



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG



Regolith

Simulation and Procedural Generation of Planetary Systems

Bachelor's thesis in Computer Science and Engineering

Jimmy Dahlberg

Filip Hansson

Isaac Johannes Ketto

Didrik Nilsson

Simon Renhult

Peter Ta

BACHELOR'S THESIS 2026

Regolith

Simulation and Procedural Generation of Planetary Systems

Jimmy Dahlberg

Filip Hansson

Isaac Johannes Ketto

Didrik Nilsson

Simon Renhult

Peter Ta



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Regolith

Simulation and Procedural Generation of Planetary Systems

Jimmy Dahlberg Filip Hansson Isaac Johannes Ketto Didrik Nilsson Simon Renhult
Peter Ta

© Jimmy Dahlberg, Filip Hansson, Isaac Johannes Ketto, Didrik Nilsson,
Simon Renhult, Peter Ta 2026.

Supervisor: Staffan Björk, Department of Computer Science and Engineering

Examiner: Arne Linde, Department of Computer Science and Engineering

Graded by teacher: Aris Alissandrakis, Department of Computer Science and
Engineering

Bachelor's Thesis 2026

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 barren terrestrial planet with clouds and a star in the background

Typeset in L^AT_EX

Gothenburg, Sweden 2026

Regolith

Simulation and Procedural Generation of Planetary Systems

Jimmy Dahlberg, Filip Hansson, Isaac Johannes Ketto, Didrik Nilsson,
Simon Renhult, Peter Ta

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

Regolith is a system capable of procedurally generating a planetary system consisting of a star, planets, and moons. Several technical sub-systems were implemented and integrated, including a gravitational N-body simulation, procedural terrain generation using marching cubes, atmospheric rendering, gas giant generation, and level-of-detail optimization. The project achieved its purpose of providing the authors with valuable experience and knowledge in procedural generation, simulation, rendering, and large-scale software development. The results of this thesis can benefit other developers by providing insight into the strategies, design decisions, and implementation methods used throughout the project. In particular, the work can serve as a reference for developers interested in creating their own procedural planetary systems or similar large-scale simulations.

Keywords: procedural generation, planet, solar system, n-body simulation, computer graphics, space, marching cubes, atmosphere

Acknowledgements

We would like to thank our supervisor, Staffan, for taking the time to help us through this project and answering our questions.

Jimmy Dahlberg, Filip Hansson, Isaac Johannes Ketto, Didrik Nilsson,
Simon Renhult, Peter Ta, Gothenburg, June 2026

Contents

List of Figures	xi
List of Tables	xiv
Glossary	xvii
1 Introduction	1
1.1 Background	1
1.2 Purpose	2
1.3 Scope	2
2 Theory	5
2.1 Celestial mechanics	5
2.1.1 Hill sphere	5
2.1.2 Roche limit	6
2.1.3 Two-body problem	6
2.1.4 N-body problem	6
2.2 Numerical methods for ordinary differential equations	6
2.2.1 Euler's method	7
2.2.2 Leapfrog integration	7
2.3 Gravitational N-body simulations	7
2.3.1 Direct methods	8
2.3.2 Approximation methods	8
2.4 Marching cubes	8
2.5 Atmospheres	9
2.6 Signed distance fields	10
2.7 Noise	10
2.7.1 Perlin noise	10
2.7.2 Fractal noise	10
2.7.3 Domain warping	11
2.7.4 Voronoi noise	11
3 Methodology	13
3.1 Unity	13
3.1.1 Lighting and shadows	13
3.2 GitHub	14

3.3	Software development methodologies	14
3.4	Planning	14
3.4.1	Research	15
3.4.2	Project plan	15
3.4.3	Minimal viable product	15
4	Process	17
4.1	Planetary system simulation	18
4.1.1	Why an N-body simulation?	18
4.1.2	Direct or approximation?	19
4.1.3	Choice of integration method	19
4.1.4	Visualizing orbits	20
4.1.5	Avoiding chaotic behavior	20
4.2	Flying camera	21
4.3	Skybox	21
4.4	Simulating the gigantic scale of space	23
4.5	Noise	23
4.6	Terrestrial generation	24
4.6.1	Marching cubes	24
4.6.2	Generating the terrain data	25
4.7	Combining the sub-systems	26
4.8	Adding moons to the simulation	27
4.8.1	Are you worthy of a moon?	28
4.9	Use of astronomical data	29
4.10	Gas planets	30
4.11	Splitting planets into chunks	31
4.12	Deforming the terrain	32
4.13	Walkable planets	33
4.14	Surface colors	33
4.15	Lighting	34
4.15.1	Shadows	35
4.16	Atmospheres	36
4.16.1	Modelling atmospheric scattering	37
4.17	Clouds	39
4.18	Optimizing planet generation	41
4.18.1	Level of detail	41
4.18.2	Moving chunk generation to the GPU	42
5	Result	45
5.1	An overview of Regolith	46
5.2	The technical sub-systems	46
5.2.1	Simulation	46
5.2.2	Terrain generation of planets and moons	47
5.2.3	Gas planets	47
5.2.4	Atmospheres and clouds	48
5.2.5	Skybox	49

6	Discussion	51
6.1	Result	51
6.2	Process	51
6.3	Ethics and sustainability	52
6.3.1	Use of generative AI during development	52
6.3.2	Contributions to open source	53
6.3.3	Procedural generation and its effect on designers	53
6.4	Future work	53
6.4.1	Planets	54
6.4.2	Planetary system	54
7	Conclusion	57
	Bibliography	59

List of Figures

1.1	Example screenshots from different games.	2
2.1	Marching cubes 14 unique triangulated cube cases By Ryoshoru—Own work, CC BY-SA 4.0, via Wikimedia Commons.	9
2.2	Examples of different kinds of noise.	12
4.1	Visualized orbits for three planets orbiting a star.	20
4.2	Visible stretching on the skybox.	22
4.3	Screenshots from different stages of implementing marching cubes. . .	25
4.4	Mapping from continentalness to elevation.	26
4.5	A moon orbiting a planet incorrectly due to inherited transform. . . .	27
4.6	Two moons orbiting a planet in the outer system at the start of the simulation.	28
4.7	Comparison of Hill sphere and Roche limit for the Sun-Earth-Moon system by Cmglee, CC BY-SA 4.0, via Wikimedia Commons.	29
4.8	Side by side comparison of 3D noise (left) and 2D noise with visible seam (right).	31
4.9	Gas planets with different parameters using colors similar to Jupiter's. .	31
4.10	Final version of the gas planet shader featuring lighting and a higher mesh resolution.	32
4.11	Planet with one chunk pulled out.	32
4.12	Atmosphere based on distance traveled inside the physical bounding sphere.	36
4.13	Atmosphere using Cornette-Shanks phase function.	37
4.14	Atmosphere with Rayleigh and Mie scattering.	38
4.15	The max height of the atmosphere showing beneath mountains. . . .	38
4.16	Screenshots of the final atmosphere.	39
4.17	First look at clouds.	39
4.18	Screenshots of clouds at current implementation stage.	40
4.19	Screenshots from different stages of implementing level of detail. . . .	41
4.20	Planet generated using clipmaps. The different colors show the different detail levels.	42
5.1	An overview of orbits in the simulation after the outermost planet has completed one rotation.	47
5.2	Examples of different terrestrial types and moons.	48

5.3 Two different gas giants.	48
---------------------------------------	----

List of Tables

3.1	MoSCoW categorization of planned features for the project.	16
4.1	List of sprints and which tasks were performed during them.	17
5.1	Implementation status of features from the MoSCoW table.	45

Glossary

bake To calculate something and store it for later use.

central body A celestial body that has considerably smaller bodies (satellites) in orbit around it.

compute shader A shader that uses the GPU for performing computational work but does not produce graphics [1].

conic section An intersection between a plane and a cone.

fragment shader A shader that runs computations for coloring and texturing on every fragment (pixel) on the screen [2].

level of detail A rendering technique that adjusts the detail of objects depending on the distance to the viewer [3].

mesh A structured discretization of a geometric domain into small, simple elements such as triangles or quadrilaterals [4].

noise function A pseudorandom function with certain statistical properties.

octree A three dimensional tree data structure that divides space by recursively splitting it in to octants.

perturbation Deviation from the motion of a celestial body that has a single force acting on it.

procedural generation The process of creating data with algorithms rather than by hand.

raycast To calculate the closest intersection between a ray and its surroundings.

raymarch To move along a ray by a given step size.

satellite A celestial body that orbits another body (central body) with considerably larger mass.

seed A number that can be used to initialize a pseudo-random number generator.

semi-major axis Half of the diameter of the longest diameter of an ellipse.

shader A program that runs on the GPU, often to produce graphical effects [2].

skybox A large cube surrounding the scene with textures applied to each of the 6 internal sides [5].

vertex shader A shader that takes in data corresponding to points (vertices) in a 3D space and transforms them into a 2D position on the screen [2].

1

Introduction

With the introduction of open world games where you can freely explore, developers are faced with a problem. How are these worlds created? One option is to manually create the worlds, placing every detail by hand. Although this gives full control over the final result, it limits the scope of exploration to what has been designed ahead of time. It is also time consuming and may affect the re-playability of the game. Another option is to procedurally generate the world, where the placement of most or all objects is determined algorithmically. This allows for possibly infinite exploration without having to spend excessive time on creating the world itself.

1.1 Background

There are many games that employ procedural generation of planets. One such example is *No Man's Sky* [6], which procedurally generates the entire universe. The game allows for exploration of both different planets and galaxies that all have unique properties. Structures and other noteworthy objects exist on planets, usually with large distances between them. *Astroneer* [7] on the other hand has a smaller scope. It allows the player to explore one planetary system with more detailed planets than *No Man's Sky*. The emphasis is on exploring the different planets rather than space between them or the planetary system. The planets themselves contain vast cave systems. Examples of both games can be seen in Figure 1.1.

An earlier example of procedural generation related to space can be found in the 1984 game *Elite* [8]. They had eight galaxies consisting of 256 planets, each of which were procedurally generated due to hardware limitations [9]. *Minecraft* [10] is another game that employs procedural generation, although not for planetary systems.

Exo Explorer [11] is a bachelor's project at Chalmers from 2023 that covers the development of a procedurally generated planetary system in the Unity game engine. They cover terrain generation with the marching cubes algorithm, ecosystem simulation on the planets and interactive exploration for the user. Sebastian Lague [12] created a notable project on the topic of procedurally generated planetary systems. It covers the development of a simulated planetary system with procedurally generated planets and moons.

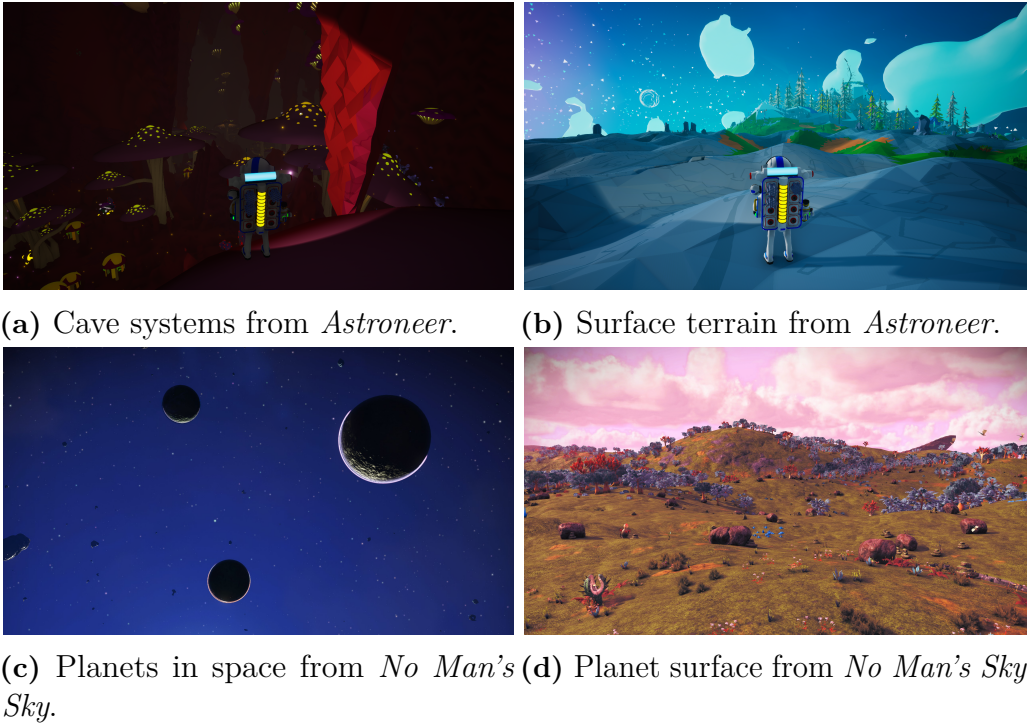


Figure 1.1: Example screenshots from different games.

1.2 Purpose

The purpose of this project is to explore and learn about procedural generation, real-time rendering, and physics simulation. To facilitate this, we aim to create a simulation consisting of procedurally generated planets and moons orbiting a central star, where their motion is determined by a simulation based on Newtonian gravity. The generation will be seeded and deterministic, allowing for many unique planetary systems while staying consistent on revisits. User exploration will be designed to be seamless when changing between traveling through space and walking on terrestrial planets, allowing the user to visit every part of the planetary system.

The project will be considered successful if we manage to implement the simulation as described. Achieving this outcome demonstrates that we have gained knowledge and experience in procedural generation, physics simulation, and real-time rendering. Based on the challenges encountered and solutions developed throughout the project, we aim to provide insights and recommendations to developers interested in implementing similar projects.

1.3 Scope

It was decided early that Regolith will not be a game, but rather a technical demonstration of what we have learned. This means that developing a storyline, UI, and other functions that provide an enjoyable player experience is considered out of scope. Instead, the intention is to focus on exploring, experimenting with,

and understanding how planetary systems can be generated and simulated through controllable parameters.

We limit the deterministic generation of planetary systems to one at a time, as the main focus of the project lies at a planet-star scale, not a galactic scale. This applies both for the physics simulation and for terrain generation. This also means that surface objects—rocks, trees, and grass—are not prioritized. Neither are fluid simulations or ecosystems, as they are deemed too complex for the limited time available. Detailed terrain generation will be limited to terrestrial planets and moons as the goal is to allow for surface exploration on them. For gas giants, a simpler surface texture technique is enough to achieve a balance between graphical fidelity and computational performance as they do not have a defined surface.

The project does not aim to be realistic, but rather takes inspiration from observational data of our own solar system, exoplanets, and exoplanetary systems. This is to achieve a simulation with believable behavior and appearance, all while maintaining computational feasibility.

2

Theory

This chapter presents the theoretical background required to understand procedural generation and simulation of planetary systems. This includes Newtonian physics, methods of integration, and techniques for procedural geometry generation. It also presents a few different noise techniques, which are commonly used in procedural generation.

2.1 Celestial mechanics

Celestial mechanics deals with the movement of celestial bodies, how they interact and influence each other. The science began in the early 16th century when Johannes Kepler devised his three laws of motion based on observational data. Later in 1687 Isaac Newton was able to mathematically explain, with his theory of gravitation and calculus, the movement Kepler observed [13]. Thus the core equation for celestial mechanics was formed. It describes the gravitational force $\vec{F}_{ij}$ on body i inflicted by body j where m_i and m_j are their respective masses, the gravitational constant G , the distance r_{ij} between them and its unit vector $\hat{r}_{ij}$.

$$\vec{F}_{ij} = -G \frac{m_i m_j}{r_{ij}^2} \hat{r}_{ij} \quad (2.1)$$

2.1.1 Hill sphere

The Hill sphere is a model of a gravitational sphere of influence and is defined as the volume of space around a satellite in which its gravitational force dominates over its central body [14]. The radius of the Hill sphere, r_{hill} , can be calculated using Equation 2.2 with the semi-major axis a of the satellites orbit, the satellite mass m and the central body mass M .

$$r_{hill} = a \sqrt[3]{\frac{m}{3M}} \quad (2.2)$$

2.1.2 Roche limit

The Roche limit is defined as the furthest distance from the center of a celestial body where a satellite would be torn apart by tidal forces [14]. It can be calculated with Equation 2.3 where r_{roche} is the Roche limit, r_M is the radius of the celestial body, ρ_M its density and ρ_m the density of the satellite.

$$r_{roche} = r_M \sqrt[3]{\frac{2\rho_M}{\rho_m}} \quad (2.3)$$

2.1.3 Two-body problem

With Kepler's laws it is possible to analytically solve for the motion of a satellite and its central body, which is called a two-body problem. The motion of each body will form a conic section. In reality, this can never be entirely precise since other forces, such as gravitational force from a third body or radiation pressure from a star, can cause perturbations. Thus the two-body problem is only an approximation for motion of two bodies in a system with multiple bodies [13].

In a two body system where the mass of one body is negligible in comparison to the other mass, the orbital velocity v can be approximated with the formula:

$$v = \sqrt{\frac{GM}{a}} \quad (2.4)$$

where G is the gravitational constant, M is the mass of the central body, and a is the semi-major axis of the satellite.

2.1.4 N-body problem

The N-body problem appears when three or more bodies exert gravitational force on each other. Unlike the two-body problem, it is not analytically solvable since the complexity increases drastically for each added body [13]. It is known as a chaotic system, which means that variation in initial conditions can cause drastic changes to the behavior of the system and is effectively indistinguishable from a random process even though it is deterministic from a mathematical perspective [15].

2.2 Numerical methods for ordinary differential equations

Ordinary differential equations appear in the modeling of almost any scientific discipline and generally do not have an exact solution, which is why numerical methods are used to approximate it. A first order differential equation can be written as an initial value problem (IVP):

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases} \quad (2.5)$$

For the IVP, a numerical integration method produces an approximate solution at discrete time steps. The choice of integrator determines the order of accuracy and conservation properties of the approximation [16].

2.2.1 Euler's method

Euler's method is a first order integrator and can be written as

$$y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n) \quad (2.6)$$

where y_{n+1} is the next state, y_n the current state, Δt is the time step and f is the function that evolves the state.

To describe movement through space, Euler's method can be applied to a position $\vec{r}$, velocity $\vec{v}$, acceleration $\vec{a}$, and time step Δt to get the following set of equations:

$$\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n \Delta t \quad (2.7)$$

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n \Delta t \quad (2.8)$$

Because Euler is a first order method, the error produced after n steps will be approximately proportional to Δt , meaning reducing the time step will only slightly improve the accuracy. Thus, a higher order method is preferred for better accuracy over time [16].

2.2.2 Leapfrog integration

The leapfrog integrator is a second order symplectic integrator. Symplectic means that over long simulations, the integrator prevents unphysical drifts in total energy, keeping the error oscillating rather than growing with time [17]. The kick-drift-kick variation can be written as

$$\vec{v}_{n+1/2} = \vec{v}_n + \vec{a}_n \frac{\Delta t}{2} \quad (2.9)$$

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_{n+1/2} \Delta t \quad (2.10)$$

$$\vec{v}_{n+1} = \vec{v}_{n+1/2} + \vec{a}_{n+1} \frac{\Delta t}{2} \quad (2.11)$$

where $\vec{r}$ is the position, $\vec{v}$ is the velocity, $\vec{a}$ is the acceleration, and Δt is the time step. In this version, one half-step is taken in the velocity (kick), then a full-step is taken in the position (drift) and finally a second kick. Because of the low computational cost for a second order symplectic integrator it is a common method used for N-body simulations [18].

2.3 Gravitational N-body simulations

A gravitational N-body simulation is a numerical solution to the equations describing the movement of n particles in a system governed by gravity. The force $\vec{F}_i$ acting on the body i with a mass of m_i in a N-body system is computed by Equation 2.12 [19].

$$\vec{F}_i = -G \sum_{j \neq i} \frac{m_i m_j}{r_{ij}^2} \hat{r}_{ij} \quad (2.12)$$

When the initial position and velocity are provided for each particle, a unique solution can be found through numerical integration (covered in section 2.2).

2.3.1 Direct methods

Direct methods, also known as brute force methods, compute the gravitational force for every particle by summing up forces from every other particle as seen in Equation 2.12. They do not introduce any approximations which make them extremely precise, but come at a cost in computational time. The number of pairwise interactions that must be computed per time step is:

$$n_{\text{pairs}} = \frac{n(n-1)}{2} \sim \mathcal{O}(n^2) \quad (2.13)$$

The quadratic time complexity of direct methods generally becomes problematic for $n > 10^4$ [19].

2.3.2 Approximation methods

To avoid the exponential time growth of direct methods, approximations such as grouping particles based on their position or by looking at the gravitational potential of the system can be made [19]. One example is the Barnes-Hut algorithm [20] which utilizes an octree to recursively divide space into cubes. These generalize the gravitational pull of distant objects in the same cube to just a singular object with the center of mass in the middle of the cube. It has a time complexity of $\mathcal{O}(n \log n)$ at the cost of accuracy for distant particles.

2.4 Marching cubes

Marching cubes [21] is a high resolution 3D surface constructing algorithm that creates polygonal surfaces from volumetric data. The algorithm processes the volumetric data in scan-line order and determines triangle vertices through linear interpolation along the grid edges. The algorithm identifies the location of the surface for a user-defined scalar value and generates a set of triangles approximating that surface.

Marching cubes uses a divide-and-conquer approach by decomposing the volumetric data into a regular grid of cubes. For each cube, the algorithm determines how the surface intersects it before proceeding to the next cube. Each of the eight corners of the cube is classified as either inside or outside the surface depending on whether its scalar value is below or above the chosen threshold. Edges that connect corners with differing classifications are intersected by the surface.

Since each cube has eight vertices with two possible states, there are $2^8 = 256$ possible configurations. However, many of these configurations are equivalent under rotation

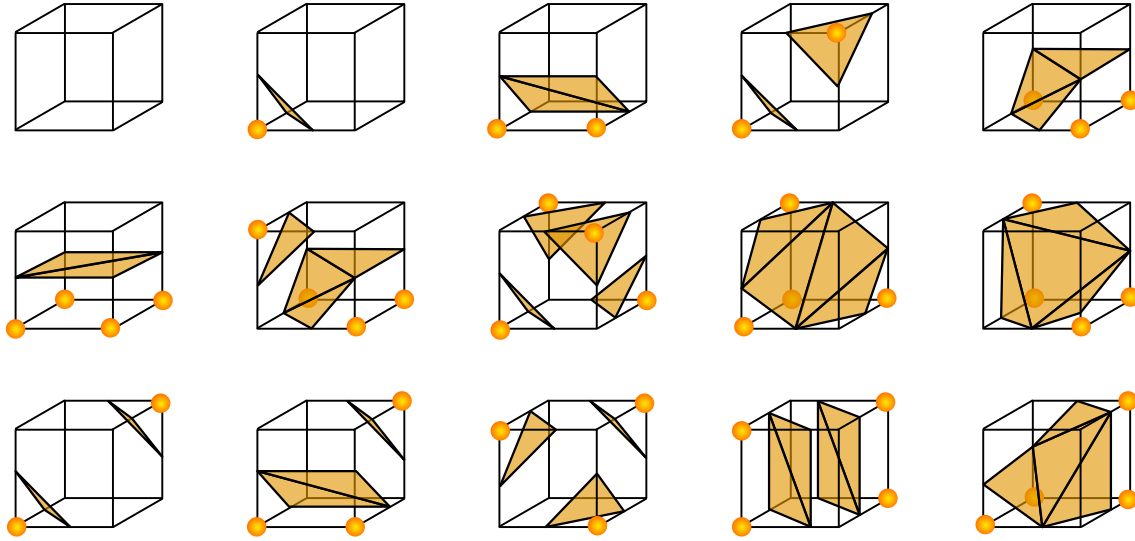


Figure 2.1: Marching cubes 14 unique triangulated cube cases By Ryoshoru—Own work, CC BY-SA 4.0, via Wikimedia Commons.

and reflection. By exploiting the symmetry of the cube, these can be reduced to 14 unique cases. For example, if all corners are on the same side of the surface, no triangles are generated. In contrast, if a single vertex differs from the others, the surface intersects three edges, forming a single triangle. Figure 2.1 shows the 14 unique triangulation cases that can be produced from a cube.

To efficiently determine the appropriate triangulation for a cube, an 8-bit index is constructed by assigning each corner a bit based on its classification. This index serves as a lookup key into a precomputed table that specifies which edges are intersected and how triangles should be formed. The exact intersection points along the edges are computed using linear interpolation between the scalar values at the vertices. The interpolation step produces a smooth surface and avoids the blocky appearance that would result from placing the vertices at the midpoints between corners.

2.5 Atmospheres

An atmosphere is defined as the mix of gases enveloping a planet. In the case of Earth, it is made up of primarily nitrogen and oxygen. The density of these gases vary depending on altitude [22]. The scale height models the distance it takes for the atmospheric density to decrease by a factor of e . It can be computed using

$$H = \frac{RT}{Mg} \quad (2.14)$$

where R is the universal gas constant, T is the temperature, M is the molecular weight and g is gravity [22]. This is useful during atmospheric calculations to get the density at a given altitude.

There are two types of scattering that occur in the atmosphere: Rayleigh and Mie scattering. Rayleigh scattering models how light interacts with air molecules. It is important to note that Rayleigh scattering is wavelength dependent and as such sunsets on earth are yellow or orange while the sky above is blue. Mie scattering models how light interacts with aerosols, which are larger particles in the air such as smog. It is largely wavelength independent [23].

2.6 Signed distance fields

Signed distance fields [24], also known as SDFs, measure the Euclidean distance from a point to a surface. If the function is negative then the point is inside the shape, positive means it is outside of it, zero means it is on the surface. They have been used in computer graphics as a modeling tool for a long time.

2.7 Noise

Noise functions create random values that can be used in a wide range of applications including procedural content generation. They are commonly used in computer graphics to produce natural and interesting looking patterns for textures ranging from rocky surfaces to water [25] or in procedural content generation to for example create natural looking terrain [26]. Many techniques exist for creating or modifying noise, some of which will be discussed in the following sections.

2.7.1 Perlin noise

Perlin noise [25] is one of the earliest noise algorithms. It produces noise by computing pseudorandom gradients for every point in a integer lattice $[x, y, z]$ where the coordinates x, y, z are the gradient values. Every point outside the lattice is then interpolated between lattice equation coefficients first in x (along the lattice edges) then y (within lattice z -faces) then finally z which is then evaluated as a linear function in $[x, y, z]$.

2.7.2 Fractal noise

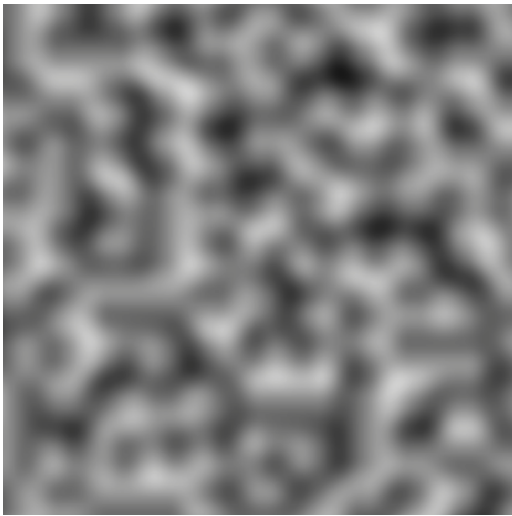
Fractal noise is a technique used on pre-existing noise algorithms to create new ones with fractal patterns and more detail. This is done by applying a function on the noise usually called “fractal brownian motion” or “fbm(x)” for short. $\text{fbm}(x)$ is computed as the sum of multiple noise octaves, that is the sum of noise sampled with increasing frequency and decreasing amplitude [27]. The result can be seen in Figure 2.2b. Compared to noises like Perlin, fractal noise has a pattern with more complex gradients and unique shapes.

2.7.3 Domain warping

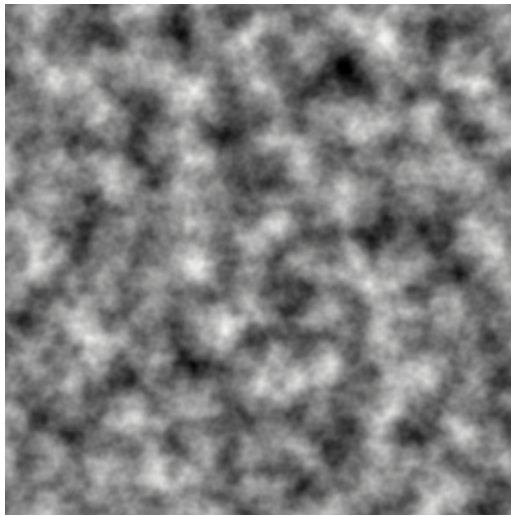
Domain warping is a technique used in computer graphics to apply deformations on graphics. When combined with noise it produces patterns that resemble smoke and marble. The distortion is applied by using fractal noise as an input to the function itself. It is calculated which is calculated as $\text{fbm}(x + \text{fbm}(x))$. Examples are given in Figure 2.2c and Figure 2.2d.

2.7.4 Voronoi noise

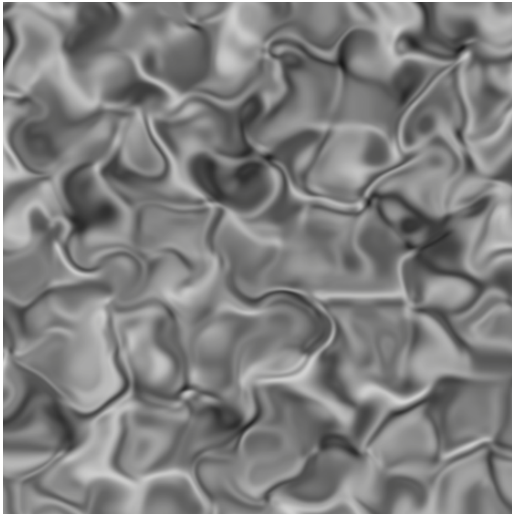
Voronoi noise [28], also known as Worley noise, is a type of noise that can mimic cell looking patterns such as skin, mosaic, modern camouflage, and stone tiles. Given a space $\mathbb{R}^n$ with randomly distributed points, a function $f(x)$ can be defined over $\mathbb{R}^n$ which returns the distance to the closest point. This distance value can then be used as noise data for other functions and algorithms. An example is given in Figure 2.2e.



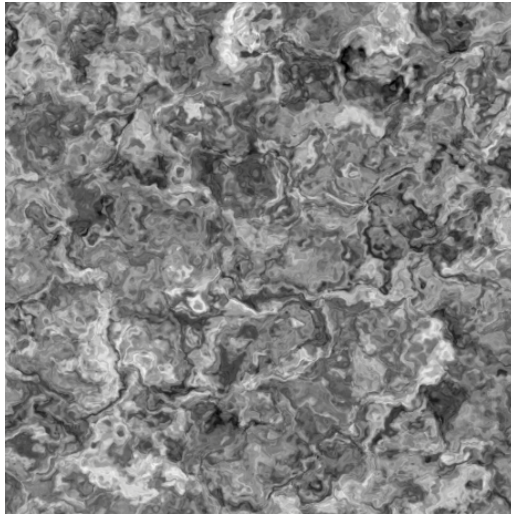
(a) Perlin noise.



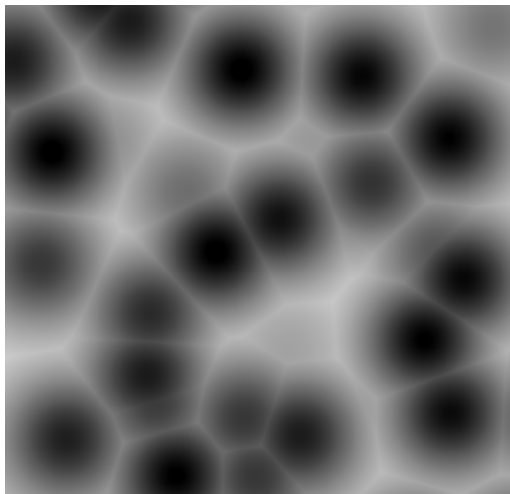
(b) Fractional noise.



(c) Warped noise.



(d) Warped fractional noise.



(e) Vornoi noise.

Figure 2.2: Examples of different kinds of noise.

3

Methodology

This chapter presents tools and strategies used throughout the project. We cover how they work and why they were used.

3.1 Unity

The simulation was implemented in Unity, a popular game engine used around the world to primarily create games [29]. It was chosen primarily due to the team's prior familiarity with the engine. This allowed focus to be directed towards the procedural generation and creation rather than on learning new unfamiliar tools.

The primary tool we used within Unity was the inspector window. It allowed us to work within a scene and change variables in our code in real time to see the changes. This shortened the development time because it allowed us to test features without having to rerun the program each time. Another frequently used tool was the transform component, which allowed us to easily calculate and change positions, rotations, and scale within the simulation. The update functions were also used often, especially the following types:

Update Is called every frame.

Fixed Update Is not frame dependent and runs every 0.02 seconds by default.

Late Update Is executed after all Update functions have been executed.

3.1.1 Lighting and shadows

Lighting is a fundamental component in real-time graphics and game development, as it directly affects the visual quality, atmosphere, and realism of a scene. In the Unity game engine [30], lighting sub-systems are used to simulate how light interacts with objects and environments through techniques such as real-time lighting, baked lighting, and mixed lighting. Unity provides different types of light sources, including directional lights, point lights, and spotlights, each serving different purposes depending on the desired visual effect [31].

Shadows are an important aspect of real-time rendering because they provide depth perception and improve the realism of virtual environments. In Unity, shadows are primarily generated using a technique called shadow mapping [32]. It works by

rendering the scene from the perspective of a light source and storing the depth information in a texture known as a shadow map. During rendering, Unity compares the depth of visible objects with the stored depth values in the shadow map to determine whether a fragment is illuminated or in shadow.

Unity's shadow sub-system supports different shadow types depending on the selected render pipeline and lighting setup. Real-time shadows are updated dynamically and are commonly used for moving lights or objects, while baked shadows are precomputed and stored in lightmaps [33] to reduce performance costs. Unity also provides settings for shadow resolution, shadow distance, and cascade shadows, allowing developers to balance visual quality with rendering performance. In addition, Unity supports soft shadows, which create smoother shadow edges to achieve a more natural appearance, although this requires additional computational resources.

3.2 GitHub

GitHub is a platform that allows developers to create and manage code [34]. It was used as the version control platform for the project to enable collaborative development and online hosting of the source code repository. It was chosen primarily due to the team's prior familiarity with Git-based workflows. This allowed the team to focus on implementing and iterating on functionality rather than learning and configuring new version control tools. GitHub provided a centralized repository with full version history, enabling safe experimentation through branching and rollback of changes when necessary.

3.3 Software development methodologies

To aid with the development of the simulation, ideas from common software development methodologies such as agile software development [35] and incremental build model [36] were used. Usually these methods are used for large scale software development projects. For this project it was more appropriate to adopt some of the concepts found in the methodologies such as incremental development and working in sprints. This has the advantage of splitting the workload of the project into smaller and more manageable goals and gives structure to the development, while continuously reiterating if a goal has unsatisfactory results.

3.4 Planning

In this section three important steps in planning a project in a group are covered. First research in to the topic has to be done, then a project plan can be developed along with a MoSCoW table to set a clear goal and purpose the group can work towards.

3.4.1 Research

Research was conducted both prior to and during the development. Information was primarily sourced through academic search tools such as Google Scholar, the ACM Digital Library, IEEE Explore Digital Library and other similar tools. If a topic lacked academic research, alternative sources such as web material were also considered. When web resources were used a proper evaluation of the source was conducted. Specifically, we researched topics related to procedural generation, terrain generation, gravity simulation, optimization, and rendering techniques. Existing projects and games such as No Man's Sky [6] and Astroneer [7] were studied to better understand the technical challenges and feasible solutions. At the same time, we experimented with Unity individually to gain familiarity with the engine and its relevant sub-systems.

3.4.2 Project plan

We developed a project plan which covered what we wanted to achieve with the project, how feasible it seemed at the time, and the theory relevant to understanding it. The main purpose of the project plan was to develop a mutual understanding in the group of what the goal was, the theory needed to reach it, and how the work should be divided. We identified different technical areas early in the process, such as terrain generation and gravity simulation, allowing members to focus on separate research and development tasks simultaneously.

3.4.3 Minimal viable product

The minimal viable product (MVP) was created as a goal that would implement the core functionality. To determine what the MVP should include we had an extensive meeting listing everything we could think of implementing. We then sorted the features into four priority levels: *must have*, *should have*, *could have*, and *won't have*. This categorization can be seen in Table 3.1. The features categorized as *must have* became our MVP and they were selected since we felt the product would not be in a usable state without them.

Priority	Feature
Must have	Planetary system with one star and one terrestrial planet Basic movement and camera A procedurally generated planet Planets rotate around the star and their own axis Background with stars Basic seeding
Should have	Planetary system with multiple planets and moons A way to land and explore planets Complex movement system Different planet types like gas and ice planets Atmospheres Level of detail Chunks Advanced seeding
Could have	Deformable planets Different types of celestial bodies Exploration of gas giants' atmospheres
Won't have	Multiple planetary systems Bodies of water on planets Ecosystems

Table 3.1: MoSCoW categorization of planned features for the project.

4

Process

This chapter covers the development process of the project. It roughly follows chronological order as most of the sub-systems were developed in parallel and merged a couple of times to ensure they worked together. Table 4.1 details what tasks were performed during which sprints.

Sprint	Task
1	Planning and pre-study N-body simulation Noise Flying camera Skybox
2	Terrestrial generation Surface colors Moons Level of detail Gas planets Demo build
3	New chunk system Refactored N-body simulation Skybox improvement Walkable planets Atmospheres Lighting
4	Shadow mapping Started UI development Improved star and gas planets
5	Procedural textures and better surface colors Shadow mapping improvements Moving chunk generation to the GPU Clouds

Table 4.1: List of sprints and which tasks were performed during them.

4.1 Planetary system simulation

This section covers the development of the gravitational simulation and generation of a stable planetary system consisting of a star and multiple planets.

4.1.1 Why an N-body simulation?

Starting out, we had to consider which approach to choose in order to simulate the movement of celestial bodies in a planetary system. From the pre-study we knew that we could either:

- calculate an exact elliptical orbit for each satellite with Kepler's laws by approximating a 2-body system as covered in subsection 2.1.3 between the satellite and its central body.
- set up a gravitational N-body simulation as covered in section 2.3 where the system evolves based on the gravitational force they exert on each other.

Both approaches have their advantages and disadvantages, which we will go through. Firstly, we had to consider the difficulty of implementation. From what we had gathered in the pre-study, setting up a simple simulation where particles move over time would be of similar difficulty with both approaches. The difference lies in system stability. With Kepler's laws the system will see no change in behavior over time, assuming no floating point errors. That is not the case for a N-body simulation as it is a chaotic system as covered in subsection 2.1.4. It depends heavily on the initial state of the system and configuring it to behave as a stable planetary system means a much higher difficulty of implementation. The stability also depends on the choice of integration method which we discuss further in subsection 4.1.3.

Secondly, there is the issue of how realistic we want the simulation to be. The first approach will ignore all planet to planet and moon to moon interactions which may cause the simulation to lose out on emergent or chaotic behavior that could be simulated with the other more realism-inspired approach.

Finally, we considered the computational complexity. The former approach gives an analytical solution for each planetary orbit and thus it grows linearly with the number of bodies in the simulation. The latter approach gives a numerical solution for each planetary orbit which simulates the interaction between all bodies. Thus it grows quadratically with the number of bodies in the simulation. The latter will also have more calculations per timestep, increasing the performance cost compared to the former approach.

Although the first approach seemed to be simpler to implement and computationally cheaper, we decided for the second approach because it offered a challenge and a more realistic simulation.

4.1.2 Direct or approximation?

The deciding factor for the type of N-body simulation was the number of particles N that should be simulated. To be able to simulate the movement of multiple planets and moons our goal was to be able to simulate around 30 particles. This falls well within the range for a direct method as covered in subsection 2.3.1. A direct method also provides accuracy over an approximation method like the Barnes-Hut algorithm as covered in subsection 2.3.2. In addition, implementation of a direct method is much simpler compared to an approximation method since no approximations has to be made. The core of a direct method is the Equation 2.12, which can then be used to integrate the system over time.

The core structure of the first iteration consisted of two scripts: `DirectNbody` that kept track of all celestial bodies and called the `UpdateState` function for every celestial body in the simulation during each fixed update of the Unity game engine (see section 3.1), and `Celestial`, which contained information about each individual body as well as the `UpdateState` function that calculated the gravitational force exerted by every other celestial body and integrated the resulting position updates.

To simulate axial rotation for a celestial body, an axis was assigned along with a rotational speed, which were used at the end of the `UpdateState` function to rotate the body at each step.

4.1.3 Choice of integration method

But what integration method is appropriate for our project? The simplest method is Euler's method as covered in subsection 2.2.1 and is what was used for the first iteration of the simulation. The first drawback with Euler's method is that it is a first order integrator, which means it does not account for change in the acceleration caused by gravitational force in the simulation. The second drawback with Euler's method is that it is not symplectic, which means the systems total energy will not be preserved over time, the error continues to build up which causes orbits to spiral inward or outward. To counter these drawbacks, we chose to implement Leapfrog integration as covered in subsection 2.2.2. It is a symplectic second order integrator and is commonly used for N-body simulations.

In the second iteration of the N-body simulation we implemented the kick-drift-kick variation of the Leapfrog integrator as covered in Equation 2.9. Another change from the first iteration is the separation of accumulating the total gravitational force from the integration. This change was made because the first iteration incorrectly integrated forces one by one for celestial bodies, affecting later force calculations. Thereby causing celestial bodies to update their state based on a different previous state, as well as increasing the number of calculation steps per simulation step. The solution was to compute the sum of all forces acting on the body and then integrate, which is the correct implementation of Equation 2.12 for N-body simulations. With this improvement we were certain our simulation could run for a long time without any noticeable behavior due to errors.

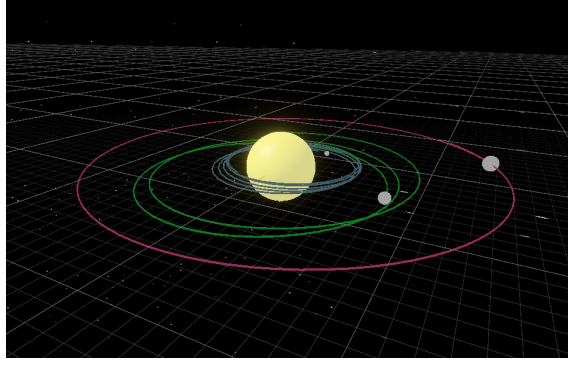


Figure 4.1: Visualized orbits for three planets orbiting a star.

4.1.4 Visualizing orbits

To be able to appreciate and debug the N-body simulation, a visualization tool for the orbits had to be created. From the pre-study we knew it was possible to create a virtual body for each celestial body and simulate its movement at a higher simulation speed up to a limit, save its position at each step and then draw a line between the points, thereby predicting the celestial bodies orbital path. With this approach, Sebastian Lague [12] who implemented it, could determine the initial velocity based on the visualized orbit. This is not the approach we chose, and thus we had to solve the initial velocity problem another way, which we cover in subsection 4.1.5. The reason we did not take a similar approach was the difficulty of implementation combined with concerns of performance when doing the forward simulation at runtime. It was also not necessary to determine the initial velocity as we had a different approach in mind.

The solution we went for was to save the position of the celestial body when it had moved a small fixed distance from the previous position, then drawing a line between it and the previous position. The reason for saving a position only after a fixed distance was to minimize the number of points needed to draw an orbit path. This created a never ending trail behind the celestial body which meant we also had to set a maximum number of points to avoid overflow. When the maximum is reached, the oldest point is removed so that the trail can be drawn continuously. An example of the orbit visualizer can be seen in Figure 4.1.

4.1.5 Avoiding chaotic behavior

N-body simulations are usually used to investigate how a system will behave and how it will look after a certain amount of time. This is different from our goal, which is to generate a system that is stable without having to run the simulation ahead of time and check if it remains stable. Thus, we are either forced to solve the N-body problem (see subsection 2.1.4) or come up with an approximate solution for an initial state that behaves as a stable planetary system. The initial state of the planetary system consists of a position and a velocity for each celestial body. By first deciding an initial position, a fitting velocity can be approximated.

For the initial position of a planet, the first approach was to build outwards from the star and set a planet at a fixed interval. This gave an unnatural look to the planetary system, and when a large planet was added, it exerted enough gravitational force on nearby planets to destabilize their orbits. The second approach was to increase the interval for each planet added by the square of the planets index from the star, this was inspired by our solar system where the distance to each planet from the Sun roughly grows quadratically. We chose to use the second approach. For the initial velocity, assuming a two body system between star and planet we can use the mean orbital velocity formula as covered in Equation 2.4.

4.2 Flying camera

In order to ease testing during development, it was decided to create a free fly camera. We started by determining what we were looking for and what would be a good implementation for it. This included smooth flying with easily changeable speeds and low movement restrictions. The unity inspector itself has a free fly camera, and we decided that something similar would be beneficial during development. By searching online we were able to find a good starting point [37]. Using this code as a skeleton we were able to create a movement sub-system that proved to be very helpful in development. It was missing some quality of life features like not being able to flip around the camera and invert the controls. It would be further developed upon both now and later, and by the end almost nothing of the original code would be left.

4.3 Skybox

Unity's default skybox was not fit for a space simulation, so we decided to create a custom one that looks like space with stars in the distance. This can be done by providing Unity with six images to create a cubemap that wraps around the world. Alternatively, you can create a material shader that Unity will then use as the skybox. We decided early on that we would use the second approach as it allowed us to develop our knowledge of procedural generation.

There are two ways to create shaders in Unity. The first is through *Shader Graph*, which is a node based visual scripting tool that is built into the engine. It works by connecting nodes representing different mathematical operation and functions. Code is then automatically generated from the nodes by Unity. The second method is to write the code by hand. The first version of the shader was created with *Shader Graph* and consisted of a gradient noise node that was connected to a power node that controlled the strength of the noise. By adjusting the power of the noise, a “starry sky” effect could be achieved. When we tried this we saw a stretched seam when panning around the camera. This can be seen in figure Figure 4.2.

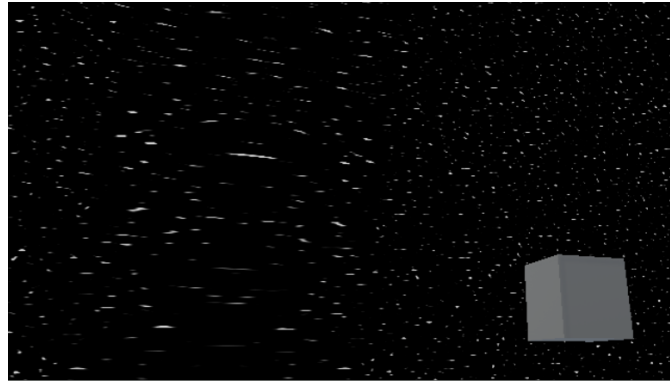


Figure 4.2: Visible stretching on the skybox.

The seam was caused by the built-in noise functions that were only implemented in 2D, resulting in incorrect wrapping. The solution would require a lot of additional nodes in the graph which was less than ideal. This is where we decided to write the code by hand as it allowed for creative freedom without being constrained by *Shader Graph*.

The first version of our own background used Voronoi noise (see subsection 2.7.4). To avoid creating a seam we tried to map the 2D coordinates to a sphere, which instead caused a condensed seam to appear at the two poles of the sphere. As this did not work, we tried cubemapping instead. This was the best version so far, but it was still not satisfactory as if you looked closely the concentration of stars was a bit higher at the edges of the cube, making them visible.

Up to this point we had only considered the stars as a 2D surface that we map to the background of the simulation. In order to solve the concentration of stars at the edges we converted the Voronoi noise to 3D and sampled it using the direction of the vertex in object space. This removed the visible cube edges, but instead we saw large rings on the horizontal axis. This was attributed to our hash function using the sine function. Switching to a different hash function resolved the issue.

To further improve the background, we decided to make the stars vary by multiplying their sizes by a noise value between 0 and 1. This made really small stars flicker, so we changed the minimum size to 0.1. In addition to varying the size we also changed the color of the star depending on its size, where larger stars became red and smaller ones blue.

These fixes made the skybox a lot more appealing but it was still unsatisfactory. To make it more varied and appealing we defined a horizon and made star density greater closer to it and scarcer further away. To not have the skybox look the same in all directions, we added some fbm noise along the horizon to make it look a bit cloudy. The plan was originally to use fbm for clouds in 3D space, but as we did not have a 3D implementation of fbm at this stage in development, we restricted it to only be used along the horizon to prevent stretching.

4.4 Simulating the gigantic scale of space

From prior experience we knew that working with large distances and sizes would be troublesome. At large distances artifacts occur such as vertex jittering, where the individual vertices of a model jump due to lack of precision. Unity's transform component which serves as the basis for any position or scale, only supports the float format. To illustrate this, the distance to the Sun is about 150 million kilometers, or $1.5 \cdot 10^{11}$ meters [38]. Representing this to the closest meter requires 38 bits of precision ($1 + \lfloor \log_2(1.5 \cdot 10^{11}) \rfloor$). The more common float format only has 24 bits of precision, and would thus not be able to accurately represent this distance at this scale.

The initial implementation of the N-body simulation used 32-bit floats to store planet data for calculations, but after discussing what scales we wanted to support, we changed it to 64-bit doubles instead. This made the N-body simulation able to handle large numbers but it was not enough to solve all problems. As a user moves through the universe their position changes, potentially reaching very large values. To combat this we introduced something known as floating origin. Normally in games or simulations, the origin of the universe is centered at position (0,0,0) and the user moves away from this point as they move. But we decided to make the player the origin of the universe. This means that everything shifts relative to the user's position and therefore we keep all values small. The origin shift occurs every time the user has moved a distance of 25000 units. Then the user is shifted to the origin and every object in the planetary system shifts by the same amount to maintain identical relative positions.

4.5 Noise

Before we looked into generating terrestrial planets, we built a small library of composable noise functions(see section 2.7). We did this preemptively so that we could use them during generation without having to implement them then. The reason we had to implement them ourselves rather than using Unity's built-in noise functions is that the ones Unity provides only take in 2D coordinates. We wanted 3D noise as that would allow us to create more interesting terrain, and so we had to build our own.

We chose to implement three types of noise: Perlin noise with Perlins suggested improvements, fractional noise (also called fractional brownian motion or fbm), and warped noise. These all have distinct looks, fbm with Perlin noise as a base being a common choice for procedural generation as it exhibits self recurring patterns that are common in nature.

To allow greater flexibility in the types of noise that can be generated, we made all noise functions follow a common interface, `Noise3`, which can be seen in Listing 4.1. Having this common interface allows us to construct noise functions that take other noise functions as a base. For example, fbm can take a `Noise3` parameter that specifies which type of noise to layer.

```
1 public abstract class Noise3 {  
2     public abstract float Sample(ulong seed, float x, float y, float z);  
3     public abstract float Sample(ulong seed, Vector3 position);  
4 }
```

Listing 4.1: Interface to the noise generators.

4.6 Terrestrial generation

With a set of noise functions at our disposal we moved on to generating the terrestrials. This work was split into two parts: generating the data that describes the terrain, and turning that data into a mesh that can be rendered on screen.

4.6.1 Marching cubes

We started the development of the marching cubes algorithm by researching how the algorithm works, what is possible with it, and how it can be incorporated into Unity. We read through the original paper about marching cubes to learn how it was made and its intended usage. This paper taught us how the data is structured and used to generate the 14 unique cases of triangulated cubes needed to create the surface of a 3D object (see section 2.4). We also watched a video from Sebastian Lague about the implementation of the marching cubes algorithm. Lague showed how the algorithm could be implemented in Unity and some complications that comes with it [39]. We did not want to copy his work, even though he gave everyone the right to use it for anything, because we wanted to develop it ourselves. Not only would we learn exactly how the algorithm is to be implemented but we also get full rights to our code.

We decided to implement the marching cubes algorithm on the CPU first. The problem with implementing on the GPU is that it is very difficult to debug the code because the algorithm is not type-checked and instead validated in runtime. This means that bugs can only be detected visually, making it harder to find the culprit and fix it. By developing the algorithm first on CPU, we will be able to implement marching cubes correctly.

For the implementation, we started with learning how Unity's meshes works. We began by making different 3D shapes using hard-coded vertices. We learned that Unity uses two arrays for the mesh: one that contains the vertices and another that contains indices to the vertices. This discovery was crucial for the development of the marching cubes algorithm in Unity. From this, we created sample data that mimics the noise density data that would make our planets. With this data, we knew if the algorithm worked if the correct shape was created.

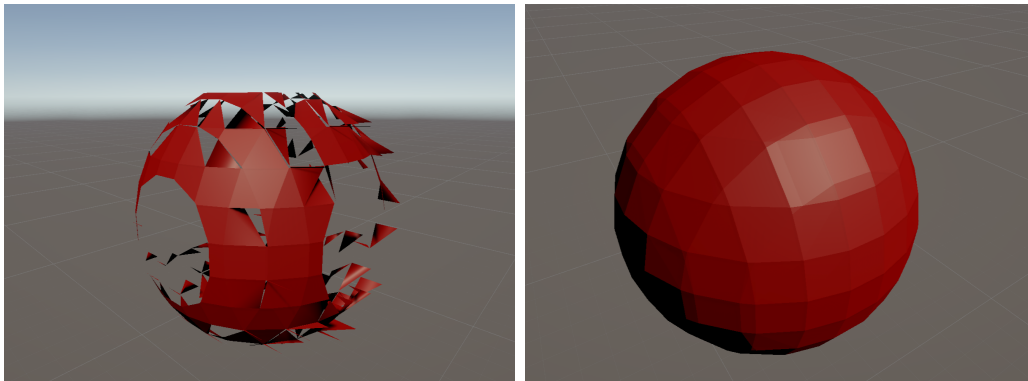
After learning how Unity's mesh works, we started implementing the algorithm. In the beginning, we started with generating only one of the triangulation cases to see if we could get something drawn on the screen. After getting it to work, we added the second case to see if we implemented the cube numbering correctly according to the original paper. We realized that writing all 256 triangulation cases by hand

```

1 Vector3 InterpolateVerts(Vector4 v1, Vector4 v2) {
2     float t = (0.0f - v1.w) / (v2.w - v1.w);
3     return Vector3.Lerp(
4         new Vector3(v1.x, v1.y, v1.z),
5         new Vector3(v2.x, v2.y, v2.z),
6         t
7     );
8 }

```

Listing 4.2: Interpolation method for the marching cubes algorithm.



(a) First iteration. The mesh is broken and incomplete. (b) Second iteration. The mesh is smooth and captures the shape.

Figure 4.3: Screenshots from different stages of implementing marching cubes.

would take a long time. We looked at Sebastian Lague’s code to see where he got the triangulation data, found his source, and used the same source for the hard-coded data [40]. This saved us hours of implementation, letting us focus on the actual algorithm rather than getting the correct vertices for each case.

The first iteration of marching cubes did not work and resulted in a broken mesh (see Figure 4.3a). The reason for the mesh being broken was that the edge index was used as an interpolation weight, which led to incorrect vertex data. This was solved by storing the cube vertices in two arrays, which allows us to interpolate the edge vertex and correctly create the mesh. We also changed the interpolation method to use fewer arguments using the `Vector4` type. This made the method more versatile if it were to be used for something else (see Listing 4.2).

With this change, the mesh got correct vertex data for its triangles, fixing the broken mesh, which concluded the second iteration of the marching cube algorithm (see Figure 4.3b).

4.6.2 Generating the terrain data

Marching cubes needs floating point data in a regular grid, where some set value represents the surface. We chose to use zero as our value for the surface. Calculating the terrain data now requires us to allocate a 3D grid of floats, and then for each

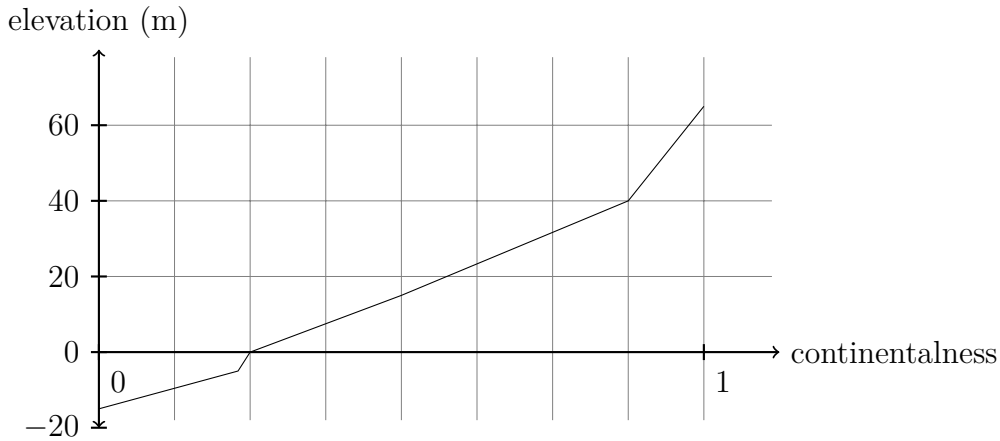


Figure 4.4: Mapping from continentalness to elevation.

float we compute a value that represents if we are inside or outside of the terrain. An easy shape to generate here is a sphere. We can measure the distance from the grid cell we are computing to the center of the grid and then subtract the radius of the sphere we want, as in Equation 4.1.

$$d(\vec{P}) = |\vec{P}| - r \quad (4.1)$$

Once we had that working, we looked into different ways of generating more interesting terrain. We found a presentation from the team behind Minecrafts terrain generation [41] and decided to adapt it to work on spherical planets. The idea we borrowed was what they call a “continentalness” value. This is a floating point number that is sampled from noise, in our case 3D fbm with a Perlin noise base. It is then used as a lookup into a table that maps it to elevation. The default mapping can be seen in Figure 4.4.

This allows us to create a heightmap for the terrain. We then add the base radius of the planet to the height. This technique does however require that along any ray from a grid cell to the center of the grid, we need to get the same continentalness. Otherwise, we will get artifacts in the terrain. For Minecraft, this is not a problem as they can just sample in the XZ-plane as they are not generating their terrain in 3D. In order to get consistent values, we normalize the grid coordinate and then scale it by a constant. The scaling enables us to adjust the size of features such as mountains.

4.7 Combining the sub-systems

As most sub-systems so far have been developed in parallel some time had to be spent making them work together to have a finished MVP. The first step was to insert the terrestrial planets as discussed in section 4.6 in to the planetary system simulation (see section 4.1) and replace the checkered spheres which had been used for the development. The first problem we encountered was that the simulation

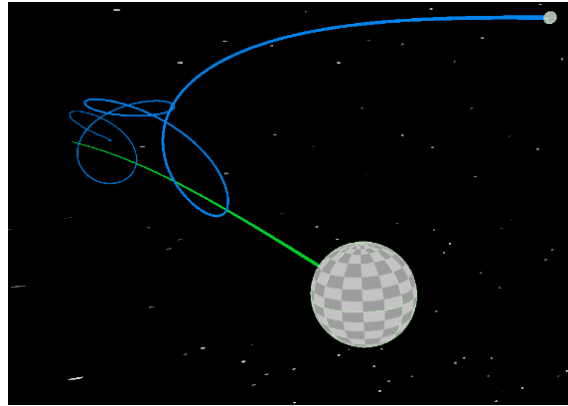


Figure 4.5: A moon orbiting a planet incorrectly due to inherited transform.

required the local origin of a celestial body to be at its center, but the terrestrial planets had their origin at a corner of the 3D grid of the terrain data as discussed in subsection 4.6.2. This was fixed by offsetting the vertices of the generated mesh by the center coordinate of the terrestrial.

The second problem we encountered when combining the planetary simulation with the procedurally generated planets was how the size and scale of objects was handled. In the simulation we scaled planets by their radius with the Unity transform to make them appear to be their size, which worked as primitive spheres were used as placeholders. When doing the same for our generated planets they broke and were not visible anymore. The problem was that the generated planets assumed that the size was only determined by the radius set at initialization. If scaling should be applied, it should be applied to the radius directly before starting the generation. Thus, the solution was to modify the radius directly, not the scale of the object.

4.8 Adding moons to the simulation

After the MVP was completed, the next step in implementing the simulation of a planetary system was to add moons. We knew that the hard part was not to add more bodies to the N-body simulation, it was implementing a system that guarantees stable orbits around planets that was going to take time.

We started with adding a single moon to each planet, at a fixed distance, with an initial velocity calculated with the same equation used for planets (see Equation 2.4). To get the moon to follow its planet around the star, we set its Unity transform to inherit from the planet. This was quickly found to be a mistake as it would update the position of the moon relative to the planet. This happened outside of the N-body simulation which is supposed to handle all movement of celestial bodies. It also messed with the axial rotation as it was also updated relative to the planet. The incorrect orbit can be seen in Figure 4.5.

The solution was to add the planets initial velocity around the star to the initial velocity of the moon around the planet instead of using Unity's transform inheritance.

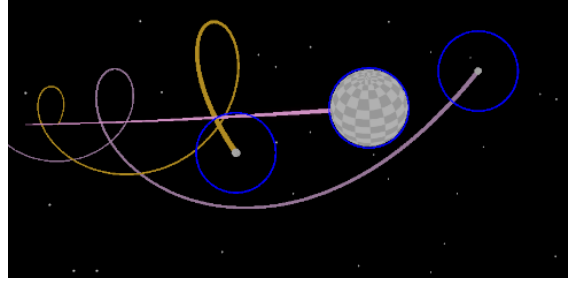


Figure 4.6: Two moons orbiting a planet in the outer system at the start of the simulation.

The next step was to have multiple moons orbiting a planet. We decided to take the same approach that was used for adding multiple planets around a star (see subsection 4.1.5). The result can be seen in Figure 4.6.

This worked fine for planets far out in the planetary system, for a short period of time in the simulation. For moons near the center of the system, the dominant gravitational force was not from the planet but from the star. This caused them to orbit in parallel with the planet around the star, not around the planet. Moons further out in the system instead got too close to the planet after a short time in the simulation. This caused the gravitational force between the moon and planet to spike, leading to the moon being shot out of orbit. Because of these two problems, we needed a precise way of deciding if a planet can have a moon and if so, how far away from the planet it should orbit.

4.8.1 Are you worthy of a moon?

It is not enough to arbitrarily decide if a planet can have a moon or not. Thus, a system that determines if a planet is eligible to house a moon is needed for accurate behavior of planetary orbits. The Hill sphere (see subsection 2.1.1) can be used to determine how far away from the planet a moon could potentially orbit. To determine how close a moon could orbit we can use the Roche limit (see subsection 2.1.2). By comparing the two we can determine whether a planet even can have a moon at all. The blue area in Figure 4.7 shows where a moon can have a stable orbit around Earth.

Since the Hill sphere model assumes a three-body system it will only be an approximation in our planetary system, especially when a planet has multiple moons whose interactions are not accounted for. The Roche limit will also be an approximation because the tensile strength of the satellite is not taken into account. For example, Jupiter's moon Metis is an exception to the Roche limit model as it orbits within the limit [14]. To account for approximation errors we only allow moons to spawn in the range $[3r_{roche}, \frac{r_{hill}}{2}]$. The margin factors were found by trial and error when running the simulation. If the range is too small, or if $3r_{roche} > \frac{r_{hill}}{2}$ then the planet is not worthy of a moon.

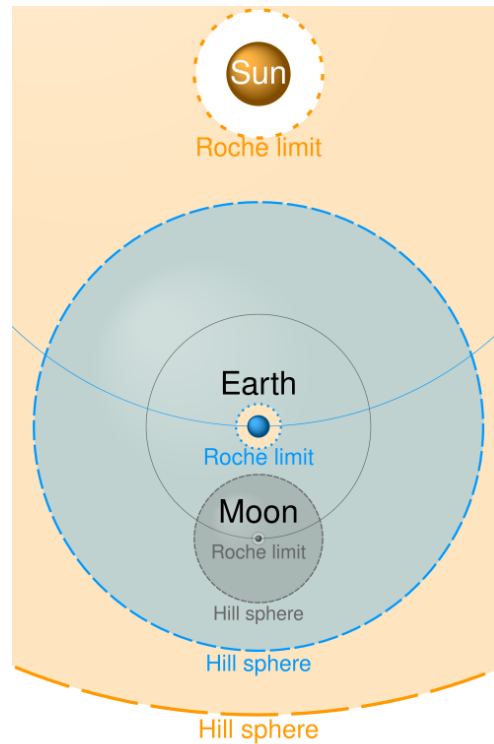


Figure 4.7: Comparison of Hill sphere and Roche limit for the Sun-Earth-Moon system by Cmglee, CC BY-SA 4.0, via Wikimedia Commons.

4.9 Use of astronomical data

In the start we set arbitrary values to the attributes of celestial bodies. For example, terrestrial planets had a radius of four times the its index counted from the star. Position and density were set similarly. This made the simulation feel abstract and hard to design for realistic behavior. To remedy this, we decided to base all attributes on observations from reality. In astronomy, planet radii are given in units of Earth radii ($R_{\oplus}$), star radii in units of Sun radii ($R_{\odot}$), and distances at the planetary system scale are given in astronomical units(AU) which is the distance between the Sun and Earth.

Terrestrial planets generally fall within $0.8\text{--}2 R_{\oplus}$ and gas planets within $6\text{--}22 R_{\oplus}$ [42]. Planets with radii Between $2\text{--}6 R_{\oplus}$ are considered Neptune-like, which we have not implemented. As for stars, we decided to limit us to main sequence stars, specifically A-K class stars which are the most similar to the Sun. These fall within $0.45\text{--}2.1 R_{\odot}$. By randomizing values within each range we get attributes of planets that imitate reality, therefore as the N-body problem is an IVP, the behavior of the simulation will likely also behave realistically.

As covered in section 4.4 the gigantic scale of space is hard to represent in a $1 : 1$ relation in a computer simulation. With the use of doubles and a floating origin as discussed in section 4.4 we can run the simulation at a $1 : 1$ scale. This creates bottlenecks elsewhere, as generating terrestrial planets requires a much more

sophisticated system than what we have been able to create. To solve this we apply a scaling factor to all radii which lets us easily control at what scale we want to generate the planetary system. Terrain generated on a scale of $1 : 10^4$ has good detail at the surface, which means an average terrestrial planet at this scale will have a radius of 640 meters.

Another problem with using values at a $1 : 1$ scale is the distances between celestial bodies, objects are so far away that they are not visible from each other. This defeats the purpose of creating a planetary system that we want to be able to visualize completely for the user. Thus we decided to scale down distances as well. Simply using the same scaling factor of $1 : 10^4$ for radii would not bring the celestial bodies close enough to be visible at the same time as distances are much larger than sizes at an astronomical scale. This is why we decided for a distance scale of $1 : 3 \cdot 10^6$.

By scaling with a separate factor, we allow for configuration of the scale of the whole simulation as well as making it easy to expand if terrain generation is improved and can handle larger planets.

4.10 Gas planets

One of the “should haves” for the project (see Table 3.1) was to add planet types beyond terrestrial planets, so we decided to add gas giants to the simulation. Since the terrain of gas planets is not generated, all of the visual representation of the planet will be done through a shader. After researching similar shaders [43], We concluded that the main idea of the shader was to wrap fractal noise, also known as fbm (see subsection 2.7.2), around a sphere and stretch it in the x-axis. This causes a distinct line pattern that gives the impression of gas flowing in a sphere since fractal noise already bears a resemblance to gas. We did consider taking a more complex approach with fluid simulation but it was ultimately rejected due to performance concerns and implementation complexity.

The first iteration of the shader was based on a 2D version of fbm noise that blends two colors, a warm color and a cool color instead of a gradient between black and white typically seen in noise. The noise is then stretched and animated in the x-axis to achieve a similar effect of gas flowing around a sphere. To increase further increase visual fidelity a warping effect was applied on the noise using the existing implementation of the fbm (see subsection 2.7.3). However, similarly to the skybox issue (see section 4.3) it became apparent that 2D implementations of noise will result in seams when used as a material in any kind of 3D context.

To remove the seam a 3D version of noise was adopted for the shader. Using 3D noise also allowed us to use the z-axis of the noise to animate it, which simplified the implementation and increased the visual fidelity of the animation (see Figure 4.8).

Most of the shader implementation was done using world space positions but this made the visuals inconsistent with respect to the scale of the gas planet. When using world space positions, scaling or rotating the planet would have no effect on the material. Therefore the switch to object space was done to make the noise scale

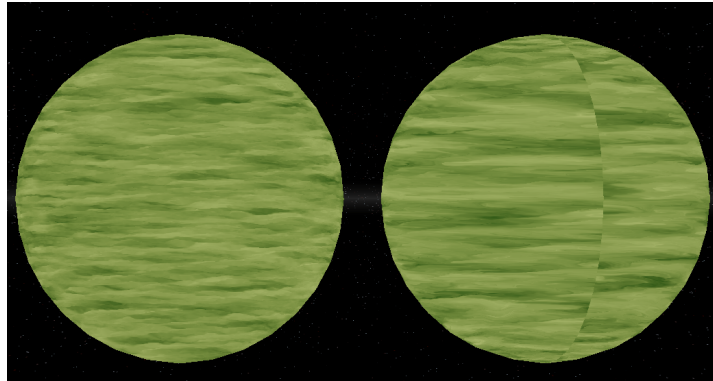


Figure 4.8: Side by side comparison of 3D noise (left) and 2D noise with visible seam (right).

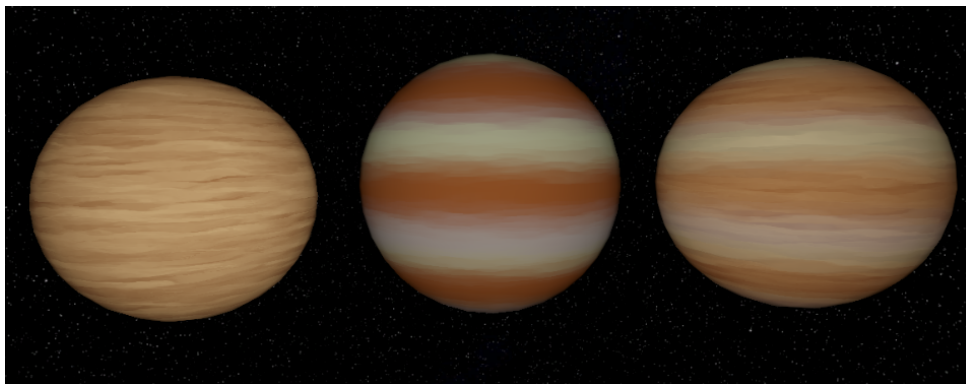


Figure 4.9: Gas planets with different parameters using colors similar to Jupiter's.

with the planet and allow rotations to affect the planet. Additionally, colored bands were added to the planets by linearly blending two band colors onto stripes created using a sine function. The stripes are then linearly blended with the noise texture to create the result seen in Figure 4.9.

The last visual improvement of the gas planets was improving their shape. The shader is used as a material on a default 3D sphere, which looks jagged as large sizes due to the low resolution of the mesh. To solve this, we raymarch against a signed distance field (see section 2.6) of a slightly smaller sphere. The result can be seen in Figure 4.10, which looks perfectly smooth.

4.11 Splitting planets into chunks

Generating whole planets in one go took several seconds at this point. This made it annoying to adjust terrain parameters, and waiting for an entire planetary system to generate took even longer. Here we decided to split planet generation into smaller parts that can be generated independently of each other, commonly referred to as chunks. As they are independent of each other, we can generate multiple of them simultaneously on separate threads. This speeds up load times and the saved time allows for more complex terrain generation down the line.

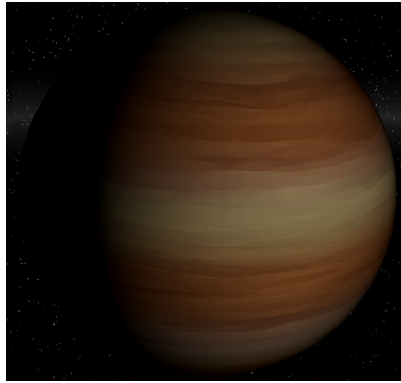


Figure 4.10: Final version of the gas planet shader featuring lighting and a higher mesh resolution.

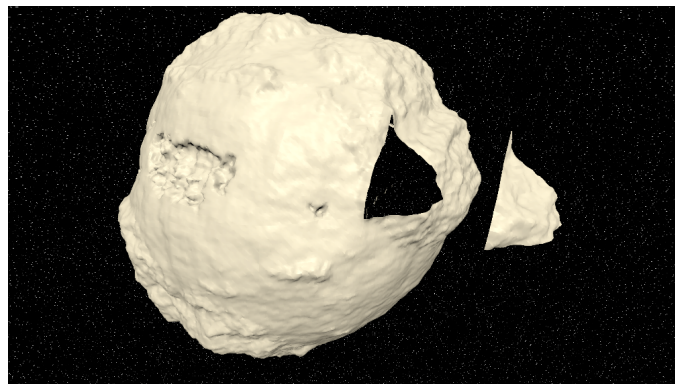


Figure 4.11: Planet with one chunk pulled out.

The first implementation split the planet into chunks on a uniform grid. This can be seen in Figure 4.11. When chunks need to be generated they submit a job to a thread pool that generates the terrain data and runs marching cubes on it to produce a mesh. Due to Unity limiting their API:s to only being run on the main thread, we cannot complete the mesh creation in the thread pool. Instead we create a list of vertices and a list of indices, transfer them to the main thread through a concurrent queue, then give them directly into Unity.

4.12 Deforming the terrain

At this point we realized that implementing terrain deformation would not take that much time. With planets split into chunks, we do not have to regenerate the mesh for the entire planet, but rather only for the chunks that are effected by the edit. There was still the question of how we store the edits. Initially, we wanted to directly modify the terrain data and store that. With our chunk size of 16^3 and each grid cell containing a float, we would require 16 KB per modified chunk.

At this point, we found Mike Turitzin's project that takes a different approach [44]. Instead of storing the modified geometry, he stores the edits as SDF operations. Each time a new edit is made, the affected chunks of the terrain are regenerated.

We then realized that our terrain data approximates a SDF, which allows us to use operations already defined for combining SDFs to apply edits.

Edits are stored as a list of planet local positions, sizes, operations (addition or subtraction), and types of SDF. These are applied with a smoothing function after the terrain data has been generated. When an edit is added, we mark all chunks that overlap with the new edit as invalid, and then let the terrain generator regenerate them. To speed up chunk generation, we filter the edits to only include those that intersect the chunk and thus remove edits that cannot possibly affect the final result. This way of handling edits allows us to use and shape that can be represented as an SDF as an edit. Our implementation supports spheres, cubes, and octahedrons.

In order to place edits we use Unity's built-in raycasting. This required us to add `MeshColliders` to all chunks so that Unity's physics sub-system knows about their geometry.

4.13 Walkable planets

After the implementation of collision meshes on planets we could add means for exploring them. For this purpose we had to create a new type of ground movement, a way to swap between the different movement types, a way to determine a planets gravity, and a way to keep the camera's relative axis the same no matter where on the planet you are walking. To do this we had to create a multitude of different scripts and use trial and error in order to figure out how to best implement it. We mapped the change of movement controls to the button `T` and made it only work if you are looking at a planet by using raycasting. When the button is pressed it will switch from the fly camera to the ground controls, fix your rotation so the relative up uses the planet as a reference, and apply the planets gravity. However we still face one problem. When planets rotated and moved around the star the user would not move with the planet, ultimately leading to infinite falling as the gravity pulled the user towards the center of the planet. However, it was easy to fix by making the user a child of the planet when ground controls are turned on.

4.14 Surface colors

To make the terrestrial planets a bit more visually interesting, we introduced two random RGB colors based on steepness. This produced unnatural looking planets. As we aimed to make a realism inspired simulation we decided to implement four different color palettes that terrestrial planets and moons would get their colors from. We called the first type *Earth-like*, here we took inspiration from Earth. It had green for the flat color and brown for the steep color. We called the second one *Barren*, with inspiration from Earth's moon. It had a lighter gray for flats and a darker gray for steep areas. The third palette, *Rust* was inspired by Mars and had two tones of rusty colors. The final one, *Volcanic* took inspiration from basalt and volcanic areas on Earth and had two tones of a dark eggplant color.

We also introduced a color based elevation and a color based on latitude. The elevation color which dominated at high altitudes was set to a light gray tone. For the latitude, a white color was applied to emulate polar caps as seen on both Earth and Mars. As we aimed for deterministic procedural generation, we introduced variance to each of the colors based on the seed. Doing this to RGB values often caused unproportionate changes to the colors which is why we used the HSV colors instead. With it we got more natural variance in the colors by changing the hue, saturation, and brightness slightly.

We also attempted to implement detailed procedural texturing to achieve a rocky look when walking on the surface. This turned out to be harder than expected as it required careful configuration and tuning of factors affecting noise functions (see section 4.5), which we deemed would take too much time for the potential result. As the planets still seemed uniform in color on a planet wide scale, we decided to saturate and darken the planet color based on down-scaled fbm noise (see subsection 2.7.2), this was inspired by the darker spots on Earth's moon.

4.15 Lighting

Initially, we thought that Unity's built-in lighting sub-system (see subsection 3.1.1) would be sufficient for simulating starlight in our project. This assumption was proven incorrect. It works well when the light source affects all objects from the same direction, but in our case the planets orbit around the star, which means that the lights direction changes depending on each planet's position relative to the star.

Our first attempt to solve this was to place a large spotlight at the center of the star to mimic starlight. However, Unity does not handle spotlights as efficiently as directional lights. As a result, Unity frequently culled the starlight whenever too many lighting calculations occurred simultaneously. This made the lighting implementation significantly more challenging than anticipated, as we now had to implement a custom lighting shader.

We started with a lighting model that illuminates surfaces facing the star. The lighting direction was calculated using the position of the star and the position of the pixel currently being shaded. After determining the light direction, we computed the dot product between the normalized light direction and the surface normal. It is important that this is done in world space to get a correct result. The result of the dot product produced a value between negative one and one, which was used as the lighting intensity for the surface. This intensity was combined with the fragment's base color to compute the final fragment color. Consequently, surfaces facing the star appeared brighter, while surfaces facing away from the star appeared darker.

We also wanted our custom lighting shader to support additional light sources provided by Unity. To achieve this, we followed Unity's official documentation on rendering additional lights in a shader using the Universal Render Pipeline (URP) [45]. One aspect we needed to define ourselves was the shadow mask used when rendering shadows from additional light sources. According to Unity, the shadow mask is

important when combining baked lighting with real-time lighting. Since Regolith exclusively uses real-time lighting, we could safely set the shadow mask value to one.

4.15.1 Shadows

The initial lighting implementation did not account for occlusion. As a result, surfaces located behind mountains or inside of caves were still illuminated even though they should have been in shadow. To address this issue, we needed to implement shadows. Implementing shadows proved to be considerably more difficult than lighting. Several issues arose during shadow rendering including shadow acne, flickering, and poor performance. Since our planets were very far from the light source, these problems became especially noticeable.

To generate shadows for a planet, we first needed to create a shadow map containing depth information. Our initial approach involved rendering a custom shadow map using an orthographic camera that drew scene geometry onto a depth texture. The shader could then compare fragment positions against the stored depth values to determine whether a surface was occluded from the light source. However, when attempting to sample the depth data inside our lighting shader, we discovered that the coordinates in the shader did not correspond to the coordinates in the shadow map. This resulted in incorrect shadows and in some cases none at all. We experimented with different rendering methods and rewrote parts of the shadow calculation process, but none of the solutions worked reliably. At the same time, performance suffered significantly due to the custom shadow map implementation, reducing the frame rate to approximately 15 frames per second. This was unacceptable for the quality standard we aimed to achieve. To solve both the coordinate mapping issues and the performance problems, we switched to Unity's built-in lighting sub-system for generating the shadow map while still using our own lighting calculations.

To obtain a functional shadow map from Unity, we added a directional light aimed toward the planet closest to the user. Functionally, this achieved the same result as our custom implementation, but Unity handled the coordinate mapping correctly. To access the shadow information, we used Unity's lighting packages and retrieved the main directional light through the `GetMainLight(float4 shadowCoord)` function. This function returns the scene's main light together with several lighting properties. We extracted the shadow information using the light's `shadowAttenuation` property and multiplied it with the final fragment color. Additionally, we adjusted the directional light's shadow bias parameters to reduce shadow acne.

At this stage our shader could correctly receive shadows on planet surfaces, but could not cast shadows onto other objects. The issue was caused by the absence of the shader pass required by Unity's rendering pipeline. This problem was resolved by extracting the `ShadowCaster` pass from Unity's built-in lighting shader and adding it to our custom one. After including this pass, our planets could cast shadows onto other planets, enabling effects such as eclipses. This also solved the issue where caves remained illuminated even though starlight could not reach their surfaces.

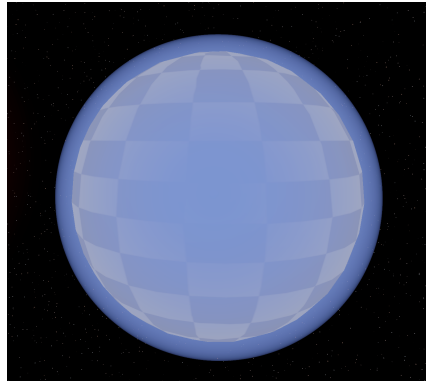


Figure 4.12: Atmosphere based on distance traveled inside the physical bounding sphere.

4.16 Atmospheres

We began by researching how atmospheres in the real world function. This helped us get an idea of how to potentially model it in Unity. The basic principle as mentioned more thoroughly in the theory section 2.5, is to simulate light's interaction with particles in the atmosphere, how they scatter and how much light eventually reach the observer. The main reference and inspiration for our implementation on atmospheres is “Accurate Atmospheric Scattering” by Sean O’Neil [46].

We began by utilizing a ray-sphere intersection test of a sphere to mathematically figure out whether a given screen pixel, is looking at the atmosphere. Using the signed distance function we could now cast a ray from a pixel on the screen in the direction of where the user looked and determine if it hit the atmosphere. If it did, we colored the pixel blue. This created a solid blue atmosphere, which was unrealistic. From research we understood that the further light travels through a medium, the more it scatters from interactions with molecules. Therefore we calculate the distance traveled from the entry into atmosphere to the exit. Using this distance we made the atmosphere more transparent when a shorter distance was covered.

This approach had two problems: light traveled through the planet's surface, and our atmosphere was thicker in the center because our shader did not account for that. We experimented with two methods to solve this. The first one we considered was doing a second ray-sphere intersection to define the ground. Since it used a mathematical function of a perfect sphere it failed to capture different surface heights like mountains or deep valleys. Because of this, we went along with the second solution, the depth buffer. It is possible to sample the depth buffer and compute positions in 3D space from the value. This did not work on the first try. After a lot of debugging it turned out that our custom mesh for the planetary surface did not draw to a depth texture and therefore the atmosphere shader had no data of it. To fix this we modified the shader for the ground material to also draw to the depth texture. After this was implemented the atmosphere looked more realistic but we were not satisfied with it.

We had read about a mathematical model called Henyey-Greenstein's phase function [47] which models the probability that light scatters in a direction. We ended

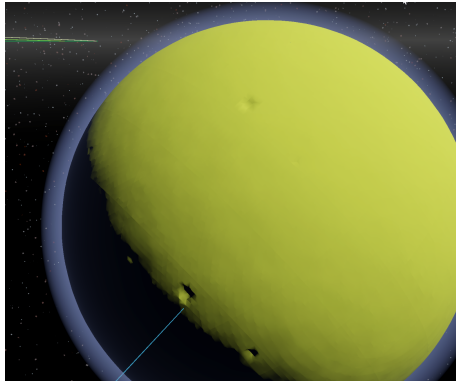


Figure 4.13: Atmosphere using Cornette-Shanks phase function.

up using the Cornette-Shanks phase function instead due to its use in modern implementations of atmospheres such as “A Scalable and Production Ready Sky and Atmosphere Rendering Technique” by Hillaire [48]. The result can be seen in Figure 4.13.

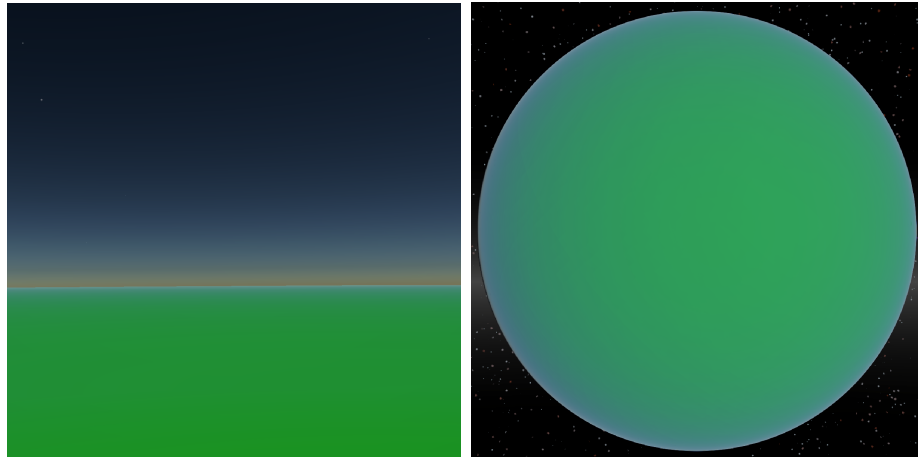
At this stage our shader assumed that the atmosphere had uniform density, which is not physically accurate, Earths atmosphere gets thinner at higher altitudes. In order to model this we calculated a value known as the scale height section 2.5. Instead of basing the atmosphere’s opacity on the distance traveled through it, we could now get the atmospheric density of our planet at a given altitude using the scale height.

4.16.1 Modelling atmospheric scattering

Now that we had a function to return the atmospheric density, we wanted to integrate Sean O’Neil’s solution for modeling atmospheres. They calculate the amount of light that reaches a given point in the atmosphere and how much of that light reaches the observer by using mathematical approximations to increase performance [46]. Since it was published in 2005, we decided that we could afford to skip the approximation and instead perform the calculations directly to save development time. This came at a performance cost that we deemed negligible for modern hardware.

Our implementation raymarches from the entry point to the exit, this being either the atmosphere’s edge or the ground. The number of steps and their size depend on the desired number of samples. At each step we compute the density at the current position based on the altitude above ground. We accumulate the density multiplied by the step size to track the total density traveled, called the optical depth. We also calculate the optical depth from the current sample point to the star in a similar fashion, stopping at the edge of the atmosphere. These values are then used to calculate the total light reaching the observer taking into account Mie and Rayleigh scattering which is covered in section 2.5. The accumulated light is then used to calculate the final output color, which takes into account both Rayleigh and Mie scattering. The result can be seen in Figure 4.14 and the code in Listing 4.3.

Using the scale height to calculate the atmospheric density by altitude was physically correct, however the atmosphere was small due to our planet being scaled down. We



(a) Atmosphere at ground level. (b) Atmosphere from space.

Figure 4.14: Atmosphere with Rayleigh and Mie scattering.

```
1 float sunAngle = dot(sunDirection, viewDirection);
2 float3 rayleighTerm = rayleighScattering * rayleighPhaseFunction(sunAngle);
3 float mieTerm = mieScattering * miePhaseFunction(sunAngle, 0.8);
4 half4 finalColor = half4(
5     light * SunBrightness * (rayleighTerm + mieTerm), 1.0
6 );
```

Listing 4.3: Final output color based on the accumulated light.

found this to be visually underwhelming. Furthermore since the the mesh generation for the planetary surface was done after creating the atmosphere, we had no way of accounting for terrain elevation. This caused issues which can be seen in Figure 4.15 where the ground bleeds through the atmosphere.

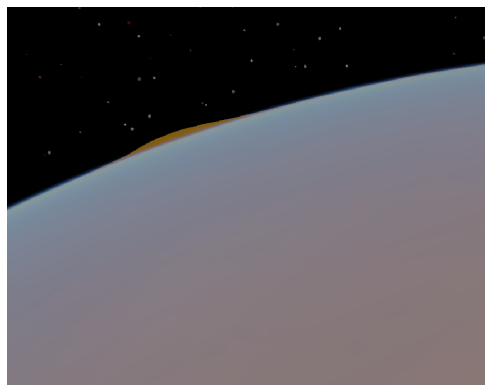
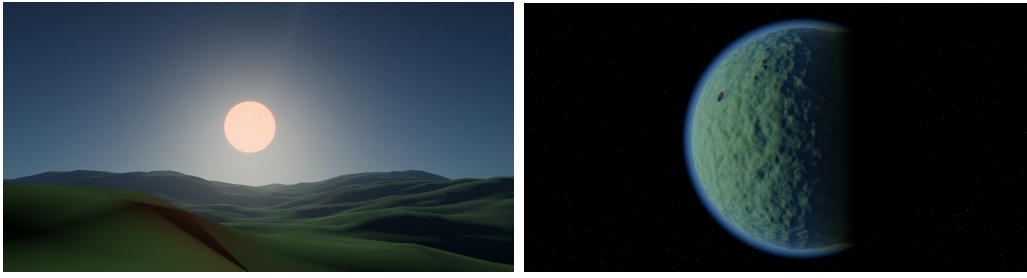


Figure 4.15: The max height of the atmosphere showing beneath mountains.

Therefore in the end we decided to go for a non realistic approach where the atmosphere density decreased semi exponentially by altitude. This decoupled atmospheric density from physics, and consequently we could tune the atmospheric density to look like we wanted. We proceeded to integrate the atmospheres into the simulation with consideration to the procedural aspect. We made the atmospheric properties



(a) Atmosphere at ground level. (b) Atmosphere from space.

Figure 4.16: Screenshots of the final atmosphere.

vary between planets by randomly selecting from a set of presets during creation. Figure 4.16 shows the final artistically tuned result.

4.17 Clouds

Continuing the work from atmospheres we experimented with the idea of clouds. From previous knowledge we understood that the basic idea of clouds were similar to atmospheres and that clouds could be created by sampling noise data. Using the existing loop to calculate outgoing atmospheric light, we sampled an fbm function on the GPU to get density at given points in the atmosphere and accumulated it. We then calculated how much light gets absorbed by the density. Denser cloud means more light get absorbed making the cloud thicker and darker. The first version can be seen in Figure 4.17.

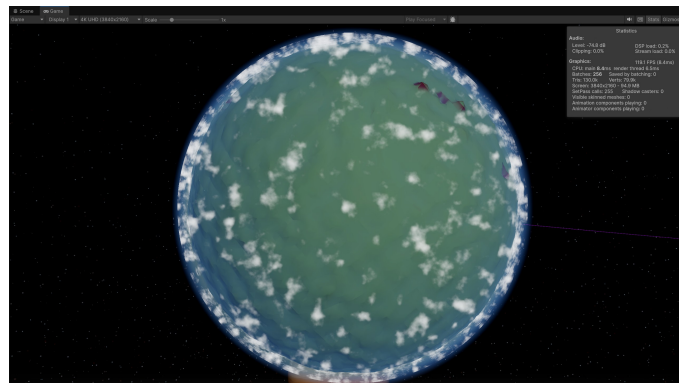


Figure 4.17: First look at clouds.

To increase performance and quality, we decided to separate the atmosphere and clouds into separate passes. This made it both easier to code and allowed us to use different number of samples for the clouds and atmosphere. Additionally, sampling from the outer edge of the atmosphere to the ground wasted samples as we did not want clouds at ground level or at the edge of space. Having clouds in a separate pass allowed us to define an inner and outer sphere acting as a shell to raymarch through and take samples along.

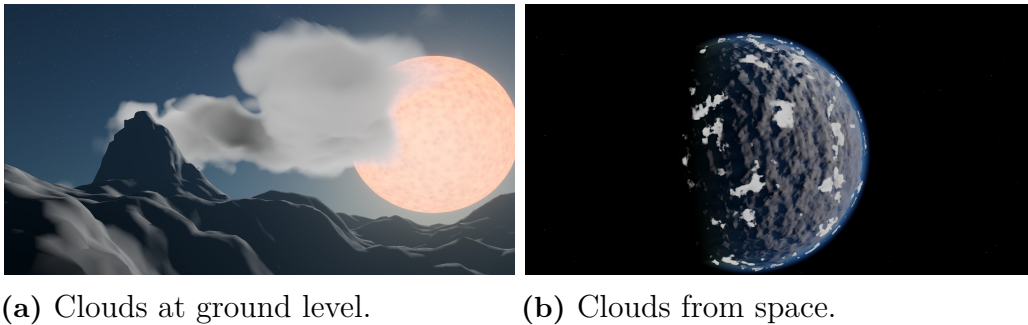


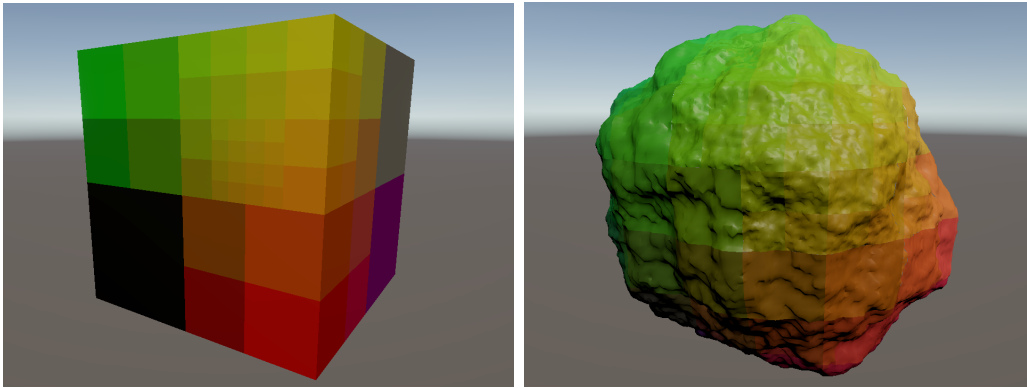
Figure 4.18: Screenshots of clouds at current implementation stage.

At this stage, the clouds only calculated direct lighting from the star, which made them very dark at the bottom as the upper part of the cloud absorbed the starlight and we had no ambient lighting. We also noticed banding in the sky due to low sample counts and uniform step size during raymarching. Increasing the sample count would help mitigate this, but it is an expensive way to solve it. Instead we randomized the step size for each ray, which traded the banding for noise that can be distracting, especially at low frame rates.

Before proceeding further we decided to tackle the performance issue. Up until now the clouds sampled 3D fbm two times per sample which is very inefficient as calculating 3D noise requires many math operations. To avoid this, we baked the noise data into 3D textures, that we then be sampled inside the shader. Sampling a texture instead of calculating noise at runtime improved the performance at the cost of increased memory usage. This did however make development slower as each time you wanted to change the noise data to change the look of the clouds, you had to exit, re-bake them, and restart the simulation.

In desire of trying to improve the clouds we researched existing cloud implementations to speed up development. Ultimately we decided to base our implementation on Guerrilla games Decima engine’s solution to clouds ‘Nubis’ [49], [50], [51]. Just like the atmosphere, creating clouds can be divided into two parts: sampling density and lighting calculations. Since we already had a way to get density, we focused on implementing better light calculations. We implemented the general outline in the talk ‘The Real-Time Volumetric Superstorms of ‘Horizon Forbidden West’ [50], where they describe how they perform their light calculations. This worked quite well and we added adjustable parameters for fine tuning things like the light extinction.

Since we had more time we also implemented similar noise sampling to theirs. This was however quite a difficult process and took the majority of the development time. With their clouds as a reference we implemented what they call a “dimensional profile”. In short, its a 2D texture that contains information about where clouds should exist and what type they are. Due to our planets being spherical we could not keep this as a 2D texture and instead made it into a cubemap texture sampled with the sample direction from the planet center. We found that creating good noise textures were very difficult, and since we could not test the noise on the fly due to baking, it took a long time to iterate on.



(a) Visualization of the different detail levels as cubes. (b) A planet with chunks at different detail levels.

Figure 4.19: Screenshots from different stages of implementing level of detail.

Lastly, due to our planets being scaled down as covered in section 4.9, our cloud layer was quite thin and very close to the ground. If you climbed a mountain you could potentially touch the clouds, creating artifacts. Our solution to this was making the atmosphere linearly larger as you approached the surface.

4.18 Optimizing planet generation

With the ability to generate planets we are faced with a few problems. Firstly, generating large planets quickly filled up the computers memory, straining it. Secondly, it took several seconds to generate them, making the user wait. We solved this with two techniques: level of detail and GPU generation.

4.18.1 Level of detail

Chunks already allow us to generate the planet in multiple parts. Level of detail (LOD) expands on this by changing the scale and amount of detail each part of the planet is generated with. Parts of the planet that are further away can be generated at a lower level of detail to conserve memory and CPU time. This also speeds up the initial generation of the planet as we can generate at a lower level of detail to begin with.

Our implementation is based on octrees. The planet is represented as one cubic volume large enough to contain the entire planet plus a margin to allow the user to deform the planet. When the user gets close “enough,” the volume splits into eight smaller volumes that act the same way. There is a limit to how many times the volume can be split. When the user moves far “enough” away, eight sub-volumes joins together into a larger one. These volumes are stored in a tree to keep track of the splits. Figure 4.19a shows this, each volume is assigned its own color. Each volume is then used as a chunk as seen in Figure 4.19b.

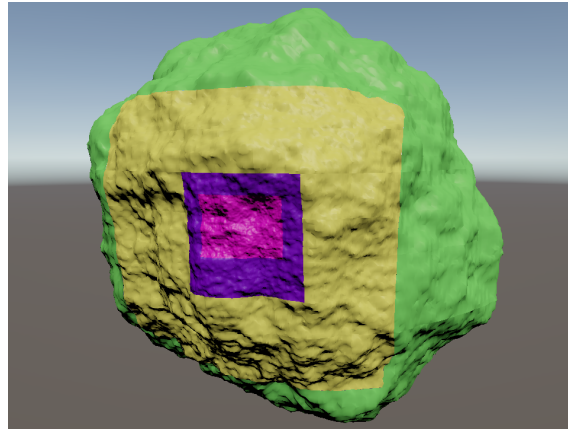


Figure 4.20: Planet generated using clipmaps. The different colors show the different detail levels.

We took care to make sure that a parent chunk is only removed once all child chunks are generated so that parts of the planet do not disappear while it is generating. One problem that remains to be solved are the holes in the mesh between LODs. These are caused by the higher detail meshes capturing details that the lower detail ones cannot, and thus they do not match up. We did not have enough time to implement a solution for this.

The LOD system worked well and allowed us to generate larger planets, but when the user is close up, the demo noticeably slows down. To remedy this, we looked into another technique called geometry clipmaps [52]. They were developed for 2D height mapped terrain data, so we had to adapt them for our 3D use case. The idea is to have nested grids at different scales that are centered around the user. This would only load data that is within a set distance of the user. We got a prototype working (seen in Figure 4.20), but it did not get finished in time, as we instead prioritized porting terrain generation to the GPU.

4.18.2 Moving chunk generation to the GPU

The CPU implementation of terrain generation took too long, so we decided that it should be ported to the GPU, utilizing its parallel nature. This required porting both the noise generation and marching cubes to the GPU. They were implemented as compute shaders, one for generating the terrain data, and one for converting it to a mesh.

Due to the lack of GPU debugging tools, we took small steps where we could easily identify which piece of code caused problems. This lack of tools is also why we decided to wait until we had a working implementation before we moved it to the GPU. We began by just generating a triangle on the GPU, transferred the data to the CPU and created a mesh from it. Then we expanded to generating a sphere with a variable resolution. At this point we had a working version of marching cubes on the GPU, so we could move the terrain data generation to the GPU. Finally, we made the chunks use the new generator.

Initially the generation was done synchronously, waiting until all work was done for one chunk before moving on to the next one. Later, we changed it to dispatch multiple compute shaders at once and then reading back the results when they were ready. To not exhaust resources, we put a limit on the number of chunks that could be generated concurrently. This required careful tracking of which chunks were waiting on being generated and which were in the process of generating.

5

Result

Regolith¹ is a Unity project that generates a procedural planetary system based on a given seed and then simulates the orbital dynamics of all celestial bodies with a direct N-body simulation. In this chapter we present all features that have been implemented from the MoSCoW table in Table 5.1.

Priority	Feature	Status
Must have	Solar system with one star and one terrestrial planet	Done
	Basic movement and camera	Done
	A procedurally generated planet	Done
	Planets rotate around the star and their own axis	Done
	Background with stars	Done
	Basic seeding	Done
Should have	Solar system with multiple planets and moons	Done
	A way to land and explore planets	Done
	Complex movement system	
	Different planet types like gas and ice planets	
	Atmospheres	Done
	Level of detail	Done
	Chunks	Done
	Advanced seeding	Done
Could have	Deformable planets	Done
	Different types of celestial bodies	
	Exploration of gas giants' atmospheres	
Won't have	Multiple solar systems	
	Bodies of water on planets	
	Ecosystems	

Table 5.1: Implementation status of features from the MoSCoW table.

¹<https://github.com/Multipacker/regolith>

5.1 An overview of Regolith

In the Regolith demo, the user starts in the inner planetary system. From here the mouse is used to rotate the camera and **W**, **A**, **S**, **D**, **CTRL**, and **SPACEBAR** to move. **SHIFT** can be pressed to increase the movement speed by a multiplier, where the arrow keys can be used to change the base acceleration and multiplier strength.

The user can travel freely to any of the celestial bodies in the simulation and press **T** while looking at a desired planet to switch to ground movement. They are also able to deform planets by clicking the right mouse button to remove terrain and the left mouse button to create it. The simulation speed can be changed at runtime and the motion of bodies in the simulation can be visualized by turning on the orbit trails.

The user can find planets and moons of different types, examples of which can be seen in Figure 5.2 and Figure 5.3. For terrestrial planets an atmosphere is generated with the potential for clouds. Moons may be found orbiting any planet, but are more common further out, especially around gas giants. They come in the same color types as terrestrial planets but have a different kind of terrain with craters of different sizes.

5.2 The technical sub-systems

This section covers the technical sub-systems and gives an overview of how they work and what they can do.

5.2.1 Simulation

The motion of celestial bodies in the planetary system is simulated with a direct N-body simulation assuming Newtonian gravity and using Leapfrog integration. The simulation can be sped up and slowed down at runtime, allowing the user to see the system develop as the orbits are visualized with a colored trail as seen in Figure 5.1. Radius, density and semi-major axis of bodies are generated based on real-world observations with influence based on the seed. These attributes, along with equations based on the two-body problem (see subsection 2.1.3), are used to determine the initial state of the simulation, that is the set of initial positions and velocities of all celestial bodies. The two-body approximation is made in every central body-satellite system, along with a margin of error, in order to avoid chaotic behavior of the whole system. Whether a planet can inhabit moons or not is decided by the Roche limit and the Hill sphere.

A scaling factor is applied to the radius to get a simulation at a $1 : 10^4$ scale where the terrain generation works. A larger scaling factor of $1 : 3 \cdot 10^6$ is applied to the semi-major axis to place celestial bodies within viewing distance of each other so that the user can see the whole planetary system at the same time. The two factors are easily modifiable to allow a simulation to be run at a larger scale eventually. Still, the scale is large enough to cause floating point errors if not handled, which is

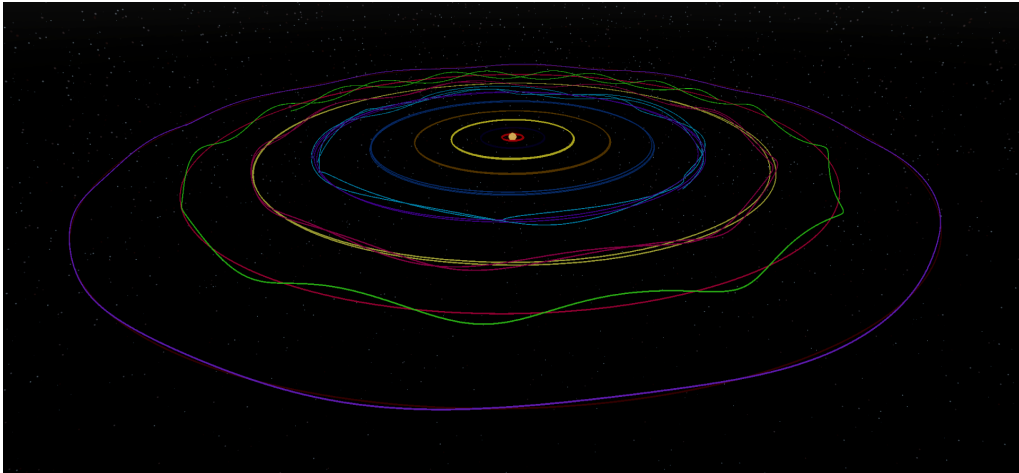


Figure 5.1: An overview of orbits in the simulation after the outermost planet has completed one rotation.

why the simulation has a floating origin and stores attributes of celestial bodies in doubles.

5.2.2 Terrain generation of planets and moons

The terrain generator uses the marching cubes algorithm (see section 2.4) to generate procedural planets and moons from volumetric density data. This is generated in a three-dimensional grid of floating point values where the value zero represents the terrain surface. The density values are generated procedurally using layered 3D noise adapted for spherical terrain generation.

Planets and moons are divided into chunks where each chunk generates its own mesh independently and asynchronously on the GPU. Terrain meshes are generated dynamically and are updated depending on the user's position. A level of detail system is used where nearby chunks use higher resolution meshes while distant chunks use lower resolution meshes.

The terrain generation supports deterministic generation through the use of seeds, allowing the same planetary terrain to be reproduced consistently. Collision meshes are generated for all chunks, allowing the user to walk on and explore the terrain surface.

5.2.3 Gas planets

Gas planets are generated using the seed described in subsection 5.2.1. When this seed determines that a gas planet should be generated the simulation creates a sphere and applies the shader described in section 4.10 as a material. The seed also determines the colors of the planet and sets the necessary properties for the n-body simulation such as density and size.

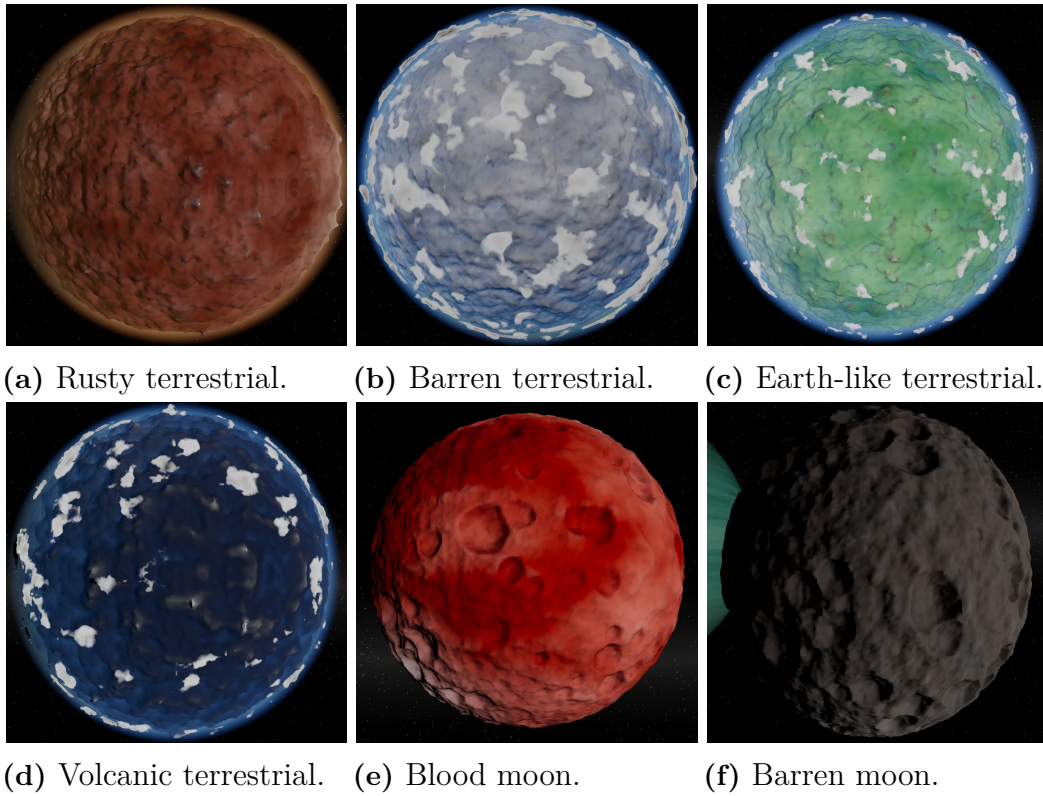


Figure 5.2: Examples of different terrestrial types and moons.

The planets feature four colors: two for the main body of the planet and two for the stripes. To keep the coloring consistent, a base color is picked from the seed, and then four colors are generated from it by slightly adjusting the hue, saturation and brightness.

5.2.4 Atmospheres and clouds

Our planetary system successfully renders different kinds of atmospheres. These atmospheres feature varying properties such as wavelengths, scattering strength and

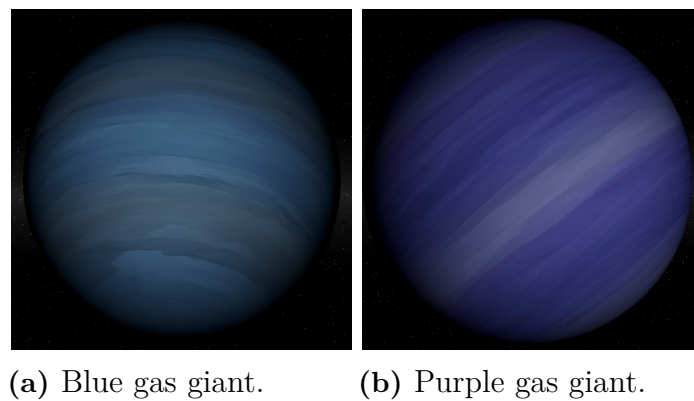


Figure 5.3: Two different gas giants.

density which dictate the look of the atmosphere. The lighting is performed in a physical manner, taking into account Rayleigh and Mie scattering. These properties create a dynamic atmosphere that changes color depending on the observer's position and view direction.

Some atmospheres contain volumetrically rendered clouds that display varying shapes depending on their location and altitude on the planet, these shapes vary from flatter to taller structures. Clouds are illuminated by both direct and ambient light, creating darker and lighter areas.

5.2.5 Skybox

The background in Regolith is a procedurally generated texture applied to the Unity skybox. It consists of blue and red stars of different sizes and strengths and a horizon band that emulates the glow from stars far away in the same galaxy. The position of stars is determined by 3D Voronoi noise along with a density bias towards the horizon band.

6

Discussion

This chapter discusses the working process and result of the project. Then ethics and sustainability from multiple angles, including the effect of procedural content generation on artists, open source contribution and the usage of generative AI. Finally, potential improvements to parts of Regolith are discussed and ways in which the simulation could be further developed are suggested.

6.1 Result

The purpose of this project was to explore and learn about procedural generation, real-time rendering, and physics simulation by creating a program that generates a planetary system that can be explored in real-time. By implementing these features in Regolith we developed a deeper understanding of how they can be designed and integrated in a real-time application. The project demonstrates practical approaches to generating large-scale environments, handling challenges in rendering, and simulating physics in an efficient manner.

Having developed Regolith, we can now provide guidance for how other developers can implement their own version of these features. One of the most important lessons learned was the value of planning for scalability early in development. Implementing level of detail (see subsection 4.18.1) and floating origin (see section 4.4) early proved essential for handling the large scales involved in planetary simulations without the need for refactoring later on. Another key takeaway was that spherical worlds introduce challenges not present in traditional Unity projects. Developers attempting similar projects should carefully consider these challenges during the design and early development phases, as they can significantly influence the architecture of the entire system.

6.2 Process

A great aid in our development was the MoSCoW table that we created in the beginning to list the features we sought out to implement for Regolith. We used it to prioritize certain core features that we wanted to implement first. With it, we had a good plan on what to do first during our development process. As seen in Table 5.1, we managed to implement every features under the “Must have” priority list. The reason for this achievement is that we planned our development phase so

that we would implement a minimal viable product (MVP) early. We deemed the MVP to be implemented when all the “Must have” features were implemented. This decision was crucial for our continued success at implementing the features listed in the MoSCoW table.

After we had our MVP, we had the opportunity to implement the rest of the features in parallel. Unlike the “Must have” features, we did not manage to implement every single feature under “Should have”. There are two reasons for this outcome. The first reason for this was because of time constraints. We did not have enough time to implement every single feature listed in the MoSCoW table, leading to us having to decide what features we needed to prioritize. As seen in Table 5.1, the features we managed to finish implementing were important for the quality of our simulation and the performance when running it. We also decided to implement one of the “Could have” features, deformable planets, because we deemed it easy to develop with our current software architecture and implemented features. The second reason for our outcome was because we divided the work and implemented the features in parallel. Developing the features in parallel was important for us to implement as many features as possible. The problem with this was that much of our time was spent integrating the features. We believe we could have benefited from more cooperation with each other. However, a positive side effect of working in parallel was that the program became more modular.

6.3 Ethics and sustainability

Here we discuss some of the ethical implications of how we have worked and how our work could be used. We have identified several possibilities, but have chosen to focus on three here: our usage of AI during development, contributions to open source, and replacing level designers. We discuss each one in turn.

6.3.1 Use of generative AI during development

With the widespread deployment of large language models, especially within software development and programming, it is important to assess our usage of it. There are several ethical questions that can be posed. How the data to train the AIs is collected is a gray-zone, but there have been several legal cases where authors are claiming that AI companies are using their copyrighted works for training. There is also question of energy use for training and running the models and how that affects the people who live nearby. In the USA, where a lot of the data centers for today’s models are located, people are reporting higher electricity bills [53]. However, analysis by researchers at LBNL did not identify data centers as a main driver for electricity price [54].

AI is available in a few different forms: AI summaries in search results, dedicated chat-bots, and coding agents integrated with development tools. The members in the team have slightly different views on AI and thus our use of it has varied. Still, we managed to agree to a limited use of AI. When used it was to provide additional consultancy during implementation of various features and writing. It is important

to note that AIs can potentially make mistakes, which is why every use of AIs has been done in moderation and evaluated.

6.3.2 Contributions to open source

By making our work open source under the MIT license we are contributing to the library of public and freely available code. Other developers now have the ability to take inspiration, learn, and improve upon our implementation. A possible downside to open source is that the product can be used toward negative or unethical goals. However, we deem this risk to be low with respect to Regolith as it does not provide any malicious functionality that we can think of.

6.3.3 Procedural generation and its effect on designers

With procedural generation being capable of generating large amounts of unique content in a short time, it inherits a discussion on the impact on designers. This has the potential for designers and artists to be replaced in the name of efficiency. We do not believe this to be a significant risk.

Procedural content might feel soulless and without intention as it can be hard to model an algorithm to capture the intentionality of handmade content. For a procedural generation algorithm to create results that feel intentional, the algorithm must be constructed and tuned to get the desired outcome. Here, designers can still come with feedback and provide insights into how they think about design. We believe this is a form of handcrafting as it is only the nature of the work that changes. Thus there is still work for designers, both to design the algorithm and set parameters, but also to modify the procedural content to achieve a intentional feel.

Therefore we believe a hybrid approach will provide the best development process, as a base can be generated that would otherwise require a lot of manual designing. On this base the designer can improve and add details that are hard to model with an algorithm. According to the study *Procedural versus human content generation: Two sides of the same coin?* [55] the use of procedural content generation can improve game development with minor influence on the player experience. Therefore we deem our work with procedural generation could contribute to an improved game development process. However, this is a complex topic and we cannot cover everything in this thesis. For further reading we refer you to the works of Rodrigues and Brancher [56] and Chia [57].

6.4 Future work

There are many possibilities to expand on our work. We believe that our implementation is built in such a way that it should be possible to easily extend it. To start with are the items from our MoSCoW table that are not completed (see Table 5.1). For example, we never added ice planets or the ability to explore the atmospheres of gas giants.

In its current state some aspects of the user interactivity could be improved upon. To correctly transition between flying and walking, the user must manually decrease the speed of the camera in order to achieve a smooth landing on the planet. The system could instead automatically interpolate the speed to a lower value. Simulation parameters are not exposed to the user, they are only available in the Unity inspector (see section 3.1). Future work of the UI could involve adding menus and UI elements that allows the user to tweak features to their will. Features such as the shape and size of edits for planetary deformation, simulation speed and changing seed could also be exposed in the UI.

6.4.1 Planets

The generation of terrestrial planets could be improved significantly. While the user is close to or on a planet, the entirety of it is still loaded at some level of detail, even parts that cannot be seen. This takes a lot of resources while not resulting in any improvement to the look of the demo. In subsection 4.18.1 we explored a different technique that could potentially result in better performance, but we did not have time to fully implement it. With improved performance, more resources can be spent on generating interesting terrain. One idea we had but could not implement due to this was caves and other underground structures.

The planets in our simulation are quite uniform. Placing objects like rocks and trees on the planets surface could make it look more interesting and feel less artificial. Another way to increase the variety would be to introducing the concept of biomes, regions placed around the planet based on some form of noise. These would then alter terrain generation, placement of surface objects, texturing, and anything else procedurally generated on the planet. Another feature that would make planets more interesting is the introduction of oceans or lakes. This could either be implemented as a proper water simulation or faked using a textured sphere. If it is simulated, it should probably react to when the user deforms the terrain. Atmospheres and clouds could be improved visually in various ways. For example, the properties that define their visual features could be based on the type of planet. Furthermore, regional weather and storms could be introduced to increase the variety of environments encountered on planets. The noise sampling for the clouds could also be improved, and introduction of better noise generation to model billowy and swirly clouds.

6.4.2 Planetary system

For Regolith to generate a new planetary system with a different seed it has to be restarted. Implementing runtime generation of systems is important to display the true power of procedural generation. We have two ideas on how it can be done. The simplest method would be to create a starting screen where a root seed can be set, from where the current state of Regolith could be executed. The second more interesting approach would be to implement the possibility to travel to other planetary systems, where the current implementation of Regolith would be executed when entering a system.

To further develop the planetary system we see more and different types of celestial bodies as the obvious step. Asteroid belts provide an interesting challenge for the N-body simulation as they contain many bodies, which would be inefficient to simulate with the direct approach we take. Thus, an approximation method such as the Barnes-Hut algorithm mentioned in subsection 2.3.2 is necessary to make it work. Rogue asteroids could also be implemented to simulate objects entering from outside the system which would fit together with using the deformable terrain to simulate collisions between celestial bodies.

Binary stars, or even N-ary star systems, would be quite interesting to implement. It would require a barycentric calculation for the star system in order to figure out their initial state in the simulation. This could potentially cause problems with light and shadow calculations. If the stars are similar they could be approximated by a single light source originating at their barycenter.

7

Conclusion

The purpose of this project was to explore and learn about procedural generation, real time rendering, and physics simulation by creating a planetary system that could be explored in real time. Over the course of the project, we developed Regolith, an application that can generate planets and moons with different looks and characteristics while maintaining efficiency and stability.

Several technical sub-systems were implemented and integrated, including a gravitational N-body simulation, procedural terrain generation using marching cubes, atmospheric rendering, gas giant generation, and level of detail optimization. Collectively, these technologies allowed us to develop deterministic planetary systems featuring different planets and moons with unique appearances. It demonstrated how procedural generation technologies like noise functions, domain warping, and seeding could be employed to create realistic planets and moons.

One of the main challenges throughout the development process was balancing realism, stability, and performance. The gravitational simulation required careful consideration of numerical integration methods and floating-point precision, while terrain generation and rendering demanded optimization techniques such as chunking, GPU computation, and level of detail. Although the simulation was not scientifically accurate in every aspect, the implemented solutions were sufficient to create stable and visually convincing planetary systems.

The project also highlighted the importance of iterative development and experimentation. Many sub-systems evolved significantly throughout development as new problems were discovered and alternative approaches were tested. This process was especially important when integrating the different sub-systems into a single coherent application. The project can also benefit other developers by providing insights into the strategies, design decisions, and implementation methods used throughout its development. In particular, Regolith can serve as a reference for developers interested in creating their own procedural planetary systems or similar large-scale simulations.

In conclusion, the project achieved its purpose of providing the authors with valuable experience and knowledge in procedural generation, simulation, rendering, and large-scale software development. It can also provide insight and recommendations to developers interested in implementing similar projects. Regolith serves both as a technical proof of concept and as a foundation for future work involving more advanced planetary simulation, optimization, and interactive exploration.

Bibliography

- [1] V. Blanco. “Compute shaders.” Vulkan Guide. Accessed: 2026-04-27. [Online]. Available: https://vkguide.dev/docs/gpudriven/compute_shaders/.
- [2] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, 4th ed. CRC Press / Taylor & Francis, 2018, ISBN: 9781138627000. DOI: 10.1201/b22086.
- [3] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner, *Level of Detail for 3D Graphics* (The Morgan Kaufmann Series in Computer Graphics). Elsevier Science, 2002, ISBN: 9781558608382. [Online]. Available: <https://books.google.se/books?id=TkNilsp6lYoC>.
- [4] M. Bern and P. Plassmann, “Chapter 6 - mesh generation,” in *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds., Amsterdam: North-Holland, 2000, pp. 291–332, ISBN: 978-0-444-82537-7. DOI: <https://doi.org/10.1016/B978-044482537-7/50007-3>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780444825377500073>.
- [5] E. Meiri. “Skybox,” Accessed: Feb. 12, 2026. [Online]. Available: <https://ogldev.org/www/tutorial25/tutorial25.html>.
- [6] Hello Games. “No Man’s Sky – about,” Accessed: Jan. 29, 2026. [Online]. Available: <https://www.nomanssky.com/about/>.
- [7] System Era. “Astroneer – a game of aerospace industry and interplanetary exploration,” Accessed: Jan. 29, 2026. [Online]. Available: <https://astroneer.space/>.
- [8] D. Braben and I. Bell, *Elite*, BBC Micro computer game, 1984.
- [9] F. Spufford, *Backroom Boys: The Secret Return of the British Boffin*. Faber & Faber, 2010.
- [10] S. Austin, *Minecraft live 2023: The recap! Highlights from the live show*, Mojang, Oct. 15, 2023. Accessed: Feb. 11, 2026. [Online]. Available: <https://www.minecraft.net/en-us/article/minecraft-live-2023-the-recap->.
- [11] J. Båtsman Hilmerisson, 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,” *Chalmers Open Digital Repository*, 2023. [Online]. Available: <https://hdl.handle.net/20.500.12380/307641>.
- [12] S. Lague, *Seblague/solar-system at episode-03*, Sep. 2020. Accessed: May 12, 2026. [Online]. Available: <https://github.com/SebLague/Solar-System/tree/Episode-03>.

- [13] J. Gagnestad, “Celestial mechanics,” *Access Science*, Jun. 2025. DOI: 10.1036/1097-8542.115600.
- [14] D. Souami, J. Cresson, C. Biernacki, and F. Pierret, “On the local and global properties of gravitational spheres of influence,” *Monthly Notices of the Royal Astronomical Society*, vol. 496, no. 4, pp. 4287–4297, Aug. 2020, ISSN: 0035-8711. DOI: 10.1093/mnras/staa1520. eprint: <https://academic.oup.com/mnras/article-pdf/496/4/4287/33498461/staa1520.pdf>. [Online]. Available: <https://doi.org/10.1093/mnras/staa1520>.
- [15] J. A. Dutton et al., “Chaos,” *Access Science*, Aug. 2025. DOI: 10.1036/1097-8542.124550.
- [16] J. Butcher, “Numerical methods for differential equations and applications,” <http://www.math.auckland.ac.nz/Research/Reports/view.php?id=370>, vol. 22, Dec. 1997.
- [17] H. Yoshida, “Construction of higher order symplectic integrators,” *Physics Letters A*, vol. 150, no. 5, pp. 262–268, 1990, ISSN: 0375-9601. DOI: [https://doi.org/10.1016/0375-9601\(90\)90092-3](https://doi.org/10.1016/0375-9601(90)90092-3). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0375960190900923>.
- [18] T. Quinn, N. Katz, J. Stadel, and G. Lake, *Time stepping n-body simulations*, 1997. arXiv: astro-ph/9710043 [astro-ph]. [Online]. Available: <https://arxiv.org/abs/astro-ph/9710043>.
- [19] M. Trenti and P. Hut, “N-body simulations (gravitational),” *Scholarpedia*, vol. 3, no. 5, p. 3930, 2008, revision #91544. DOI: 10.4249/scholarpedia.3930.
- [20] 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.
- [21] W. E. Lorensen and H. E. Cline, “Marching cubes: A high resolution 3d surface construction algorithm,” in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’87, New York, NY, USA: Association for Computing Machinery, 1987, pp. 163–169, ISBN: 0897912276. DOI: 10.1145/37401.37422. [Online]. Available: <https://doi.org/10.1145/37401.37422>.
- [22] J. T. Houghton, *The physics of atmospheres*. Cambridge University Press, 2002.
- [23] T. Nishita, T. Sirai, K. Tadamura, and E. Nakamae, “Display of the earth taking into account atmospheric scattering,” in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’93, Anaheim, CA: Association for Computing Machinery, 1993, pp. 175–182, ISBN: 0897916018. DOI: 10.1145/166117.166140. [Online]. Available: <https://doi.org/10.1145/166117.166140>.
- [24] M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and Z. Corse, “Local optimization for robust signed distance field collision,” *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, no. 1, May 2020. DOI: 10.1145/3384538. [Online]. Available: <https://doi.org/10.1145/3384538>.
- [25] K. Perlin, “An image synthesizer,” *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296, Jul. 1985, ISSN: 0097-8930. DOI: 10.1145/325165.325247. [Online]. Available: <https://doi.org/10.1145/325165.325247>.
- [26] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games* (Computational Synthesis and Creative Systems), 1st ed. Cham:

- Springer International Publishing, 2016, ISBN: 978-3-319-42716-4. DOI: 10.1007/978-3-319-42716-4.
- [27] P. G. Vivo and J. Lowe. “Fractal brownian motion.” Chapter 13 of *The Book of Shaders*. Accessed: 2026-04-09. [Online]. Available: <https://thebookofshaders.com/13/>.
 - [28] S. Worley, “A cellular texture basis function,” in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '96, New York, NY, USA: Association for Computing Machinery, 1996, pp. 291–294, ISBN: 0897917464. DOI: 10.1145/237170.237267. [Online]. Available: <https://doi.org/10.1145/237170.237267>.
 - [29] Unity Technologies, *Unity — real-time 3d development platform*, <https://unity.com/>, Accessed: 2026-05-10, 2026.
 - [30] Unity Technologies. “Introduction to lighting,” Accessed: May 11, 2026. [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/LightingInUnity.html>.
 - [31] Unity Technologies. “Light types overview,” Accessed: May 11, 2026. [Online]. Available: <https://learn.unity.com/tutorial/light-types-overview>.
 - [32] Unity Technologies. “Shadow mapping,” Accessed: May 11, 2026. [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/shadow-mapping.html>.
 - [33] Unity Technologies. “Introduction to lightmaps and baking,” Accessed: May 11, 2026. [Online]. Available: <https://docs.unity3d.com/6000.4/Documentation/Manual/Lightmappers.html>.
 - [34] GitHub, *Github: The future of building happens together*, <https://github.com/home>, Accessed: 2026-05-17, 2026.
 - [35] Agile Alliance, *What is agile?* Published June 29, 2015, Jun. 2015. Accessed: Jan. 31, 2026. [Online]. Available: <https://www.agilealliance.org/agile101/what-is-agile/>.
 - [36] Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*, 7th ed. Newtown Square, PA: Project Management Institute, 2021.
 - [37] ashleydavis, *Unity freecam (github gist)*, <https://gist.github.com/ashleydavis/f025c03a9221bc840a2b>, Accessed: 2026-05-14, 2026.
 - [38] A. Barnett. “Our sun: Facts,” NASA, Accessed: Feb. 12, 2026. [Online]. Available: <https://science.nasa.gov/sun/facts/>.
 - [39] S. Lague, *Coding adventure: Marching cubes*, 2019. Accessed: Apr. 28, 2026. [Online]. Available: <https://www.youtube.com/watch?v=M3iI2l0ltbE>.
 - [40] P. Bourke, *Polygonising a scalar field*, <https://paulbourke.net/geometry/polygonise/>, 1994. Accessed: Apr. 29, 2026.
 - [41] H. Kniberg, *Minecraft terrain generation in a nutshell*, Feb. 2022. Accessed: Mar. 3, 2026. [Online]. Available: <https://www.youtube.com/watch?v=CSa506knuwI>.
 - [42] F. Fressin et al., “The false positive rate of *kepler* and the occurrence of planets,” *The Astrophysical Journal*, vol. 766, no. 2, p. 81, Mar. 2013, ISSN: 1538-4357. DOI: 10.1088/0004-637x/766/2/81. [Online]. Available: <http://dx.doi.org/10.1088/0004-637x/766/2/81>.

- [43] H. Hutchinson. “Gas giant!” Shader program published on Shadertoy, Shadertoy, Accessed: May 10, 2026. [Online]. Available: <https://www.shadertoy.com/view/1lj3zV>.
- [44] M. Turitzin, *I’m making a game engine based on dynamic signed distance fields (sdfs)*, Jan. 2026. Accessed: Mar. 16, 2026. [Online]. Available: <https://www.youtube.com/watch?v=i1-TXbn5iMA>.
- [45] Unity Technologies. “Render additional lights in a shader in urp,” Accessed: May 9, 2026. [Online]. Available: <https://docs.unity3d.com/6000.3/Documentation/Manual/urp/use-built-in-shader-methods-additional-lights-fplus.html>.
- [46] S. O’Neil, “Accurate atmospheric scattering,” in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley Professional, 2005, ch. 16.
- [47] L. G. Henyey and J. L. Greenstein, “Diffuse radiation in the galaxy.,” *Astrophysical Journal*, vol. 93, pp. 70–83, Jan. 1941. DOI: 10.1086/144246.
- [48] S. Hillaire, “A scalable and production ready sky and atmosphere rendering technique,” *Computer Graphics Forum*, vol. 39, no. 4, pp. 13–22, 2020. DOI: <https://doi.org/10.1111/cgf.14050>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14050>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14050>.
- [49] A. Schneider and N. Vos, *The real-time volumetric cloudscapes of horizon zero dawn*, Presentation slides, SIGGRAPH 2015 Course: Advances in Real-Time Rendering in Games, 2015. Accessed: May 10, 2026. [Online]. Available: <https://d3d3g8mu99pzk9.cloudfront.net/AndrewSchneider/The-Real-time-Volumetric-Cloudscapes-of-Horizon-Zero-Dawn.pdf>.
- [50] A. Schneider, *The real-time volumetric superstorms of ‘Horizon Forbidden West’*, Game Developers Conference (GDC) Vault, 2022. Accessed: May 10, 2026. [Online]. Available: <https://www.gdcvault.com/play/1027688/The-Real-Time-Volumetric-Superstorms>.
- [51] A. Schneider, *Nubis3: Methods (and madness) to model and render immersive real-time voxel-based clouds*, Presentation slides, SIGGRAPH 2023 Course: Advances in Real-Time Rendering in Games, 2023. Accessed: May 10, 2026. [Online]. Available: [https://advances.realtimerendering.com/s2023/Nubis%20Cubed%20\(Advances%202023\).pdf](https://advances.realtimerendering.com/s2023/Nubis%20Cubed%20(Advances%202023).pdf).
- [52] F. Losasso and H. Hoppe, “Geometry clipmaps: Terrain rendering using nested regular grids,” *ACM Trans. Graph.*, vol. 23, no. 3, pp. 769–776, Aug. 2004, ISSN: 0730-0301. DOI: 10.1145/1015706.1015799. [Online]. Available: <https://doi.org/10.1145/1015706.1015799>.
- [53] Bloomberg News, *2025 ai data centers electricity prices*, <https://www.bloomberg.com/graphics/2025-ai-data-centers-electricity-prices/>, 2025. Accessed: May 31, 2026.
- [54] A. J. Lawson, M. C. Offutt, N. R. Ortiz, and L. Zhu, “Data centers and their energy consumption: Frequently asked questions,” Library of Congress, Congressional Research Service, CRS Report R48646, May 2026. Accessed: May 31, 2026. [Online]. Available: <https://www.congress.gov/crs-product/R48646>.

- [55] L. Rodrigues, R. Bonidia, and J. Brancher, “Procedural versus human level generation: Two sides of the same coin?” *International Journal of Human-Computer Studies*, vol. 141, p. 102 465, 2020, ISSN: 1071-5819. DOI: <https://doi.org/10.1016/j.ijhcs.2020.102465>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1071581920300677>.
- [56] L. Rodrigues and J. Brancher, “Procedurally generating a digital math games levels: Does it impact players in-game behavior?” *Entertainment Computing*, vol. 32, p. 100 325, 2019, ISSN: 1875-9521. DOI: <https://doi.org/10.1016/j.entcom.2019.100325>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1875952119300308>.
- [57] A. Chia, “The artist and the automaton in digital game production,” *Convergence: The International Journal of Research into New Media Technologies*, vol. 28, 2 Feb. 2022. DOI: <https://doi.org/10.1177/13548565221076434>. [Online]. Available: <https://journals.sagepub.com/doi/full/10.1177/13548565221076434>.