Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Procedural planetoid generation in videogames

Generación procedural de planetoides en videojuegos

Ginés Cruz Chávez

La Laguna, July 11, 2024

Mr. **Jesús Miguel Torres Jorge**, Profesor Contratado Doctor attached to the Department of Computer and Systems Engineering of the University of La Laguna, as tutor.

Mr. **Manuel Alejandro Bacallado López**, Profesor Contratado Laboral de Interinidad attached to the Department of Computer and Systems Engineering of the University of La Laguna, as co-tutor.

C E R T I F Y

That the report titled:

"Procedural planetoid generation in video games"

has been carried out under their direction by Mr. **Ginés Cruz Chávez**.

And for the record, in compliance with the legislation in force and for the appropriate purposes, they sign this document in La Laguna on July 11, 2024

Acknowledgements

To my parents, Ramón and Jacqueline, for the support and encouragement they have given me throughout my life.

To my tutor Jesús and my co-tutor Manuel, for giving me the opportunity to do my undergraduate thesis in the field of video games, in addition to their show of support, constant monitoring and work attitude.

To the friends I have met throughout these 4 years of university, with whom I have been able to share a very important stage of my life.

License

© This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Resumen

El objetivo de este trabajo ha sido estudiar y utilizar la generación procedural de contenidos (PCG) para crear niveles de videojuego. Se ha desarrollado un algoritmo que permite generar planetoides en tres dimensiones a gran escala, además de describir las deformaciones en su superficie utilizando funciones de ruido Simplex. Para demostrar la viabilidad de este algoritmo, se ha desarrollado un prototipo de videojuego donde se utiliza el planetoide resultante como nivel que el jugador puede explorar.

Tanto el algoritmo como el prototipo de videojuego se han desarrollado utilizando el lenguaje C# y el motor de videojuegos libre Godot. Para los efectos visuales presentes en el prototipo, se ha utilizado el software de modelado 3D Blender, además de las herramientas que aporta Godot para animaciones y efectos de sonido.

Palabras clave: PCG, Generación procedural, Desarrollo de videojuegos, Godot, C#

Abstract

The objective of this work has been to study and use procedural content generation (PCG) to create videogame levels. An algorithm has been developed that allows generating large-scale three-dimensional planetoids, as well as describing its surface's deformations using Simplex noise functions. To demonstrate the viability of this algorithm, a videogame prototype has been developed where the resulting planetoid is used as a level that the player can freely explore.

The algorithm and videogame prototype have been developed using the C# programming language and the open source game engine Godot. For the visual effects present in the prototype, the 3D modelling software Blender was used, as well as the tools for animations and sound effects that Godot provides.

Keywords: PCG, Procedural generation, Videogames development, Godot, C#

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Procedural content generation (PCG) in video games is a technique where content is created utilising algorithms, in contrast to manual content creation, where the data is predefined and stored beforehand. This technique lets the developer define the algorithms that will be used to generate rooms, enemies, textures or any other type of content in a video game in a more dynamic way.

As time goes on, more video games have started to use PCG, as an almost infinite amount of content can be generated from a single algorithm simply by creating variations in its parameters, without the developer needing to manually define all possible combinations. If used correctly, PCG can give a video game more variation in content, and as the processing power of modern computers increases, more examples of video games that incorporate this technique can be observed.

1.1 Description

This project is focused on the creation of a procedural generation algorithm capable of producing three-dimensional large-scale planetoids. The viability of this algorithm is demonstrated with the creation of a video game prototype where the resulting planetoid is used as a level that the player can freely explore.

1.2 Objectives

This project's objectives are defined by the following items:

1. Develop a procedural generation algorithm capable of creating three-dimensional large-scale planetoids, defining its surface deformations using noise functions.
2. To demonstrate the algorithm's viability, using open source tools, create a video game prototype that:
    a) Uses the Godot game engine.
    b) Includes the resulting planetoid as a level that the player can freely explore.
    c) Has at least one mechanic and objective that the player must complete to finish the game.

1.3 Motivations

This line of work was chosen for several reasons. First, to study the generation of procedural planets as well as the algorithms used in the process. Furthermore, the Godot game engine has recently gained popularity as the best choice for indie developers to make their games in. In 2020, Godot was awarded a $250,000 Epic MegaGrant (Godot Engine, 2020), which is given to software that improves upon computer graphics development. This project presents a lot of challenges that, if solved, help towards learning how the Godot engine works, as well as gaining experience on implementing PCG algorithms using the C# programming language. Finally, this project also serves as an opportunity to experiment which game mechanics could work well when paired with procedurally generated planets.

1.4 Technologies used

In this chapter, the technologies used for this project will be described, including the reasons why they were chosen. Every software present in this list is either free and open-source, and thus can be acquired at no cost, or has a free license option for university students and professors.

1.4.1 Godot

Godot is a cross-platform, free and open-source game engine released under the MIT license (Godot Engine, 2024b). The engine has recently gained a lot of popularity among the game development community, as it allows users to create 2D or 3D games using the GDScript, C++ (using GDExtension) or C# programming languages. In Figure 1.1 a screenshot of the engine's editor can be observed.

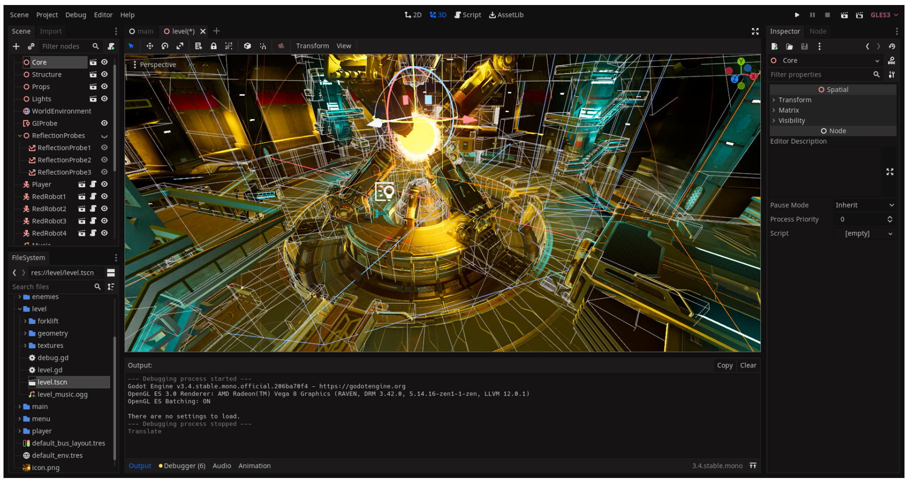
The development process in Godot is structured around a hierarchy of nodes. Every node has a type, and new classes can derive from a node type in order to inherit and extend its behavior. Scenes contain one or multiple nodes, and these can be reused and instantiated anywhere. This workflow is very powerful even for creating user interfaces, as the Godot editor is made using Godot itself.

Godot's MIT license allows the user to make any changes to the engine's code if necessary, although the need for this didn't arise during the project's development. Its support for the C# programming language allows algorithms to be ported over to other game engines or applications, since C# is an established language with a large community of developers.

1.4.2 JetBrains Rider

JetBrains Rider is a powerful integrated development environment (IDE) designed specifically for C# and game development (JetBrains, 2024), offering a suite of tools for coding, debugging, and testing applications. It is widely adopted in the software and game development industries due to its robust features and seamless integration with various frameworks and game engines.

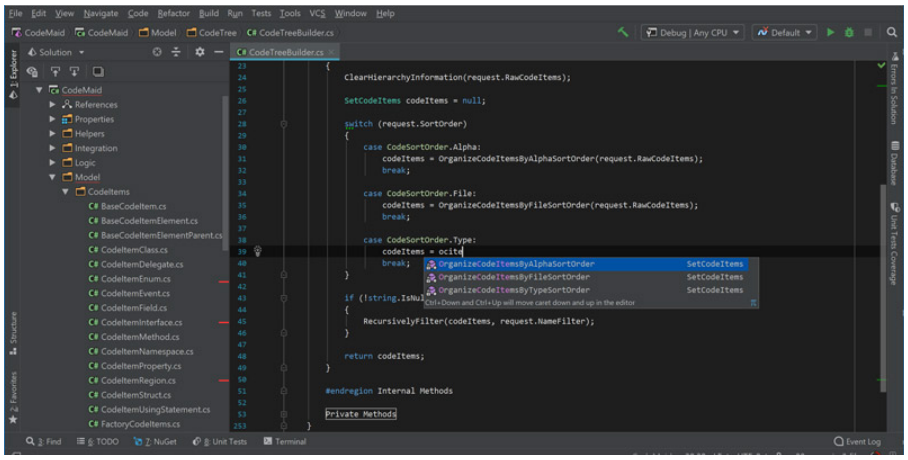
Rider is a popular choice among game developers due to its advanced code analysis capabilities, refactoring tools and intuitive interface, which facilitates some areas of development. In Figure 1.2 a screenshot of the IDE can be observed.



```
--- Debugging process started ---
Godot Engine v3.4.stable.mono.official.206ba70f4 - https://godotengine.org
OpenGL ES 3.0 Renderer: AMD Radeon(TM) Vega 8 Graphics (RAVEN, DRM 3.42.0, 5.14.16-zen1-1-zen, LLVM 12.0.1)
OpenGL ES Batching: ON

There are no settings to load.
--- Debugging process stopped ---
Translate
```

Figure 1.1: Godot game engine

This development environment was chosen because of its integration with the Godot game engine, allowing for both C# and GDScript syntax highlighting, unit testing, launching debuggers for the Godot editor or the game and more. It also offers good performance compared to other IDEs, even with large codebases, ensuring a smooth development experience.

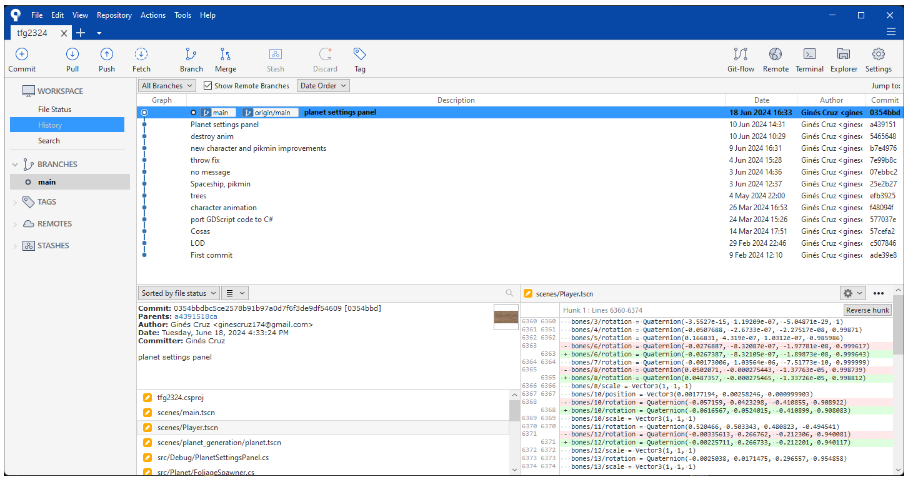


```
        {
            ClearHierarchyInformation(request.RawCodeItems);

            SetCodeItems codeItems = null;

            switch (request.SortOrder)
            {
                case CodeSortOrder.Alpha:
                    codeItems = OrganizeCodeItemsByAlphaSortOrder(request.RawCodeItems);
                    break;

                case CodeSortOrder.File:
                    codeItems = OrganizeCodeItemsByFileSortOrder(request.RawCodeItems);
                    break;

                case CodeSortOrder.Type:
                    codeItems = ocite
                    break;
            }

            if (!string.IsNul

            {
                RecursivelyFilter(codeItems, request.NameFilter);
            }

            return codeItems;
        }

        #endregion Internal Methods

        Private Methods
    }
```

Figure 1.2: JetBrains Rider IDE

1.4.3 Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency (Git, 2024). Due to its open-source license and easy availability, it has been picked as the version control system for this project. To facilitate working with Git on Windows, the free Git GUI Sourcetree (Sourcetree, 2024) was chosen. In Figure 1.3, a screenshot of the Sourcetree GUI can be observed.



Graph	Description	Date	Author	Commit
	main origin/main **planet settings panel**	**18 Jun 2024 16:33**	**Ginés Cruz <gines**	**0354bbd**
	Planet settings panel	10 Jun 2024 14:31	Ginés Cruz <gines	a439151
	destroy anim	10 Jun 2024 10:29	Ginés Cruz <gines	5465648
	new character and pikmin improvements	9 Jun 2024 16:31	Ginés Cruz <gines	b7e4976
	throw fix	4 Jun 2024 15:28	Ginés Cruz <gines	7e99b8c
	no message	3 Jun 2024 14:36	Ginés Cruz <gines	07ebbc2
	Spaceship, pikmin	3 Jun 2024 12:37	Ginés Cruz <gines	25e2b27
	trees	4 May 2024 22:00	Ginés Cruz <gines	efb3925
	character animation	26 Mar 2024 16:53	Ginés Cruz <gines	f48094f
	port GDScript code to C#	24 Mar 2024 15:26	Ginés Cruz <gines	577037e
	Cosas	14 Mar 2024 17:51	Ginés Cruz <gines	57cefa2
	LOD	29 Feb 2024 22:46	Ginés Cruz <gines	c507846
	First commit	9 Feb 2024 12:10	Ginés Cruz <gines	ade39e8

Commit: 0354bbdbc5ce2578b91b97a0d7f6f3de9df54609 [0354bbd]
Parents: a4391518ca
Author: Ginés Cruz <ginescruz174@gmail.com>
Date: Tuesday, June 18, 2024 4:33:24 PM
Committer: Ginés Cruz

planet settings panel

- tfg2324.csproj
- scenes/main.tscn
- scenes/Player.tscn
- scenes/planet_generation/planet.tscn
- src/Debug/PlanetSettingsPanel.cs
- src/Planet/FoliageSpawner.cs

scenes/Player.tscn

Hunk 1 : Lines 6360-6374

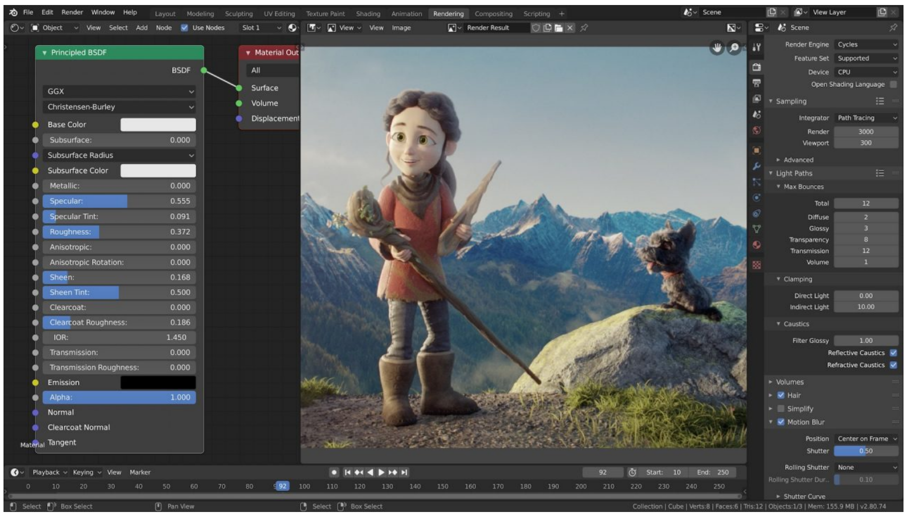
```
  bones/3/rotation = Quaternion(-3.5527e-15, 1.19209e-07, -5.04871e-29, 1)
  bones/4/rotation = Quaternion(-0.0507688, -2.6733e-07, -2.27517e-08, 0.99871)
  bones/5/rotation = Quaternion(0.166831, 4.319e-07, 1.0312e-07, 0.985986)
- bones/6/rotation = Quaternion(-0.0276887, -8.32087e-07, -1.97781e-08, 0.999617)
+ bones/6/rotation = Quaternion(-0.0267387, -8.32105e-07, -1.89873e-08, 0.999643)
  bones/7/rotation = Quaternion(-0.00173006, 1.03564e-06, -7.51773e-10, 0.999999)
- bones/8/rotation = Quaternion(0.0502071, -0.000275443, -1.37763e-05, 0.998739)
+ bones/8/rotation = Quaternion(0.0487357, -0.000275465, -1.33726e-05, 0.998812)
  bones/8/scale = Vector3(1, 1, 1)
  bones/10/position = Vector3(0.00177194, 0.00258246, 0.000999903)
- bones/10/rotation = Quaternion(-0.057159, 0.0423298, -0.410855, 0.908922)
+ bones/10/rotation = Quaternion(-0.0616567, 0.0524015, -0.410899, 0.908083)
  bones/10/scale = Vector3(1, 1, 1)
  bones/11/rotation = Quaternion(0.520466, 0.503343, 0.480823, -0.494541)
- bones/12/rotation = Quaternion(-0.00335613, 0.266762, -0.212306, 0.940081)
+ bones/12/rotation = Quaternion(-0.00225711, 0.266733, -0.212201, 0.940117)
  bones/12/scale = Vector3(1, 1, 1)
  bones/13/rotation = Quaternion(-0.0025038, 0.0171475, 0.296557, 0.954858)
  bones/13/scale = Vector3(1, 1, 1)
```

Figure 1.3: Sourcetree

1.4.4 Blender

Blender is an open-source software suite which allows users to create many forms of digital content, including 3D modeling, rendering, animation and video editing (Blender Foundation, 2024a). It is widely utilized in industries such as film and gaming due to its robust capabilities and flexibility. Many video game developers tend to choose Blender to create their art assets, as it offers all of the needed tools to create any 3D model from scratch. In Figure 1.4 a screenshot of the software can be observed.

Concerning this project, Blender was chosen because it's a free and open-source tool that provides a complete solution for 3D modeling without the need of licensing fees, making it accessible for students and people on a limited budget. Furthermore, it integrates very well with the Godot game engine (Godot Engine, 2024c), as assets can be exported using the .glb format and seamlessly imported into any game without using external tools. Blender's community support and forums (Blender Foundation, 2024b) also provide a useful learning guide for students or beginners.


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object View Select Add Node Use Nodes Slot 1
View Image Render Result
Scene
View Layer
Principled BSDF
BSDF
Material Output
All
Surface
Volume
Displacement
Normal
Clearcoat Normal
Tangent
Material
Playback Keying View Marker
Select Box Select Pan View

GGX
Christensen-Burley
Base Color
Subsurface: 0.000
Subsurface Radius
Subsurface Color
Metallic: 0.000
Specular: 0.555
Specular Tint: 0.091
Roughness: 0.372
Anisotropic: 0.000
Anisotropic Rotation: 0.000
Sheen: 0.168
Sheen Tint: 0.500
Clearcoat: 0.000
Clearcoat Roughness: 0.186
IOR: 1.450
Transmission: 0.000
Transmission Roughness: 0.000
Emission
Alpha: 1.000

Render Engine: Cycles
Feature Set: Supported
Device: CPU
Open Shading Language
Sampling
Integrator: Path Tracing
Render: 3000
Viewport: 300
Advanced
Light Paths
Max Bounces
Total: 12
Diffuse: 2
Glossy: 3
Transparency: 8
Transmission: 12
Volume: 1
Clamping
Direct Light: 0.00
Indirect Light: 10.00
Caustics
Filter Glossy: 1.00
Reflective Caustics ✓
Refractive Caustics ✓
Volumes
Hair ✓
Simplify
Motion Blur ✓
Position: Center on Frame
Shutter: 0.50
Rolling Shutter: None
Rolling Shutter Dur..: 0.10
Shutter Curve

Start: 10 End: 250

Collection | Cube | Verts:8 | Faces:6 | Tris:12 | Objects:1/3 | Mem: 155.9 MB | v2.80.74

Figure 1.4: Blender

Chapter 2
Background and state of the art

Procedural content generation is a very diverse topic: as a technique it has found applications in video games, film and other industries. Before diving into explaining how procedural planetoid generation works, a brief introduction will be given showing commonplace PCG algorithms, as well as video games that use this technology.

2.1 Algorithms

2.1.1 Cellular Automaton

A cellular automaton is defined as a grid of cells, with each one having a state and a set of rules that determine the state a cell transitions to based on the state of it and its neighborhood (WikiDot, 2024). All cells must share the same rule-set, but not their state. In each iteration of the game, every cell looks at its neighbors and chooses the state it will have in the next iteration, with all state changes being done at the same time.

The most prominent example of a cellular automaton is *Conway's Game of Life*. The rules for *Game of Life* were made by the British mathematician *John Horton Conway* in 1970, and state the following (Martin Gardner, 1970):

1. Every cell with two or three neighboring cells survives for the next generation.
2. Each cell with four or more neighbors dies from overpopulation. Every cell with one neighbor or none dies from isolation.
3. Each empty cell adjacent to exactly three neighbors is a birth cell. A cell is placed on it at the next move.

In Figure 2.1 an image of possible patterns in the *Game Of Life* can be observed.

Cellular automata have many applications in PCG, with their most popular use case being dungeon generation, as they tend to create natural looking patterns depending on the used rule-set. The website *Procedural Dungeon Generation: Cellular Automata* (jrheard's blog, 2016) describes how to create an algorithm that can create caves/levels like the one shown in Figure 2.2. All code and parameters present on the website's algorithms can be edited in order to observe how they affect the resulting level.

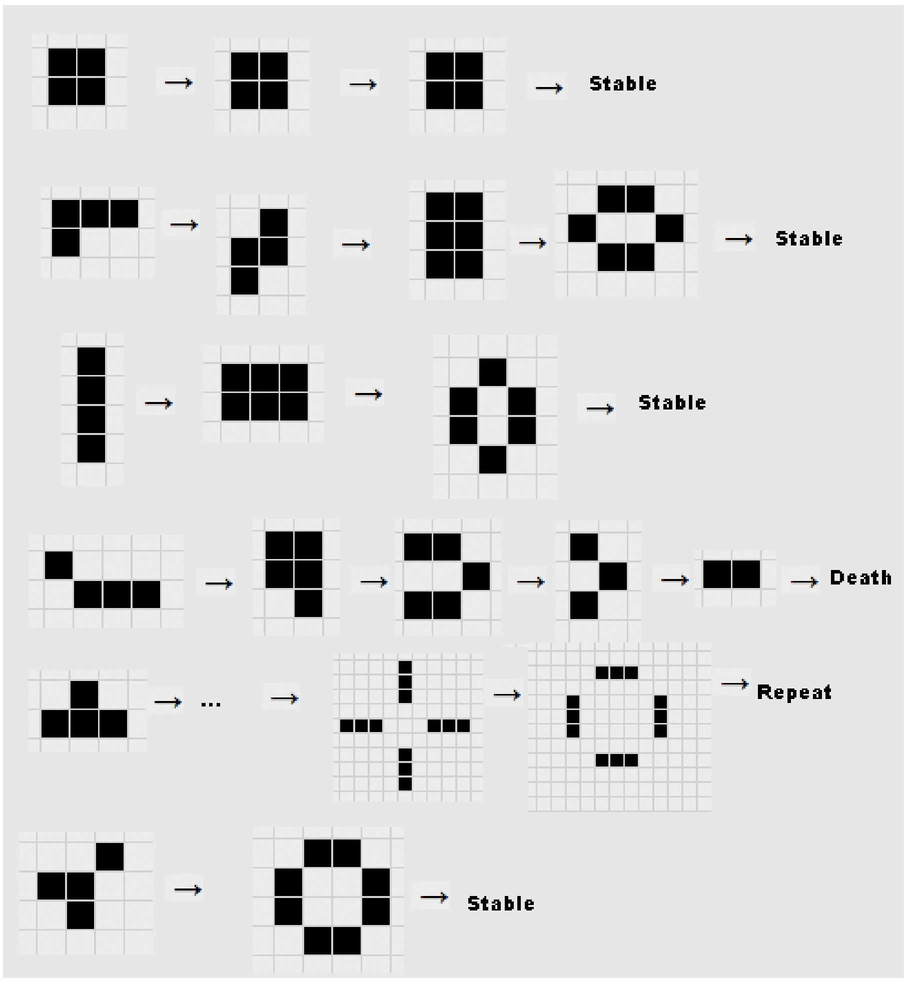


Figure 2.1: John Conway's Game Of Life

2.1.2 Random Walk

Random Walk, sometimes also called Drunkard Walk, is an algorithm for generating maps on a 2D grid, similar to what was previously seen in subsection 2.1.1. The reason for the name Drunkard's Walk is that the patterns this algorithm creates are very staggered, like how a drunkard would walk.

The algorithm (Noveltech, 2023) works as follows:

1. Create a grid of size NxM.
2. Choose a random starting position (x, y) in the grid.
3. Set the position as visited.
4. Move in any cardinal direction (north, east, south, west).
5. If the position is not out of bounds, set this position as the new position.
6. Go back to step 4 until the condition is met (e.g. number of iterations)

This algorithm guarantees connectivity from the starting position (x, y), which means that all areas of the map are accessible. Some modifications may be done to this algorithm to increase its effectiveness, for example, biasing the walking direction towards the center of the grid in order to avoid hitting the

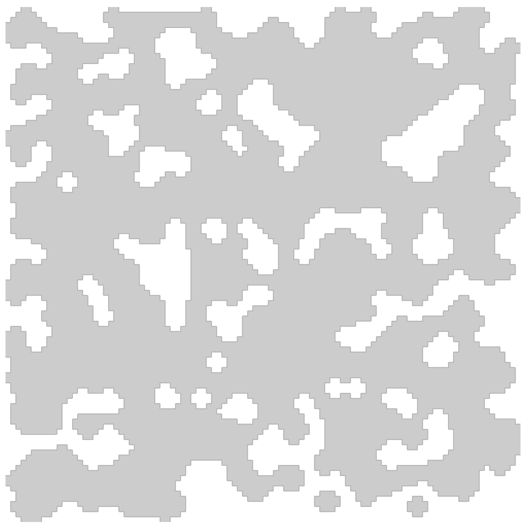


Figure 2.2: Level generated with Cellular Automata

edges. If the walking direction is biased towards the last direction it travelled, it will create longer corridors. In Figure 2.3, a 256x256 map with 65535 iterations was generated.

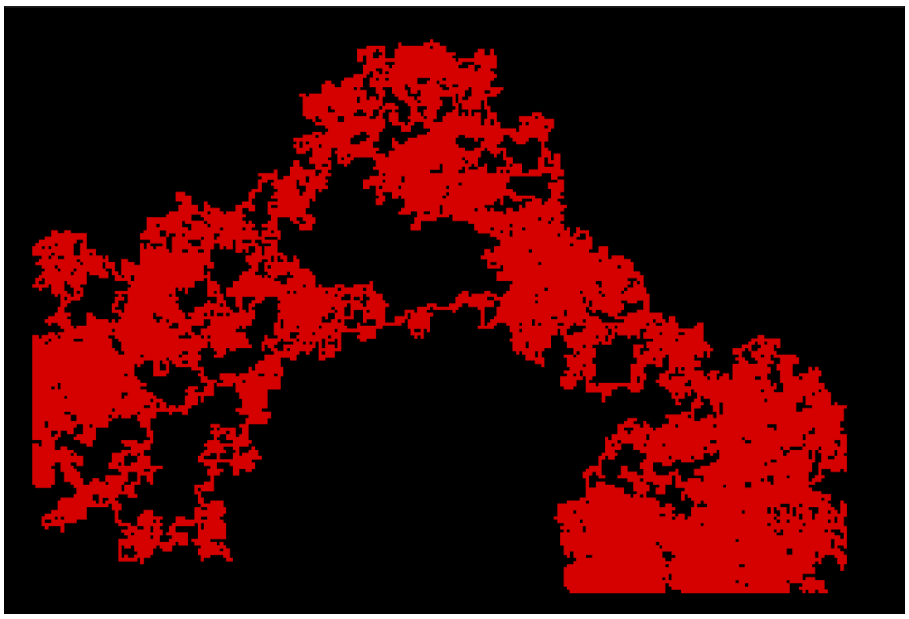


Figure 2.3: 256x256 Random Walk generated map with 65535 iterations

2.1.3 Perlin Noise

Perlin noise (Ken Perlin, 1999) is a type of gradient noise developed by Ken Perlin in 1983 which generates its patterns by interpolating random gradients at the corners of a grid. Unlike simple random noise, which can appear disjointed and artificial, Perlin noise gives a more natural result. This is achieved through the interpolation of gradients, which ensures that the transitions between points are smooth.

This algorithm works in the following way:

1. Divide the space into a grid, where each vertex has a random directional unit vector assigned to it. (example given in Figure 2.4)
2. To find the value for any point, find the grid cell in which the point resides. Identify the corners for that cell, and for each corner calculate a displacement vector from that corner to the desired point.
3. For each corner, calculate the dot product between the offset vector and gradient vector. (example given in Figure 2.5)
4. Interpolate all dot products with a chosen function. In the example seen in Figure 2.6, the smoothstep function was used. This step is responsible for giving Perlin noise its characteristic look, and results may vary if other interpolation functions are used.

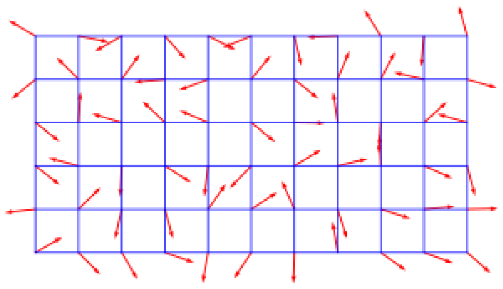


Figure 2.4: Perlin noise grid

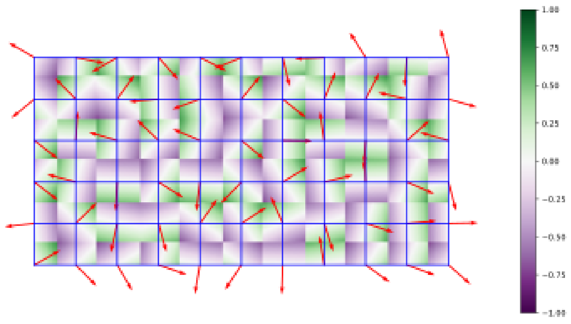


Figure 2.5: Perlin noise grid with dot products

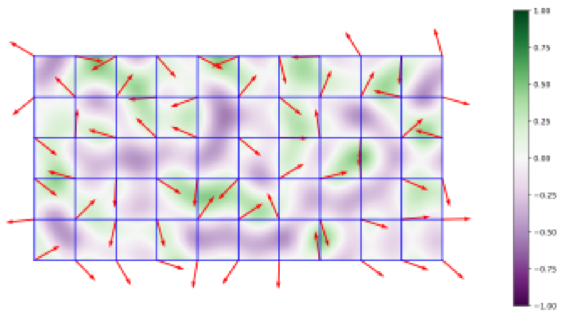


Figure 2.6: Perlin noise grid with interpolation

In Figure 2.7, an example of generated Perlin noise can be seen.

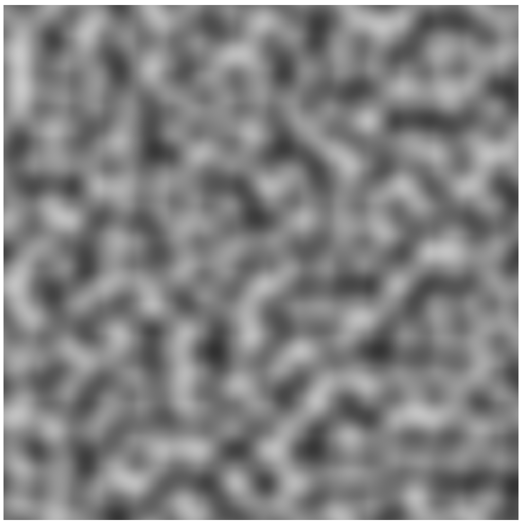


Figure 2.7: Two-dimensional slice through 3D Perlin noise at $z = 0$

In the context of procedural planet generation, Perlin noise can be used to create elevation maps for terrain, simulating cloud patterns or other natural phenomena like the wind. Its ability to produce intricate and naturalistic patterns is why it remains a staple in procedural generation, although for real time usages it has been deprecated in favor of more performant approaches like Simplex noise.

2.2 Video games

The history of procedural content generation in video games is a diverse topic, as this technique started to be used in the first video games due to storage limitations. Reduced disk space in computers at the time limited games' amount of content, so procedural generation algorithms started to be used as a method to generate content in real time adapted to the developer's needs.

2.2.1 Rogue

One of the oldest and most notable examples of procedural content generation is the game Rogue, released in 1980. The gameplay is turn-based, and the player must traverse a dungeon full of enemies and objects, descending through levels until recovering the Amulet of Yendor and carry it back to the surface. This game is characteristic for representing all of its graphics using ASCII characters, something common at the time due to graphical limitations, and using procedural generation to place its rooms, enemies and rewards. In Figure 2.8, a screenshot of the game can be observed.

2.2.2 .kkrieger

.kkrieger was released in 2004 by the .theprodukkt team for the Breakpoint competition, an event for showcasing demos. This game is remarkable for achieving a fully playable first person shooter in only 96 Kilobytes. The low disk size was achieved by having all of the game's content be generated using PCG (textures, sounds, music, 3D models). In Figure 2.9, a screenshot of the game can be observed.

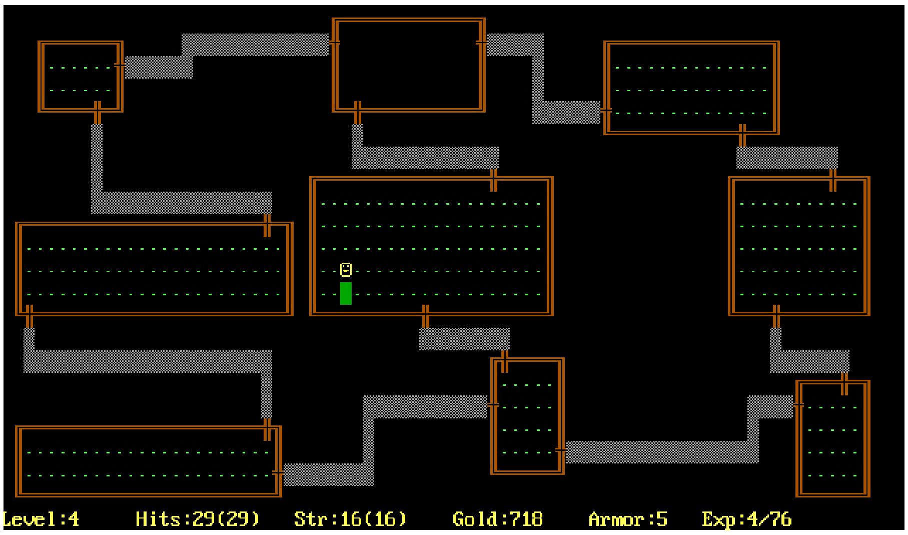


Figure 2.8: Rogue

Figure 2.9: .kkrieger

2.2.3 No Man's Sky

No Man's Sky was released in 2016, and it is the most noteworthy case of procedural planetoid generation. Players can explore a fully procedurally generated universe with over 18 trillion planets. To achieve this scale, the game uses procedural content generation algorithms that take the player's position as an input and generates what they can see around them. Thus, the game does not need to store data for all possible planets, which would take a very large amount of space, and can generate what it needs at a specific point in time discarding all other information. In Figure 2.10, a screenshot of the game can be observed.

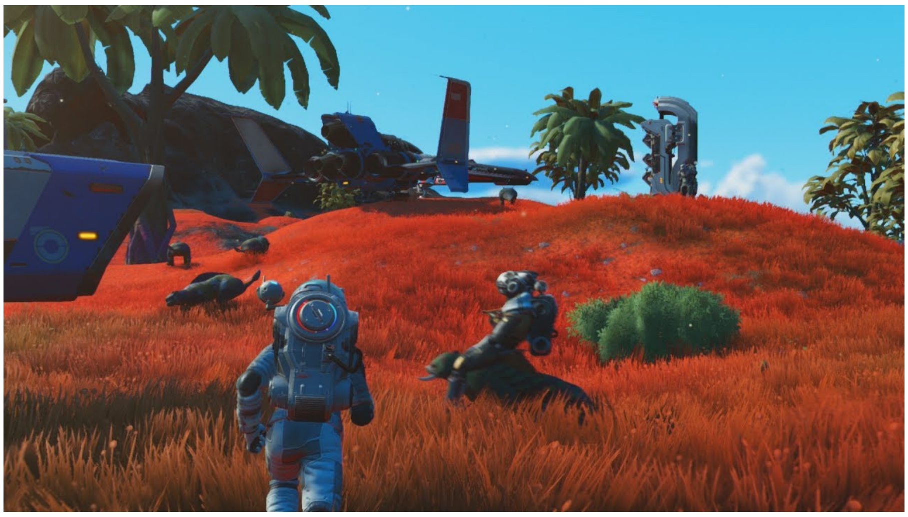


Figure 2.10: No Man's Sky

Chapter 3
Planet generation

The developed planet generation algorithm has five main components: planet terrain with simplex noise, level of detail to increase rendering performance, foliage placement, ocean rendering and atmosphere rendering. Each of these components will be thoroughly explained in the following chapters.

3.1 Planet terrain

The process of creating the terrain for a procedural planet begins with a sphere. There are many ways of creating a sphere 3D mesh procedurally, but the approach used in this work is the *cube sphere* (Jasper Flick, 2024a), also called *quad sphere*.

This sphere is created by starting with a regular cube mesh with many points along its surface. The number of points is important, as it will determine the amount of quality in the resulting terrain. These points can be interpreted as vectors from the center of the mesh to that point's position, and by normalizing them (turning them into vectors with length 1) every point now has the same distance to the mesh's center. In Figure 3.1, an example of this can be seen.

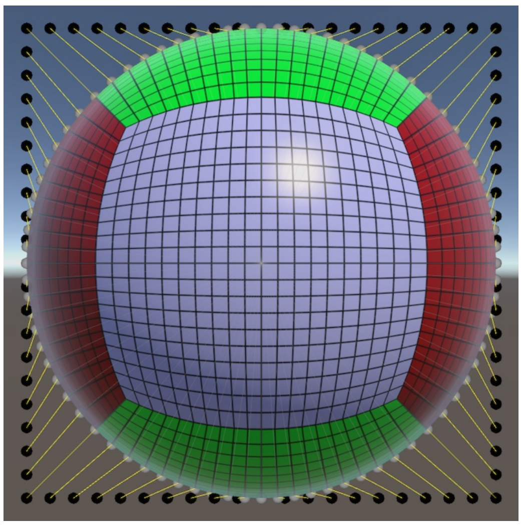


Figure 3.1: Projecting a cube to a sphere

With this primitive constructed, the planet's surface can now be defined by applying a translation to each of these points. This could be done with any function, but to achieve natural looking results, Simplex noise was used.

3.1.1 Simplex noise

Simplex noise (Gustavson, 2005) was designed by Ken Perlin in 2001 as an improvement to his Perlin noise (seen in subsection 2.1.3), allowing for higher dimensions with less performance cost. Perlin noise can suffer from visual artifacts and computational inefficiencies, especially in higher dimensions, and Simplex noise was created to fix this problem.

While Perlin noise uses a regular square grid, Simplex noise employs a simplex grid. In 2D, this grid consists of equilateral triangles (seen in Figure 3.2), and in 3D it uses tetrahedra. This structure reduces the complexity of interpolations and diminishes directional artifacts present in Perlin noise.

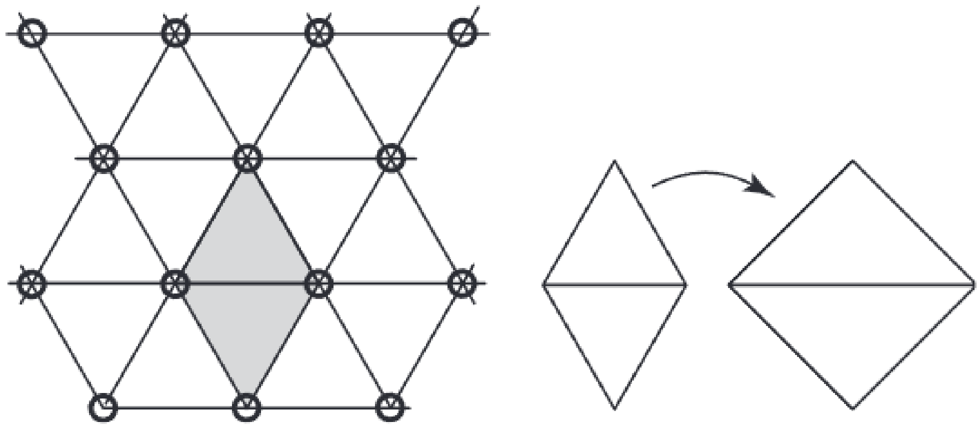


Figure 3.2: 2D Simplex grid (on the left)

Gradient vectors are assigned to the vertices of the simplex grid, designed to minimize correlation between adjacent cells and ensuring a more uniform distribution of noise. This process is very similar to what was seen in Figure 2.4 but using a Simplex grid instead.

To efficiently map points from Cartesian coordinates to the Simplex grid and vice versa, skewing and unskewing transformations are applied. These transformations simplify the calculations needed to determine which simplex the point falls into.

The number of vertices involved in the noise computation is reduced from 2^n in Perlin noise to $n+1$ in Simplex noise (n being the number of dimensions), also achieving significant performance gain when calculating noise in higher dimensions. Simplex noise generates visually appealing patterns without the grid-like artifacts often seen in Perlin noise. The equilateral triangle structure helps produce isotropic noise, which looks more natural. In Figure 3.3, an example of 2D Simplex noise can be seen.

Godot does not implement the original Simplex noise algorithm as it was patented by Ken Perlin (although this patent expired in 2022), but the FastNoiseLite library (GitHub, 2024) used by Godot to calculate different types of noise implements the OpenSimplex2 algorithm, which makes some changes to avoid the patent but is visually the same.

OpenSimplex2 modifies the gradient computation and lattice structure in order to minimize visual artifacts even more effectively than Simplex noise. It also introduces new methods for gradient selection and interpolation, resulting in smoother and more isotropic noise patterns.

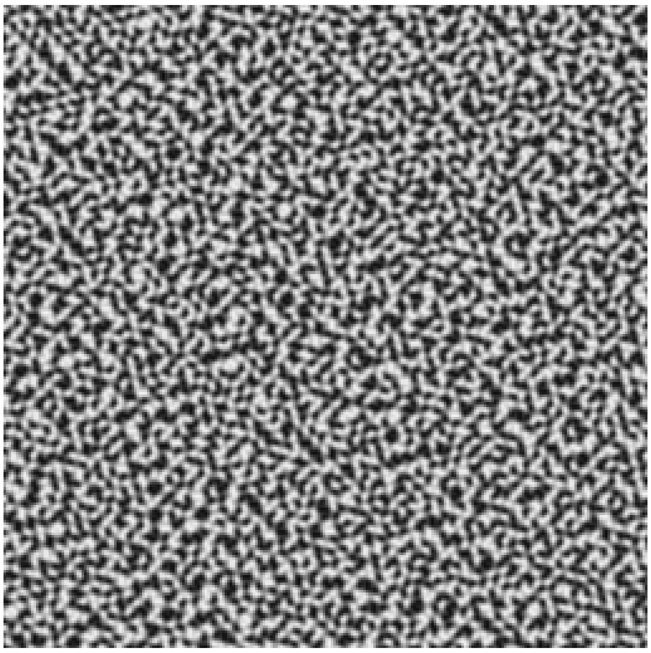


Figure 3.3: 2D Simplex noise

3.1.2 Noise layers

To achieve realistic planet terrain, performing one pass of Simplex noise over the terrain is not enough. The solution to this problem is the introduction of *noise layers*, which break up the patterns found when using noise functions by themselves.

Each noise layer has its own noise map (defined by a seed and frequency), an amplitude and a minimum height value. If using more than one noise layer, the option to use the first noise layer as a mask is also available. The following code defines a noise layer:

```
using Godot;

namespace tfg2324.Planet;

[GlobalClass]
public partial class ProceduralPlanetNoiseLayer : Resource
{
    [Export]
    public FastNoiseLite NoiseMap;
    [Export]
    public float Amplitude = 1;
    [Export]
    public float MinHeight = 1;
    [Export]
    public bool UseFirstLayerAsMask = false;
}
```

Listing 3.1: ProceduralPlanetNoiseLayer

Godot's FastNoiseLite class (Godot Engine, 2024a) allows for Value, Voronoi, Perlin and Simplex noise, but only Simplex noise is used in this context. The ProceduralPlanetSettings class has a method PointOnPlanet that, given a point on a sphere's surface, returns that point after all noise layers have been applied.

```
public Vector3 PointOnPlanet(Vector3 pointOnSphere)
{
    float elevation = 0;
    float baseElevation = 0;

    // Calculate first layer separately to use as mask
    if (NoiseLayers.Length > 0)
    {
        baseElevation = NoiseLayers[0].NoiseMap.GetNoise3Dv(pointOnSphere * 100.0f) *
            NoiseLayers[0].Amplitude;
        baseElevation = (baseElevation + 1 / 2.0f) * NoiseLayers[0].Amplitude;
        baseElevation = Mathf.Max(0, baseElevation - NoiseLayers[0].MinHeight);
    }

    // Apply all noise layers
    foreach (var layer in NoiseLayers)
    {
        if (layer.UseFirstLayerAsMask && layer == NoiseLayers[0])
        {
            continue;
        }

        var mask = layer.UseFirstLayerAsMask ? baseElevation : 1;
        var layerElevation = layer.NoiseMap.GetNoise3Dv(pointOnSphere * 100.0f);
        layerElevation = (layerElevation + 1 / 2.0f) * layer.Amplitude;
        layerElevation = Mathf.Max(0, layerElevation - layer.MinHeight) * mask;
        elevation += layerElevation;
    }

    MinHeight = Mathf.Min(MinHeight, elevation);
    MaxHeight = Mathf.Max(MaxHeight, elevation);
    return pointOnSphere * (Radius + elevation);
}
```

Listing 3.2: PointOnPlanet

In Figure 3.4, a planet was generated with 0, 1, 2 and 3 noise layers, all with different frequency and amplitude parameters. More layers can give the planet finer detail, especially when observed up close, but using more than 3 layers increases computational cost for generation while making a very small difference in the final result.

3.2 Level of detail

While rendering the procedural planet, not all parts of it can be seen at the same time and from the same distance. Parts of the terrain that are up close require more detail, thus needing a higher number of vertices, and parts that are far away can be rendered with a lower vertex count in order to save on computational resources. For this reason, a level of detail (LoD) system was implemented.

The data structure chosen for this LoD solution is a *quadtree*. A quadtree is a tree structure in which each node has exactly four children, and they are most often used to partition two-dimensional

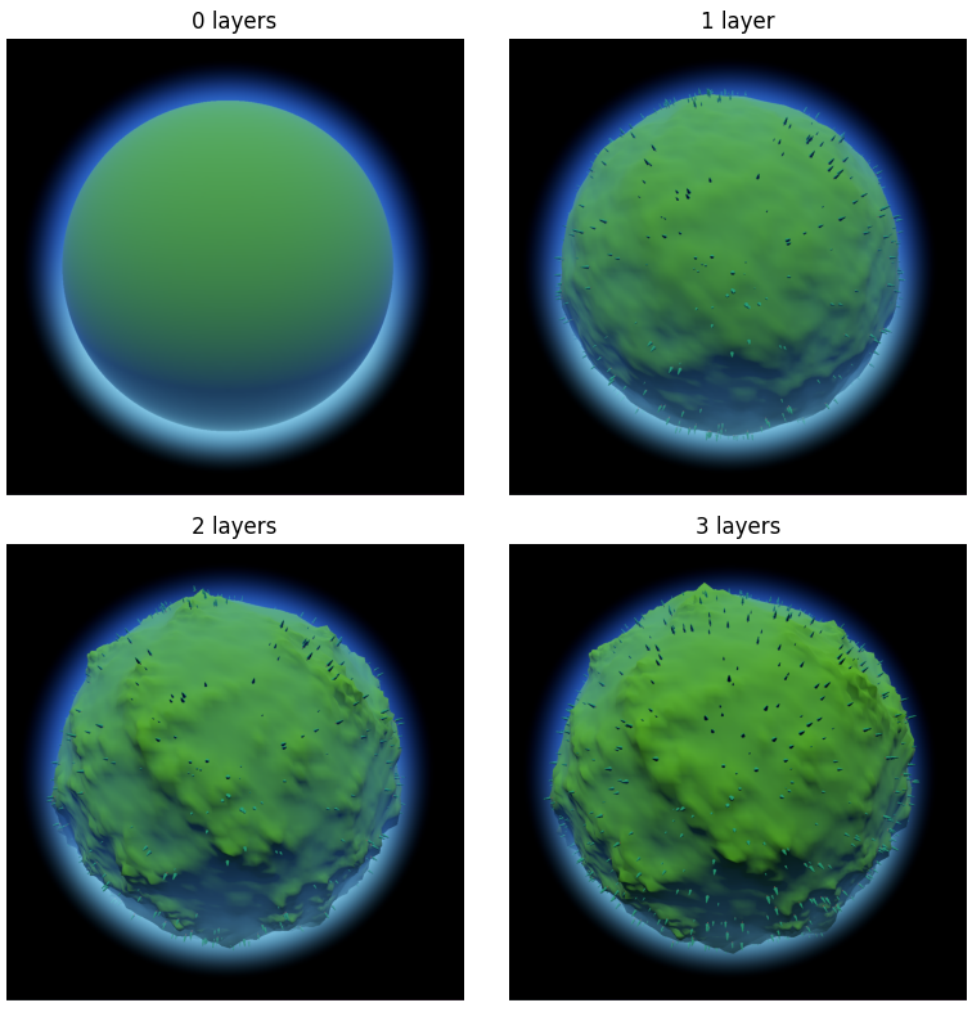


Figure 3.4: Planet generated with 0, 1, 2, and 3 noise layers

space. In this case, if we consider Figure 3.1 the surface can be split up into 6 different quadtrees, one for each side of the cube.

Each quadtree starts with a single node. Every game frame, the distance between the player's position and the center of each node is compared, and if it is below a defined threshold, the node splits into four children. If the distance is higher than the threshold, its children are merged back into a single node. This process is repeated until the maximum depth is reached, which is another parameter that can be manually defined. In Figure 3.5, an example quadtree diagram can be seen, with the red point representing the player's position.

After this process, the generation algorithm considers that each node in the quadtree is a *terrain chunk*, which is defined as a mesh with fixed vertex count that displays a specified portion of the planet's surface. Each time a new quadtree node is created, its associated terrain chunk is generated using the Job Queue. In Figure 3.6, the level of detail system can be observed in wireframe mode.

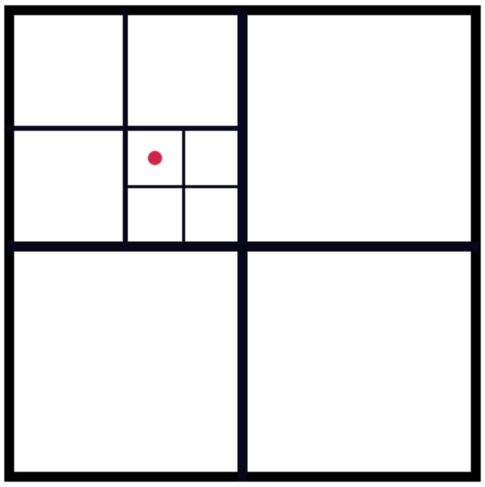


Figure 3.5: Example quadtree with red point representing player position

3.2.1 Job Queue

Due to the level of detail system, chunks are generated and destroyed very frequently if the camera is moving. Performing this generation task on the main thread would cause the game to freeze while executing the algorithm, and the game experience would be severely affected.

The job queue solves this issue by introducing multi threaded generation. This system is structured by the following parts:

- JobQueue: The main class responsible for handling all worker threads. It contains a queue of terrain jobs waiting to be processed and a list of terrain jobs being currently processed. New TerrainJobs can be created using this class.
- TerrainJob: Instantiates a new terrain chunk and builds it using the data provided. When it finishes, it notifies the job queue so it can be removed from the list of processing jobs.
- WorkerThread: The class which contains the code running in all generation threads. It has a reference to the job queue semaphore and waits until a new terrain job is available. When the semaphore is signaled, it fetches a job from the queue and runs it. Once completed, the thread waits for another job to be available.

3.3 Ocean

The ocean of the procedural planet is rendered using a post-processing effect. In the Godot engine, this is achieved by overlaying a plane in front of the camera with a custom shader that applies a filter to what is being rendered on the screen. The Godot shader language allows written shaders to access what has been rendered to the screen as a texture (called the screen texture) as well as the depth for all pixels in the screen texture (called the depth texture). In Figure 3.7, an example of these textures can be seen.

To render a sphere with a screen shader, a ray casting approach has been used. For each pixel in the screen, a ray is fired towards the scene and if it intersects with the planet's ocean sphere, the ocean

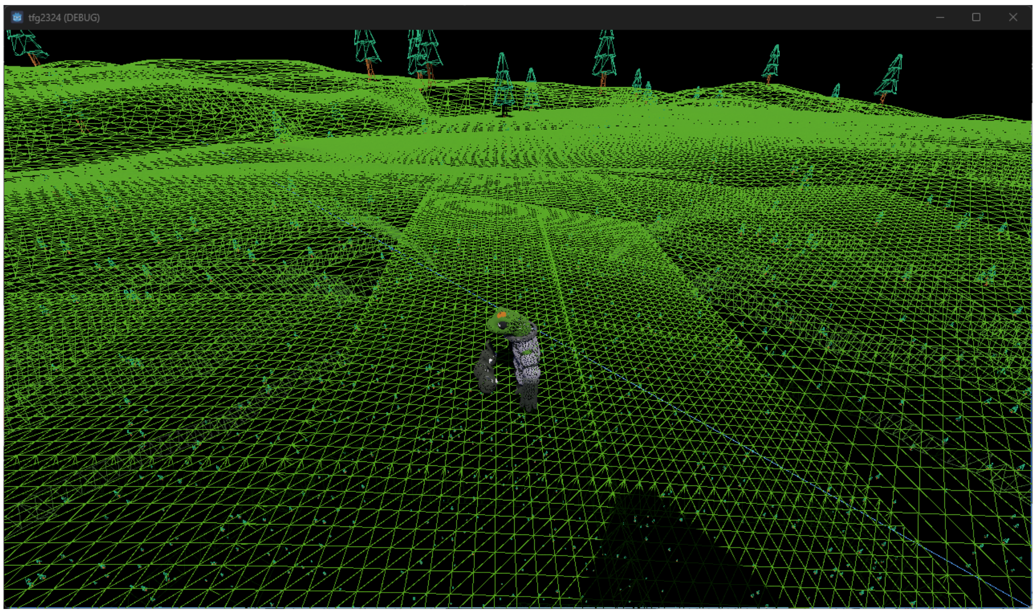


Figure 3.6: Level of detail system in wireframe mode

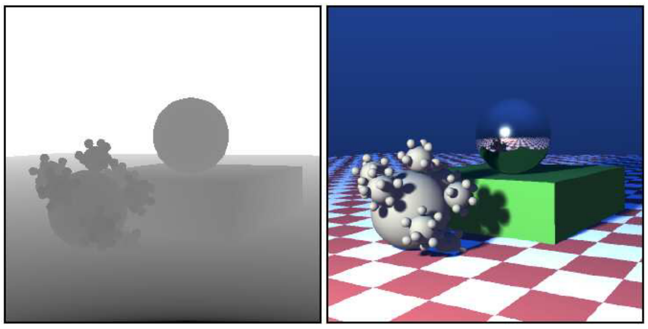


Figure 3.7: On the left: depth texture, on the right: screen texture

is rendered for that specific pixel. This process is explained further in subsection 3.3.2. With the information in the depth texture and the distance obtained in the ray-sphere intersection process, the depth of the water for that pixel is calculated. With this value, an effect was implemented where shallower parts of the ocean were rendered almost transparent, while deeper parts ocean the water's original color.

The waves effect is achieved by using two normal maps (shown in Figure 3.8) that are scrolled along the object at different speeds, mapped with triplanar mapping (Jasper Flick, 2024b) in order to avoid texture distortions in some parts of the sphere. The speed and size of the waves can be controlled in the shader with a parameter.

3.3.1 Phong shading

The lighting is calculated using the Phong shading model, seen in Figure 3.9. This model describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces.

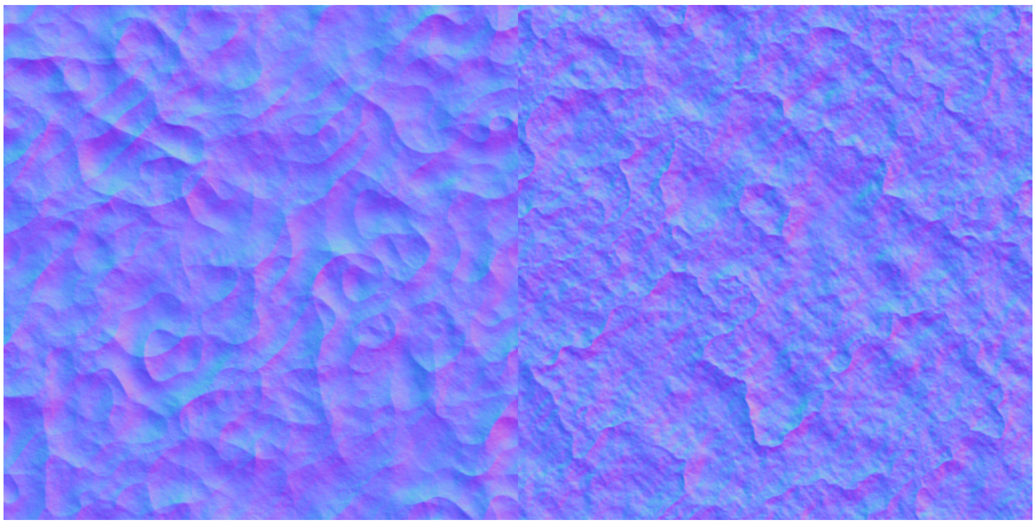


Figure 3.8: Normal textures for ocean rendering

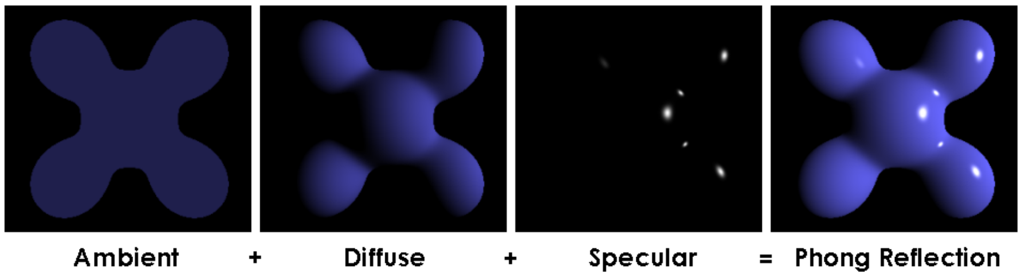


Figure 3.9: Phong shading model.
By Brad Smith - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1030364

The shader receives the sun's direction as a parameter, as well as the smoothness of the ocean. The following code is responsible for the calculation of the diffuse and specular components:

```
float diffuse_lighting = clamp(dot(ocean_sphere_normal, to_sun_direction), 0.0, 1.0);
float specular_angle = acos(dot(normalize(to_sun_direction - ray_direction),
    wave_normal));
float specular_exponent = specular_angle / (1.0 - smoothness);
float specular_highlight = exp(-specular_exponent * specular_exponent);
```

Listing 3.3: Phong shading calculation

These components are all combined to calculate the final pixel color for the ocean. In Figure 3.10, the result of the ocean rendering can be seen.

3.3.2 Ray-sphere intersection

To render the ocean and atmosphere for a procedural planet, the problem of ray-sphere intersection must be solved first. In Figure 3.11, a diagram of ray-sphere intersection can be seen. This is one of the simplest cases of ray-geometry intersection, with different ways to approach it. In this case, an analytic approach was chosen (scratchapixel, 2024).

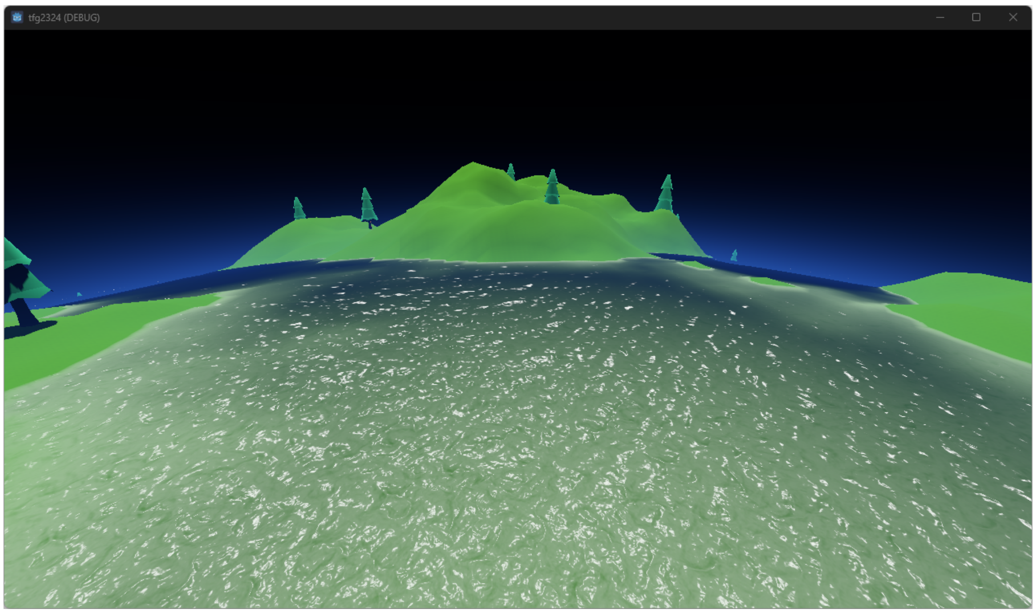


Figure 3.10: Result of ocean rendering

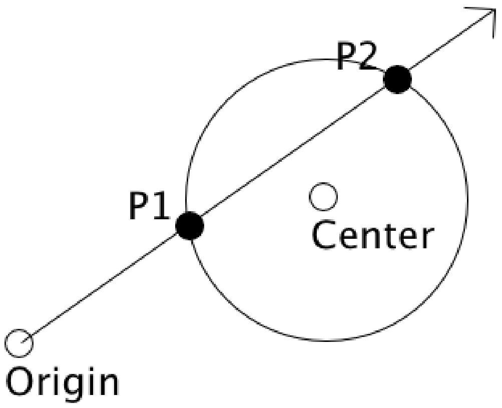


Figure 3.11: Ray-sphere intersection

A ray can be expressed as $O + tD$, where O is the ray's origin, D is a vector denoting the ray's direction, and t is the function's parameter. On the other hand, spheres can be defined using an algebraic form: $x^2 + y^2 + z^2 = R^2$ where x, y and z are Cartesian coordinates for a point and R is the radius of the sphere centered at the origin. If x, y and z are considered as coordinates of point P, it can be expressed as $P^2 - R^2 = 0$.

If P in the sphere's equation is substituted with the ray's equation (which defines all points along the ray), the following is obtained: $(O+tD)^2 - R^2 = 0$. Expanding this results in $O^2 + D^2t^2 + 2ODt - R^2 = 0$, which is an equation of the form $ax^2 + bx + c$ (with $a = D^2, b = 2OD, c = O^2 - R^2$), also known as a quadratic equation.

The discriminant of a quadratic equation is calculated with $\Delta = b^2 - 4ac$, and depending on its value the following cases may apply:

1. If $\Delta > 0$, the ray intersects the sphere at two points.

2. If $\Delta = 0$, the ray intersects the sphere at one point, which means the ray's origin is inside the sphere.
3. If $\Delta < 0$, the ray does not intersect the sphere.

Finally, the distance from the ray to the sphere can be calculated with $\frac{-b+\sqrt{\Delta}}{2a}$, and the distance traveled through the sphere with $\frac{-b-\sqrt{\Delta}}{2a}$.

3.4 Atmosphere

Atmosphere rendering works very similarly to ocean rendering, using a post-processing effect and ray-sphere intersections. The atmosphere shader has parameters that control its radius, height, density and color settings for day/night time. The sun's position and depth texture are also passed to the shader for lighting purposes.

If the rays fired from the camera intersect the atmosphere, a function is called to compute the atmospheric scattering for each pixel. This function steps through the atmosphere in small distance increments, calculating the density and light contribution at each point. The density is influenced by the distance from the planet's surface and the predefined atmosphere height, while the light contribution is determined by the angle between the sun direction and the local up vector at each point.

The accumulated light contributions and density factors are then used to determine the final color of the atmosphere. The shader blends the day and night colors based on the calculated atmospheric and light factors, resulting in a smooth transition between day and night. By adjusting parameters such as the atmosphere height, density and color, the shader can generate a variety of different atmospheric effects.

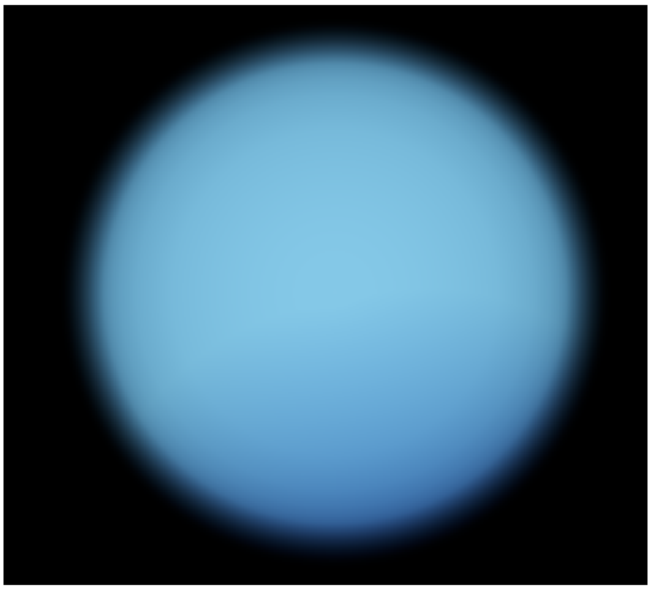


Figure 3.12: Atmosphere

3.5 Foliage placement

The final step in procedural planet generation is placing foliage throughout the surface to add decoration. The number of foliage instances is determined by a factor of the planet's radius, as to proportionally fill the planet with foliage no matter its size. The random number generator seed is very important, as it will determine where the trees will be placed. The algorithm is described below in algorithm 3.1.

Algorithm 3.1: SpawnFoliage

```
1:  procedure SpawnFoliage(planetSettings)
2:      SeedRandomNumberGenerator()
3:      foliageAmount ← Round(planetSettings.planetRadius * 3)
4:      for $i = 0$ to foliageAmount do
5:          randomSpherePoint ← RandomVector3().Normalized()
6:          pointOnPlanet ← planetSettings.PointOnPlanet(randomSpherePoint)
7:          if pointOnPlanet.Length < planetSettings.oceanHeight then
8:              continue
9:          foliageScene ← RandomFromList(foliageScenes)
10:         foliageInstance ← foliageScene.Instantiate()
11:         foliageInstance.Position ← pointOnPlanet
12:         foliageInstance.Scale ← RandomBetween(5.0, 15.0)
13:         foliageInstance.OrientUp()
```

The model used for the tree comes from the free asset pack Kenney's Nature Kit (Kenney, 2020). In Figure 3.13, an example of a planet with generated foliage can be seen.

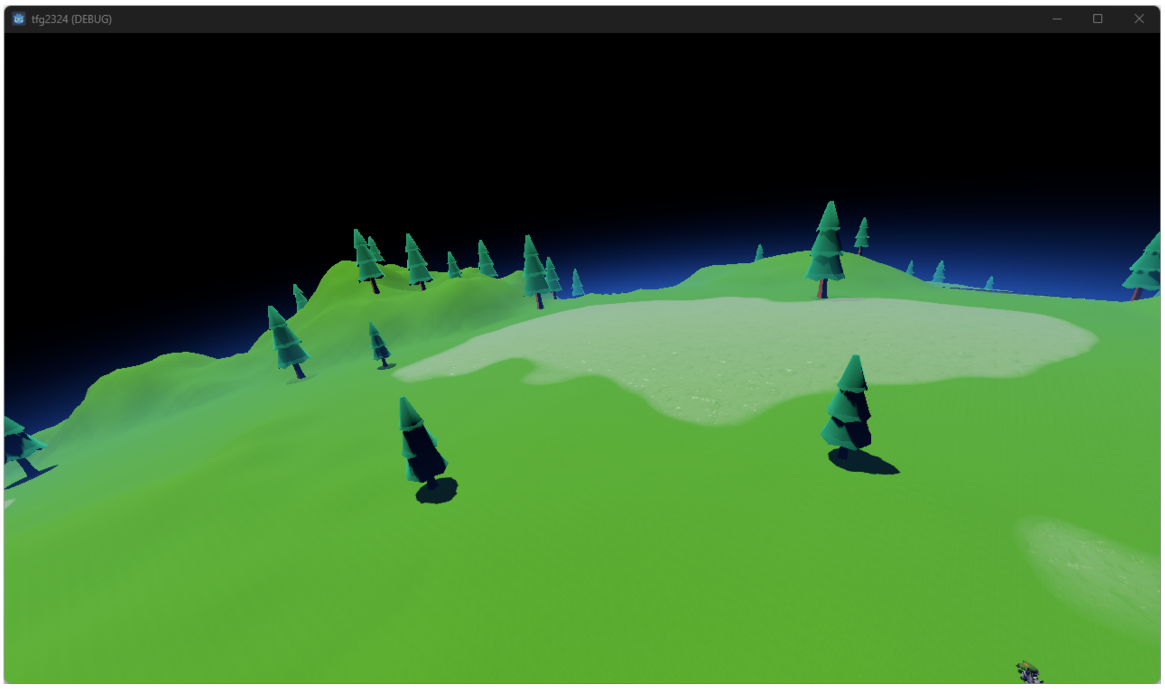


Figure 3.13: Example of foliage generation

Chapter 4
Game prototype

To demonstrate the feasibility of this algorithm applied to a video game, a playable prototype was built. In this chapter, the gameplay and mechanics of this prototype will be explained.

The prototype is divided into two main stages: the planet creator and resource collecting game.

4.1 Planet creator

The planet creator allows the player to see the procedural planet from a far away view, rotating the camera by clicking and dragging the mouse. The view can also be zoomed in or out by using the mouse's scroll wheel. In Figure 4.1, a screenshot of the planet creator can be observed.

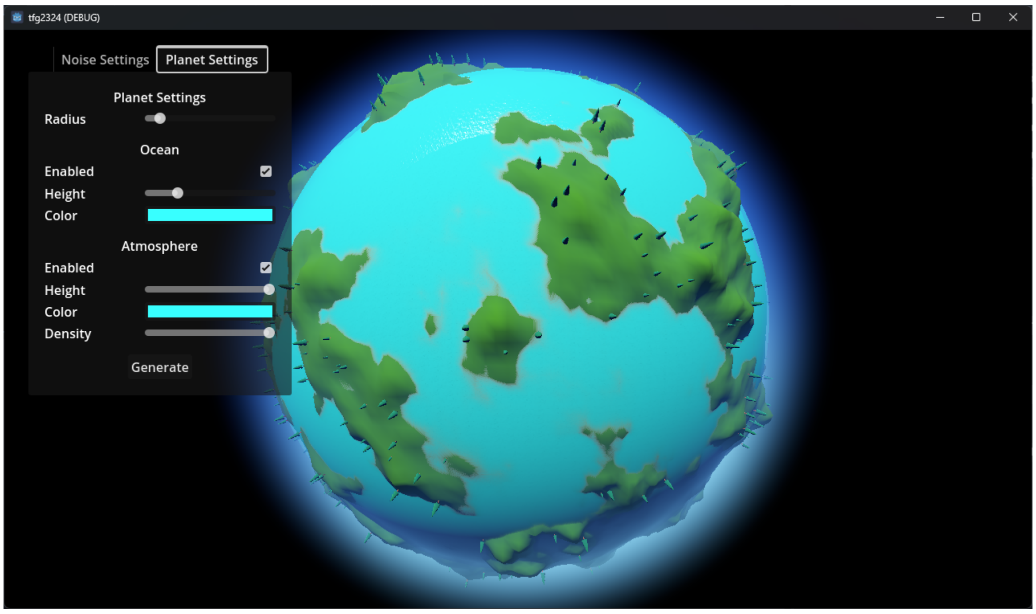


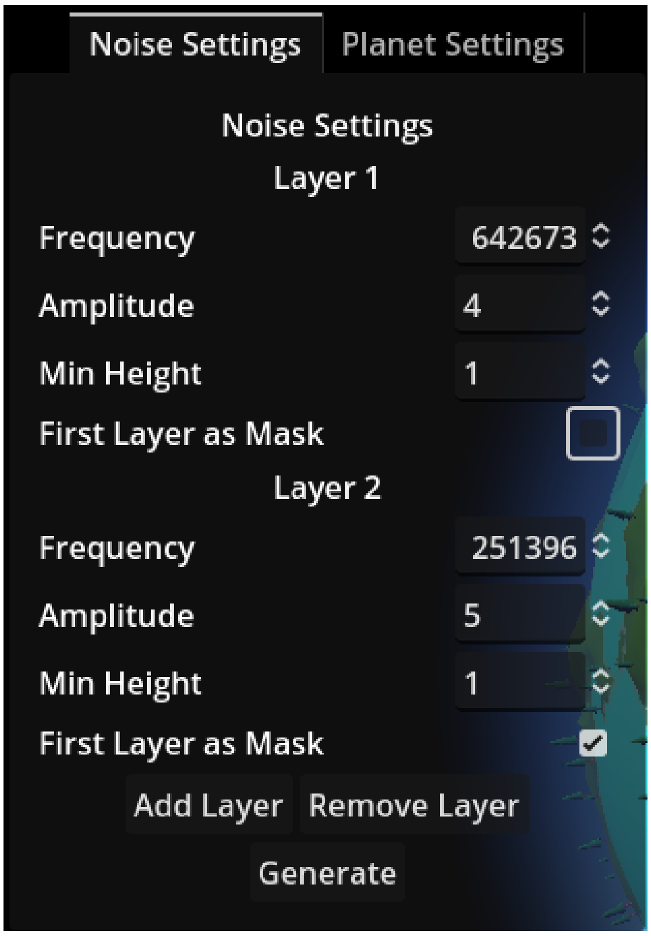
Figure 4.1: Planet creator

The *Planet Settings* tab allows the user to customize the following settings:

1. The radius of the procedural planet.
2. Enable or disable the ocean, change its height or change its color.
3. Enable or disable the atmosphere, change its height, change its color or change its density.

In the *Noise Settings* tab (seen in Figure 4.2), the user can customize the appearance of the planet's surface by adding a noise layer (previously seen in subsection 3.1.2) with the *Add Noise Layer* button, or removing a noise layer with the *Remove Noise Layer* button.

For each layer, its frequency, amplitude and minimum height can be customized, as well as the option to use the first noise layer as a mask so no new islands are created. The planet is updated when the user presses the "Generate" button.



Noise Settings

Layer 1

Frequency	642673
Amplitude	4
Min Height	1
First Layer as Mask	☐

Layer 2

Frequency	251396
Amplitude	5
Min Height	1
First Layer as Mask	✓

Figure 4.2: Noise layers tab

The user can transition to the resource collecting segment at any time by right clicking on any point on the planet's surface. A cutscene will be shown where the camera flies from the current position to behind the player.

4.2 Resource collecting

The resource collecting segment is based on the game Pikmin (seen in Figure 4.3), released by Nintendo EAD for the Nintendo GameCube in 2001. In this game, you play as Captain Olimar, who crashes his ship onto a mysterious planet and must use the *Pikmin* creatures that inhabit the planet to recover the lost ship parts in less than 30 days.

Figure 4.3: Pikmin

This segment aims to combine procedural planets with Pikmin gameplay, allowing you to explore the planet by foot or on a spaceship while hunting for resources using your minions. In Figure 4.4, a screenshot of the prototype in action can be seen.

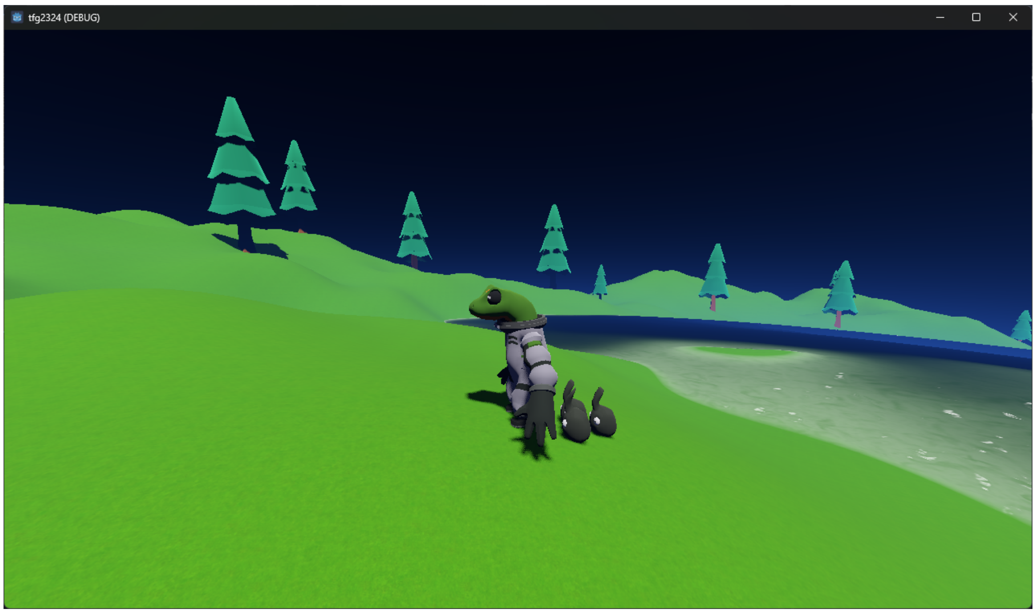


Figure 4.4: Prototype gameplay

4.2.1 Player movement

The player controls an astronaut frog (seen in Figure 4.5) and can move around using the W, A, S and D keys. The mouse is used to rotate the camera around.

The movement was done using Godot's **CharacterBody3D** node, which facilitates creating player controllers by supplying logic for collisions, walking on slopes and moving platforms. Planetary gravity

Figure 4.5: Astronaut frog (from poly.pizza)

is achieved by always pulling the player towards the planet's center, and orienting it so that its up direction is facing away from the surface. The dust cloud effect observed when the player is running was made using Blender (seen in subsection 1.4.4) and Godot's **GPUParticles3D**.

4.2.2 Minions

Minions (seen in Figure 4.6) are this prototype's version of Pikmin. They follow the player and can be commanded to attack resources found in the planet surface. To throw a minion, the left mouse button is used. They function very similar to the player, being able to move around the planet but are controlled by an AI state machine. The following states are possible:

1. **Idle**: The minions are standing idle waiting for a player to call them.
2. **Throwing**: A minion is being thrown by a player. Once it hits the ground, it detects nearby resources and runs towards them. If none are detected, they follow the player.
3. **Follow Player**: The minions run towards the player, stopping when they are close enough.
4. **Follow Objective**: The minions have found a resource to attach and are approaching it.
5. **Attacking**: The minions attack a resource until it runs out of health. If multiple minions attack the same resource, its health decreases faster.

4.2.3 Spaceship

To pilot the spaceship (seen in Figure 4.7), the player must go near it and press the E key to board. With the W key, the spaceship will take off and can be controlled using the W and S keys to propel forward or backward and the A and D keys to roll left or right. The mouse is used to change the spaceship's direction. If the ship is near the planet's surface, the Space key can be used to land, and once landed the player can exit the spaceship with the E key.

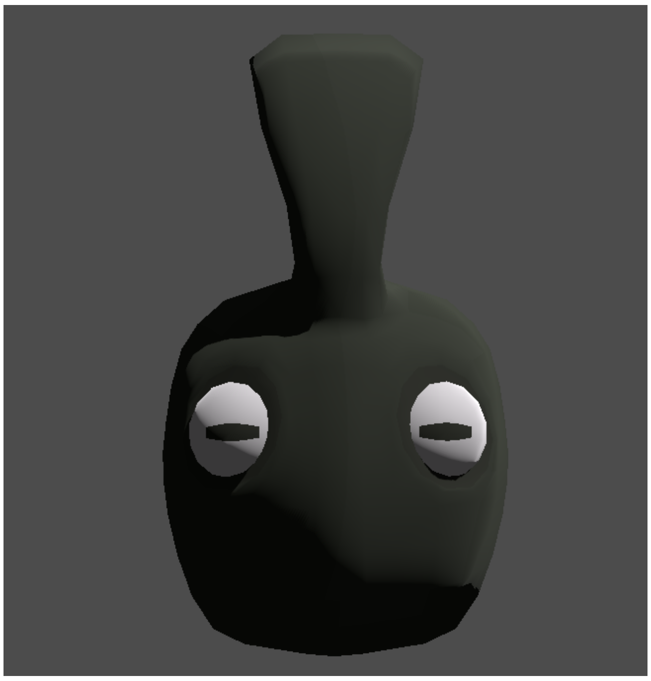


Figure 4.6: Minion (from poly.pizza)

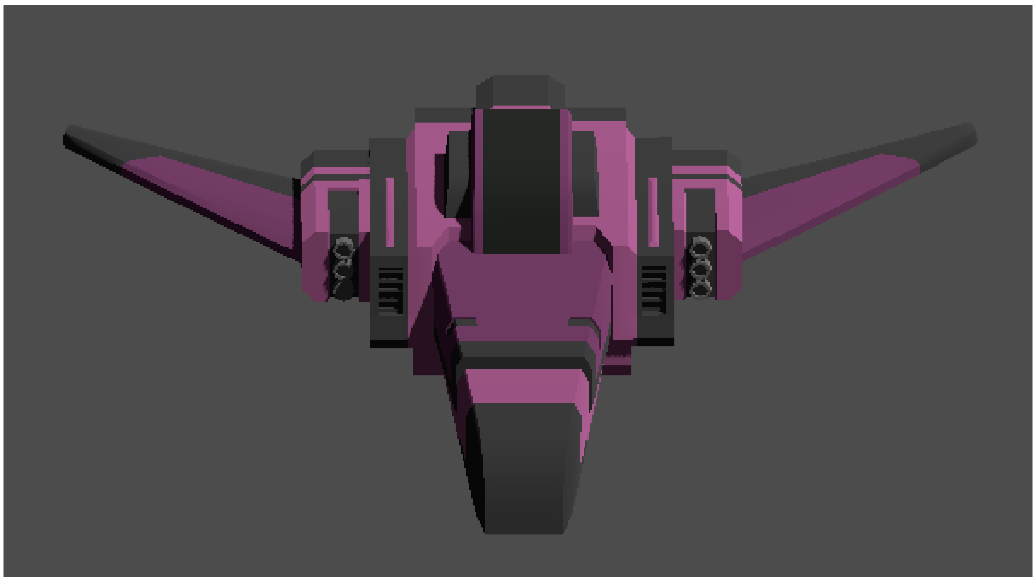


Figure 4.7: Spaceship (from poly.pizza)

Chapter 5
Conclusiones y líneas futuras

En este proyecto se han aplicado algoritmos PCG para crear un sistema capaz de generar planetoides tridimensionales a gran escala. Para demostrar la viabilidad de este algoritmo, se creó un prototipo de juego utilizando el motor de videojuegos Godot que permite al jugador explorar libremente el planeta, así como recolectar los recursos que se encuentran en su terreno. A este prototipo se le añadió el renderizado de océanos y atmósferas planetarias mediante shaders para aumentar su fidelidad visual. Se utilizaron herramientas adicionales, como el IDE JetBrains Rider para el desarrollo C#, Blender para el modelado 3D y Git para el control de versiones.

El prototipo también tiene un modo en el que el usuario puede ver el planeta desde un punto de vista lejano, utilizando el ratón para rotar la cámara. Los ajustes de la superficie, el océano y la atmósfera del planeta pueden modificarse mediante la interfaz gráfica de usuario situada a la izquierda de la pantalla.

5.1 Líneas futuras

Esta sección describirá las posibles líneas futuras para este proyecto:

- Poblar el mundo del juego con varios planetas procedurales para crear un universo procedural, y permitir al jugador viajar entre estos planetas. Este cambio traería consigo nuevos retos técnicos, como el gestionar un mundo de gran escala evitando las limitaciones de precisión de punto flotante y cargar/descargar los recursos en memoria según sea necesario.
- Añadir diferentes tipos de planetas (por ejemplo: nieve, bosque, desierto, volcán) con recursos únicos que coleccionar. Los esbirros serían clasificados por color dependiendo de los tipos de planeta en los que sean capaces de sobrevivir.
- Explorar otros tipos de función de ruido para estudiar cómo afectan al terreno del planeta. El ruido estriado podría crear planetas similares a asteroides, mientras que los cráteres serían capaces de generar lunas.
- Añadir enemigos a los planetas para hacer que la recolección de recursos sea una tarea más difícil. El jugador se ve obligado a utilizar sus esbirros u otros recursos a su disposición para luchar en contra de estas amenazas.
- Añadir asteroides alrededor de los planetas para hacer que el espacio se sienta más poblado, además de que cumplan la función de un obstáculo que el jugador tendría que esquivar mientras viaja. Si son rotos con la nave del jugador, se conseguirían recursos adicionales.

Chapter 6
Conclusion and future lines of action

In this project, PCG algorithms have been applied to create a system capable of generating three-dimensional large-scale planetoids. To demonstrate the feasibility of this algorithm, a game prototype was created using the Godot game engine that allows the player to freely explore the planet, as well as collecting resources found on the terrain. Ocean and atmosphere rendering were added to this prototype using shaders to increase its visual fidelity. Additional tools were used, such as the JetBrains Rider IDE for C# development, Blender for 3D modeling and Git for version control.

The prototype also has a mode where the user can see the planet from a far-away point of view, using the mouse to rotate the camera. The planet's surface, ocean and atmosphere settings can be altered using the graphical user interface (GUI) situated on the left side of the screen.

6.1 Future lines of action

This section will describe the possible future lines of action for this project:

- Populate a game world with multiple procedural planets to create a procedural universe, and let the player travel between planets. This addition would create new technical challenges, like being able to handle a large game world without running into floating point precision errors and managing asset loading/unloading.
- Add different types of planets (e.g. snow, forest, desert, volcano) with their own unique resources to collect and environmental hazards. Minions would now be classified by color depending on the types of planet they can survive in.
- Explore other types of noise to study how they affect the planet's terrain. Ridged noise could create asteroid-like planets, while craters would generate moons.
- Add enemies to the planets in order to make resource collecting a more challenging task. The player is obligated to use the minions or other resources at their disposal to fight back these threats.
- Add asteroids around planets to make space feel less empty, as well as an obstacle that the player would need to avoid while traveling. If the player destroys these with their spaceship, they would gain additional resources.

Chapter 7
Budget

This chapter will go over the cost of the work station used to develop the procedural generation algorithm and game prototype, as well as the cost for hours spent on research, development and report writing.

Table 7.1: Equipment budget

Description	**Quantity**	**Cost (€)**
HP Victus 16-s0011ns	1	1,399

Table 7.2: Labor budget

Hours worked	**Cost (€) per hour**	**Total cost (€)**
300	25	7,500

Table 7.3: Total project cost

Description	**Total cost (€)**
Equipment cost	1,399
Labor cost	7,500
Total project cost	**8,899**

Bibliography

The following bibliographic references are presented in alphabetic order by author. References with more than one author appear ordered according to the first author.

Blender Foundation. (2024a). *Blender*. Retrieved June 26, 2024, from https://www.blender.org/

Blender Foundation. (2024b). *Community - blender.org*. Retrieved June 26, 2024, from https://www.blender.org/community

Git. (2024). *Git*. Retrieved June 26, 2024, from https://git-scm.com/

GitHub. (2024). *Fastnoiselite*. Retrieved June 26, 2024, from https://github.com/Auburn/FastNoiseLite

Godot Engine. (2020). *Godot engine was awarded an epic megagrant*. Retrieved June 26, 2024, from https://godotengine.org/article/godot-engine-was-awarded-epic-megagrant/

Godot Engine. (2024a). *Fastnoiselite - godot engine (stable) documentation in english*. Retrieved June 26, 2024, from https://docs.godotengine.org/en/stable/classes/class_fastnoiselite.html

Godot Engine. (2024b). *Godot engine*. Retrieved June 26, 2024, from https://godotengine.org/

Godot Engine. (2024c). *Importing 3d scenes - godot engine (4.1) documentation in english*. Retrieved June 26, 2024, from https://docs.godotengine.org/en/4.1/tutorials/assets_pipeline/importing_scenes.html

Gustavson, S. (2005). Simplex noise demystified.

Jasper Flick. (2024a). *Cube sphere, a unity c# tutorial*. Retrieved June 26, 2024, from https://catlikecoding.com/unity/tutorials/cube-sphere/

Jasper Flick. (2024b). *Triplanar mapping*. Retrieved June 26, 2024, from https://catlikecoding.com/unity/tutorials/advanced-rendering/triplanar-mapping/

JetBrains. (2024). *Rider: The cross-platform .net ide from jetbrains*. Retrieved June 26, 2024, from https://www.jetbrains.com/rider/

jrheard's blog. (2016). *Procedural dungeon generation: Cellular automata*. Retrieved June 26, 2024, from https://blog.jrheard.com/procedural-dungeon-generation-cellular-automata

Ken Perlin. (1999). *Making noise*. Retrieved June 26, 2024, from https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/

Kenney. (2020). *Nature kit*. Retrieved June 26, 2024, from https://kenney.nl/assets/nature-kit

Martin Gardner. (1970). *The fantastic combinations of john conway's new solitaire game "life"*. Retrieved June 26, 2024, from https://web.stanford.edu/class/sts145/Library/life.pdf

Noveltech. (2023). *Generating a 2d map using the random walk algorithm*. Retrieved June 26, 2024, from https://www.noveltech.dev/procgen-random-walk

scratchapixel. (2024). *A minimal ray-tracer*. Retrieved June 26, 2024, from https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-sphere-intersection.html

Sourcetree. (2024). *Atlassian*. Retrieved June 26, 2024, from https://www.sourcetreeapp.com/

WikiDot. (2024). *Cellular automata*. Retrieved June 26, 2024, from http://pcg.wikidot.com/pcg-algorithm:cellular-automata