimenting with the fBm parameter values until ranges that produced visually satisfactory results were identified. Then, the parameters were randomized, according to the logic presented in the following pseudocode:

```
void RandomizeParameters() {
    Random random = new Random(planetSeed);
    // NextInt(a,b) returns a random int between a and b
    octaves = random.NextInt(4, 8);
    // NextFloat(a,b) returns a random float-value between a and b
    amplitude = random.NextFloat(0.0f, 20.0f);
    if(amplitude < 4)   frequency = random.NextFloat(0.0f, 12.0f);
    else                frequency = random.NextFloat(0.0f, 2.0f);
    if (frequency < 4)  lacunarity = random.NextFloat(0.0f, 4.0f);
    else                lacunarity = random.NextFloat(0.0f, 8.0f);
    persistence = random.NextFloat(0.1, 0.25);
}
```

However, this method was non-deterministic, causing planets to be generated differently on subsequent visits. To fix this, planets were generated using a seed derived from its solar system's seed (as later described in Section 4.8.3).

4.1.7. Coloring the fBm planets

The code for coloring the planets was reused from the first planet implementation, with one extension. Cliff edges could be simulated by calculating the dot product between the direction of a specific vertex normal and the direction to the planet center. If the resulting value was close to one, the corresponding fragments at that position were assigned the cliff color (see Figure 20). Some color themes were created to get aesthetically pleasing results (see examples in Figure 21).

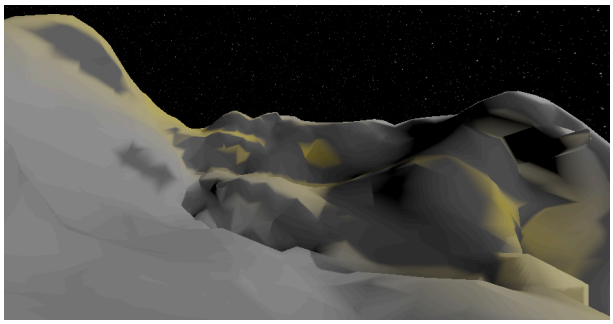
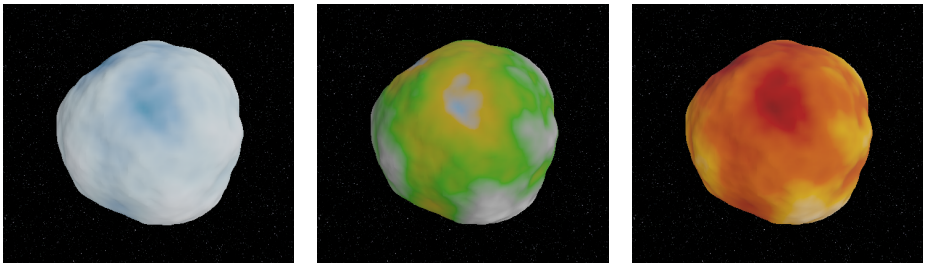


Figure 20: Cliff faces colored in dark gray



a) Ice world b) Forest world c) Lava or Red Desert world

Figure 21: A few select color themes available when generating the planets.

Each theme was assigned a warmth value in the range [0, 1] (0 for coldest, 1 for warmest) and grouped by these values. This facilitated temperature-based coloring, where a planet's color theme was determined by its distance from its sun, using the following normalized warmth formula to simulate temperature falloff:

$$\text{normalized\_warmth} = \frac{\text{planet\_warmth} - \text{min\_warmth}}{\text{max\_warmth} - \text{min\_warmth}}$$

4.2. Optimizing the Planet Generation

Replacing height-maps with Marching Cubes in mesh generation significantly increases computational demands due to the added dimensionality. This section describes how the planet generation was optimized to address this challenge.

4.2.1. Compute Shader

The first optimization step involved moving the Marching Cubes from the CPU to a compute shader on the GPU. Unlike the standard rendering pipeline, compute shaders are standalone programs invoked directly by the CPU, with custom-defined inputs [45].

The main motivation for this shift was to leverage the GPU's strong parallelization capabilities. Given that the Marching Cubes processes each scalar independently, it inherently lends itself to parallel execution, making the GPU well-suited for this task.

After transitioning the Marching Cubes algorithm to a compute shader, this GPU-based mesh generation approach was tested against its CPU counterpart. The testing involved feeding identical scalar fields to both the CPU and GPU implementations of the algorithm and measuring the time required to generate the resulting mesh. The tests were performed across a range of different sizes to ensure broader applicability of the results.

Surprisingly, the GPU approach performed worse (see Table 1). This was presumably due to overhead from buffer setup and data retrieval from the buffer, a common bottleneck in compute

shader workflows. Additionally, the triangle buffer was configured for the worst-case polygonization (up to five vertices per polygon(s)), which added to retrieval time.

In the end, this approach did not achieve the intended reduction in planet generation time; however, as alternative methods remain available, the focus shifted to exploring a different solution.

Table 1: Average time to generate vertices on PC-4 (Table A-4)

Scalar field size	**CPU**	**GPU**
32x32x32	1.34ms	98ms
48x48x48	3.33ms	116.5ms
64x64x64	8ms	172.67ms

4.2.2. Worker Thread Pooling

An alternative approach involved distributing the workload across multiple threads. In addition to the main thread, a dedicated thread was introduced to handle planet generation requests. Previously, all operations, including the loop that generated each planet during solar system creation, ran on the main thread. This caused performance issues, as the generation process led to noticeable stuttering; the frame could not advance until all meshes were fully generated. By offloading the generation tasks to a separate thread, the main thread could dispatch requests and proceed without delay.

To further optimize the two-threaded approach, a thread pool [46] was introduced to distribute planet generation across multiple threads instead of relying on a single one. A thread pool pre-allocates a set number of threads at startup; tasks such as planet generation are queued and handled by available workers. This improves performance by reusing threads and avoiding the overhead of constantly creating new ones.

This multi-threaded adjustment reduced stuttering between frames and allowed for a smoother experience (see Table 2).

Table 2: Frame time metrics on PC-4 (Table A-4) - Worker Thread Pool

	Average	**1% high**	**0.1% high**
From	4,21 ms	8,55 ms	40,1 ms
To	4,2ms	7,73 ms	33,74 ms

4.2.3. Chunking & Level-of-detail

A chunking system was implemented to optimize planet generation. To further boost performance, a level-of-detail (LOD) mechanism was added, reducing mesh complexity for distant planetary regions by using fewer scalar values, which is especially important when using the Marching Cubes algorithm. This speeds up loading as rendering less detailed areas is quicker.

The implementation works by having an *Octree* class, which is initialized with a size matching the planet's diameter, while the *OctreePlanetSpawner* class manages mesh generation. A resolution variable sets the number of scalar values per chunk (e.g., 32^3 for a resolution of 32). As the player nears a leaf node, it subdivides, unless it is at a predefined max depth (to avoid infinite subdivisions), generating higher-resolution meshes while disabling the parent mesh. Moving away reverses this process, with subdivisions removed and the parent mesh restored. The root node always remains intact. This dynamic adjustment ensures that only nearby regions are rendered in high detail, improving efficiency.

However, a limitation of the Marching Cubes algorithm with varying LODs is occasional gaps between chunks of different resolutions (as partially evident in Figure 23), as higher-resolution chunks capture more surface points while lower-resolution might miss them.

There are several possible solutions to address this issue. One approach involves the use of a skirt, which mitigates the problem by layering vertices along the edges of a chunk, effectively creating a surrounding skirt. Another method is stitching, where, in cases of adjacent chunks with differing LODs, the edges of neighboring chunks are connected to ensure continuity. A more advanced solution is the Transvoxel algorithm [47], specifically designed to handle transitions between chunks with varying LODs. However, due to time constraints, none of these solutions were implemented, and the issue remained unresolved.

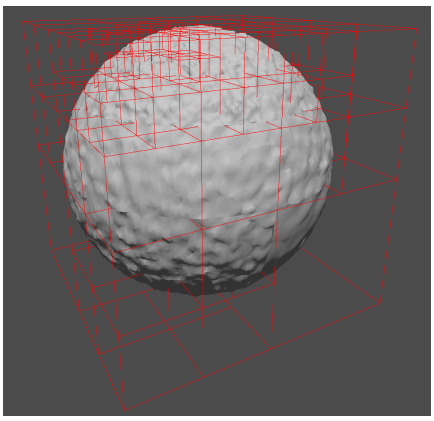


Figure 22: Octree planet outlining the chunks

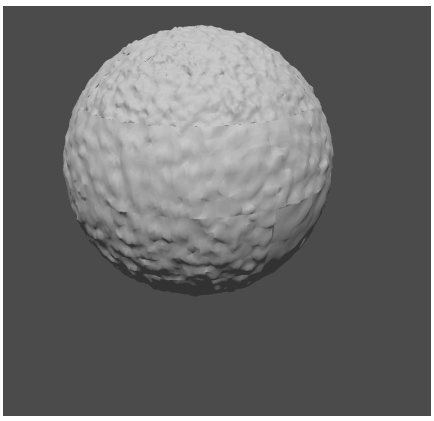


Figure 23: Octree planet showing the varying LODs

4.3. Planetary Features

This section describes the process of creating the additional planetary features, including surface details, surface features, oceans, and atmospheres.

4.3.1. Surface Details

Surface details, such as grass, are classified in *Population of a Large-Scale Terrain* [48] as elements rendered in close proximity to the player. Their positioning is not deterministic due to their high density and the player's limited attention to individual instances.

To generate surface details, the system first identifies the current chunk the player is in, along with adjacent chunks within a defined range. It then iterates through all triangles in the mesh data of each relevant chunk. For each triangle, a random point is generated within its bounds using barycentric coordinates [49].

Given a triangle with vertices $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$, and two random variables r_1 and r_2, a random point $\boldsymbol{d}$ within the triangle can be computed using the following formula [49]:

$$\boldsymbol{d} = (1 - \sqrt{r_1})\boldsymbol{a} + \sqrt{r_1}(1 - r_2)\boldsymbol{b} + \sqrt{r_1}r_2\boldsymbol{c}$$

This method ensures that the point $\boldsymbol{d}$ lies uniformly within the triangle. To determine the orientation of the surface detail, the normal vector of the triangle is calculated via the cross product of its edge vectors:

$$\boldsymbol{n} = (\boldsymbol{b} - \boldsymbol{a}) \times (\boldsymbol{c} - \boldsymbol{a})$$

The resulting normal vector $\boldsymbol{n}$ is then normalized. An orthonormal basis is constructed using the normal as the y-axis. The x-axis vector is chosen based on the vertical component of the normal to maintain numerical stability:

```
Vector3 upVector = normal;
Vector3 xVector;
if (Mathf.Abs(upVector.Y) < 0.99f)
  xVector = new Vector3(0, 1, 0).Cross(upVector).Normalized();
else
  xVector = new Vector3(1, 0, 0).Cross(upVector).Normalized();
```

The z-axis vector is obtained as the cross product of the x and y vectors, forming a complete orthonormal basis used to orient surface details.

To evaluate this approach, a simple plane mesh was utilized, yielding the following result:

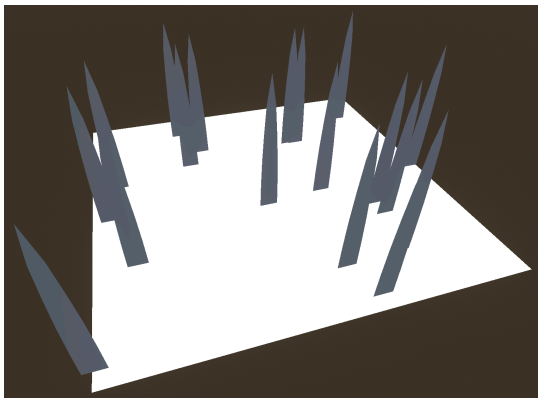


Figure 24: Simple Grass Spawning

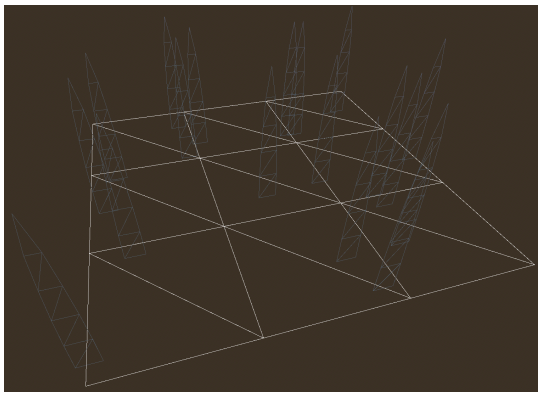


Figure 25: Wire-frame of Simple Grass Spawning

An initial issue encountered was that only a single surface detail was rendered per face (see Figure 24 and Figure 25), making the total number of rendered instances dependent on the face count of the mesh.

To address this, a density parameter was introduced to control the desired number of instances. In order to distribute the instances proportionally across the mesh, the total mesh area is first computed and then multiplied by the density value to determine an overall target number of instances. For each face, the fraction of the total area is multiplied by the total target to obtain an "ideal instance count."

In certain cases, the combination of area and density parameters may yield an instance count of less than 1. To address this, when the instance value is below 1, a random check decides whether to add a single instance or none. Otherwise, the count is rounded to an integer. This approach distributes instances proportionally to each face's area.

```
int totalInstanceTarget = Mathf.Max(1, Mathf.FloorToInt(totalArea * density));

float faceAreaRatio = faceArea / totalArea;
float idealInstanceCount = totalInstanceTarget * faceAreaRatio;

int faceInstances;
if (idealInstanceCount < 1.0f)
{
  // Probabilistic approach - chance equals the fractional ideal count
  faceInstances = GD.Randf() < idealInstanceCount ? 1 : 0;
}
else
{
  // Round to nearest integer instead of floor for better distribution
  faceInstances = Mathf.RoundToInt(idealInstanceCount);
}
```

By iterating through each face and applying this calculation, the number of instances is determined and their positions and orientations are computed accordingly. With the density parameter set to 10, the resulting distribution is shown in Figure 26.

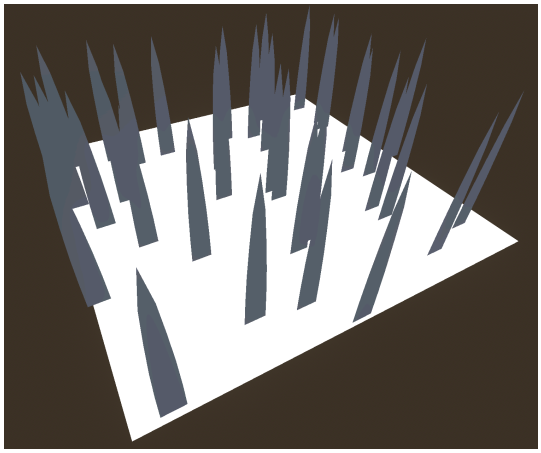


Figure 26: Density-Based Grass Spawning

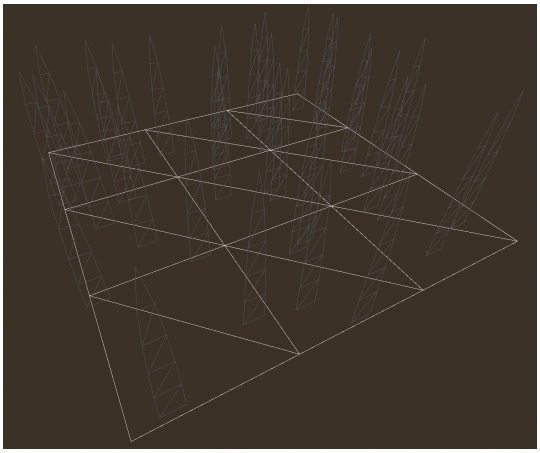


Figure 27: Wire-frame of Density-Based Grass Spawning

Applying a grass shader to the grass blade meshes resulted in enhanced surface detail on the planet, as demonstrated in the final visual output.

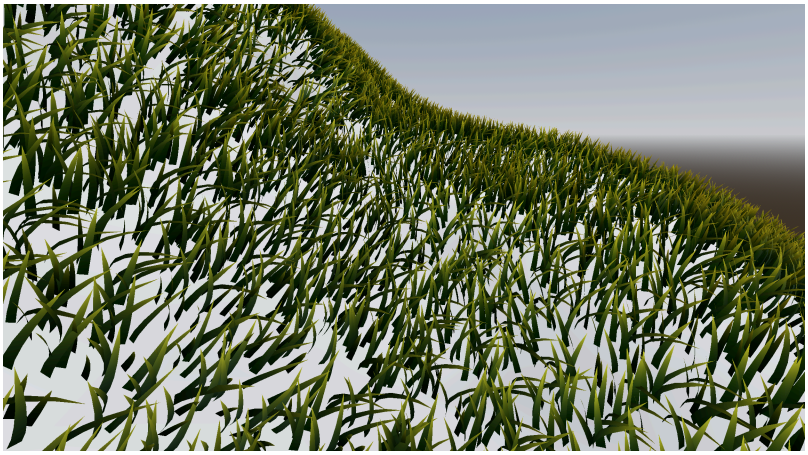


Figure 28: Grass on a planet

4.3.2. Surface Features

Surface features, such as trees, are classified in *Population of a Large-Scale Terrain* [48] as elements that must remain visible from a distance and require consistent placement. Positional irregularities in these features are easily noticeable and can disrupt visual coherence.

Unlike surface details, surface features must remain visible at greater distances and cannot rely on dense mesh data, which is incompatible with the octree-based terrain system. The reduced triangle count at lower octree resolutions renders geometry-dependent techniques ineffective and inconsistent.

To address this, an alternative approach based on ray-casting was implemented. Rays are cast inward from randomly sampled points on the planet's axis-aligned bounding box (AABB). The AABB is a rectangular box that encompasses an object, aligned with the coordinate axes (x, y, z). Upon terrain intersection, hit positions and surface normals are extracted to procedurally instantiate surface features using a construction pipeline similar to Surface Detail but decoupled from mesh density.

Points on the AABB are generated using uniform random sampling with bilinear interpolation across each face with the following formula:

$$\boldsymbol{p} = \boldsymbol{v_1} + u(\boldsymbol{v_2} - \boldsymbol{v_1}) + v * (\boldsymbol{v_4} - \boldsymbol{v_1})$$

where v_1, v_2, and v_4 are the three vertices defining an AABB face, and u and v are random values in the range $[0, 1]$. This assumes rectangular faces, which is true for AABBs.

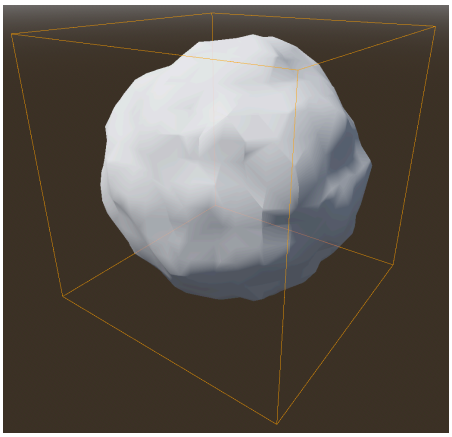


Figure 29: The AABB (marked yellow) encapsulating the planet

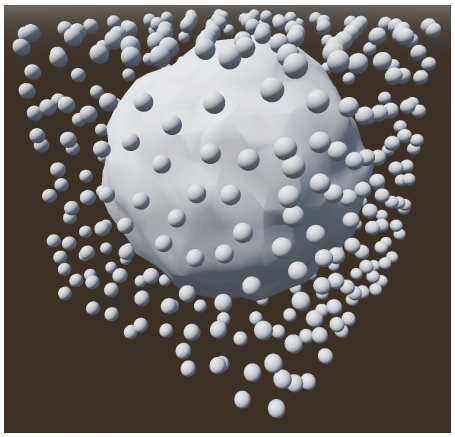


Figure 30: Randomized points on the AABB

Although this uniform sampling technique is computationally efficient and straightforward to implement, it tends to produce uneven spatial distributions. Specifically, it may result in clustering of points in some regions and sparse coverage in others—an outcome that is undesirable when attempting to ensure consistent surface feature placement across the planetary surface (see Figure 31).

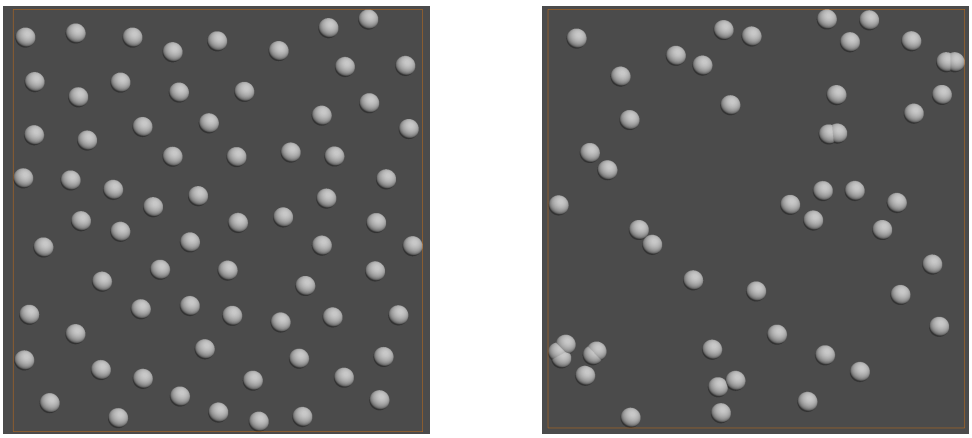


Figure 31: Poisson (left) vs Uniform (right) distribution

To achieve a more spatially uniform distribution of points, the sampling method was subsequently replaced with Poisson-disc [50] sampling. Unlike uniform sampling, Poisson-disc sampling ensures that each point is separated by a minimum distance r, thereby avoiding clustering and producing a more even distribution of samples. The algorithm follows the method described in Fast Poisson Disk Sampling in Arbitrary Dimensions [50]. It operates as follows, with the input parameter r representing the minimum distance and the constant k indicating the number of samples before rejection:

Step 1. Initialize a background grid with cell size $\frac{r}{\sqrt{n}}$ where n is the dimensionality of the space (in this case, $n = 3$).

Step 2. Randomly select an initial sample within the domain and insert it into both the active list and the background grid.

Step 3. While the active list is non-empty:

- Randomly choose a sample from the active list.
- Generate up to k candidate points within an annulus of radius r to $2r$ around the chosen sample.
- For each candidate:
  - Reject it if it lies outside the domain or is closer than r to any existing sample.
  - If it is valid, add it to both the active list and the grid.
- If none of the k candidates are accepted, remove the sample from the active list.

After establishing a more spatially uniform distribution of points, the next step involved determining an effective method for distributing varied environmental features such as large trees, small trees, and rocks.

One effective approach for achieving this distribution is the Alias method [51]. The Alias method enables sampling from a discrete probability distribution in constant time. Given a list of n probabilities, it allows for the selection of an index i, where $1 \leq i \leq n$, according to the specified distribution.

For example, consider a case where three types of features are to be placed in a scene: large trees, small trees, and rocks. Suppose the desired probabilities for selecting each feature are as follows:

- Large trees: 50%
- Small trees: 30%
- Rocks: 20%

Using the Alias method, this distribution is encoded into two tables—one for probabilities and one for aliases. At runtime, a uniformly random index is selected, and the final outcome is determined using the stored values, thereby ensuring that features are placed according to the specified probabilities in an efficient manner.

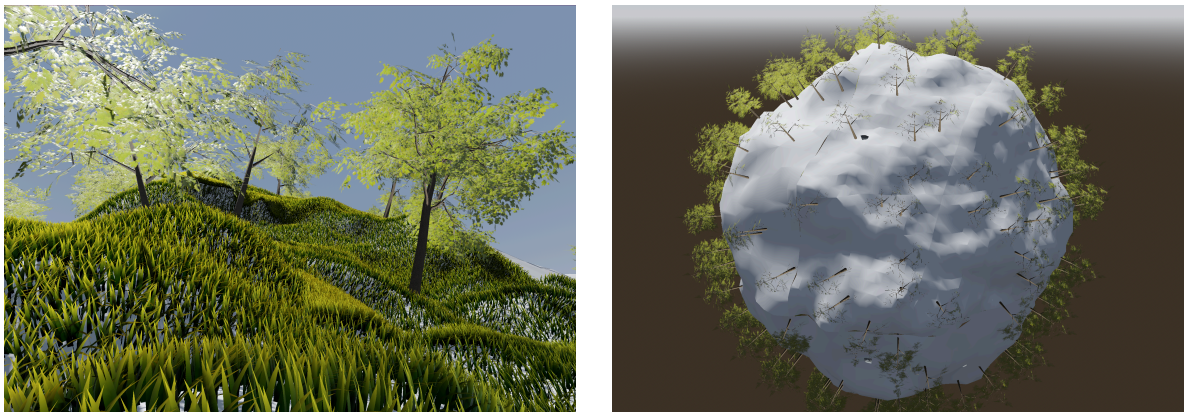


Figure 32: Surface features on a planet. Tree asset by user *LOLIPOP*, via sketchfab.com, licensed under CC BY 4.0.

4.3.3. Oceans

The ocean layer was implemented as a simple blue sphere, expanded from the planet's center to some configurable radius (typically equal to or slightly less than the planet radius). This sphere then intersects with the planet terrain, creating an ocean. However, only having a smooth blue sphere as water was deemed not visually appealing, thus a water shader was implemented.

Surface motion was implemented using bump mapping [52], scrolling two separate noise textures in different directions on top of the water surface, and using them as normal maps [52] to displace each vertex normal during lighting calculations, creating visual "bumps" on the surface. To add actual wave movement, displacement mapping [13] was used to physically alter the vertices of the mesh using another noise texture as a height-map. Figure 33 shows the result of both of these techniques.

One challenge with applying these techniques to a sphere (rather than flat terrain) is texture seams and pinching at the poles (see Figure 34). Godot's support for seamless noise resolved the seam issue. To address polar pinching, one solution was found: triplanar mapping [53]. However, since the pinching is mostly hidden by terrain and rarely visible (and in the interest of time), the issue was considered acceptable and left unaddressed. Triplanar mapping was later implemented in Section 4.6.3 but was not applied to the ocean, since it used a mesh without built-in support.

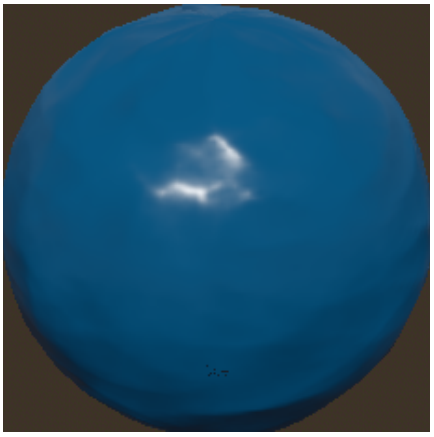


Figure 33: Ocean layer by itself

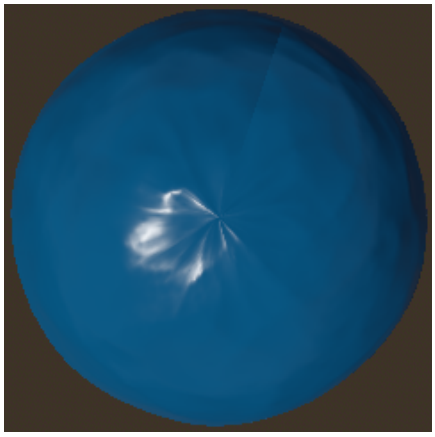


Figure 34: Visible pinching on the ocean's poles

Another glaring issue was flickering due to the ocean and the atmospheres (Section 4.3.4) clashing. This was resolved by making the atmosphere have a higher render priority than the ocean layer, meaning that the ocean gets rendered before the atmosphere.

The oceans looked cohesive with the planets, so the next step was to make the water transparent and to color the water based on its depth. This was achieved by using a combination of the depth texture [54] to calculate the distance of each pixel to the water surface, and the screen texture [55] to blend the planet's surface color with the water color. The water color was also divided into two separate colors, one brighter color for shallow parts and one darker color for deeper parts. Using the Godot built-in function for blending two colors [56], the final per-pixel water color was calculated:

```
vec3 water_depth_color = mix(deep_color, shallow_color, smooth_shallow_depth);
vec3 transparent_water_depth_color = mix(screen * smooth_shallow_depth,
water_depth_color, transparency_blend);
```

Finally, the color around the edges of the ocean (where the ocean surface meets the terrain), was colored white to appear as foam. This was done by blending the previously calculated color with a foam color:

```
vec3 transparent_water_depth_color_with_foam = mix(transparent_water_depth_color,
foam_color, smooth_foam_depth);
```

Put together, the ocean greatly improved the visuals of the planets. The result is demonstrated in Figure 35.

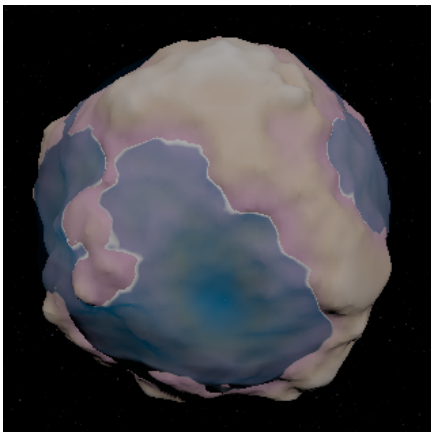


Figure 35: Planet with ocean and water shader

4.3.4. Atmospheres

Atmosphere development began with research into existing solutions, resulting in two approaches: a simpler method by Martin Donald [57] and a more physically accurate one by Sebastian Lague [58]. Both implementations utilize a post-processing shader on a cube with flipped faces.

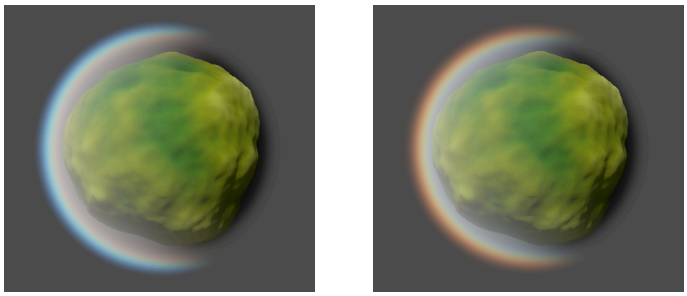
The simpler version was implemented using ray-sphere intersections to create a transparent, uniformly colored, sphere around the planet (see Figure 36). The color remains uniform since it does not account for the sun's position. Attempts to improve shading involved calculating the dot product between each vertex normal and the sun ray directions, meant to influence the resulting color. However, due to issues with different coordinate spaces and fetching the sun's position, this was unsuccessful and caused a shift towards the more realistic approach.

Figure 36: Implementation of a simple atmosphere

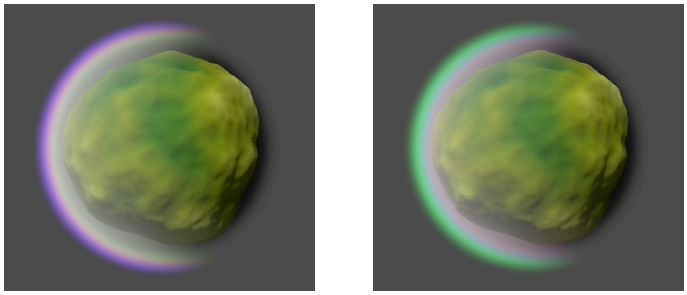
The second iteration aimed for a more physically accurate atmosphere involving Rayleigh scattering. Rayleigh scattering is a physical phenomenon that describes how light interacts with particles smaller than its wavelength [59], allowing for atmospheric light scattering, density falloff with altitude, and sunsets. The algorithm roughly works by approximating light scattering along a ray cast from the camera through the atmosphere, using a series of sampling points for optical depth and scattering calculations.

The color is based on a 3D vector representing different light wavelengths corresponding to the different parts of the visible light spectrum. However, basing it on only Rayleigh scattering limited the achievable color options. For example, a red Mars-like atmosphere was not possible through this method alone. To address this, one of the wavelengths was set lower than the others, amplifying the

scattering of that wavelength, and thereby altering its color. Finally, a set of preset wavelength vectors were created and are chosen at random as the planets are generated. See Figure 37.



a) Blue atmosphere b) Orange Atmosphere



c) Purple atmosphere d) Green atmosphere

Figure 37: Main colors of the different atmosphere variations.

Initially, atmospheric rendering was performance-heavy due to costly light scattering calculations. Two solutions were considered: offloading calculations to a compute shader, and implementing a LOD system. The latter was chosen for its simplicity, and is done by dynamically reducing scattering and optical depth samples with distance to the player.

Originally set to 30, the number of sampling points along the rays traveling through the atmosphere caused significant performance drops. After testing, a sample count of 10 was found to balance visual fidelity and performance effectively, reducing average frame time from 132.2 ms (at 30 samples) to 18.56 ms in a single-planet benchmark on PC-3 (see Table A-3). Furthermore, with the LOD in place, the amount of sampling points reduces from the set maximum value, downwards to 1, as the distance to the player increases. This further increases performance since not all atmospheres in a system have to render at full quality at the same time.

4.4. Player Controller

The player controller was initially implemented as a flying camera for free exploration of the galaxy, without collision or surface interaction. After terrain generation was completed, support for planetary landings began its development. This required simulating local gravity, surface-aligned movement, and jumping.

Planetary gravity fields were implemented using an Area3D node [60] with a spherical collision shape (see Figure 38). This node allows any physics body that enters its collision shape to inherit the gravity direction and strength set by the Area3D. The gravity direction is calculated by subtracting the camera's world position from the planet's world position. This is updated during each physics step to allow the direction to always point towards the planet center.

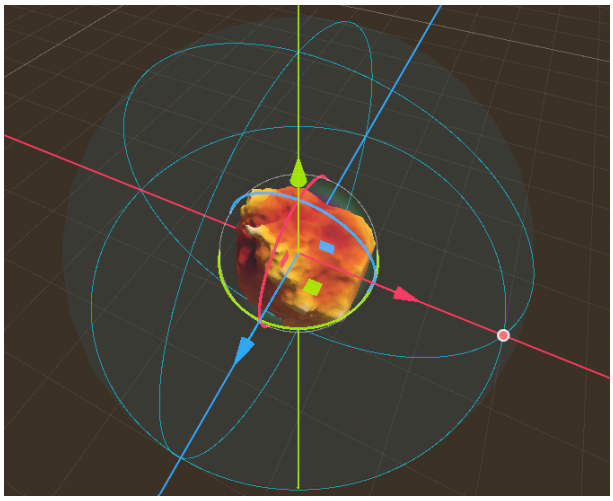


Figure 38: Area3D node with collision shape around a planet

To facilitate exploration and simulate orbital behavior, the player inherits a planet's total velocity upon entering its gravitational field, ensuring they remain in orbit. The final step was implementing surface landing and movement mechanics.

Rotating the player while moving on the planetary surfaces was achieved by linearly interpolating the player's basis toward a target basis with its 'up' vector aligned against the gravity vector. Finally, the ability to jump was implemented by adding an impulse along the opposite direction of the gravity vector.

Several issues emerged during testing. At high velocities, the player could fall off planets, which was resolved by lowering the base speed. Uncontrollable bouncing, likely caused by uneven terrain at high speeds, was mitigated by adding a downward raycast to support the collision system. Furthermore, many issues were ultimately traced to planets moving at high speeds while the player was also in motion, combined with inaccurate planetary collision shapes.

4.5. Physics Engine

Simulating the gravitational interactions within a galaxy, containing a vast number of stars and planets, presents a significant computational challenge known as the N-body problem [61]. The goal is to calculate the net gravitational force acting on each body at discrete time steps and use this information to update their positions and velocities over time. This section details the progression of methods implemented to tackle this problem within our project, moving from a simple baseline to an optimized approximation algorithm, and discusses the performance analysis that guided these choices.

4.5.1. Direct Summation

One approach to solving the N-body problem is the direct summation method. This technique relies directly on Newton's Law of Universal Gravitation [62], calculating the gravitational force between every pair of particles in the system.

The force $\vec{F}_{12}$ exerted on particle 1 by particle 2 is given by:

$$\vec{F}_{12} = G\frac{m_1 m_2}{|\vec{r}_{12}|^3}\vec{r}_{12} \quad \text{where} \quad \vec{r}_{12} = \vec{r}_2 - \vec{r}_1$$

The total acceleration $\vec{a}_i$ on particle i is the sum of accelerations caused by all other particles $j \neq i$ in the system, as illustrated in Figure 39:

$$\vec{a}_i = \sum_{j \neq i} G \frac{m_j}{|\vec{r}_{ij}|^3} \vec{r}_{ij}$$

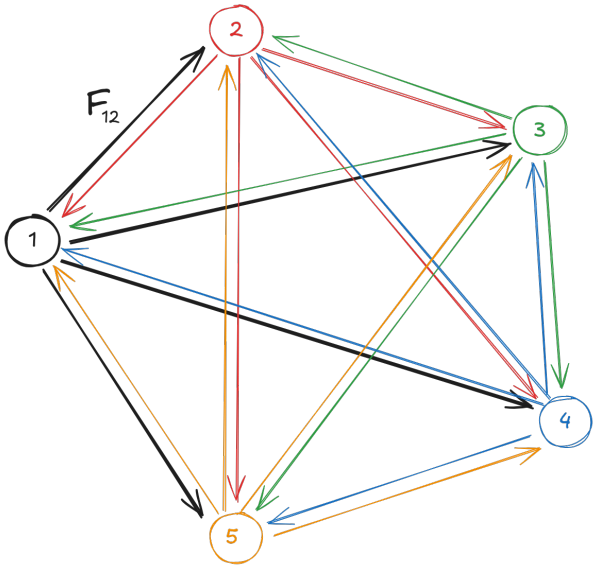


Figure 39: Diagram illustrating pairwise force calculation between five bodies

This requires $\frac{N(N-1)}{2}$ pairwise calculations per time step, resulting in a computational complexity of $O(N^2)$. Our implementation used nested loops as illustrated in the pseudocode in Appendix B.

While simple and accurate, the $O(N^2)$ complexity made direct summation computationally prohibitive for large N within the target performance goals, necessitating an approximation method.

4.5.2. Barnes-Hut Approximation

To efficiently simulate large numbers of bodies, the Barnes-Hut algorithm was implemented, reducing the computational complexity to $O(N \log N)$ [63]. The core idea was to use an octree [22] (described in Section 2.4.1) to group distant particles together. The gravitational influence of these groups was then approximated by treating the group as a single point mass located at the group's center of mass (CoM). This approximation leveraged Newton's shell theorem [62] and is effective when the distance to the group is large compared to the group's size.

In dynamic N-body simulations, constantly changing particle positions quickly invalidate the octree, typically requiring reconstruction each time step (physics frame). This demands an efficient construction algorithm, for which a **Morton-code-based linear octree** [64] was chosen, enabling:

1. **Parallelizable Steps:** Both calculating Morton codes and sorting particles are trivial to parallelize.
2. **Cache Efficiency:** Processing spatially local data sequentially after sorting can lead to better CPU cache utilization compared to pointer-chasing in traditional octree implementations.
3. **Efficient Partitioning:** The sorted order allows for fast partitioning of particles into child nodes using binary search rather than geometric tests.

The construction process proceeds as follows:

1. **Morton Codes:** A 64-bit Morton code is calculated for each particle by mapping a particle's 3D position within the global simulation bounds to a 1D integer by interleaving the bits of its scaled coordinates (see Figure 40). This mapping largely preserves spatial locality since nearby particles

tend to have numerically close Morton codes [65]. This step is parallelized using `rayon` [66] for particle counts larger than a set threshold, determined via benchmarking (see Section 4.5.4).

2. **Sorting:** Particles are sorted based on their Morton codes, effectively grouping spatially adjacent particles in memory, as illustrated in Figure 41. The sorting is parallelized using `rayon`, providing significant speedup.

3. **Tree Construction:** An explicit linear tree structure is built recursively from the sorted list of particles. The construction works by dividing the particle list into smaller ranges:
   - If a range contains one particle or the maximum depth is reached, a leaf node is created, and its `GravityData` (mass and center of mass) is computed.
   - For internal nodes, the range is partitioned into 8 sub-ranges (octants). This is done efficiently by performing a binary search on the sorted Morton codes, checking the relevant 3 bits at the current depth to find the split points.
   - Recursive calls are made for each non-empty octant, and parent nodes aggregate `GravityData` from their children.

   The tree-building step is run sequentially, but there's potential for parallelization.

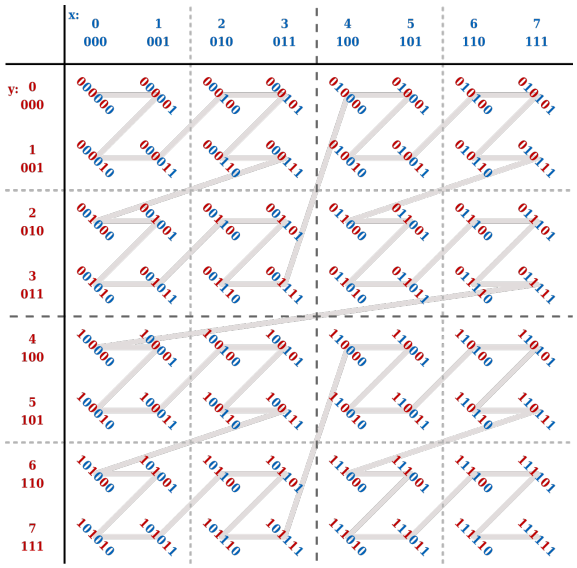


Figure 40: 2D version of Morton codes, showing how interleaving works. Nomen4Omen, CC BY-SA 4.0

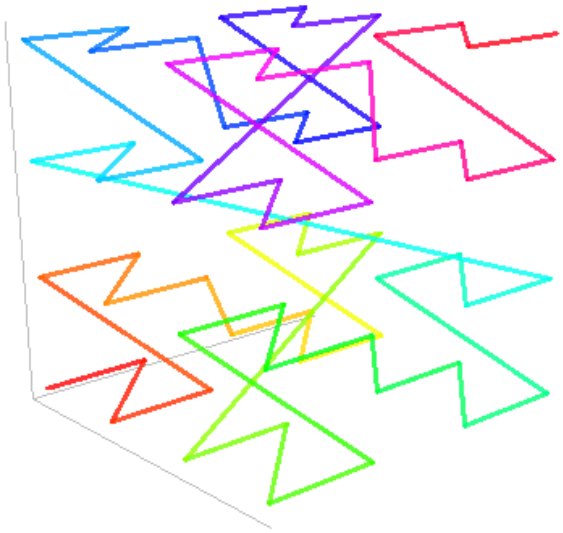


Figure 41: How points are ordered in 3D when sorted using Morton codes. Robert Dickau, CC BY-SA 3.0

The `Node` struct stores the necessary information for the algorithm:

```
struct Node {
  bounds: BoundingBox,
  children: [Option<NonZeroUsize>; 8],
  body_range: Range<usize>, // Range in sorted_indices
  depth: u32,
  data: GravityData, // Mass, CoM
}
```

Once the octree is built for the current time step, each particle's acceleration is computed by traversing the tree, using the *Multipole Acceptance Criterion (MAC)* [63]. This criterion determines whether a node (a region of space containing particles) is sufficiently far from the target particle to be approximated as a single point mass. The principle is illustrated in Figure 42, which uses a 2D quadtree for clarity; an octree applies the same logic in 3D by subdividing space into eight octants instead of four quadrants.

The MAC involves these steps during tree traversal for a target particle:

1. Compute the distance d between the target particle and the current node's center of mass (CoM).
2. Determine the node's characteristic size s (e.g., bounding box width).
3. If $s^2 < \theta^2 d^2$ (where θ is a predefined threshold parameter, typically between 0.5 and 1.0), the node is considered "far enough." Its gravitational effect on the target particle is then approximated using the node's total mass and CoM, and the traversal down this branch of the tree stops.
4. If the node is "too close" ($s^2 \geq \theta^2 d^2$):
   - If it is an internal node, the algorithm recursively traverses its non-empty children, applying the MAC test to each.
   - If it is a leaf node, direct summation (Section 4.5.1) is performed between the target particle and each particle within that leaf. A softening factor ε is used in these calculations (d^2 is replaced by $d^2 + \varepsilon^2$) to prevent numerical instability from particles being too close.

Because each particle's traversal is an independent tree traversal, this process is trivially parallelized using `rayon`, greatly improving performance:

```
accelerations = (0..particles.len())
    .into_par_iter() // Only need this line to pararellize computation
    .map(|i| octree.calculate_accel_on_particle(g, i))
    .collect()
```

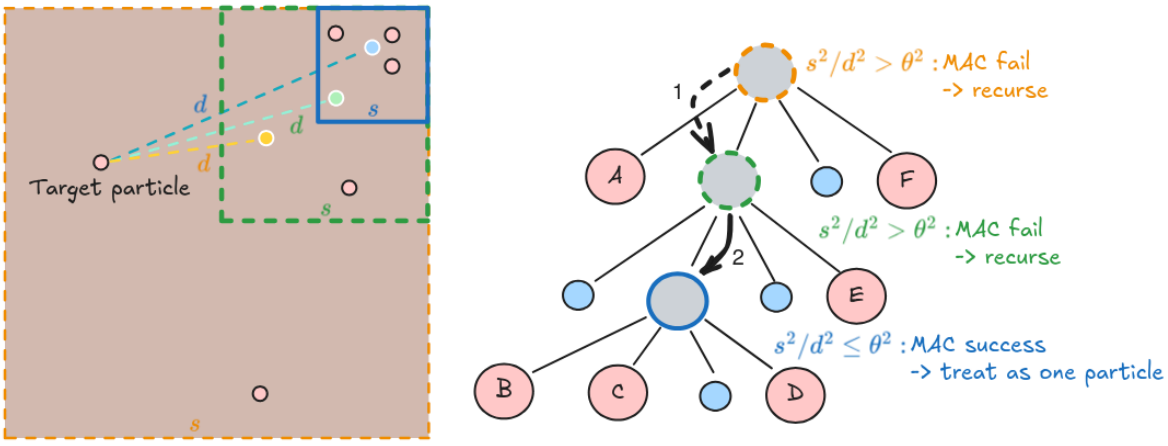


Figure 42: Illustration of the Barnes-Hut Multipole Acceptance Criterion (MAC). Left: Spatial view showing a target particle and nested quadtree cells. The ratio of cell size (s) to distance (d) from the target particle to the cell's Center of Mass determines if the cell can be approximated. Right: Corresponding tree traversal. For each node, if $\frac{s^2}{d^2} \geq \theta^2$ (MAC fail), the algorithm recurses to its children. If $\frac{s^2}{d^2} < \theta^2$ (MAC success), the node's entire mass is treated as a single point particle for force calculation.

4.5.3. Integration

Finally, the calculated accelerations (whether from Direct Summation or Barnes-Hut) were used to advance the simulation state via numerical integration. The implementation used the Forward Euler method [67] (specifically, symplectic Euler, which offers better long-term stability for orbital mechanics [68]) to update velocities and positions based on the accelerations computed in parallel for the current time step Δt:

$$v_{\text{new}} = v_{\text{old}} + \vec{a} \cdot \Delta t \quad \texttt{(body.vel += acc * delta)}$$

$$p_{\text{new}} = p_{\text{old}} + \vec{v}_{\text{new}} \cdot \Delta t \quad \texttt{(body.pos += body.vel * delta)}$$

This combination of parallelized efficient octree construction via Morton codes, $O(N \log N)$ force calculation with parallel traversals, and simple Euler integration allowed the simulation of large-scale galactic systems.

4.5.4. Performance Benchmarking and Threshold Tuning

To guide optimization and inform decisions about algorithms and parallelization, rigorous performance benchmarking was conducted on the core components of the physics engine. The Rust library criterion [69] was used, offering statistical insights through repeated runs, regression detection, and detailed reports.

The benchmarks focused on performance under varying workloads, particularly different numbers of simulated bodies (N). Key tests compared acceleration calculations (e.g., Direct Summation vs. Barnes-Hut) and parts of the Morton-based octree construction. Consistent synthetic test data ensured reproducibility across runs.

1. **Algorithm Selection Thresholds for Force Calculation:** The key results for the Direct Summation vs. Barnes-Hut benchmark are summarized in Figure 43. This graph plots the average computation time (in milliseconds, on a logarithmic scale) against the number of bodies for four variants: sequential Direct Summation (`direct/sequential`, yellow line), parallel Direct Summation (`direct/parallel`, blue line), sequential Barnes-Hut (`barnes_hut/sequential`, green line), and parallel Barnes-Hut (Morton-based octree, `barnes_hut/parallel`, red line). Each data point represents the average of 100 samples from benchmarks run on PC-1 (see Table A-1). Since these benchmarks are hardware-specific, the established performance thresholds might vary on other systems, especially regarding the number of CPU cores. This visualization was crucial for determining these thresholds.

   - For very small numbers of bodies ($N < 100$), the `direct/sequential` method (yellow line) is the most performant. Its low intrinsic overhead makes it ideal for these scenarios, despite its $O(N^2)$ complexity. All other algorithms show an overhead, especially the parallel ones.
   - As N increases beyond approximately 100, the `direct/parallel` method (blue line) surpasses sequential direct summation and also remains faster than `barnes_hut/parallel` (red line) for a significant range. This is because the parallelization of the N^2 calculations effectively utilizes multiple cores, and this benefit outweighs the Barnes-Hut octree construction overhead until the N^2 factor becomes too dominant.
   - The final crossover occurs at approximately $N = 440$, where the `barnes_hut/parallel` method (red line) becomes the most efficient. Beyond this point, the $O(N \log N)$ complexity of Barnes-Hut, combined with parallelism, provides superior performance over both direct summation variants.

These results directly informed the dynamic switching system in the physics controller. Based on the number of bodies, the simulation adaptively selects the best-performing algorithm: sequential for few bodies, parallel direct sum for moderate counts, and parallel Barnes-Hut for large numbers. The 60 FPS threshold in the diagram illustrates their real-time viability.

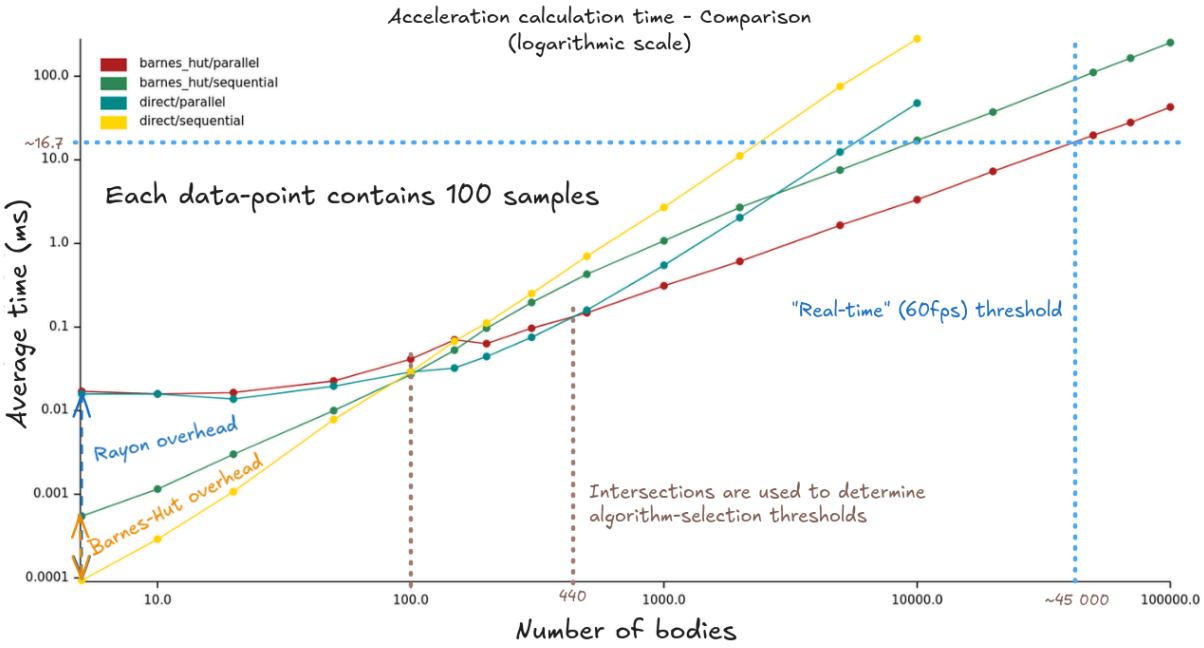


Figure 43: Criterion benchmark results for N-body acceleration calculations, illustrating the performance crossover points that informed the selection of algorithm-switching thresholds. Average time per 100 samples (ms, log scale) vs. number of bodies. Key thresholds for Direct Summation vs. Barnes-Hut, and parallelization overheads are noted.

2. **Parallel Morton Encoding Threshold:** This benchmark group specifically compared the performance of calculating Morton codes sequentially versus in parallel using rayon. The results are visualized in Figure 44. This graph plots average encoding time against the number of bodies, for both sequential and parallel implementations.

   As the diagram illustrates, for smaller numbers of particles, the sequential encoding is faster due to the overhead associated with initializing and managing parallel tasks. However, as the number of bodies increases, the benefits of parallel computation become apparent. The intersection point, where the parallel version starts to outperform the sequential one, is around $N \approx 4000$. Based on these empirical results, for particle counts below 4000, Morton codes are calculated sequentially, while for counts at or above 4000, the parallel `rayon` implementation is utilized to leverage multi-core processing.

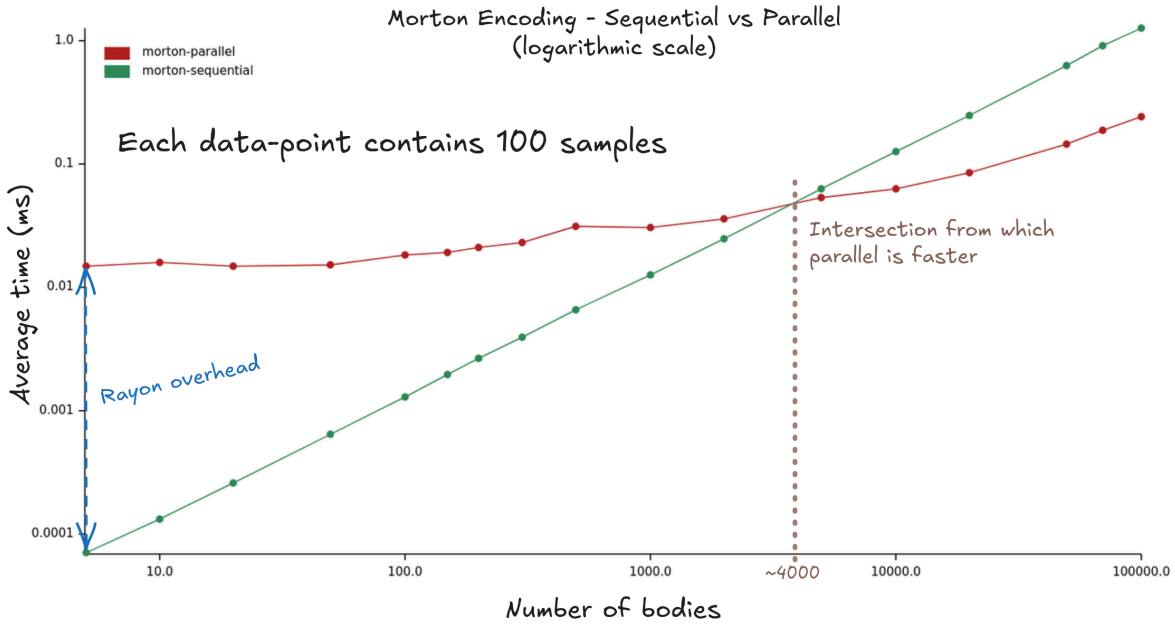


Figure 44: Morton encoding criterion benchmark results demonstrating the benefit of parallelization for larger datasets. Average time per 100 samples (ms, log scale) vs. number of bodies. Parallel encoding surpasses sequential performance around N ≈ 4000, after overcoming initial parallelization overhead incurred by rayon.

In summary, using `criterion` for systematic benchmarking was crucial for optimizing the physics engine. It provided the quantitative data necessary to justify algorithmic choices and fine-tune parameters such as parallelization and algorithm-switching thresholds, resulting in a more performant and scalable simulation.

4.5.5. Trajectory Simulation and Visualization

To understand orbital dynamics and aid in system design, a trajectory simulation system was implemented to predict and visualize the future paths of celestial bodies. This system runs a separate N-body simulation for a configurable number of future steps and time increment, using the same core physics logic (Direct Summation or Barnes-Hut) and semi-implicit Euler integration as the main simulation. The resulting sequences of future positions for each body are rendered as colored line strip meshes. Trajectories can also be calculated relative to a central body.

Given the computational cost, especially for many steps or bodies, trajectory calculations are offloaded to a background thread. The main thread sends messages to this worker and retrieves results asynchronously using a queue-and-poll mechanism. This prevents the main game loop from freezing during intensive calculations. The worker is designed to process the latest request if multiple are queued.

The accuracy of these predicted trajectories is subject to the same numerical errors as the primary Euler integration, accumulating with the number of steps; smaller time deltas improves accuracy at the cost of computation. However, it was observed that less precise trajectories (larger time delta) tended to overestimate instability, providing a useful heuristic: if a system appeared stable with coarse predictions, it was generally stable in practice.

This trajectory visualization proved invaluable for the System Generation process (Section 4.6), allowing for iterative tuning of orbital parameters to achieve stable or aesthetically desirable

configurations. It served as a key diagnostic tool for debugging physics and visually confirming the immediate future dynamics of generated solar systems.

4.6. System Generation

System generation involved deterministically generating stable and aesthetically plausible solar systems and their contained celestial bodies. This included calculating orbital positions for planets and moons to ensure long-term stability, and assigning unique seeds to each celestial body for reproducible procedural generation.

The generation of systems should balance realism, aesthetics, and stability. Generating stable systems with realistic distances lead to planets appearing too small to be visible, which deteriorates the game play experience. On the contrary, smaller distances with large planets could lead to instability, specifically for moons. The moons' orbit radius increase with the planets' radius, increasing the risk of moons interacting neighboring planets.

To verify a system's stability, the previously implemented trajectories (see Section 4.5.5) were used (see Figure 45). However, these trajectories were not entirely accurate; numerical errors accumulate over time, meaning that initially stable orbits may eventually become unstable.

These errors tended to grow when bodies undergo large transformations between physics steps. The errors were reduced by utilizing a higher physics frame rate and slowing down the orbit speeds. Conversely, faster orbits could lead to a higher accumulation of errors, making stable systems appear unstable in the simulation.

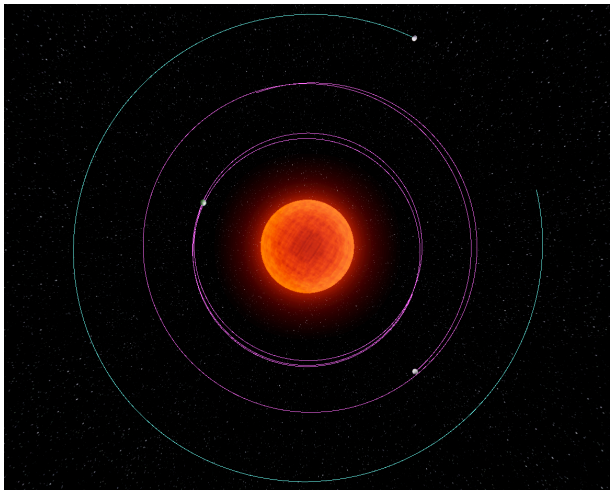


Figure 45: Three planet system with visible trajectories.

4.6.1. Solar System Stars

The stars were at first implemented as simple yellow spheres with constant masses and fixed positions and radiuses. Other than the colors, the constant variables would not be randomly generated as the stars would not physically interact with other stars.

To increase variety between solar systems and increase visual interest, a star shader snippet [70] was found online and implemented. Additionally, an array of star color presets was defined, from which a color was randomly selected during generation (see Figure 46).

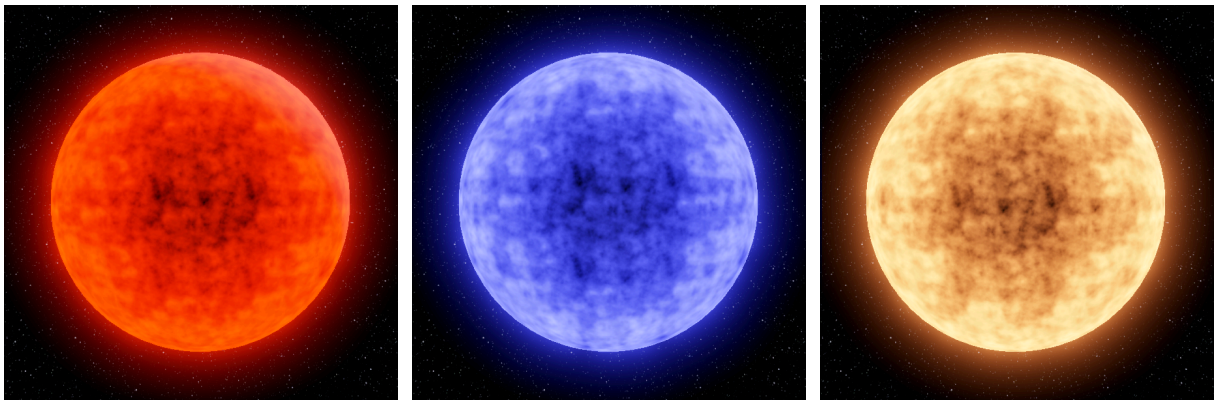


Figure 46: Example solar system star colors (red, blue, and orange).

4.6.2. Planets

To fulfill the goal of generating solar systems, planets needed to be placed into the systems along with the stars. The planets were initially assigned orbit radii which increased linearly with a base distance from the sun. This proved to be unstable for systems containing moons and with smaller distances between planets. After experimentation, scaling the orbit radii linearithmically proved to provide a better balance between aesthetics and stability. This design choice is inspired by the structure of our own solar system, where outer planets are spaced farther apart than inner ones [71].

In order for a planet to stay at the same distance from its star, it was assigned an initial velocity perpendicular to the vector in the direction of the star. The velocity v scales with the star's mass m, the gravitational constant G, and inversely with the orbit radius r:

$$|v| = \sqrt{G \cdot \frac{m}{r}}$$

Besides the placements of planets, the generator also randomizes the angle around the star it should be placed at, mass, and radius of planets. Unlike reality, in which mass scales with the radius and placing moons, the mass and radius are randomized and independent from each other.

4.6.3. Moons

Procedurally placing the moons followed the same procedure as the planet placement, except using the planet as the central reference point instead of the star. Only the planets from the fourth position outward were allowed to have moons, to avoid gravitational interference due to their closer proximity.

The moons' orbit radii increased linearly and the spacing between moons was computed by dividing the planet's orbit increase value by a constant. A higher ratio resulted in moons being closer to their planet and to each other.

The moon's orbital velocity and the initial angle (orbit angle) were calculated in the same way as for planets, except that its planet's mass and position was used instead of the star's. Furthermore, the radius of the moon was randomly chosen based on its planet's radius.

The moon's appearance was created by generating a sphere, followed by randomly positioning craters along its surface. Each vertex was then processed by iterating through all predefined craters to calculate height adjustments based on a method created by Sebastian Lague [72], using the following formulas:

```
cavity = x * x - 1;

rimX = Min(x - 1 - rimWidth, 0);
```

```
rim = rimSteepness * rimX * rimX;

craterShape = Max(cavity, floorHeight);
craterShape = Min(craterShape, rim);

craterHeight += craterShape * crater.Radius;
```

An issue that arose when using the *Max* and *Min* functions is that only one of the values would be utilized, which could result in the formation of harsh shaped craters (see Figure 47). To address this, a smooth minimum and maximum was employed. These functions were based on the approach described in Inigo Quilez's article *Smooth minimum for SDFs* [73].

The smooth minimum function can be formulated as follows (Inigo Quilez ©):

```
SmoothMin(float a, float b, float smoothnessFactor)
{
  var h = Clamp((b - a + smoothnessFactor) / (2.0 * smoothnessFactor), 0.0, 1.0);
  return a * h + b * (1.0 - h) - k * h * (1.0 - h);
}
```

The smooth maximum function can be derived by inverting the smoothness factor as follows:

```
SmoothMax(float a, float b, float smoothnessFactor)
{
  return SmoothMin(a, b, -smoothnessFactor);
}
```

Enabling this smoothing mechanism allowed for better smoothness over craters, as illustrated in Figure 48.

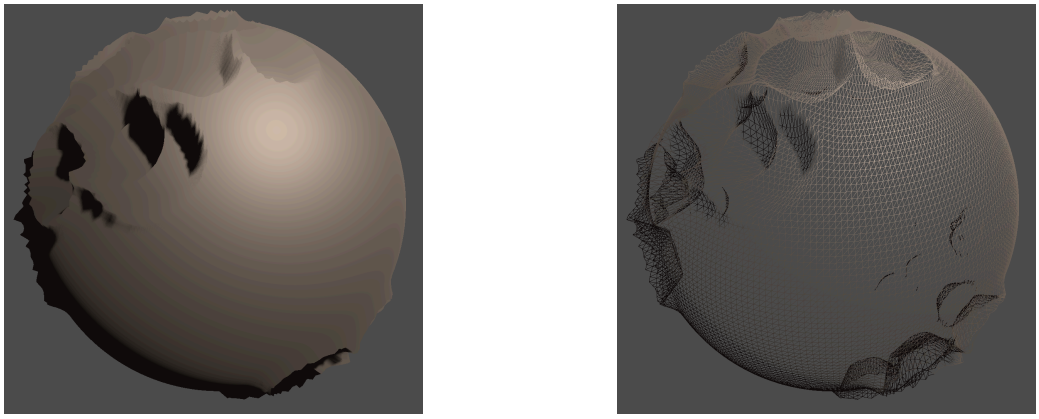


Figure 47: Craters with smoothness set to 0

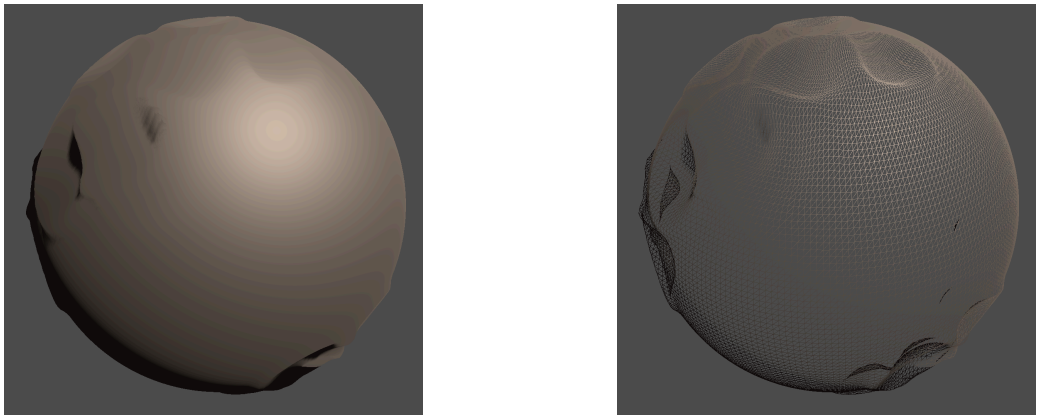


Figure 48: Craters with smoothness set to 1

To further enhance the visuals of the moons, textures and a normal map were added. However, incorporating textures presented a challenge: the textures became stretched due to vertex manipulations used to create craters (see Figure 49). While vertex positions were adjusted, the UV mapping—which determines how texture coordinates correspond to the 3D model's surface—was not updated accordingly.

To resolve this issue, triplanar mapping [53] was employed. This technique involves sampling the texture by projecting it from the three basis vectors (x, y, z), effectively "wrapping" the texture around the object. Since triplanar mapping is a built-in feature in Godot, it was simply enabled in the material setting. By applying both a color texture and a normal texture, the moon achieved a rocky surface with well-defined craters as shown in Figure 50.

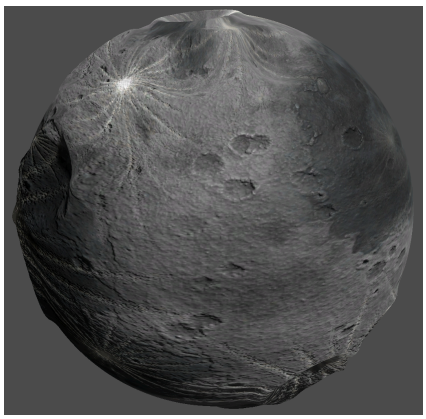


Figure 49: Moon with a texture (triplanar disabled)

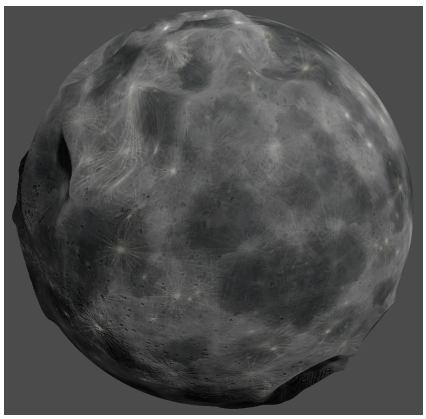


Figure 50: Moon with a texture (triplanar enabled)

4.7. Galaxy

In the context of this project, the galaxy [74] represents the largest scale of the simulation, a vast space populated by procedurally placed solar systems. The following sections introduce the various iterations the galaxy underwent during development.

4.7.1. Star Field

The first version of the galaxy, a 3D distribution of stars that we called a 'star field' (see Figure 51). Points were sampled pseudo-randomly with a uniform distribution within a finite cube, and seeding it, to determine where each star would be instantiated. The stars were constructed using a single spherical mesh (see Figure 52).

Figure 51: Star field

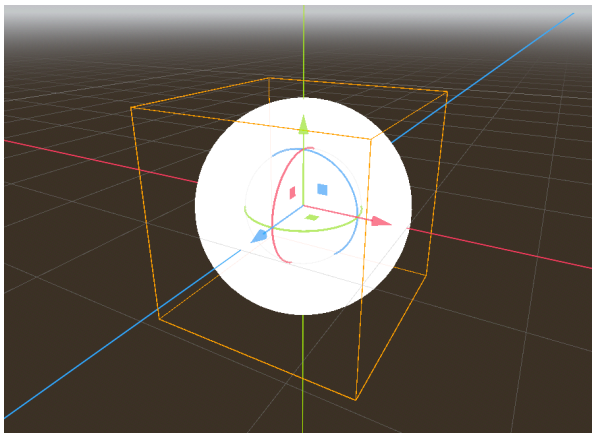


Figure 52: Star mesh

4.7.2. Disc Galaxy

Thereafter, a version of the galaxy that imitates the formation of a disc galaxy was created. A disc galaxy is characterized by a flat, rotating disc structure, with a greater concentration of stars at the center [75].

This was achieved by sampling pseudo-randomly with a uniform distribution, within a sphere instead of a cube. The distribution was also influenced by reducing the probability of a star being placed the further away it was located from the galaxy center, thus:

- reducing vertical spread, which would mimic the flattened shape of a disc.
- increasing the probability of stars being placed near the center, resulting in a greater concentration of stars near the center.

This resulted in a galaxy with a disc-like distribution, as shown in Figure 53.

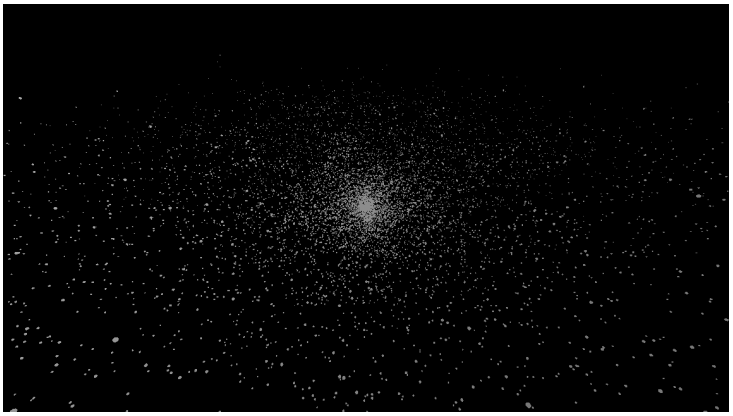


Figure 53: Disc galaxy

4.7.3. Skybox

A traditional skybox was created in Blender [76], [77] to serve as a pre-rendered galaxy background. Unlike the procedurally generated star fields, it does not contain actual 3D stars, but instead imitates a dense star field using a single static image, as seen in Figure 54.

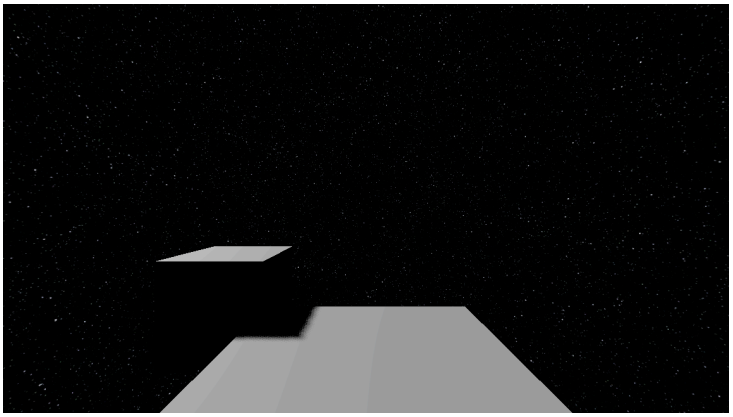


Figure 54: Skybox testing environment

This approach was mainly used for presentation and testing purposes. Since the final goal was to use a backdrop composed of actual, explorable stars, this implementation was not intended for the final product.

4.7.4. Infinite Galaxy

This version is based on the original star field concept from Section 4.7.1, this time, expanding infinitely in all directions rather than being limited to a confined structure (see Figure 55).

Additionally, star placement was further refined by sampling from a noise texture. This approach was used to influence clustering, creating areas of higher and lower star densities, to make the galaxy more varied and visually interesting.

Figure 55: Infinite galaxy

To support infinite exploration, the galaxy space was divided into discrete chunks (see Section 2.4). Only chunks in the player's closest vicinity are generated and rendered, while distant chunks are culled to save performance. As the player moves, new chunks are generated procedurally, giving the illusion of an endless galaxy. See an example "Star chunk" in Figure 56.

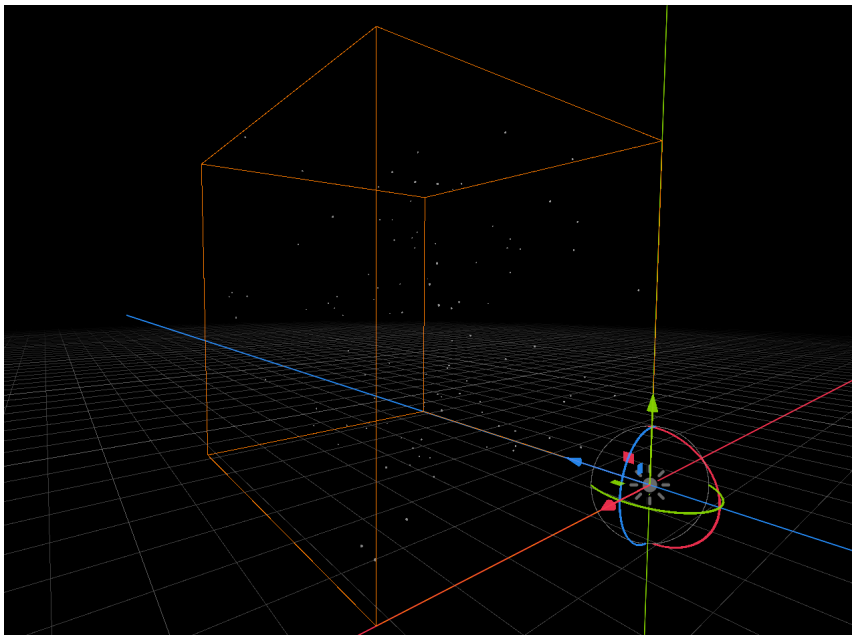


Figure 56: Star chunk

4.7.5. Finite physics-based galaxy

An Infinite galaxy is a compelling concept, but applying physics to stars of an ever-expanding galaxy is not doable. Since such galaxies are infinitely vast, there is not any fixed point of reference making any attempt at global physics calculations not work.

With great advancements in the physics engine (Section 4.5), an attempt to simulate physics of a finite disc-shaped galaxy was performed, with the physics no longer only confined to the bounds of the solar system.

To test this idea the disc galaxy implementation (Section 4.7.2) was revisited and repurposed. It was retrofitted with new stars containing mass and velocity, to interact with each other through the physics engine. The resulting galaxy is demonstrated below in Figure 57 and Figure 58:

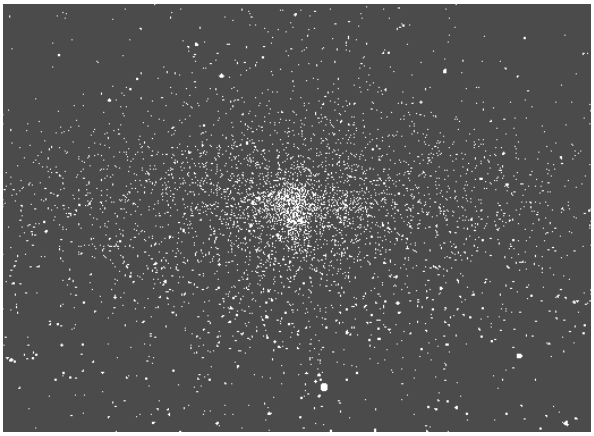


Figure 57: Physics galaxy - Before

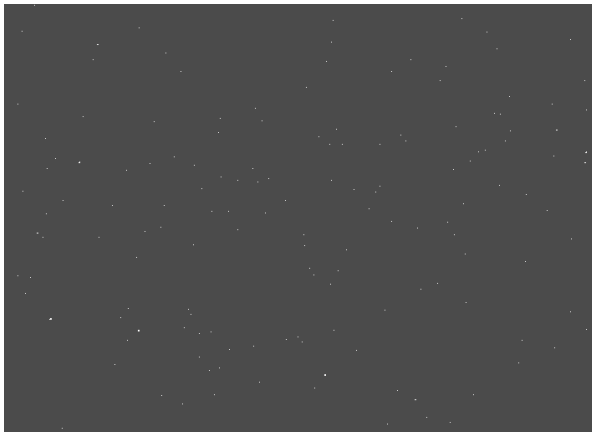


Figure 58: Physics galaxy - After

All masses were set equal and no initial velocity was set, resulting in an unstable galaxy. However, it still demonstrated the potential of simulating a galaxy using the physics engine. With 10,000 stars in this initial setup, the performance impact remained minimal. Given more time, setting appropriate masses and velocities would have been explored further.

4.8. Galaxy Map

The Galaxy Map unified the project's various components, connecting the procedurally generated infinite galaxy (Section 4.7.4) with explorable solar systems (Section 4.6) and their planets, offering a cohesive experience.

4.8.1. Selectable Stars

To enable interaction with individual stars, the selectable star was implemented. Players could now hover over a star with the mouse cursor and click to select it. This was achieved by adding a spherical collision shape to the star object, which detects mouse input events within the collider. See the star, and its collider in Figure 59.

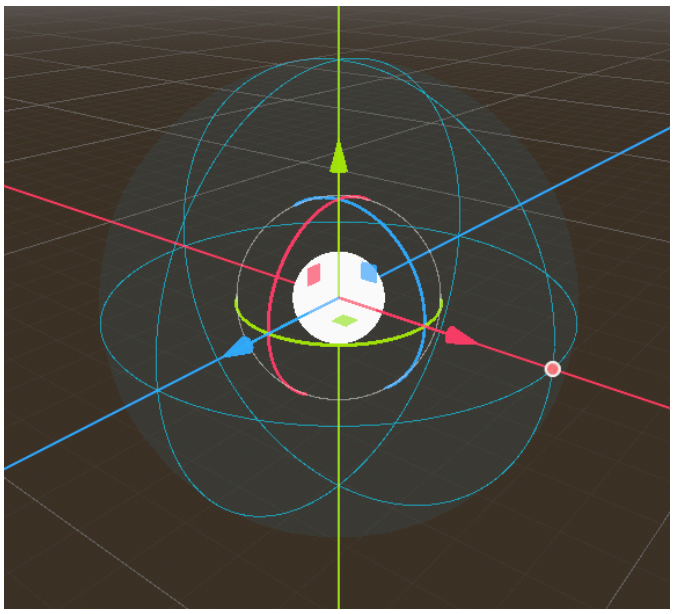


Figure 59: Selectable star

To indicate that a star has been selected, the star's location in space is highlighted, together with a distance measured in "Astronomical Units" (AU) [78]. This is visible in the center of Figure 60. In addition, the coordinates and unique seed of the star are displayed in the bottom-right corner.

4.8.2. Navigation

Two modes of transportation were implemented for navigating the Galaxy Map.

1. Manual movement (see Section 4.4).
2. Fast travel: Once a star is selected, press the "->"-button in the bottom-right of Figure 60. This moves the player rapidly towards it, stopping a short distance away.

To explore the solar systems themselves, the "Explore"-button in the bottom-right of Figure 60, can be used to enter the solar system currently selected. When pressed, a solar system is generated based on the selected star's seed and transitions the player into it. This system exists in a separate scene from the Galaxy Map.

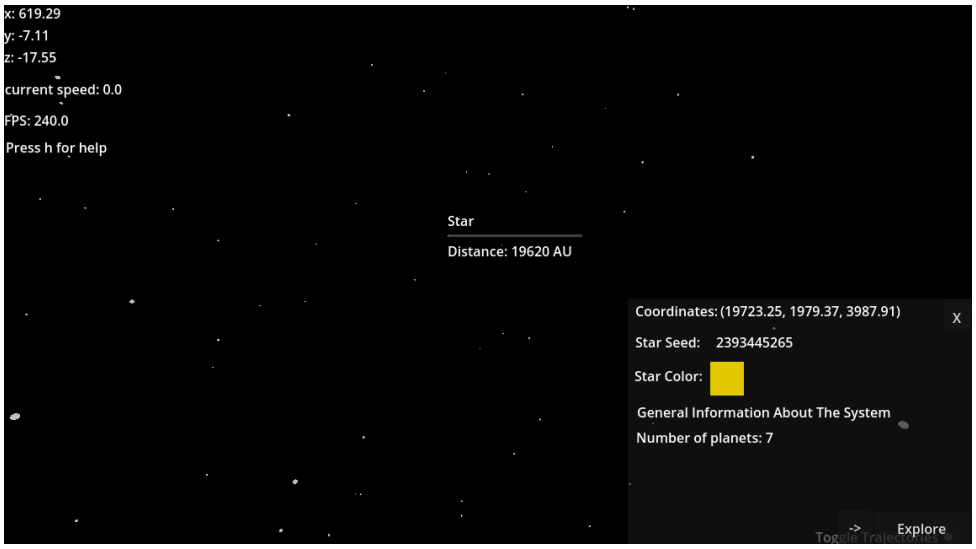


Figure 60: Galaxy map

4.8.3. Seed Creation

The galaxy utilizes a set seed, the same used in Section 4.7.4, to deterministically generate the placement of stars. With the implementation of explorable solar systems, a need arose to generate new seeds for each system. Were they to utilize the same seed, all solar systems would be identical.

To address this, a custom hash function was developed [79], allowing for the generation of unique and deterministic seeds for each star. This function takes into account both the initial Galaxy Seed, and the X, Y, and Z coordinates of a star's position, to produce a star-specific seed. This new seed is then propagated into the stars generation algorithm, which results in unique solar systems while still ensuring deterministic generation.

4.8.4. Multi-Meshed Stars & Star Finder

As the galaxy expanded, loading new star chunks (Figure 56) caused significant stuttering. This performance issue was attributed to the high overhead of instancing hundreds of individual stars, each comprising a `MeshInstance3D` [80] and a collider (Section 4.8.1).

To address this, star rendering was refactored to use Godot's `MultiMeshInstance3D` [81]. This drastically reduced the number of scene nodes and GPU draw calls (from one per star to one per chunk). As shown in Table 3, this optimization substantially mitigated frame time spikes (1% and

0.1% highs) and decreased average memory usage from 115.9 MB to 81.7 MB, enhancing performance stability.

Table 3: Frame time metrics on PC-2 (Table A-2) - Multi-mesh stars

	Average	**1% high**	**0.1% high**
From	0.39 ms	3.88 ms	33.43 ms
To	0.44 ms	1.04 ms	1.9 ms

However, `MultiMeshInstance3D` only instances visual meshes, rendering the collider-based star selection unusable. Two solutions were considered: instantiating separate colliders, or a ray-based approach. We implemented the latter, termed “Star Finder” (see Figure 61). When the player clicks, it casts a ray and checks for nearby known star positions at intervals, allowing selection without individual collider objects. This avoided potentially reintroducing the node instancing overhead that the `MultiMeshInstance3D` was intended to solve. The Star Finder features adjustable parameters, like increasing check radius with distance for easier selection of remote stars.

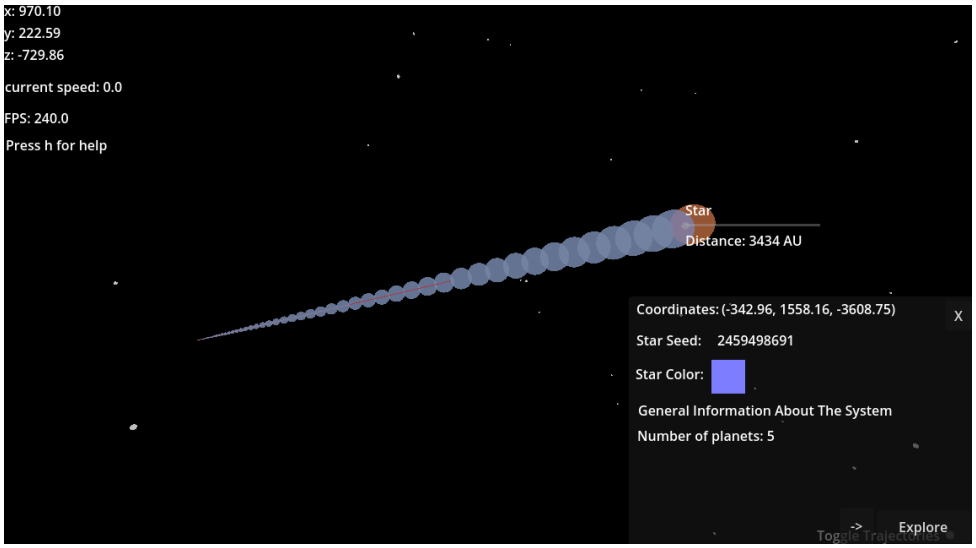


Figure 61: Star Finder tool

4.8.5. Seamless Galaxy

With great improvements in optimizing planet generation, as detailed in Section 4.2, new opportunities emerged. Previously, transitioning from the galaxy scale into individual solar systems was a static process, triggered by a button click (Section 4.8.2), which loaded a separate system scene. But now, system scenes could be dynamically instantiated in real-time as the player moves towards a star. This allows the galaxy to be populated by fully realized solar systems that load seamlessly during exploration. An example of multiple systems visible at the same time is shown in Figure 62.

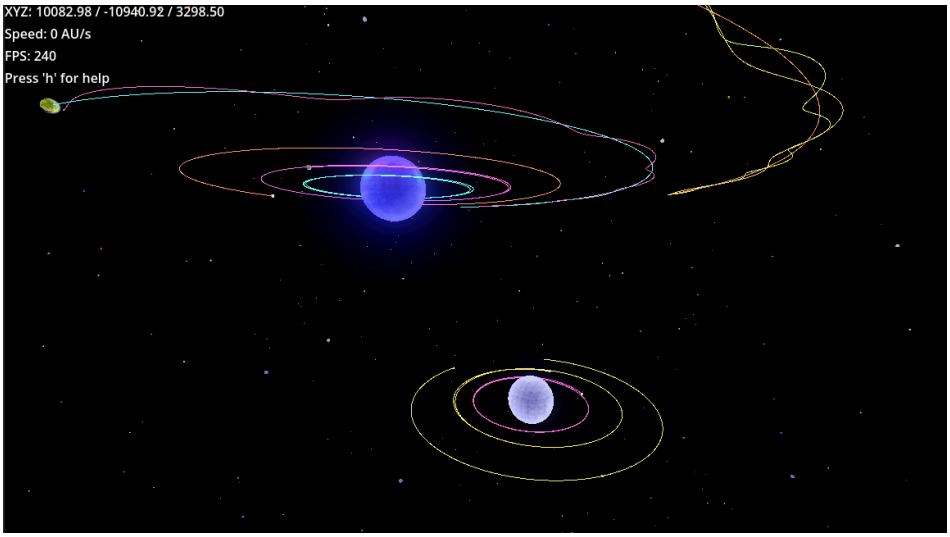


Figure 62: Seamless galaxy showing two solar systems at once

Using the Star Finder (Section 4.8.4), stars in a radius around the player could be continuously detected and instantiated at a set distance. As the player approached, these solar systems would scale up slightly until they reached their full size.

However, this approach had notable performance consequences, even with the planet generation optimizations (detailed in Table 4). Although the average frame time remained stable, some runtime stutters occurred, indicated by the 1% and 0.1% high frame times.

Table 4: Frame time metrics on PC-2 (Table A-2) - Seamless galaxy

	Average	**1% high**	**0.1% high**
From	4.17 ms	5.16 ms	11.86 ms
To	4.21 ms	9.05 ms	34.31 ms

In addition, the UI received updates (seen in Figure 63) to display more information about each solar system.Together with some added flair of an assigned stellar classification [82] roughly associated to its color, as well as a randomly selected star catalogue acronym, followed by an integer number [83] (the system's seed).

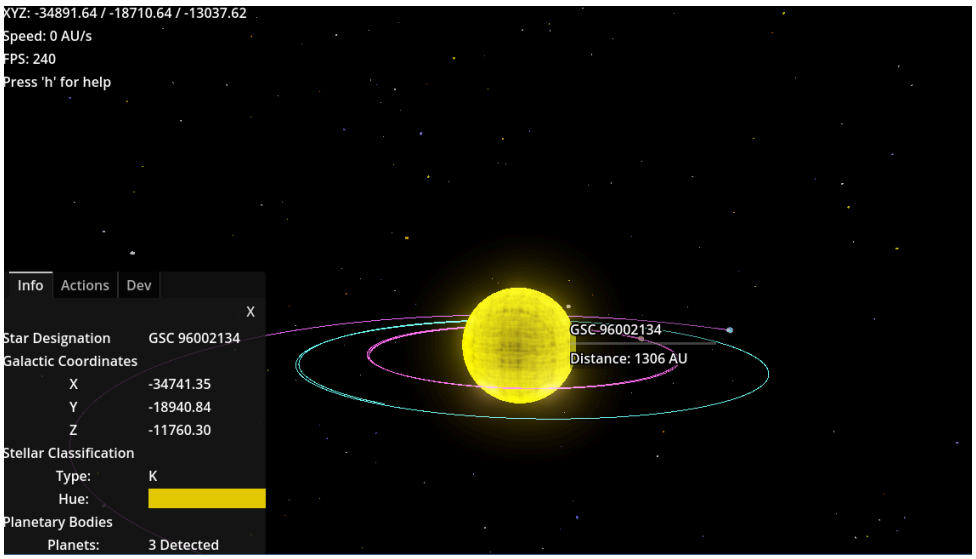


Figure 63: Updated Star Select UI

5. Result

The following section presents the final result of the project. First, a brief overview of the final product is provided in Section 5.1. Then, in subsequent sections, a more technical in-depth look at the final product will be presented.

5.1. Overview

The final product includes a galaxy map where users can freely navigate between stars, where each star represents the central body of its own solar system. The galaxy map and all systems are generated deterministically, meaning that each time a user enters the same system, it will be generated in exactly the same manner as the first time.

Entering a solar system causes its belonging celestial bodies (planets and moons) to be generated. The celestial bodies are generated procedurally using 3D-noise as well as the Marching Cubes algorithm, which enables complex terrain generation with overhangs. The moons have craters and a "moon-like" appearance to them and planets are colored based on their distance from the central sun; the closest planets have a "warmer" color palette while the further way planets have a "cooler" color palette. Furthermore, the planets have atmospheres, as well as oceans and vegetation procedurally generated on their surfaces.

Finally, the end product includes a robust physics engine that is capable of updating thousands of objects simultaneously in real-time.

5.2. Planets

The final planets included mesh generation using the Marching Cubes algorithm as well as planetary atmospheres, diverse color schemes, vegetation and oceans. The finished planets are shown in Figure 64 and Figure 65;

5.2.1. Mesh Generation

The planets' meshes were generated using the Marching Cubes algorithm (Section 2.3.2), using a scalar field generated using noise (Section 2.2) and fBm (Section 4.1.5) as input.

5.2.2. Atmospheres

The planetary atmospheres were based on ray marching and Rayleigh scattering, which allowed for simulating realistic atmospheric light scattering. Furthermore, multiple color presets were created to ensure visually pleasing results (see Figure 37 for results).

5.2.3. Color Schemes

The planets were given color schemes based on their distance from the sun, representing how warm they were. The color schemes were given variation by coloring the fragments based on distance to the lowest point on the planet Cliff faces were also colored. See Figure 20 and Figure 21 for results.

5.2.4. Surface Details and Features

Surface details (Section 4.3.1), such as grass, were generated by extracting the planets mesh data and calculating the number of grass blades per face based on the ratio between the total mesh area and the area of the current face. Subsequently, the surface detail was oriented according to the face's normal vector.

For surface features (Section 4.3.2) such as trees, generation was achieved by selecting random points within the planet's axis-aligned bounding box (AABB) and performing raycasting. The orientation of the surface feature was then determined based on the result of the ray hit.

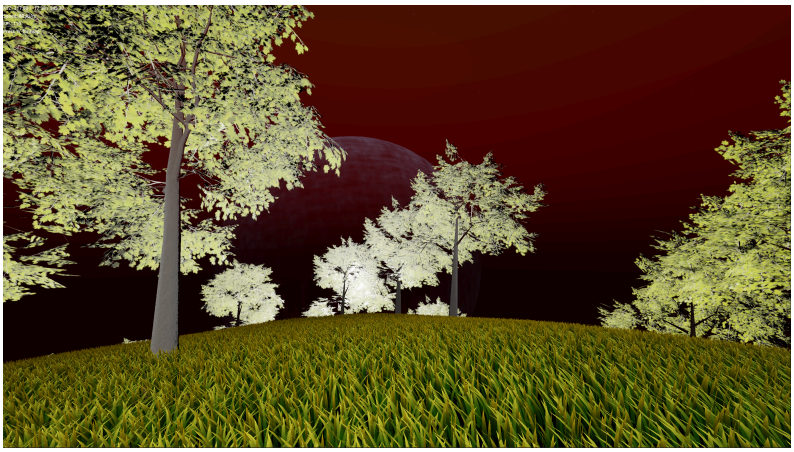


Figure 64: Close-up of a planet with trees and grass

5.2.5. Oceans

The oceans were generated by applying a water shader to a sphere. The water shader displaced the vertices using noise textures along with normal maps to simulate waves. Additionally, the water color was based on depth extracted from the depth texture. Furthermore, transparency was achieved by utilizing the screen texture. Finally, a foam effect was implemented by blending the water color with a white color at the shallow parts of the ocean.

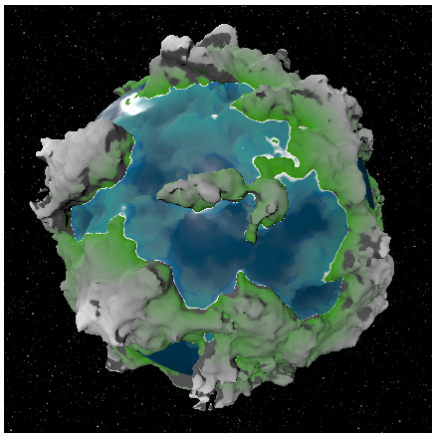


Figure 65: Planet with ocean

5.3. Systems and Orbits

Systems are deterministically generated from a seed, containing a single star and 3 to 8 planets, with moons placed around the outer planets. While systems are initially stable, numerical errors in the simulation accumulate over time, leading to eventual instability. This instability is more likely when celestial bodies move at higher velocities, such as when the gravitational constant G is set to a higher value.

5.4. Physics Engine

The Rust-based N-body physics engine (Section 4.5) was successfully developed, incorporating parallelized versions of both Direct Summation and a Barnes-Hut algorithm. Benchmarks (Section 4.5.4) validated its design and optimization. As illustrated in Figure 43, the parallel Barnes-Hut method demonstrated its capability to calculate accelerations for tens of thousands of bodies (up to approximately 45,000) within a 16.7 ms frame budget, suitable for real-time simulation of large systems at 60 FPS.

In practice, the engine primarily performs small scale calculations to govern solar system dynamics. For this scale, it defaults to the Direct Summation method, due to its superior performance for small N (see Section 4.5.4). This means that the advanced Barnes-Hut optimization for large N was not utilized in the main gameplay loop. While this underutilizes the engine's large-scale capabilities, the choice supports stable and efficient local orbital mechanics in line with gameplay priorities, which emphasize galaxy-scale exploration over high-resolution interstellar dynamics.

The engine's large-scale capabilities were demonstrated in the "Finite physics-based galaxy" (Section 4.7.5), which simulated 10,000 interacting stars in real-time. Although currently used for small-scale systems, the engine is well-suited for future expansion to more complex simulations with a greater amount of bodies.

5.5. Galaxy

The galaxy is explorable by the player through movement via the player controls, as well as through selecting a star and fast-traveling to it, with an action under the 'Actions' tab (Figure 66). Solar systems are instantiated and appear as the player approaches them.

With The Galaxy Map implementation (Section 4.8), the distribution of stars from the Infinite Galaxy connects seamlessly with the Solar Systems implementation, which in turn connects to the planets. Each step in scale is demonstrated in the following figures, the galaxy-scale (Figure 66), towards the system-scale (Figure 67), eventually reaching the planet-scale (Figure 68).

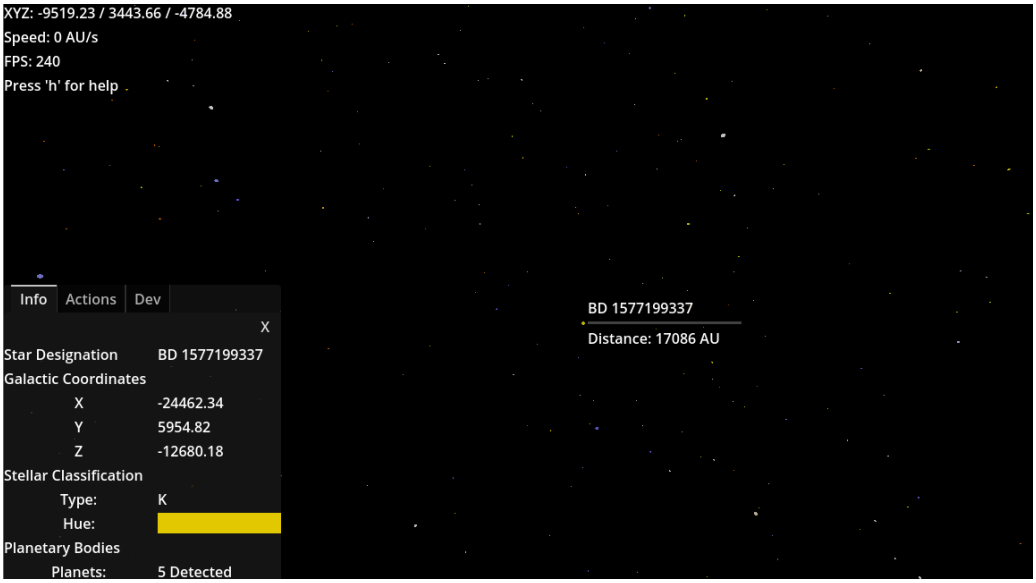


Figure 66: Galaxy Map at the galaxy-scale

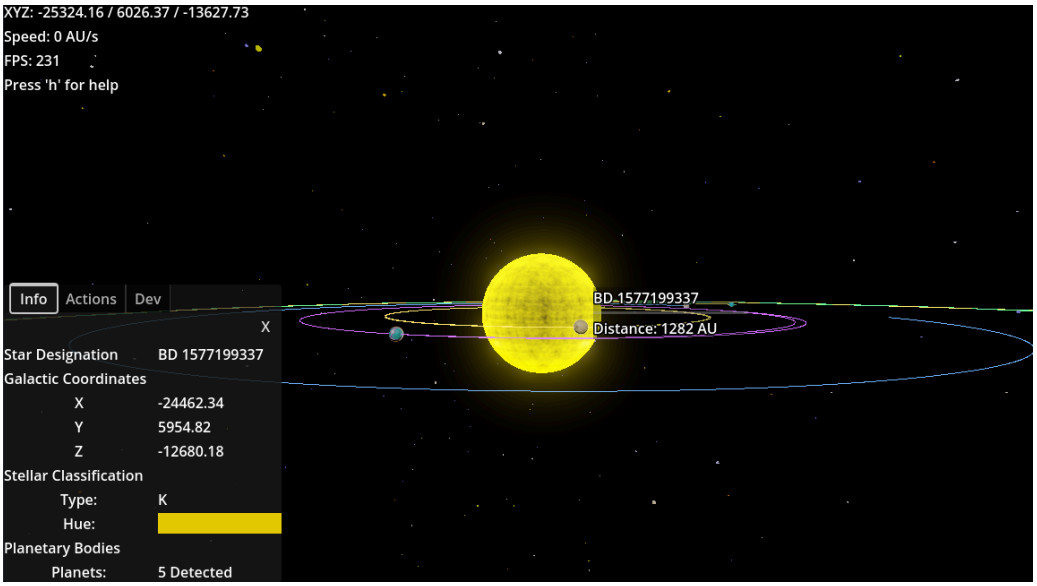


Figure 67: Galaxy Map at the system-scale

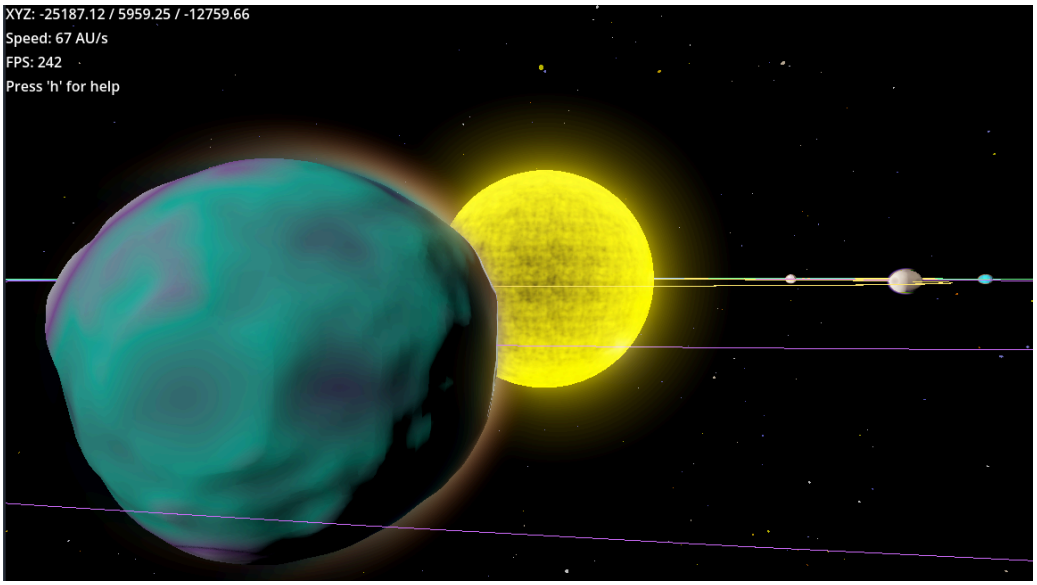


Figure 68: Galaxy Map at the planet-scale

The seamless instantiation of systems was left as a toggleable option. By enabling it, you get the result that is seen in the images above. However, the resulting performance implications on the dedicated benchmarking computer (PC-1, Specs: Table A-1) is significant, see Table 5:

Table 5: Frame time metrics on PC-1 (Table A-1) - Seamless galaxy

	Average	**1% high**	**0.1% high**
Seamless Galaxy disabled	0.88 ms	1.67 ms	4.51 ms
Seamless Galaxy enabled	1 ms	5.35 ms	17.08 ms

Compared to our performance goals, as dictated earlier in Section 3.4, looking at Table 5 with the seamless galaxy enabled, the average FPS is 1003.8 and the greatest frame time disparity is 16.8 ms. This, achieves one out of two of set goals.

By disabling seamless instantiation, the average FPS is 1135.6 and the greatest frame time disparity is 3.63 ms. This, falls within our performance goals.

6. Discussion

This section presents a discussion and reflection on the results, process, and methodology of the project. It also addresses the generalizability and validity of the findings. Furthermore, societal and ethical considerations are discussed. The section concludes with an exploration of potential directions for future work within the project.

6.1. Result Discussion

The following subsection discusses the project's results, discussing the role of the MoSCoW table as a tool in achieving these outcomes. It also addresses the considerations involved in balancing performance with the trade-offs between realism and gameplay. Finally, a comparison is made with Exo Explorer to discuss the differences.

6.1.1. The MoSCoW Table

As shown in Appendix C, the majority of the project tasks were successfully completed. This outcome may be attributed to a well-structured planning phase and effective scope management. Additionally, the inherent vagueness of certain features may have contributed, as their ambiguous nature allowed them to be interpreted either as substantial or minor tasks.

Using a MoSCoW table helped us think carefully about the priority of each feature, making sure the project stayed focused on its main purpose. This meant that even if we didn't manage to finish all of the *Should Have* or *Could Have* items, the core of the project would still be complete and aligned with its purpose. For example, the vegetation feature, listed as a *Could Have*, was something we thought would add visual interest, but it wasn't essential to the project's objectives. Keeping these priorities in mind helped the team stay focused on what mattered most and avoid spending time on features that didn't contribute meaningfully.

However, the application of MoSCoW was not without shortcomings. Some features were defined imprecisely, leading to occasional overlap and ambiguity. For example, the design of the navigation interface lacked clarity and appeared to intersect conceptually with the galaxy traversal system, resulting in confusion regarding their integration and implementation.

A notable challenge was the project's open-ended nature, which, while fostering creativity, impeded the formulation of concrete and well-defined concepts. For instance, the feature "*get interesting info about planets and solar systems*" lacked specificity, as the term "*interesting*" was not clearly defined beyond a few examples. Nevertheless, this issue was largely mitigated during the task breakdown phase for the Kanban board, which provided the necessary clarity to proceed effectively.

6.1.2. Balancing performance, visuals and user experience

While visuals were a consideration in this project, the main focus was to ensure that the program ran smoothly and performed well. In some cases, such as deciding how detailed the planets should be, we had to balance performance and visuals. Ultimately, visual fidelity was kept within reasonable limits, as long as it didn't compromise our set performance goals.

Initially, the resolution of the planets directly correlated with the amount of data points within them, meaning that a planet with a larger radius required more iterations to generate. This posed a problem when scaling up the galaxy as the planets became too demanding to generate. Therefore, by separating the radius from the amount of data points (the resolution of the planet) it became possible to scale the planets without increasing their resolution. When creating a planet with low resolution, the performance of generating planets became better, at the expense of the visual fidelity of the planet.

To reduce stuttering when loading in a new solar system (due to the planets generating), the planet generation was offloaded to different threads (Section 4.2.2). To avoid planets popping in during gameplay, they were given a temporary mesh that got replaced once their real mesh had been constructed. This was an example of where the visuals were directly impacted by the performance, albeit temporarily during gameplay. It also affected the user experience due to players having to wait for the planets to be constructed. We felt that this was a good compromise between user experience and visuals because otherwise players might get frustrated when the game freezes each time a solar system loads.

6.1.3. Balance Between Realism and Gameplay

Balancing realism and gameplay was a recurring challenge. The solar systems are not to scale, as realistic distances made planets too far apart to be visible during exploration. To address this, planets where moved closer to each other. As talked about in Section 4.6, this increased risks of instability.

Planetary motion also posed challenges for surface exploration, as moving planets affected player physics. One considered solution was to freeze a planet's movement when a player was on it, simplifying implementation by removing velocity effects. However, this was rejected in favor of maintaining physical accuracy, therefore, planets continue moving at all times.

6.1.4. Exo Explorer Differences

This project explored similar areas as the previously mentioned *Exo Explorer*, but with a greater emphasis on an advanced physics engine, proper benchmarking, and real-time exploration of a procedurally generated galaxy. By contrast, Exo Explorer focused on a single solar system, allowing for deeper detail in planetary environments, whereas this thesis prioritizes scalable procedural systems suitable for rendering and simulating large-scale space exploration.

A key difference between the two projects lies in their approach to realism. As briefly discussed in Section 6.1.3, one proposed solution to address issues related to planetary landings involved freezing a planet's motion when a player is present on its surface and transitioning to a geocentric reference frame. This method was inspired by *Exo Explorer*, which adopts a similar approach.

Finally, the most significant distinction is the implementation and optimization of a large-scale physics model for simulating celestial dynamics, which was an element that was not developed in *Exo Explorer*.

6.2. Process and Method Discussion

This subsection reflects on the process and methods used, including workflow choices, use of multiple programming languages and AI, and changes in the original plan.

6.2.1. Multiple Programming Languages

This project utilized a multi-language approach, via Godot's GDExtension system [36] (detailed in Section 3.3). We utilized a mixture of GDScript, C# and Rust, allowing us to select the optimal language for specific tasks.

For computationally intensive components, the physics engine (Section 4.5) in particular, Rust was chosen. Its strengths in raw performance, memory safety, and concurrency enabled significant optimization [84].

However, it also introduced complexities. Managing a multi-language code base, debugging across languages, and passing data between the languages required setup and proved to be cumbersome on occasions.

Overall the experience was positive, leveraging each language's strengths was advantageous for achieving the project's simulation goals despite the added overhead.

6.2.2. Workflow and Collaboration

The group followed an Agile-inspired workflow (Section 3.1) utilizing a GitHub Projects Kanban board for task management, Git for version control with a feature-branch model (Section 3.2), and Discord for team communication. GitHub also facilitated pull request (PR) reviews and issue tracking.

This structured approach was generally effective, with tasks discussed, prioritized, and assigned during weekly team meetings. However, sometimes PRs would lay waiting for a review on GitHub a longer time, since we had not established any process to assign reviews, and it was up to the initiative of each member. This could lead to PRs getting outdated, creating problematic merge conflicts and requiring more time to bring branches up to-date. Implementing a stricter system of handling these reviews would therefore have been beneficial.

Despite these challenges, core elements such as feature branches, mandatory PRs, and centralized task tracking were crucial for effective collaborative development throughout the project.

6.2.3. Use of Generative AI

Artificial intelligence (AI) was utilized at various stages throughout the project. During the development phase, tools such as ChatGPT [85] and GitHub Copilot [86] were employed to support the coding process. Copilot was also integrated into the PR review workflow, providing quick feedback on code submissions. While AI-generated reviews were not considered substitutes for peer-reviewed evaluations, they offered an efficient means of identifying obvious issues (e.g. noticing when a wrong variable was used) that might otherwise be overlooked.

Additionally, AI was employed during the report writing phase. Tools such as ChatGPT and Google Gemini [87] were occasionally used to refine already written text and improve the overall quality of the writing, by for example, finding grammatical errors and spelling mistakes

6.2.4. Project Purpose

The project's purpose underwent a greater change after early feedback on the initial planning. The original purpose was as follows:

> "The aim of this project is to simulate solar systems through procedurally generated planets, utilizing computer-generated noise such as Perlin noise, together with the Marching Cubes algorithm. The composition of the solar systems can vary – from a sun with a single planet to more complex systems with multiple planets and additional celestial bodies such as moons. To mimic the natural movements of these celestial bodies, a simplified physics simulation will be implemented."

This project also aims to explore and combine different techniques for optimization to ensure that the simulation will run in a performance-efficient manner."

It became clear that the project's objectives were not clearly defined. After internal discussions the team reached a consensus on the purpose of the project. Any changes, particularly to the purpose, sought to address the following three problems:

1. A great deal had been explained specifically about solar systems in the planning, while the team, in reality, had drifted towards wanting to create an entire galaxy of solar systems instead.

2. An entire section of the planning was dedicated to performance and optimization, as well as a part of the purpose. This played a part in making it unclear whether this project's major focus was about optimization, or something else.
3. It was unclear how this project differs from the similar bachelor's thesis project Exo Explorer [18].

Through internal discussions, consultation with our supervisor, and study of the aforementioned feedback, the purpose was rewritten to refine the project's goals and scope (see the rewritten purpose in Section 1.1).

This helped align the team as a whole. Putting the focus on implementing a galaxy of solar systems, rather than a single one. As well as reducing the focus on performance and optimization techniques, instead focusing on finding techniques to reach an at least viable performance.

6.2.5. MoSCoW Changes

The MoSCoW method was used to structure and prioritize project features into tiers of importance. As the project progressed and its scope became clearer, the MoSCoW table underwent change. Due to the agile-inspired workflow of the project, features were re-prioritized, removed or added.

While most features remained unchanged throughout, some of the most notable changes were:

- Camera/player controls: Moved from "Should have" to "Must have" as the focus on being able to explore the planets, systems, and galaxy, was deemed very important.
- Space background: Removed from "Must have", since it was deemed not critical for project's success. Although, a space background was eventually given by itself as the stars were distributed in the galaxy.
- Galaxy traversal: Added as a "Should have" to allow for traversal at different scales. On the solar galaxy level, the solar system level, and on the planet level.
- Other celestial bodies e.g. asteroids or nebulae, as well as planet vegetation, were added as potential features to be added. In the end planet vegetation was explored.

6.2.6. Performance Methodology

At first, in the initial planning, the key performance metrics to evaluate the application were:

- **Memory consumption:** Evaluating the amount of system memory utilized during execution.
- **Scene generation time:** Measuring the time it takes to generate a scene.
- **Frames per second (FPS):** Assessing the rendering performance of the simulation.

Shortly thereafter, a target of maintaining an average of 60 FPS was determined as a concrete benchmark to strive toward. This target was to be achieved on a specific benchmarking computer with specific hardware specifications, those specifications were detailed in **PC-1** (see Table A-1).

However, as development progressed and with greater research into real-time performance, the team improved its understanding of what constitutes a smooth and responsive performance. This, with the focus shifting towards consistency in frame times instead, this updated benchmarking methodology is described in Section 3.4.

Regarding the other original metrics:

- **Scene generation time:** This refers to how long it takes to initialize and load new scenes and was initially marked as a performance metric. However, since the majority of elements are streamed at runtime, rather than through traditional loading screens in between, the metric was eventually deemed less critical. Even so, it was still utilized as a metric for some operations, such as during planet generation, to compare different implementations and optimizations.

- **Memory consumption:** While initially a concern, it proved not to be a limiting factor in practice. This is likely due to the content being generated procedurally at runtime, rather than pre-loaded or stored in memory. Although it was continuously monitored, no memory-related issues occurred, making it a non-critical performance metric for this project.

6.3. Generalizability and Validity

This section considers the broader applicability of the project's components and the soundness of its simulation results.

6.3.1. Generalizability

Many techniques employed in this project are generalizable. The implemented N-body algorithms are standard methods applicable to other systems governed by inverse-square laws (like gravity or electrostatics). The Morton-based octree construction could be used for additional particle simulations. Optimization strategies such as octree-driven LOD, chunking, and CPU parallelism (e.g., with rayon) are common techniques, as well as procedural generation methods with noise and mesh generation with Marching Cubes.

6.3.2. Validity

Evaluating the validity of this project requires considering the goal of achieving the most scientifically accurate physics model possible within the constraints of a smooth, real-time simulation.

1. **Physics Simulation**: The simulation is grounded in established physical principles, using Newton's Law of Universal Gravitation and standard N-body algorithms like Direct Summation and Barnes-Hut. However, achieving the necessary performance led to certain trade-offs regarding accuracy:
   - *Integration Method*: We implemented the symplectic Euler method for time integration. While selected for its improved stability over explicit Euler (crucial for preventing orbits from rapidly degrading in real-time), it is a first-order integrator. This means it is more prone to numerical errors and energy conservation errors over long simulations compared to higher-order symplectic methods (e.g. Leapfrog or higher-order Runge-Kutta [88]), which are often used in non-real-time scientific studies. This choice represents a direct trade-off between accuracy and the computational budget per frame needed for smoothness.
   - *Omitted Physics*: To maintain real-time feasibility, complex physical interactions beyond basic Newtonian gravity were omitted. This includes relativistic effects, physical collisions between bodies and non-gravitational forces.
   - *Outcome*: The resulting simulation produces visually plausible orbital mechanics suitable for an interactive exploration context, but its accuracy for scientific purposes is limited by these simplifications.
2. **Procedural Generation:** The validity here pertains to internal consistency and alignment with aesthetic goals. The use of deterministic, seeded generation ensures that the galaxy and its systems are consistently reproducible. While using physically inspired concepts (noise, fBm), the procedural generation prioritizes visual diversity and exploration potential over astrophysical realism.

In essence, the project's validity lies in its successful implementation of recognized N-body algorithms and optimization techniques to create a large-scale, real-time simulation that aims for physical realism while accepting necessary compromises to ensure interactivity and performance targets were met.

6.4. Societal and Ethical Aspects

Two main points of discussion were identified: how procedural content generation (PCG) affects game designers, particularly level designers, as well as its impact on players.

As PCG and artificial intelligence (AI) advances, there's a risk that it could replace human developers. While PCG reduces cost and development time [89], the concerns that it may become proficient enough to replace human creativity are still present. An example for this is when the Swedish game company Mindark announced plans to fire half of their employees, primarily world builders, in favor of AI-driven content generation [90].

However, the development process in this project revealed that significant manual effort remained necessary. Generating aesthetically pleasing content and scaling and positioning the celestial bodies in a plausible manner all required substantial manual tweaking of parameters and settings. Moreover, PCG algorithms have the potential to generate content that may be faulty, unrealistic or unnatural. Solutions to these issues include setting specific constraints before executing the algorithms, or involving a human designer at the end of the content generation [89]. Although it is still a possibility that future PCG algorithms can automate these processes, current implementations seem to depend on a collaborative relationship between human designers and algorithmic systems.

Another concern was that PCG-generated content must be sufficiently engaging and playable to avoid negatively impacting the player experience. Games containing PCG are at risk of containing repetitive content, which may influence a player's sense of immersion or reduce replayability. An example is No Man's Sky, where the procedurally generated planets felt overly repetitive and basic [91]. Additionally, PCG can also create environments that hinder gameplay, such as untraversable terrain, negatively affecting the overall experience.

Although the primary objective of this project was not to create an engaging game play experience, certain measures were nonetheless taken to reduce repetitiveness. For instance, planetary coloration was randomized based on each planet's distance from the sun, with additional randomization applied to simulate variations in atmospheric thickness. These methods introduced greater diversity in the generated content, demonstrating that careful parameterization and randomness can effectively counteract some of the inherent risks associated with procedural generation.

6.5. Future work

There are several directions in which this project could be expanded. A number of planned features were not implemented due to time constraints, and these could serve as valuable additions in future iterations.

In particular, the planet generation system offers considerable room for improvement. Currently, generated planets feature basic elements like vegetation (e.g., trees and grass) and bodies of water (e.g., oceans), but they remain relatively unengaging. Future enhancements could include the addition of fauna, biome-specific details, and subterranean structures such as cave systems, this would also make greater use of the existing Marching Cubes implementation, which is well-suited to handle mesh generation of caves and overhangs.

Additionally, as identified in Section 4.2.3, where gaps appear between differing LOD chunks. This issue can be solved in future iterations through the implementation of the Transvoxel algorithm [47], which is designed to solve this issue.

Another potential extension involves incorporating a wider range of celestial bodies, such as gas giants, meteoroids, and comets, which would significantly enhance the diversity of the galaxy. Multiple of these, in particular the solid-surface bodies, could make use of the same, but modified,

planet generation system. Additional celestial bodies was listed as a “Could have” in the MoSCoW prioritization but could not be implemented within the project timeframe.

Another opportunity for future development is making further use of the existing physics engine (see Section 4.5). The engine’s large-scale capabilities were demonstrated with the Finite physics-based galaxy (Section 4.7.5), as well as studied in Section 4.5.4. Initially, work would continue on the physics-based galaxy to make it stable.

7. Conclusion

This project set out to develop and simulate a physics-based, procedurally generated, and explorable galaxy within the Godot engine, addressing the challenges of computational scale, performance, and plausibility in creating such vast virtual environments. The core objective was to produce a deterministic and computationally efficient model capable of real-time interaction.

Development resulted in a comprehensive system. Planets were successfully generated using a combination of noise functions (Perlin, fBm) and the Marching Cubes algorithm, resulting in complex and varied terrains. These planets were further enhanced with procedurally placed features, including oceans with shader-based wave effects, vegetation distributed via Poisson-disc sampling, and physically-inspired atmospheres simulating Rayleigh scattering.

Crucially, significant performance optimizations were implemented and validated; multi-threading effectively offloaded intensive planet generation tasks, while octree-based chunking with Level of Detail (LOD) dynamically managed mesh complexity, and Multi-Mesh instancing drastically reduced draw calls for star rendering. A robust N-body physics engine, leveraging a parallelized Barnes-Hut algorithm implemented in Rust, was integrated to simulate celestial mechanics efficiently for numerous bodies. The use of seeded randomization throughout ensured deterministic generation, allowing for reproducible galaxies and solar systems. Exploration capabilities were successfully implemented, allowing navigation across multiple scales via a galaxy map and seamless transitions into solar systems down to planetary surface interaction with a physics-aware controller.

The project successfully met its primary goals, fulfilling all “Must Have” and “Should Have” requirements defined during planning. The focus on achieving consistent frame times, rather than solely maximizing average FPS, proved effective in delivering a smoother user experience during exploration and simulation.

Ultimately, this work contributes a practical implementation and analysis of techniques essential for simulating large-scale procedural galaxies. It demonstrates the successful integration of advanced procedural generation, N-body physics simulation, and targeted optimization strategies within the Godot engine, offering a viable and efficient model for developers aiming to create expansive, dynamic, and interactive celestial environments. While acknowledging necessary simplifications for real-time performance, the project establishes a solid foundation upon which future enhancements, such as greater celestial diversity or more complex physical interactions, can be built.

Bibliography

[1] "graphics processing unit." [Online]. Available: https://www.britannica.com/technology/graphics-processing-unit

[2] "central processing unit." [Online]. Available: https://www.britannica.com/technology/central-processing-unit

[3] 3D Studio, "What is a Polygon Mesh and How to Edit It?." [Online]. Available: https://3dstudio.co/polygon-mesh/

[4] GPU Shader Tutorial, "Shader Basics - The GPU Render Pipeline." [Online]. Available: https://shader-tutorial.dev/basics/render-pipeline/

[5] K. Group, "Shader - OpenGL Wiki." [Online]. Available: https://www.khronos.org/opengl/wiki/Shader

[6] K. Group, "Vertex Shader - OpenGL Wiki." [Online]. Available: https://www.khronos.org/opengl/wiki/Vertex_Shader

[7] K. Group, "Fragment Shader - OpenGL Wiki." [Online]. Available: https://www.khronos.org/opengl/wiki/Fragment_Shader

[8] K. Group, "compute Shader - OpenGL Wiki." [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader

[9] M. J. Noor Shaker Julian Togelius, *Procedural Content Generation in Games*, 1st ed. Springer Cham, 2016. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-42716-4

[10] Godot Engine Contributors, "Random number generation — Godot Engine 4.4 documentation." [Online]. Available: https://docs.godotengine.org/en/latest/tutorials/math/random_number_generation.html

[11] K. Perlin, "An image synthesizer," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, Jul. 1985, doi: 10.1145/325165.325247.

[12] W. E. Lorensen and H. E. Cline, "Marching cubes: a high resolution 3D surface construction algorithm," in *Seminal Graphics: Pioneering Efforts That Shaped the Field, Volume 1*, New York, NY, USA: Association for Computing Machinery, 1998, pp. 347–353. [Online]. Available: https://doi.org/10.1145/280811.281026

[13] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, p. 167.

[14] T. Tamasi, "What Is FPS, and How It Helps You Win Games," 2019, [Online]. Available: https://www.nvidia.com/en-us/geforce/news/what-is-fps-and-how-it-helps-you-win-games/

[15] S. Wasson, "Inside the second: A new look at game benchmarking," 2011, [Online]. Available: https://web.archive.org/web/20201111221009/https://techreport.com/review/21516/inside-the-second-a-new-look-at-game-benchmarking/8/

[16] Godot Engine Contributors, "Nodes and Scenes — Godot Engine 4.4 documentation." [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/step_by_step/nodes_and_scenes.html

[17] M. Beukman, C. W. Cleghorn, and S. James, "Procedural content generation using neuroevolution and novelty search for diverse video game levels," in *Proceedings of the Genetic and Evolutionary Computation Conference*, in GECCO '22. Boston, Massachusetts: Association for Computing Machinery, 2022, pp. 1028–1037. doi: 10.1145/3512290.3528701.

[18] J. Båtsman Hilmersson, E. Forsberg, I. Gustafsson, I. Hansson, M. Hästmark, and D. Persson, "Exo Explorer: A procedurally generated solar system - An application for interactive exploration of a procedurally generated solar system containing diverse ecosystems," 2023. [Online]. Available: https://odr.chalmers.se/items/cb1b0a57-aa0f-41d0-9d73-b11797e926e1

[19] R. J. Vitacion and L. Liu, "Procedural Generation of 3D Planetary-Scale Terrains," in *2019 IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, 2019, pp. 70–77. doi: 10.1109/SMC-IT.2019.00014.

[20] E. of Mathematics, "Scalar field." Accessed: May 15, 2025. [Online]. Available: http://encyclopediaofmath.org/index.php?title=Scalar_field&oldid=12207

[21] 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.

[22] D. Meagher, "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer," 1980.

[23] P. R. Computing, "GPU Computing." [Online]. Available: https://researchcomputing.princeton.edu/support/knowledge-base/gpu-computing

[24] Unity, "Introduction to compute shaders." [Online]. Available: https://docs.unity3d.com/Manual/class-ComputeShader-introduction.html

[25] Mojang Studios, "Minecraft." [Online]. Available: https://www.minecraft.net/

[26] Mobius Digital, "Outer Wilds." Accessed: May 16, 2025. [Online]. Available: https://www.mobiusdigitalgames.com/outer-wilds.html

[27] Noclip, "The Making of Outer Wilds - Documentary." Accessed: Apr. 30, 2025. [Online]. Available: https://youtu.be/LbY0mBXKKT0?si=aI-4r7Z_cRVI5byn

[28] Unity Technologies, "Unity Home page." [Online]. Available: https://unity.com/

[29] Agile Alliance, "Agile 101." Accessed: Feb. 03, 2025. [Online]. Available: https://www.agilealliance.org/agile101/

[30] "Kanban." [Online]. Available: https://www.crisp.se/gratis-material-och-guider/kanban

[31] Git, "Git –fast-version-control." [Online]. Available: https://git-scm.com/

[32] GitHub, [Online]. Available: https://github.com/

[33] Git, "git-branch." [Online]. Available: https://www.w3schools.com/git/git_branch.asp

[34] The Godot Foundation, "Godot - Your free, open-source game engine.." [Online]. Available: https://godotengine.org/

[35] Godot Commuity, "Godot Features." Accessed: Feb. 11, 2025. [Online]. Available: https://godotengine.org/features/

[36] Godot Commuity, "What is GDExtension?." Accessed: Feb. 13, 2025. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html#supported-languages

[37] Gamers Nexus, "Dragon's Dogma 2 is a Mess: GPU & CPU Benchmarks, Bottlenecks, & Crashes," 2024, [Online]. Available: https://gamersnexus.net/game-benchmarks-graphics-guides/dragons-dogma-2-mess-gpu-cpu-benchmarks-bottlenecks-crashes

[38] NVIDIA Developer Guides, "Analysing Stutter – Mining More from Percentiles," [Online]. Available: https://developer.nvidia.com/content/analysing-stutter-%E2%80%93-mining-more-percentiles-0

[39] Gamers Nexus, "What Are 1% & 0.1% Lows?." [Online]. Available: https://www.youtube.com/watch?v=uXepIWi4SgM

[40] A. Hudaib, R. Masadeh, M. H. Qasem, and A. Alzaqebah, "Requirements Prioritization Techniques Comparison," *Modern Applied Science*, vol. 12, Jan. 2018, doi: 10.5539/mas.v12n2p62.

[41] W. C. Chow, "Fractal (fractional) Brownian motion," *WIREs Computational Statistics*, vol. 3, no. 2, pp. 149–162, 2011, doi: https://doi.org/10.1002/wics.142.

[42] I. Quilez, "fBm." [Online]. Available: https://iquilezles.org/articles/fbm/

[43] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, doi: https://doi.org/10.5281/zenodo.10916799.

[44] S. H. Liu and L. Pietronero, "Fractal," *AccessScience*, 2020, doi: 10.1036/1097-8542.270750.

[45] "OpenGL Wiki - Compute Shader." [Online]. Available: https://www.khronos.org/opengl/wiki/Compute_Shader

[46] Godot Engine Contributors, "WorkerThreadPool — Godot Engine 4.4 documentation." Accessed: Apr. 11, 2025. [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_workerthreadpool.html

[47] E. Lengyel, "Voxel-Based Terrain for Real-Time Virtual Simulations," Davis, CA, 2010.

[48] "Population of a Large-Scale Terrain." Accessed: Apr. 18, 2025. [Online]. Available: https://jorisar.github.io/portfolio/posts/population/

[49] C. Choy, "Barycentric Coordinate for Surface Sampling." Accessed: Apr. 18, 2025. [Online]. Available: https://chrischoy.github.io/research/barycentric-coordinate-for-mesh-sampling/

[50] R. Bridson, "Fast Poisson disk sampling in arbitrary dimensions.," *SIGGRAPH sketches*, vol. 10, no. 1, p. 1, 2007, [Online]. Available: https://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf

[51] M. Vose, "A linear algorithm for generating random numbers with a given distribution," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 972–975, 1991, doi: 10.1109/32.92917.

[52] U. Technologies, "Introduction to normal maps (bump mapping)." [Online]. Available: https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html

[53] S. Weiss, F. Bayer, and R. Westermann, "Triplanar Displacement Mapping for Terrain Rendering," in *Eurographics 2020 - Short Papers*, A. Wilkie and F. Banterle, Eds., The Eurographics Association, 2020. doi: 10.2312/egs.20201016.

[54] G. E. Contributors, "Advanced post-processing." [Online]. Available: https://docs.godotengine.org/en/latest/tutorials/shaders/advanced_postprocessing.html

[55] G. E. Contributors, "Screen-reading shaders." [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/shaders/screen-reading_shaders.html#screen-texture

[56] G. E. Contributors, "Built-in functions." [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/shaders/shader_reference/shader_functions.html#shader-func-mix

[57] Martin Donald, "Planet atmospheres, ray-sphere intersections.." [Online]. Available: https://www.youtube.com/watch?v=OCZTVpfMSys

[58] Sebastian Lague, “Coding Adventure: Atmosphere.” [Online]. Available: https://www.youtube.com/watch?v=DxfEbulyFcY

[59] “Rayleigh scattering.” [Online]. Available: https://www.britannica.com/science/Rayleigh-scattering

[60] “Area3D.” [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_area3d.html

[61] J. Gangestad, “Celestial mechanics,” 2025, doi: 10.1036/1097-8542.115600.

[62] I. Newton, *Philosophiæ naturalis principia mathematica*. London: Jussu Societatis Regiæ ac Typis Joseph Streater, 1687.

[63] J. Barnes and P. Hut, “A hierarchical O(N log N) force-calculation algorithm,” *Nature*, vol. 324, no. 6096, pp. 446–449, Dec. 1986, doi: 10.1038/324446a0.

[64] I. Gargantini, “An Effective Way to Represent Quadtrees,” *Communications of the ACM*, vol. 25, no. 12, pp. 905–910, 1982, doi: 10.1145/358728.358738.

[65] G. M. Morton, “A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing,” Ottawa, Canada, 1966.

[66] Rayon developers, “Rayon: Simple work-stealing parallelism for Rust .” Accessed: May 01, 2025. [Online]. Available: https://github.com/rayon-rs/rayon

[67] L. Euler, *Institutionum calculi integralis*. Petropoli [St. Petersburg]: Impensis Academiae Imperialis Scientiarum, 1768.

[68] M. Brorson, *1.7: Symplectic integrators*. Northeastern University, 2022. Accessed: May 01, 2025. [Online]. Available: https://math.libretexts.org/Bookshelves/Differential_Equations/Numerically_Solving_Ordinary_Differential_Equations_(Brorson)/01%3A_Chapters/1.07%3A_Symplectic_integrators

[69] J. Aparicio and B. Heisler, “criterion.rs: Statistics-driven micro-benchmarking library.” Accessed: May 01, 2025. [Online]. Available: https://github.com/bheisler/criterion.rs

[70] DeveloperOats, “3D – Sun.” [Online]. Available: https://godotshaders.com/shader/3d-sun-shader/

[71] NASA, “Planetary Fact Sheet - Metric,” [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/factsheet/

[72] S. Lague, “Coding Adventure: Procedural Moons and Planets.” Accessed: May 15, 2025. [Online]. Available: https://youtu.be/lctXaT9pxA0?si=exV4Qonfe3ODPRhs

[73] I. Quilez, “Smooth minimum for SDFs.” Accessed: May 16, 2025. [Online]. Available: https://iquilezles.org/articles/smin/

[74] McGraw-Hill, *McGraw-Hill Dictionary of Scientific and Technical Terms, 6th ed.* New York: McGraw-Hill, 2003.

[75] Swinburne University of Technology, “Disk Galaxies.” [Online]. Available: https://astronomy.swin.edu.au/cosmos/D/Disk+Galaxies

[76] Blender Foundation, “Blender - Home of the Blender project - Free and open 3D Creation Software.” [Online]. Available: https://www.blender.org/

[77] 5 Minutes Blender, “Create Stars In 2 Minutes In Blender | Night Sky & Starfield | Easy & Dirty Method | Blender Eevee.” [Online]. Available: https://www.youtube.com/watch?v=NJPFHRrTKXo

[78] Editors of Encyclopaedia Britannica, “Astronomical unit.” [Online]. Available: https://www.britannica.com/science/astronomical-unit

[79] GeeksforGeeks, “Hash Functions and Types of Hash functions.” [Online]. Available: https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/

[80] Godot Engine Contributors, “MeshInstance3D — Godot Engine 4.4 documentation.” [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_meshinstance3d.html

[81] Godot Engine Contributors, “MultiMeshInstance3D — Godot Engine 4.4 documentation.” [Online]. Available: https://docs.godotengine.org/en/stable/classes/class_multimeshinstance3d.html

[82] The Editors of Encyclopaedia Britannica, “Stellar classification.” [Online]. Available: https://www.britannica.com/science/stellar-classification

[83] International Astronomical Union (IAU), “Buying Stars and Star Names.” [Online]. Available: https://www.iau.org/public/themes/buying_star_names/

[84] S. Klabnik and C. Nichols, *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. [Online]. Available: https://doc.rust-lang.org/book/

[85] OpenAI Studios, “ChatGPT.” [Online]. Available: https://chatgpt.com/

[86] Microsoft, “Copilot.” [Online]. Available: https://copilot.microsoft.com/

[87] Google, “Gemini.” [Online]. Available: https://gemini.google.com/

[88] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. USA: Cambridge University Press, 2007.

[89] L. Lazaridis and G. F. Fragulis, “Creating a Newer and Improved Procedural Content Generation (PCG) Algorithm with Minimal Human Intervention for Computer Gaming Development,” *Computers*, vol. 13, no. 11, 2024, doi: 10.3390/computers13110304.

[90] A. Sjögren, “Hälften av de anställda sparkas – ersätts av AI.” [Online]. Available: https://www.aftonbladet.se/nyheter/a/4oAKWe/halften-av-de-anstallda-sparkas-ersatts-av-ai

[91] R. P. de Araujo and V. T. Souto, “Game Worlds and Creativity: The Challenges of Procedural Content Generation,” *Design, User Experience, and Usability: Designing Pleasurable Experiences*. Springer, Cham, pp. 443–455, 2017. doi: 10.1007/978-3-319-58637-3_34.

Appendix A : Specifications for benchmarking computers

Benchmarking was planned to be performed on a dedicated benchmarking computer named PC-1 (Table A-1), and was used for the final result. However, during Section 4 other machines are occasionally used when PC-1 is not available. These are PC-2 (Table A-2), PC-3 (Table A-3), and PC-4 (Table A-4).

Table A-1: PC-1 Specifications

Component	**Specification**
CPU	Intel i7-11800H
GPU	Nvidia GeForce RTX 3050 Ti Mobile
RAM	64 GB DDR4 3200 MHz

Table A-2: PC-2 Specifications

Component	**Specification**
CPU	AMD Ryzen 7 7800X3D
GPU	NVIDIA GeForce RTX 4080
RAM	32GB DDR5 6000MHz

Table A-3: PC-3 Specifications

Component	**Specification**
CPU	Intel Core i7-8850H
GPU	NVIDIA Quadro P1000
RAM	32GB DDR4 2267MHz

Table A-4: PC-4 Specifications

Component	**Specification**
CPU	Intel Core i7-13700K
GPU	NVIDIA GeForce RTX 4080
RAM	32GB DDR5 5600MHz

Appendix B Pseudocode for the Direct Summation algorithm

function DIRECT-SUMMATION-GRAVITY(G, particles)
    ▷ Get the number of particles
    $n \leftarrow$ **length** (particles)

    ▷ Initialize acceleration array
    acc $\leftarrow$ **array** (**Vec3** (0, 0, 0); n)

    ▷ For each particle i, calculate acceleration from each particle $j \neq i$
    for i **in** $0..n$ **do**
        for $j \neq i$ **in** $0..n$ **do**
            $\vec{r}_{ij} \leftarrow$ particles[j].**position** - particles[i].**position**
            $m_j \leftarrow$ particles[j].**mass**

            ▷ Note: Softening factor ε omitted for brevity, but used in implementation
            if $|\vec{r}_{ij}| > 0$ **then**
                $\vec{a}_{ij} \leftarrow G\frac{m_j}{|\vec{r}_{ij}|^3}\vec{r}_{ij}$
                acc[i] $\leftarrow$ acc[i] $+\vec{a}_{ij}$
    return acc

Appendix C MoSCoW Table

Table C-1: The project's MoSCoW Table showing which tasks were finished (marked green), and which tasks were unfinished (marked red). "Won't have" is elided since the relevant parts are detailed in Section 1.2

Must Have	Should Have	Could Have

• Working terrain-generation using computer generated noise.
• There should be at least one procedurally generated planet and sun.
• The planet should orbit the stars with as accurately as possible to the laws of physics.
• A way to explore the generated terrain using a camera controller.
	• To explore different performance techniques to achieve effective procedural generation of planets
	  ▸ Occlusion, backface, etc, -culling
	  ▸ Levels of detail (LOD)
	  ▸ Applied spatial data structures (E.g octrees and grids).
	  ▸ Chunking
	• A technique to draw complex terrain with overhangs and the like, e.g Marching Cubes.
	• Solar systems can have multiple planets.
	• Some planets should have moon(s).
	• Galaxy. A procedurally generated galaxy constructed out of solar systems
	• Galaxy traversal. Ways of navigating between solar systems, using a UI galaxy map or traveling from solar systems in real-time. Zooming out: Solar system level, solar systems in proximity, galaxy level. (examples)
	• There should be shaders to improve the look of things in the solar system, such:
	  ▸ Planet atmospheres
	  ▸ Water on planets
	• Player abides by the laws of physics, interacting with planets by walking for example.
		• Get interesting info about planets and solar systems, such as information about a planet's mass, amount of moons
		• Planet properties like temperature that change according to, for example, distance from the sun. Hot at equator, cold at poles, etc.
		• UI for navigating between planets/solar systems/ galaxies (fast travel)
		• Other celestial bodies. E.g. asteroids, black holes, asteroid belts, gas planets, nebulae, etc.
		• Vegetation