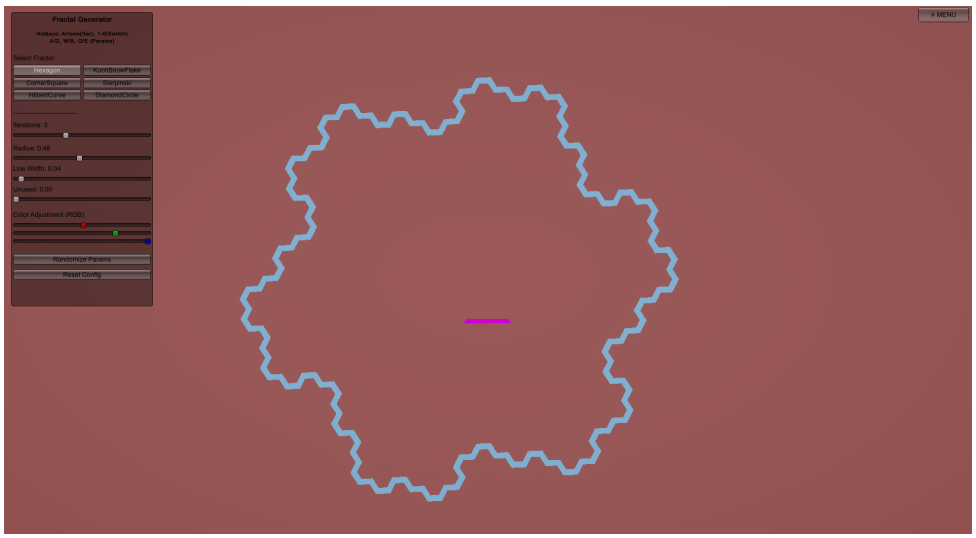
Coursework Report:
Fractal Generation System

Student Name: Yandong Wang

Date: November 28, 2025



Module: MATHEMATICS FOR GAMES AND V/AR

Goldsmiths, University of London

1 Abstract

This project presents a comprehensive Fractal Generation and Showcase System developed in Unity using C#. The application aims to demonstrate both the fundamental implementations and advanced applications of procedural fractals, ranging from deterministic geometric fractals to stochastic natural simulations. The system employs the Strategy Pattern to build a scalable architecture, managing eight distinct fractal types, including L-System based curves, recursive geometric shapes, and advanced Fractal Brownian Motion (fBm) algorithms applied to 2D terrain and 3D planetary generation. Key features include a robust user interface based on IMGUI, a JSON-based persistence system for parameter configuration, and real-time visualization of algorithmic data.

2 Introduction & Objectives

The primary objective of this coursework is to explore the mathematical foundations of fractal geometry and implement them within a real-time rendering engine. The specific objectives of this project are to:

- Implement classic deterministic fractals (e.g., Koch Snowflake, Hilbert Curve) to understand recursion and self-similarity.
- Apply stochastic fractals (fBm) to simulate natural phenomena, specifically embodied in the generation of a 2D mining map and a 3D planet.
- Develop a user-friendly tool that allows for real-time parameter manipulation via both GUI and hotkeys.
- Ensure compliance with best practices in software engineering through a modular, data-driven architecture.

3 System Architecture & Data Design

To meet the requirement of managing eight different configuration files and multiple fractal types, I avoided hard-coding individual scripts in favor of adopting a modular design.

3.1 The Strategy Pattern

The core of the "Basic Fractals" scene is built upon an inheritance model acting as a Strategy Pattern.

- **BaseFractal (Abstract Class):** Defines the contract for all fractal generators. It enforces methods like `InitFromConfig()`, `OnUpdateParameter()`, and `OnUpdateIteration()`.

- **FractalConfig (Data Class):** A serializable class acting as a Data Transfer Object (DTO). It standardizes parameters (floating-point numbers P1, P2, P3), enabling the JSON serializer to handle different fractal types uniformly.
- **FractalMasterController:** Acts as the context manager. It handles input (Keyboard/IMGUI), reads JSON files from the `StreamingAssets` folder, and injects the configuration into the current active fractal strategy.

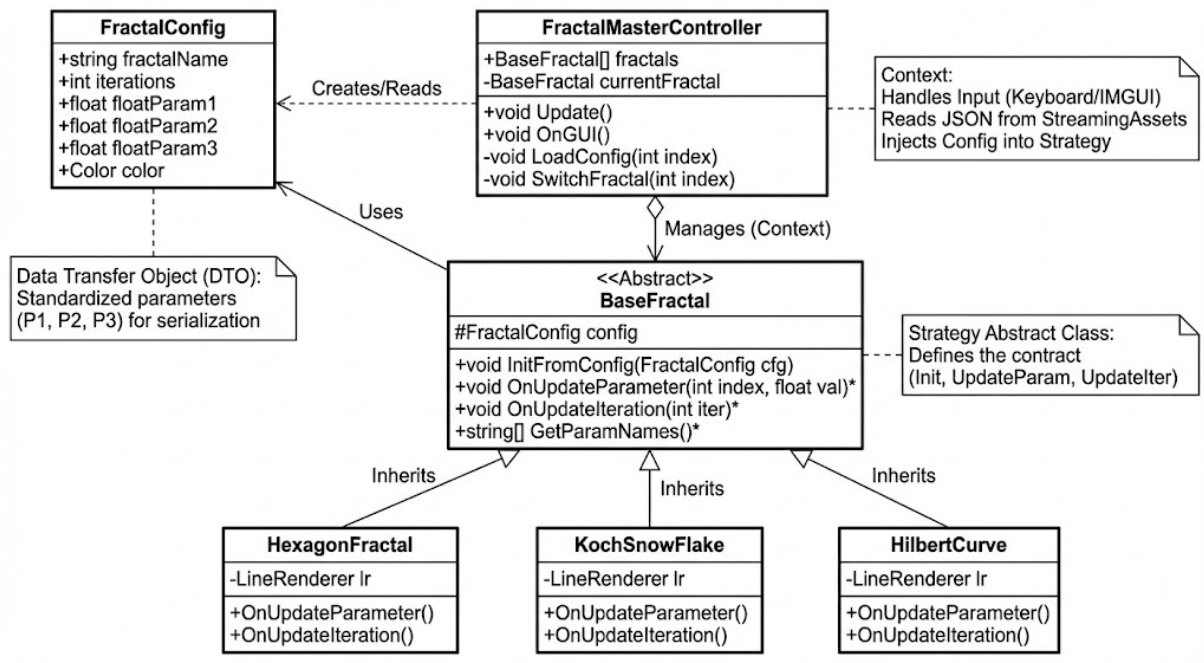


Figure 1: UML Class Diagram of the Strategy Pattern Architecture

3.2 2D and 3D Application Architecture

For the advanced scenes (Mining and Planet), the architecture evolves to handle more complex data:

- **PlanetData & MiningConfig:** Custom data structures designed to serialize complex Unity types, such as `Gradient` (via a custom `SerializedGradient` wrapper) and nested `NoiseSettings`.

```
[System.Serializable]
18 个引用
public class PlanetData
{
    public string planetName;
    public int seed;
    public float radius; //对应 resolution/size

    public NoiseSettings terrainNoise; //地形参数

    public SerializedGradient biomeGradient;//颜色参数
    public float colorSpread;

    //云层参数
    public NoiseSettings cloudNoise;
    public float cloudThreshold;
    public float cloudOpacity;
}
```

Figure 2: Data Structure for Planet Configuration

```
using UnityEngine;

[System.Serializable]
8 个引用
public class MiningConfig
{
    public string saveName = "DefaultMap";
    public float areaWidth = 20f;
    public float areaHeight = 10f;
    [Range(0.1f, 1.0f)] public float blockSize = 1.0f;
    public float noiseScale = 10f;
    public int octaves = 3;
    public float persistence = 0.5f;
    public float lacunarity = 2.0f;
    public int seed = 0;
    public Vector2 offset;
    public float[] thresholds = new float[] { 0.0f, 0.4f, 0.7f, 0.9f };
}
```

Figure 3: Serialized Gradient Implementation

- **Instance Isolation:** To prevent common issues with shared meshes in Unity Prefabs, the system implements strict initialization logic, using `GetInstanceID()` to ensure that each generated planet or cloud layer possesses a unique mesh instance

in memory.

```
string uniqueName = $"PlanetMesh_{this.GetInstanceID()}_{i}";

if (meshFilters[i].sharedMesh == null || meshFilters[i].sharedMesh.name != uniqueName)
{
    meshFilters[i].sharedMesh = new Mesh();
    meshFilters[i].sharedMesh.name = uniqueName;
}
```

Figure 4: Instance Isolation Logic in Initialization

4 Implementation of Core Algorithms

4.1 Recursive Geometric Fractals

Two custom recursive fractals were implemented to demonstrate geometric self-similarity: the **Corner Square Fractal** and the **Diamond Circle Fractal**.

The Corner Square Fractal: This algorithm recursively subdivides a square space. For every iteration, the system calculates a `childExtent` (1/3 of the current size) and generates five new iterations at the center and four corners.

- **Code Analysis:** The `DrawPattern` function uses recursion depth d. The base case ($d = 1$) triggers the drawing of a square using a `LineRenderer`.
- **Parameter Mapping:** The system maps the generic `floatParam1` to `size` and `floatParam2` to `spacing`. This allows users to intuitively visualize how changing the spacing ratio disconnects the recursive elements.

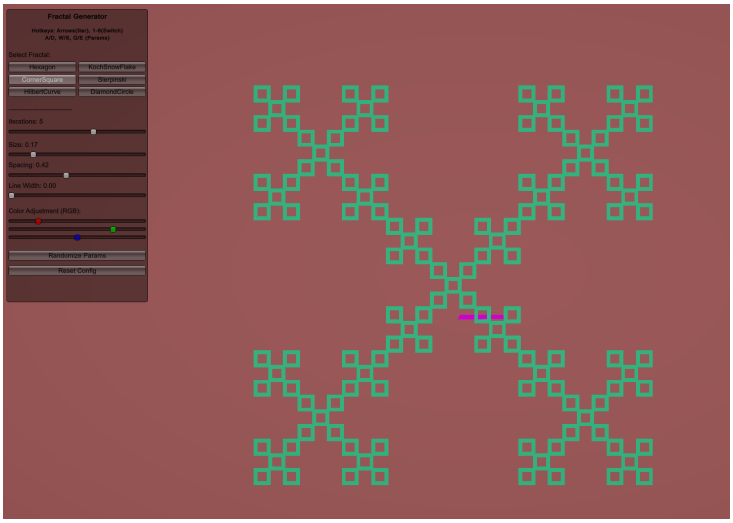


Figure 5: Corner Square Fractal: Varying Iterations

The Diamond Circle Fractal: This fractal demonstrates a node-rewrite system.

- **Algorithm:** Starting from a center point, the algorithm calculates four child positions (Top, Right, Bottom, Left). It recursively calls itself on these child nodes while drawing "bridges" (lines) between them to visualize connectivity.
- **Termination:** When recursion depth reaches zero, a circle is rendered.
- **Visual Output:** This creates a mesh structure resembling a diamond crystal lattice or a complex network graph.

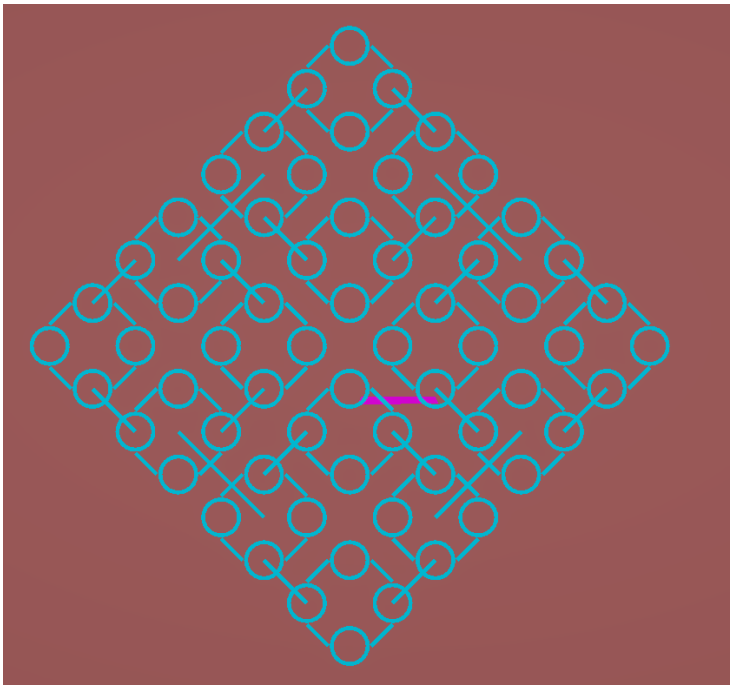


Figure 6: Diamond Circle Fractal Connectivity

4.2 L-Systems and Space Filling Curves

The project implements standard L-System parsing for generating the **Hilbert Curve** and **Gosper Island**.

- **String Rewriting:** A `StringBuilder` is used to iteratively replace characters (e.g., 'A' $\rightarrow$ "-BF+AFA+FB-") to generate drawing instructions.

```
string LSystem(int iter)
{
    StringBuilder sb = new StringBuilder("A");
    for (int i = 0; i < iter; i++)
    {
        StringBuilder n = new StringBuilder();
        foreach (char c in sb.ToString())
        {
            if (c == 'A') n.Append("-BF+AFA+FB-");
            else if (c == 'B') n.Append("+AF-BFB-FA+");
            else n.Append(c);
        }
        sb = n;
    }
    return sb.ToString();
}
```

Figure 7: Hilbert Curve L-System

- **Turtle Graphics:** The resulting string is interpreted geometrically, converting symbols into Vector3 positions for the `LineRenderer`.

```
List<Vector3> Turtle(string s, float step)
{
    List<Vector3> pts = new List<Vector3>();
    Vector3 pos = Vector3.zero;
    Vector3 dir = Quaternion.Euler(0, 0, startAngle) * Vector3.right;
    pts.Add(pos);
    foreach (char c in s)
    {
        if (c == 'F') { pos += dir * step; pts.Add(pos); }
        else if (c == '+') dir = Quaternion.Euler(0, 0, -90) * dir;
        else if (c == '-') dir = Quaternion.Euler(0, 0, 90) * dir;
    }
    return pts;
}
```

Figure 8: Gosper Island Boundary

4.3 Stochastic Fractals: Fractal Brownian Motion (fBm)

Furthermore, I implemented fBm for natural simulation.

Mathematical Basis: The system sums multiple layers (octaves) of Perlin/Simplex noise.

$$Noise(x, y) = \sum_{i=0}^{n} A^i \times P(f^i \times x, f^i \times y) \tag{1}$$

Where A is persistence (amplitude), and f is lacunarity (frequency).

Case Study 1: 2D Mining Map

- **Implementation:** A heatmap is generated where noise values are normalized.
- **Threshold Mapping:** I implemented a `MineralType` struct instead of a simple grayscale map. The noise value determines the resource type (Stone $<$ Coal $<$ Gold $<$ Diamond).

- **Visualization:** A custom IMGUI panel renders a real-time waveform graph, allowing users to intuitively understand how changing `Persistence` affects the "roughness" of the noise wave.

Figure 9: 2D Mining Map with Real-time Waveform Visualization

Case Study 2: 3D Procedural Planet

- **Cube Sphere:** To avoid texture distortion at poles, the sphere is constructed by normalizing a subdivided cube.
- **3D Noise Sampling:** Unlike the 2D map, the planet samples noise in 3D space (x, y, z). This ensures seamless terrain generation across the sphere surface.
- **Biomes & Atmosphere:** A `Gradient` evaluation maps height to color (Ocean $\rightarrow$ Land $\rightarrow$ Snow). A secondary mesh layer with transparent noise generation simulates a dynamic cloud atmosphere.

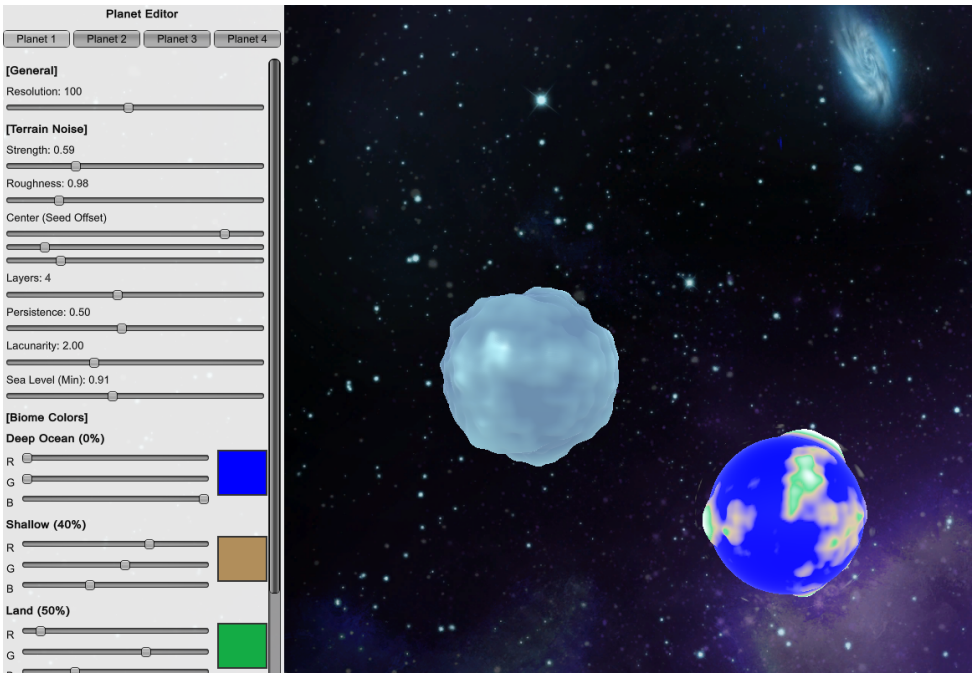


Figure 10: 3D Procedural Planet with Dynamic Atmosphere

5 User Interface & Experience (UI/UX)

The project utilizes Unity's IMGUI system to create a robust, debug-style interface that does not rely on scene objects.

- **Scene Navigation:** A persistent `GlobalNavigation` singleton allows seamless transition between the three demo scenes.
- **Parametric Control:** All floating-point parameters are normalized (0.0 - 1.0) in the GUI and remapped to physical values (e.g., line width of 0.01 - 0.15) within the fractal classes using a custom `Map()` helper function. This prevents user error and ensures stability.
- **Data Persistence:** A custom JSON serialization system allows users to save and load their creations. For the Planet scene, this includes complex serialization handling for Unity's `Gradient` class, as it does not support native serialization.

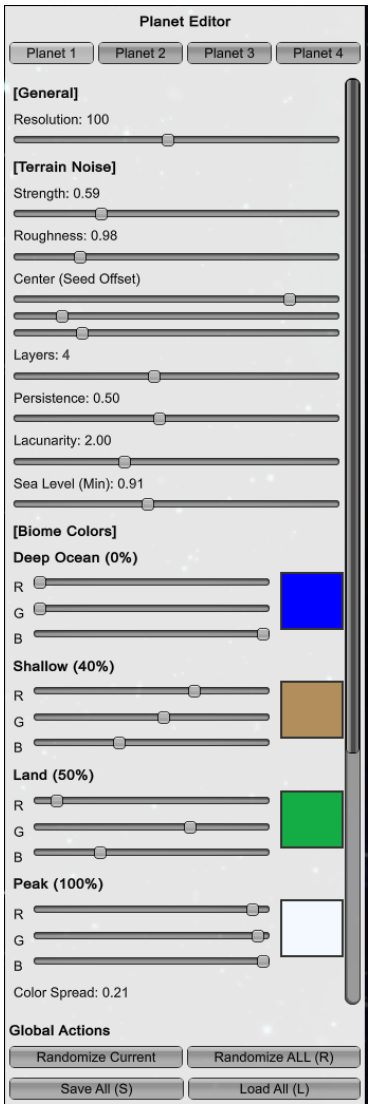


Figure 11: Custom IMGUI Controls

6 Results & Critical Evaluation

6.1 Results

The system successfully renders all 8 required fractal types.

- **Performance:** The 2D fractals use `LineRenderer`, which is efficient for low iteration counts. However, running at high iteration counts (> 7) causes the vertex count to increase exponentially, leading to minor CPU lag.
- **Visuals:** The 3D Planet generator successfully creates varied planetary types (Terran, Desert, Alien) by using a random color gradient algorithm in HSV color space.

6.2 Challenges & Solutions

1. Topological Distinction of the Gosper Island Boundary

- **Challenge:** Initially, I attempted to implement the Gosper Island using standard Gosper Curve L-System rules. However, I struggled to distinguish between the space-filling Peano-Gosper curve (which fills the interior) and the boundary contour I aimed to visualize. The transformation rules for the boundary are topologically distinct and difficult to describe purely through string replacement, as they involve complex vector rotations that do not align with standard 60° or 90° grids.

- **Solution:** I shifted my approach from string rewriting to a **recursive geometric subdivision algorithm**. By analyzing the geometry, I derived the specific mathematical constants required for the boundary: a rotation angle of $\arctan(\frac{\sqrt{3}}{5}) \approx 19.1°$ (`AngleA`) and a scaling factor of $1/\sqrt{7} \approx 0.378$ (`ScaleFactor`). I implemented a `Subdivide()` function that iterates through the vertices of the initial hexagon. Instead of replacing characters, the algorithm calculates vector mathematics using `Quaternion.Euler`. It replaces each edge with three specific vector segments (v_1, v_2, v_3) calculated via the derived angle and scale, ensuring the contour remains closed and perfectly self-similar across iterations.

2. The "Pole Pinching" Singularity in 3D Noise Mapping

- **Challenge:** Initially, I attempted to generate planetary terrain by mapping 2D Perlin Noise onto the sphere using UV coordinates (Latitude/Longitude). However, this resulted in severe texture distortion and visual artifacts at the poles (singularities), where the texture coordinate density becomes infinite.

- **Solution:** I transitioned from 2D texture mapping to **3D Cartesian Noise Sampling**. Instead of passing (u, v) coordinates to the noise function, I passed the normalized 3D vertex position (x, y, z). This approach eliminates the need for UV unwrapping and ensures uniform noise distribution across the entire spherical manifold without any seams or distortions.

3. Gaussian Bias in Fractal Brownian Motion (fBm)

- **Challenge:** In the "2D Mining Map" generator, I utilized a threshold-based system to spawn rare resources (e.g., Diamonds require noise value > 0.9). However, due to the Central Limit Theorem, summing multiple layers of noise results in a value distribution that approximates a Gaussian (Bell) curve, clustering heavily around the mean (0.5). This made extreme values (rare minerals) statistically almost impossible to generate.

- **Solution:** I implemented a **Dynamic Normalization** pass. Before rendering, the algorithm iterates through the generated noise map to identify the actual global minima and maxima. It then applies an `InverseLerp` function to stretch the values to the full $[0, 1]$ range. This ensures that the highest peak in any given seed is always mapped to 1.0, guaranteeing that rare resources appear regardless of the random seed's variance.

6.3 Future Work

- **Compute Shaders:** To handle higher iterations for the Hilbert Curve or Sierpinski Mesh, calculating vertices on the GPU would significantly improve performance.
- **LOD (Level of Detail):** The planet generator could benefit from a QuadTree-based LOD system to render higher detail only when the camera is close to the surface.

7 References

References

[1] "Gosper curve," Wikipedia. https://en.wikipedia.org/wiki/Gosper_curve (accessed Nov. 15, 2025).

[2] K. Perlin, "An Image Synthesizer," *ACM SIGGRAPH Computer Graphics*, 1985.

[3] F. K. Musgrave, *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[4] Unity Technologies, "Unity Scripting API: Mesh." https://unity.com (accessed Nov. 20, 2025).

[5] S. Lague, "Fractals and the chaos game," YouTube, 2023. [Online]. Available: https://www.youtube.com/watch?v=wbpMiKiSKm8 (accessed Nov. 20, 2025).