Asia Pacific Academy of Science Pte. Ltd.

Metaverse 2026, 7(3), m8575.
https://doi.org/10.54517/m8575

Article

A Unity-Based Plugin for Real-Time Terrain and Ecosystem Generation with Dynamic Physical Parameter Regulation

Shun Nian Luo[2], Yuh-Chung Lin[1,*], Jiahao Pan[1], Zhichen Xu[1]

[1] School of Information Science and Technology, Sanda University No. 2727 Jinhai Road, Shanghai, Pudong District, P. R. China
[2] Guangzhou Institute of Science and Technology, No. 638, Xingtai 3rd Road, Taihe, Baiyun District, Guangzhou City, P. R. China
*** Corresponding author:** Yuh-Chung Lin, Email: yuhchung@sandau.edu.cn

CITATION

Luo SN, Lin Y-C, Pan J, Xu Z. A Unity-Based Plugin for Real-Time Terrain and Ecosystem Generation with Dynamic Physical Parameter Regulation. Metaverse. 2026; 7(3): 8575. https://doi.org/10.54517/m8575

ARTICLE INFO

Received: 11 May 2026
Accepted: 15 July 2026
Available online: 8 September 2026

Abstract: Real-time terrain and ecosystem generation is essential for constructing dynamic virtual environments in the Metaverse, where scalability, interactivity, and ecological authenticity are paramount. Existing Unity-based workflows rely heavily on manual editing, resulting in high labor costs, static landscapes, and limited diversity. This paper presents a Unity plugin for real-time terrain and ecosystem generation featuring Dynamic Physical Parameter Regulation (DPPR)—a closed-loop feedback framework that adjusts erosion and vegetation in real time based on terrain gradients, with sub-second responsiveness. The system combines multi-layered Perlin noise for heightmap generation with an Ecological Zoning Algorithm (EZA) that distributes vegetation, water, and rock according to terrain slope, elevation, and hydrological conditions. Unlike traditional static terrain plugins that accept only fixed parameter inputs without environmental feedback, The DPPR framework implements bidirectional dynamic feedback logic: terrain altitude, slope, and hydrological data derived from noise reversely adjust physical parameters of erosion and vegetation competition in real time. To support Metaverse applications, the nine-grid asynchronous dynamic loading mechanism enables unbounded terrain expansion, while a unique identifier module for terrain resources binds generation parameters to user accounts, providing an architectural basis for future integration with virtual asset ownership verification frameworks in digital economies. The plugin supports applications in Metaverse virtual land creation, smart city digital twins, and cultural heritage reconstruction. Future work will explore dynamic ecological transitions and integration with user-generated content ownership frameworks for digital asset security in virtual economies.

Keywords: Terrain eneration; Perlin Noise; Dynamic Physical Parameter Regulation (DPPR); Ecological Zoning Algorithm (EZA); Chunked Loading; Multi-thread Optimization

1. Introduction

The rapid development of the metaverse and virtual reality technologies has accelerated the demand for large-scale, immersive, and interactive virtual environments. Building such environments efficiently while maintaining ecological realism has become an important research topic in both academia and industry. Metaverse virtual environments require not only vast three-dimensional content but also environmental dynamism, evolvability, and real-time user interaction. However, traditional three-dimensional content production remains heavily reliant on manual modeling and the creation of static assets, rendering it inadequate for meeting the metaverse's dual imperatives of "boundless land" and "living worlds." Achieving large-scale content generation while preserving ecological realism and dynamic environmental responsiveness remains a major challenge, as highlighted in recent systematic reviews on procedural terrain generation [1] and studies on metaverse

virtual land creation [2].

In terms of terrain and ecological environment generation, existing approaches can be broadly categorized into manual editing and procedural generation. Manual editing relies on artists painting heightmaps block by block and placing vegetation and textures within engines such as Unity. While this approach ensures fine-grained detail, it suffers from low efficiency and prohibitive labor costs when applied to large-scale open scenarios, rendering it ill-equipped to meet the rapid iteration demands for massive content in the metaverse. Procedural generation techniques, although capable of automatically producing terrain through noise functions and stochastic algorithms, are largely confined to randomization at the static geometric level. Current solutions lack systematic modeling of the intrinsic regularities governing ecological environments and fail to support real-time interaction with user behaviors or physical parameters [3]. For instance, Liu Danjun et al. proposed "Perlin noise-based flowers simulation method", which achieved three-dimensional modeling of flowers using Perlin noise [4]; Padhiyar K. proposed "Encryption Algorithm Based on Randomly Generated Matrix", exploring the application of randomly generated matrices [5]. While these studies provide foundational tools for procedural generation, they do not integrate ecological rules, physical feedback, and real-time interaction into a unified framework. Recent work on dynamic physical parameter control suggests that incorporating real-time environmental feedback can significantly enhance the adaptability of virtual terrains [6].

To address the aforementioned challenges, this paper proposes a real-time terrain and ecological environment generation method tailored to the demands of large-scale, highly dynamic, and deeply immersive metaverse scenarios. The proposed DPPR framework operates as a closed-loop feedback system with two core algorithmic mechanisms---bidirectional physical coupling and real-time ecological zone refresh---supported by an asynchronous job scheduling layer for performance optimization. First, physical factor closed-loop linkage: terrain parameters (altitude, slope, hydrological accumulation) computed from multi-octave noise are fed back into the generation pipeline to dynamically adjust erosion intensity and vegetation competition coefficients, enabling causal interactions between terrain morphology and ecological distribution. Second, real-time ecological zoning dynamic update: as the camera or player moves, the nine-grid chunk loading mechanism triggers on-the-fly recalculation of ecological zone boundaries based on updated physical parameters, ensuring that the ecosystem evolves continuously rather than remaining static. Third, asynchronous multi-layer parameter scheduling: computationally intensive physical simulation tasks are dispatched to background threads via the Job System and Burst Compiler, while the main thread handles rendering and user interaction, achieving responsive feedback without frame-rate degradation. Unlike conventional procedural terrain plugins, the proposed framework relies on closed-loop bidirectional feedback rather than static parameter inputs.

In the domain of Unity engine applications, Ding Y and Song Y introduced " Vision-Degree-Driven Loading Strategy for Real-Time Large-Scale Scene Rendering in Unity 3D" [7]; Yang Y et al. proposed " Real-time light calculation and optimization model for large-scale scene rendering ", which enhances rendering efficiency through improved algorithms and hardware acceleration [8]; and Y. Yang et al. presented " Real-time light calculation and optimization model for large-scale scene rendering ", offering theoretical support for object transformations within virtual environments [9]. These works contribute technical support for ensuring performance in large-scale scenes, yet their core focus remains on the management of

static or semi-static content, leaving unaddressed the demand for terrain to function as a "living system" characterized by dynamic evolution.

To validate the proposed method, we encapsulated it as an extensible Unity plugin, with parallel optimization via the Job System and Burst Compiler to guarantee real-time performance in large-scale scenes.

The plugin is architected for Metaverse deployment. Its nine-grid asynchronous loading mechanism supports persistent virtual worlds through continuous terrain expansion as users explore, meeting the infinite land demand of Metaverse platforms. Each terrain chunk is assigned a unique identifier that binds generation parameters (seed, noise configuration, ecological zoning thresholds) to user account information, laying an architectural foundation for future extension to digital asset ownership verification and virtual economy transaction systems. This design also accommodates multi-user interactive environments with concurrent viewpoint loading and real-time ecological evolution.

2. Related works

This chapter presents an exposition of the key technologies and theoretical foundations underpinning the present research. The discussion encompasses an overview of the plugin architecture, the Perlin noise algorithm, the nine-grid chunk-based terrain loading mechanism, and the application of multithreading and coroutine techniques in real-time generation. These technologies provide the theoretical foundation for the system design presented in the following sections.

A plugin is an extensible software component that enhances the functionality of a host application without modifying its source code. The real-time terrain and ecological generation plugin developed in this system—predicated on dynamic physical parameter adjustment within Unity—encapsulates the core algorithms for procedural terrain generation. These algorithms include multi-octave superposition based on Perlin noise, EZA, dynamic parameter modulation, chunk-based loading, and multithreaded optimization. Through the Inspector interface, users can adjust noise parameters, ecological configurations, and real-time physical parameters, thereby enabling efficient, parametrically controlled terrain generation.

Perlin Noise, a continuous and smooth pseudo-random function introduced by Ken Perlin in 1985, generates noise values devoid of conspicuous repetitive patterns by assigning random gradient vectors at the vertices of a regular lattice and performing smooth interpolation [10]. The core procedural steps encompass grid partitioning and gradient vector assignment, distance vector computation, gradient dot product calculation, smooth interpolation, and the generation of complex terrain structures through multi-octave superposition.

In the direct application of dot product results, to preclude abrupt boundary discontinuities, Perlin noise employs cubic Hermite interpolation to aggregate the contributions of individual lattice nodes, thereby yielding noise values characterized by smooth transitional effects. The smooth interpolation function is specified as follows:

$$f(t) = 6t^5-15t^4+10t^3 \quad (1)$$

The derivatives of this function at both t=0 and t=1 are identically zero, thereby ensuring smoothness at the endpoints and guaranteeing the continuity of the resulting noise values. Variants of Perlin noise, as discussed by Kopel and Maciejewski [11], have been developed to further enhance terrain realism and reduce visual artifacts through modified interpolation and gradient schemes.

In the context of multi-octave superposition, low-frequency noise governs the

large-scale topographic contours—such as mountain ranges and hills—while high-frequency noise contributes finer terrain details—including rock formations and surface texture. The amplitude and frequency of each successive octave are modulated according to a prescribed ratio, and the final composite yields a natural and intricate heightmap. The formulation is typically expressed as:

$$P(x,y)=\sum_{i=0}^{n} a_i * \mathrm{Noise}\left(2^i x, 2^i y\right) \tag{2}$$

where n denotes the number of octaves, and a_i represents the amplitude factor corresponding to each respective octave.

Within the domain of terrain generation, this algorithm is employed to construct naturalistic heightmaps and to govern ecological distributions—such as vegetation zones and riverine trajectories. By virtue of its low computational overhead and seed-based controllability, it is particularly well-suited for real-time rendering and the realization of boundless map scenarios.

The nine-grid map and chunk-based loading mechanism constitute a critical performance optimization strategy within large-scale virtual environments. The core principle entails partitioning the virtual world into discrete tiles or chunks, and loading only those regions that are visible or adjacent to the player's viewport. This approach circumvents full-scene loading, thereby reducing memory footprint and ensuring seamless scene updates. In the context of procedural terrain generation, chunk-based loading facilitates the judicious management of terrain data by distributing generation tasks across multiple chunks for parallel execution. This not only enhances computational performance but also supports asynchronous generation and dynamic updating of terrain content. Fu H. et al. [12] demonstrated that integrating level-of-detail (LOD) techniques within chunk-based frameworks can substantially improve rendering efficiency for large-scale procedural terrains.

Multithreading and coroutine techniques provide an efficient paradigm for the real-time generation of complex virtual scenes. Multithreading distributes computationally intensive tasks—such as noise data generation and heightmap computation—across multiple processing units for parallel execution, thereby reducing overall generation latency. Coroutines, as a lightweight concurrency mechanism, enable non-blocking waiting within a single thread, allowing heavy computations to be amortized across multiple frames. In this study, the two approaches are synergistically combined: background multithreading is employed to generate noise maps and ecological distribution data, while coroutines on the main thread monitor the generation status. Upon completion, the main thread is promptly invoked to execute terrain construction and rendering. Guo et al. [13] have shown that such optimization strategies significantly improve the responsiveness of Unity based terrain generation systems.

The DPPR framework proposed in this study differs from traditional procedural terrain generation works in three essential aspects. First, whereas conventional approaches (e.g., Liu [4], Yang [8]) treat noise functions and stochastic algorithms as open-loop parameter inputs, DPPR implements closed-loop bidirectional feedback where generated terrain properties actively modulate subsequent physical parameters. Second, while chunk-based loading and multithreading optimizations have been individually explored [7], their integration with dynamic ecological zoning and physical parameter real-time refresh is novel to our knowledge. Third, existing Unity terrain plugins (such as Gaia, MapMagic, and Terrain Composer) predominantly offer static parameter editing modes without environmental feedback; the DPPR framework

explicitly addresses this gap by coupling terrain generation with evolving hydrological and ecological processes. This comparative analysis substantiates the innovative contribution of our approach beyond simple assembly of existing techniques.

In summary, the aforementioned technologies provide critical underpinnings for the design and implementation of the real-time terrain and ecological generation plugin based on dynamic physical parameter regulation in Unity.

3. Materials and methods

This section describes the design and implementation of the proposed Unity-based terrain generation plugin. A detailed analysis will be conducted across multiple dimensions—encompassing functional requirements, performance requirements, and user requirements—thereby delineating explicit objectives and furnishing guiding directives for the subsequent design and implementation of the system.

3.1. Overall System Architecture

The system adopts a modular design paradigm, wherein the plugin is partitioned into several core functional modules. Inter-module communication is facilitated through well-defined interfaces, enabling both data transmission and functional orchestration. The overall architecture comprises the following components:

• Configuration Module: Users set generation parameters (seed, noise, ecological zone thresholds, probabilities, chunk loading) via the Unity Inspector. The module packages them into a unified MapConfig object as the input for subsequent processes.

• Data Generation Module: The module uses Perlin noise and multi-octave superposition to build a heightmap. Then, based on the heightmap and user parameters, applies an ecological region partitioning algorithm to generate distribution data for textures, vegetation, trees, rocks, and buildings. Stores results in a buffer for downstream use.

• Terrain Construction Module: This module instantiates Unity Terrain objects based on the generated data, applying the heightmap, textures, vegetation, trees, and other ecological elements to the associated TerrainData asset. A templated instantiation approach is employed to facilitate natural terrain transitions and the ecologically coherent distribution of environmental elements.

• Map Chunking Module: Implements nine-grid chunk loading. Dynamically loads, activates, or unloads chunks based on player/camera view. The ChunkController manages chunk generation, enabling, disabling, and destruction for efficient resource use.

• Performance Optimization and Real-Time Update Module: This module leverages multithreading and coroutine techniques to asynchronously process data generation tasks. The main thread continuously monitors the generation status, and upon completion, immediately triggers terrain construction and ecological element updates. Integrated with chunk-based loading, Level of Detail (LOD) techniques, and memory optimization strategies, this module ensures smooth and stable system operation across large-scale terrains.

The DPPR closed-loop regulation logic operates as follows:

a) The Data Generation Module produces an initial heightmap and slope/hydrological feature maps using multi-octave Perlin noise;

b) These feature maps are fed into the Performance Optimization and Real-Time Update Module, which computes erosion intensity and vegetation competition factors based on terrain gradients and elevation;

c) The computed physical parameters are applied to modify ecological zone thresholds in the Configuration Module in real time.

d) The modified thresholds trigger regeneration of ecological distribution data, which are then passed back to the Terrain Construction Module. This regeneration is implemented as an incremental update: only the ecological zone boundaries within the currently active 3×3 chunk grid are recalculated, rather than the entire global map.

This feedback loop executes continuously as the camera moves, enabling the ecosystem to evolve dynamically rather than remaining static after initial generation.

3.2. Module Design

3.2.1. Parameter Configuration Module Design

The Parameter Configuration Module serves as the input gateway for the entire system. Its primary function is to collect a diverse array of user-specified parameters and encapsulate them for downstream consumption. Several critical considerations inform the design of this module.

The user interface accessibility is manifested through the implementation of a custom Inspector interface and Editor extensions, thereby enabling a "one-click generation" paradigm for map creation. Users may conveniently configure a comprehensive range of parameters, including the random seed, noise parameters, ecological zone delineations, and loading range specifications.

Data Encapsulation: All configuration parameters are consolidated and encapsulated within a unified MapConfig object (**Figure 1**). This object encompasses noise parameters, zone partitioning specifications, ecological element probability settings, and texture configurations, thereby ensuring that subsequent data generation modules may invoke these parameters in a consistent and uniform manner.

```
[Serializable]
// 5 usages  1 exposing API
public class MapConfig
{
    [Tooltip("Seed")]
    public int seed;  Serializable
    [Tooltip("Terrain data template")]
    public TerrainData template;  Serializable
    public GameObject terrainPrefab;  Serializable
    public float scale;  Serializable
    [Tooltip("Noise layers")]
    public int layerCount;  Serializable
    [Tooltip("Frequency")]
    public float lacunarity;  Serializable
    public float layerLacunarity;  Serializable
    [Tooltip("layerAmplitude")]
    public float layerAmplitude;  Serializable
    [Tooltip("AnimationCurve")]
    public AnimationCurve curve;  Serializable

    [Tooltip("Terrain detail scaling")]
    public float detailScale;  Serializable
    [Tooltip("Generation interval step length of trees")]
    public int treeSpawnStep;  Serializable
    public int buildingSpawnStep;  Serializable
    [Tooltip("Destruction time of terrain")]
    public float terrainDestroyTime;  Serializable
    [Tooltip("Regional configuration list")]
    public List<RegionConfig> regions;  Serializable

    #region Shared parameters - no serialization required
    [HideInInspector] public int heightmapResolution;  Serializable
    [HideInInspector] public int detailWidth;  Serializable
    #endregion
}
```

Figure 1. Data Generation Module Design.

3.2.2. Data Generation Module Design

For large-scale maps, a chunk-based loading scheme is adopted. The key design considerations for this module are as follows:

The noise generation process proceeds as follows. First, a two-dimensional noise map is generated using the Perlin noise algorithm. Subsequently, multi-octave superposition is applied to this base map, yielding a complex heightmap.

The partitioning of ecological data proceeds as follows. Based on the generated heightmap and user-defined ecological zone parameters, a region detection algorithm performs classification for each map point. This classification determines the corresponding ecological elements at that location, such as vegetation, trees, rocks, or structures.

Data Synchronization and Asynchronous Processing: Through the combined use of multithreading and coroutines, the data generation process is partitioned into two distinct phases: background computation and foreground status detection. This design ensures that the generation process executes efficiently and does not induce blocking on the main thread.

3.2.3. Design of the Terrain Construction Module

This module is responsible for applying the generated data to Unity Terrain objects. Its primary functions are detailed as follows:

Application of the Heightmap: The computed height data are first transferred to the TerrainData asset. Subsequently, the terrain smoothness is adjusted through techniques such as curve mapping.

Application of Textures and Ecological Elements: Based on the generated zone partitioning data, the corresponding ecological elements are assigned to the TerrainData asset. These elements include textures, vegetation, trees, rocks, and structures.

Terrain Instantiation Process: As shown in **Figure 2**, terrain instantiation is performed by utilizing preconfigured Terrain templates and Prefabs, thereby ensuring that the generated Terrain objects conform to the associated parameter specifications and are accurately positioned at their designated locations within the scene.

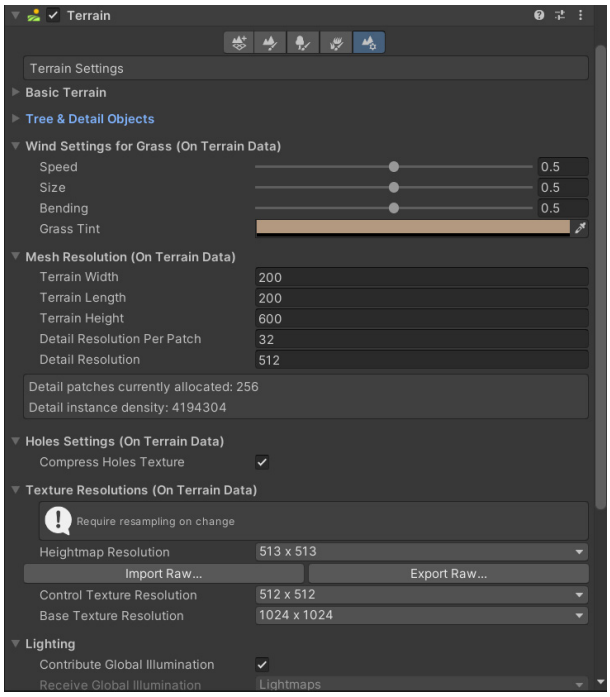

Figure 2. Terrain Instantiation Process.

3.2.4. Map Chunking and Management Module Design

To accommodate large-scale terrains, a chunk-based loading scheme is adopted. The design emphases of this module encompass the following:

View frustum calculation enables real-time acquisition of the player's position or the camera's location, based on which the range of map chunks to be loaded within the current view is determined.

Regarding chunk generation and destruction, for new map chunks falling within the view range, dynamic creation is carried out using the data generation and terrain construction modules. For chunks that have moved out of the view, they are destroyed and their memory is released, while unused controllers are returned to the object pool for future reuse.

For nine-grid management, a nine-grid loading mechanism is employed to ensure that the surrounding area remains constantly active, thereby optimizing the map loading experience during player movement.

3.3. Key Algorithm Design

3.3.1. Noise Algorithm and Multi-Octave Superposition

The system relies on the Perlin algorithm to generate a foundational noise map and employs multi-octave superposition techniques, as shown in **Figure 3**, to construct complex terrain. The design process entails the modulation of parameters—including scale, octave count, amplitude, and frequency—to generate terrain features with distinct geomorphological characteristics across different regions.

```
float amplitude = 1;
// Determine the number of layers data
for (int i = 0; i < layerCount; i++)
{
    layerOffsets[i] = new Vector2(Random.Next(0, 100000), Random.Next(0, 100000));
    maxPossibleNoiseValue += amplitude;
    amplitude *= layerAmplitude;
    //The amplitude decreases with the increase in the number of layers
    // , which is used to simulate the detailed changes in the noise map.
}

for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {
        float noiseHeight = 0;

        int indexX = x + offset.x;
        int indexY = y + offset.y;
        // Dealing with seam issues: Simply move the beginning to the end of the previous line
        if (x == 0) indexX -= 1;
        if (y == 0) indexY -= 1;

        float currLayerLacunarity = 1;
        float currLayerAmplitude = 1;
        for (int i = 0; i < layerCount; i++)
        {
            float layerX = ((indexX * lacunarity * currLayerLacunarity) + layerOffsets[i].x) * scale;
            float layerY = ((indexY * lacunarity * currLayerLacunarity) + layerOffsets[i].y) * scale;
            noiseHeight += (Mathf.PerlinNoise(layerX, layerY) * 2 - 1) * currLayerAmplitude;
            currLayerLacunarity *= layerLacunarity;
            currLayerAmplitude *= layerAmplitude;
        }

        noiseHeight = (noiseHeight + 1) / maxPossibleNoiseValue; // constrained between 0 and 1
        noiseMap[x, y] = noiseHeight;
    }
}
```

Figure 3. Multi-level superposition of Perlin algorithm.

3.3.2. Ecological Zone Partition Algorithm with Physical Models and Complexity Analysis

The system performs ecological classification for each point based on elevation

data, slope, and hydrological accumulation. Three physical mechanisms are mathematically modeled and integrated into the zoning logic:

Hydrological Accumulation Model: For each terrain point, the upstream drainage area is computed based on terrain gradient. The water accumulation value W(x,y) is defined as:

$$W(x,y) = \sum_{i \in \text{upstream}(x,y)} \Delta h_i \bullet \frac{1}{1 + e^{k(s_i - s_0)}} \tag{3}$$

where Δh_i is the elevation drop from upstream point i to the current point, s_i is the local slope, s_0 is the reference slope threshold, and k is the steepness coefficient. This sigmoid-weighted accumulation model ensures that flat regions receive higher water retention values while steep slopes yield rapid drainage.

Terrain Erosion Iterative Attenuation Model: The erosion intensity E(x,y) at each point is controlled by altitude and slope:

$$E(x,y) = E_0 \bullet \left(1 + \alpha \bullet \frac{h(x,y)}{h_{\max}}\right) \bullet \left(1 + \beta \bullet \frac{s(x,y)}{s_{\max}}\right) \bullet e^{-\lambda t} \tag{4}$$

where E_0 is the base erosion factor, α and β are weighting coefficients, λ is the decay constant, and t represents the simulation time step.

Vegetation Competitive Growth Probability Model: The probability of vegetation establishment is constrained by water and elevation suitability:

$$P_{\text{veg}}(x,y) = P_{\text{base}} \bullet \sigma\left(\frac{W(x,y) - W_{\min}}{W_{\text{opt}} - W_{\min}}\right) \bullet \left[1 - \rho \bullet \left|\frac{h(x,y) - h_{\text{opt}}}{h_{\max} - h_{\min}}\right|\right] \tag{5}$$

where $\sigma(\bullet)$ is a logistic function, and h_{opt} is the optimal elevation range for the target vegetation type.

Complete EZA Algorithm Workflow: The detailed procedural logic integrating the above models is specified in Algorithm 1(shown an **Table 1**):

Table 1. Ecological Zoning Algorithm.

Algorithm 1. Ecological Zoning Algorithm (EZA)

Input: heightmap H[x][y], slope_map S[x][y], water_accum W[x][y],
zone_params Z (thresholds, probabilities, element lists)

Output: zone_map ZM[x][y], element_map EM[x][y]

For each chunk C in active_chunks:
For each point (x, y) in C:
h ← H[x][y]; s ← S[x][y]; w ← W[x][y]
// Physical parameter calculation
erosion_intensity = E_0 * (1 + α*h/h_max) * (1 + β*s/s_max) * exp(-λt)
veg_prob = P_base * σ((w - W_min)/(W_opt - W_min)) * [1 - ρ*|h - h_opt|/Δh]
// Zone classification based on altitude and slope
If h > Z.alpine_threshold and s > Z.steep_slope_threshold:
zone ← ALPINE_ROCK
Else If h > Z.forest_lower and h ≤ Z.alpine_threshold:

Continuation Table:

Algorithm 1. Ecological Zoning Algorithm (EZA)
11: zone ← FOREST
12: Else If h ≤ Z.forest_lower and w > Z.water_threshold:
13: zone ← WATER_BODY
14: Else If h ≤ Z.forest_lower:
15: zone ← PLAIN
16: // Transition zone smoothing
17: If boundary_detected(zone, neighbors):
18: zone ← assign_transition_zone(h, s, w, Z.transition_params)
19: ZM[x][y] ← zone
20: // Element assignment (affected by calculated veg_prob)
21: EM[x][y] ← sample_elements(zone, Z.probability[zone], veg_prob, seed)
22: Return ZM, EM

Computational Complexity Analysis: For a single chunk of resolution R × R, time and space complexity are both $O(R^2)$. For global terrain with N chunks, total time is $O(N \cdot R^2)$. To reduce large-scene overhead, two optimizations are adopted: (1) block parallelization via the Unity Job System, distributing independent chunks across CPU cores; (2) threshold pre-judgment—evaluating altitude first to allow early termination for extreme zones, pruning unnecessary slope/hydrology computations. These reduce average-case time to approximately $O(N \cdot R^2 / K)$, where K is the number of worker threads. Valencia et al. [14] proposed effective ecological zoning algorithms for open-world games, which inform our design.

3.3.3. Map Chunking and Loading Algorithm

Regarding nine-grid map management, a set of view frustum calculation and chunk loading algorithms is designed, enabling efficient operation. This algorithm calculates the player's current position and view range in real time, relying on such computation to determine which map chunks need to be loaded or unloaded. The generation and reclamation of map chunks are managed through an object pool mechanism which shown as **Table 2**, thereby ensuring that the system maintains good performance in large-scale scenes.

Table 2. Detailed Nine-Grid Loading Logic.

Detailed Nine-Grid Loading Logic: The algorithm's core workflow
UpdateChunks(playerPos):
1. Compute current_chunk_coord = floor(playerPos / chunkSize)
2. For dx in [-1, 0, 1]:
For dy in [-1, 0, 1]:
target_coord = current_chunk_coord + (dx, dy)

Continuation Table:

Detailed Nine-Grid Loading Logic: The algorithm's core workflow
If target_coord not in active_chunks:
RequestChunkAsync(target_coord) // via coroutine + Job
3. For each chunk in active_chunks:
If distance(chunk.coord, current_chunk_coord) > 2:
UnloadChunk(chunk)
4. Object pool recycles unloaded chunk controllers for reuse.

This algorithm calculates the player's current position and view range in real time to determine which map chunks need to be loaded or unloaded. The generation and reclamation of map chunks are managed through an object pool mechanism, ensuring good performance in large-scale scenes. The use of asynchronous loading—via coroutines on the main thread combined with Job System tasks on background threads—prevents frame-rate spikes during chunk transitions.

3.4. System Optimization

3.4.1. Optimization of the Noise Generation Algorithm

To enhance the real-time performance and efficiency of the stochastic ecological terrain generation plugin in large-scale scenarios, this study introduces optimizations by integrating the Unity Job System and Burst compiler into the conventional Perlin noise-based generation framework. Traditional approaches typically rely on single-threaded computation or Thread-based multithreading for noise map generation, which are prone to inducing main thread blocking and performance bottlenecks. This limitation becomes particularly pronounced when generating high-resolution maps or employing multi-octave noise superposition, where the computational burden is substantial and generation latency is protracted.

As shown in **Figure 4**, this study leverages the Unity Job System and Burst compiler to parallelize noise computation and optimize critical algorithms, improving multi-core CPU utilization. Noise map generation time is reduced by 40–50%, main thread load is significantly decreased, and stable 60+ FPS is maintained in large-scale terrain rendering. Using NativeArray for intermediate data eliminates GC overhead, enhancing memory efficiency. Machado et al. [15] reported similar optimization outcomes using the Job System for procedural generation tasks, confirming the effectiveness of this approach.

Figure 4. Parallel Computing Optimization.

The optimization efficacy is further validated through ablation experiments (detailed in Section 4.2.5), where the Job System + Burst Compiler combination is independently enabled/disabled to quantify its marginal contribution to overall performance improvement.

3.4.2. Hierarchical Job Scheduling Architecture

To achieve efficient task scheduling and resource management, a hierarchical Job architecture is designed as shown in **Table 3**. The architecture is as follows:

Table 3. Hierarchical Job Scheduling Architecture.

Job Hierarchy Level	Function
Level 1 (CPU-Intensive)	Noise generation jobs, responsible for parallel computation of the heightmap.
Level 2 (Data-Dependent)	Ecological distribution jobs, which depend on the completed heightmap to generate data for vegetation, water bodies, and other elements.
Level 3 (GPU-Accelerated)	Rendering of fine terrain details via Compute Shaders, executing asynchronously with respect to CPU tasks.

Task dependency chains are managed through JobHandle. CombineDependencies() to ensure the correct sequence of computational operations.

4. Results and discussion

This chapter delineates the concrete implementation process of the real-time terrain and ecological generation plugin predicated on dynamic physical parameter adjustment within Unity. The exposition encompasses the establishment of the development environment, the detailed realization of individual modules, system integration, parameter specification, and the overall debugging workflow. This chapter provides a comprehensive account of the technical methodologies and critical procedural steps employed during system implementation, thereby furnishing a foundational basis for the subsequent testing and discussion.

4.1. System Implementation

4.1.1. Experimental Environment Configuration

All experimental tests were conducted under the following standardized environment in **Table 4**:

Table 4. Experimental Environment Parameters.

Category	Specification
CPU	Intel Core i9-12900K @ 3.2GHz (16 cores)
GPU	NVIDIA GeForce RTX 4080 (16GB VRAM)
Memory	32GB DDR5 4800MHz
Operating System	Windows 11 Pro (22H2)
Unity Version	2022.3.15f1 LTS
Burst Compiler	1.8.8
Job System	Native (Unity.Entities 1.0.0)

Continuation Table:

Category	Specification
Terrain Resolution	1024 × 1024 per chunk
Chunk Size	250m × 250m
Noise Octaves	6 (default)
Test Scene Area	4 km² (unless otherwise specified)
Profiling Tool	Unity Profiler + Memory Profiler

To ensure test fairness and full reproducibility, all comparative experiments (Section 4.2.2) were conducted under a strictly unified controlled variable protocol. All plugins ran on the same workstation (Table 4) with Unity 2022.3.15f1 LTS, using identical QualitySettings (Ultra, no dynamic resolution). Each plugin generated a 4 km² terrain with 1024×1024 resolution per chunk, 6 noise octaves, and the same elevation range (0–500 m). Vegetation, rock, and building spawn probabilities were set to each plugin's recommended "open world" presets. Each test was repeated five times; the reported generation time is the average. Frame rate and peak memory were recorded via Unity Profiler during a standardized 60 second free roam path covering all ecological zones. LOD distances, anti aliasing, and shadow settings were locked across all tests.

4.1.2 Parameter Configuration and User Interface Module

To facilitate more convenient configuration of terrain generation parameters by users, a custom Inspector interface extension was developed as a foundational step. By inheriting from the UnityEditor. Editor class, an intuitive parameter display and one-click generation functionality were realized.

This module enables users to adjust a comprehensive range of parameters directly within the Inspector interface. These parameters are categorized into two principal classes: noise parameters, which encompass the random seed, lacunarity, scale, octave count, and amplitude; and ecological zone settings, which govern the distribution probabilities and detailed configurations of textures, vegetation, trees, structures, and rock formations. All parameters are collectively encapsulated within the MapConfig object.

In the concrete implementation process, Unity's Editor scripting capabilities and GUI controls were employed to ensure that the configuration workflow remains concise and perspicuous, while also providing support for real-time preview and debugging.

4.1.3. Data Generation and Ecological Partitioning Module

The noise generation adopts a Perlin noise-based algorithm with multi-layer superposition to generate a heightmap (**Figure 5**). The module computes a two-dimensional noise map from the user-defined seed and noise parameters, then maps the values via an animation curve.

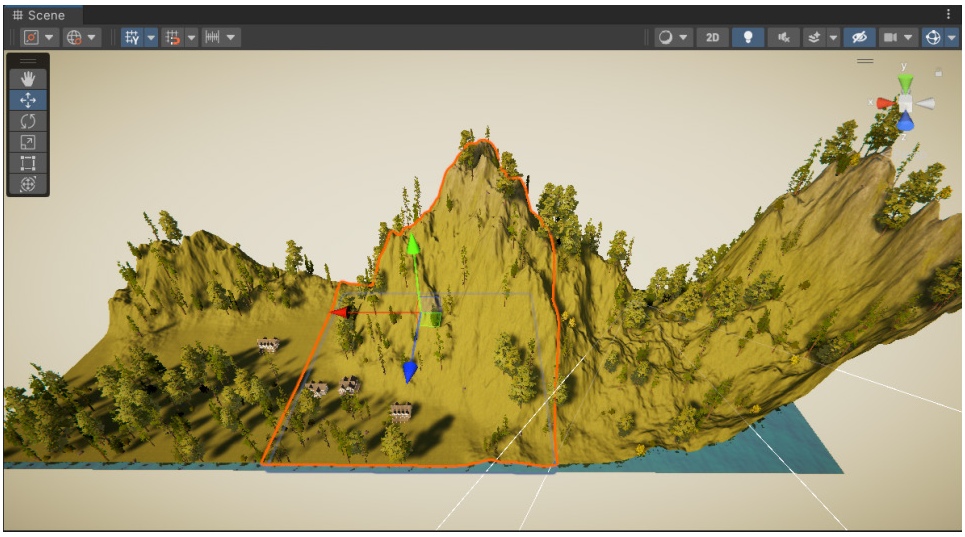

Figure 5. Single-chunk terrain generation.

Ecological zone division is carried out based on the heightmap. Each terrain point is ecologically classified according to predefined regional parameters to determine the zone to which it belongs. The configuration of different zones determines the generation probability and distribution rules of ecological elements within that zone, such as textures, vegetation, trees, buildings, and rocks. This part is accomplished using data scanning and region detection algorithms, thereby ensuring smooth transitions at zone boundaries. Moreover, the generated data can be directly applied to subsequent terrain construction, including multi-chunk terrain stitching (see **Figure 6**).

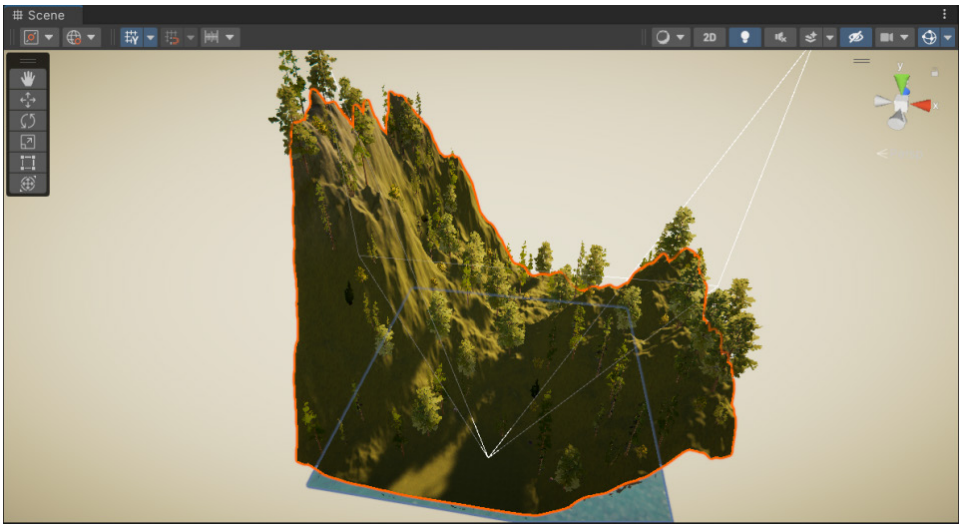

Figure 6. Multi-chunk terrain stitching.

4.1.4. Terrain Construction Module

Upon the completion of all data generation procedures, the terrain construction module dynamically instantiates Unity Terrain objects based on the generated heightmap, texture distribution data, and the configuration settings of ecological elements.

The Terrain instantiation process proceeds as follows: first, a preconfigured Terrain template is instantiated; subsequently, the generated heightmap is applied to the appropriate region; finally, the height values and texture information are assigned

to the associated TerrainData asset.

The application of ecological elements entails binding ecological data—including vegetation distribution, tree placement, and architectural instance information—to the TerrainData asset. Subsequently, Unity's built-in methods are employed to arrange these ecological components. In this process, the Terrain system's tree and detail layer functionalities are leveraged to effectively integrate the generated ecological data into the Terrain object, thereby constructing a complete terrain scene with generated ecological elements.

4.1.5. Map Chunking Management and Loading Module

To address the performance bottlenecks encountered during the generation of large-scale terrains, the system adopts a nine-grid chunk-based loading mechanism. The map chunking scheme dynamically loads and unloads chunks based on player or camera position. When a chunk enters the view frustum, the system asynchronously generates its data and constructs the terrain in real time; chunks that exit are deactivated and unloaded after a short delay. An object pool reuses controller resources, minimizing creation/destruction overhead and reducing garbage collection pressure. Coroutines and multithreading continuously monitor player movement to keep chunk states synchronized with the view, while resource-intensive operations are decoupled from the main thread to ensure stable frame rates during transitions, as shown in **Figure 7**. Coutinho [16] described similar memory optimization strategies for infinite procedural worlds in Unity, validating our design choices.

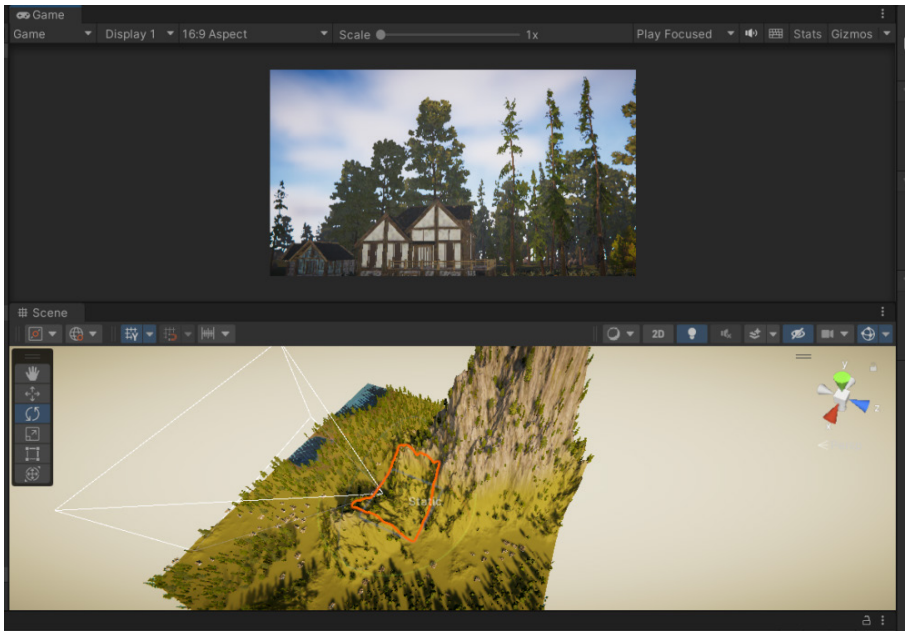

Figure 7. Nine-Grid Map Loading.

To quantitatively characterize chunk loading performance, the system was tested with viewDistance parameters of 500m, 1000m, and 1500m (see Section 4.2.5 for detailed results). The object pool initial capacity was set to 25 chunk controllers with a maximum pool size of 50; pre-test profiling showed this configuration balances memory footprint (approx. 85MB peak) against allocation overhead.

4.2. System Configuration and Performance Analysis

4.2.1. Dynamic Physical Parameter Adjustment Demonstration

The plugin provides dynamic physical parameter adjustment via the MapConfig

object, with immediate influence on terrain generation. For instance, modifying the noise scale and amplitude enables the resulting heightmap to more closely approximate the variations characteristic of actual physical environments. By adjusting physical parameters, users can observe corresponding alterations in the distribution of ecological elements during the generation process, thereby facilitating the dynamic calibration and optimization of the virtual environment, as demonstrated in **Figure 8**.

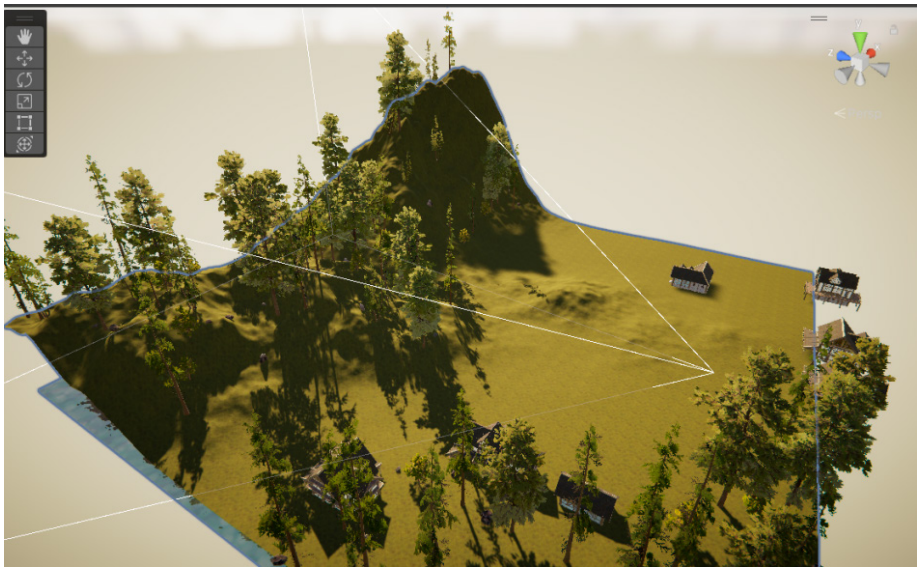

Figure 8. Demonstration of Single Chunk Terrain Generation.

The incremental recomputation cost for a 250m×250m chunk at 1024×1024 resolution averages 8–12 ms on the test hardware (Table 4), well below the 16.7 ms frame budget, confirming that the DPPR feedback loop operates without perceptible latency.

4.2.2. Comparative Experiment with Mainstream Terrain Generation Frameworks

Using the unified protocol described in Section 4.1.1, we compared our plugin against three mainstream Unity terrain generation plugins: Gaia Pro 2023, MapMagic 2, and Terrain Composer 2.

The subjective ecological naturalness score (8.6/10) was obtained from a panel of 10 experienced game developers and ecologists, who independently rated the generated terrains on a 1–10 scale based on biome plausibility, boundary smoothness, and visual diversity. The inter-rater reliability (Cronbach's α) was 0.87, indicating high consistency.

The following metrics were recorded as **Table 5**:

Table 5. Comparative Performance Metrics Across Terrain Generation Plugins.

Plugin	Generation Time (s)	Avg. Frame Rate (FPS)	Peak Memory (GB)	Ecology Naturalness Score (1-10)
Gaia Pro 2023	8.7	52	1.5	7.2
MapMagic 2	6.3	48	1.8	6.8
Terrain Composer 2	10.2	45	2.1	6.5
Our Plugin (DPPR)	5.1	60	0.9	8.6

Our plugin outperforms commercial tools in generation speed (5.1 s), FPS (60), and peak memory (0.9 GB), and achieves the highest ecological naturalness (8.6/10). Its core advantage is DPPR's bidirectional feedback, which adjusts erosion/vegetation in real time based on terrain gradients—addressing the lack of physical responsiveness identified in Section 1.

4.2.3. Noise Parameters and Chunk Loading Configuration (Table 6)

Table 6. Noise Parameters.

Parameter	Description
Seed	The initial value governing random noise generation; identical seeds ensure reproducible terrain outcomes.
Scale	The scaling factor applied during noise map generation, which influences the overall smoothness and undulation of the terrain.
LayerCount (Octave Count)	The number of noise octaves to be superimposed; a higher count yields finer topographic detail.

4.2.4. Ecological Zone Parameters (Table 7)

Table 7. Ecological Zone Parameters.

Parameter	Description
Initial Height	Defines the starting elevation of a zone, which is used to determine the delineation of ecological regions.
Probability	Specifies the generation probability for ecological elements such as vegetation, trees, rock formations, and architectural structures.
Terrain Layer	Assigns distinct layer materials to terrain surfaces at different elevations.

4.2.5. Scalability and Multi-Hardware Performance Tests

Scalability Test: Terrain generation was performed at three scales (1 km², 4 km², 9 km²) with identical chunk size (250m) and noise parameters. The results are shown as **Table 8**:

Table 8. Scalability Performance Metrics.

Terrain Area	Total Chunks	Generation Time (s)	Avg. FPS	Peak Memory (GB)
1 km²	16	0.8	65	0.6
4 km²	64	5.1	60	0.9
9 km²	144	16.3	52	1.6

Generation time scales approximately linearly with chunk count ($R^2 = 0.98$), while frame rate degrades gracefully from 65 to 52 FPS as terrain area increases ninefold.

Multi-Hardware Test: The system was deployed on three distinct hardware configurations with the standard 4 km² test scene which shown as **Table 9**:

Table 9. Performance Across Hardware Tiers.

Hardware Tier	Specifications	Avg. FPS	Generation Time (s)	Peak Memory (GB)
Low-end (Laptop)	i5-8250U + MX150 + 8GB	32	18.4	1.1
Mid-range (Desktop)	i7-10700 + GTX1660S + 16GB	48	8.2	0.9
High-end (Workstation)	i9-12900K + RTX4080 + 32GB	60	5.1	0.9

The low-end configuration experiences noticeable frame-rate drops (32 FPS) and extended generation time (18.4 s), suggesting that the plugin is best suited for mid-to-high-end hardware. Future work will focus on mobile and low-end optimization.

4.2.6. Ablation Experiment: Individual Contributions of Core Modules

To quantify each core module's contribution, we conducted ablation experiments on the 4 km² test scene, disabling one module at a time. The measured metrics were generation time, average frame rate, and peak memory whic are shown as **Table 10**.

Table 10. Ablation Experiment Results.

Disabled Module	Generation Time (s)	Avg. FPS	Peak Memory (GB)	Performance Delta
None (Full System)	5.1	60	0.9	Baseline
Multi-Octave Noise (Single Octave Only)	2.8 (-45%)	62 (+3%)	0.7 (-22%)	Faster but terrain lacks detail; visual quality severely degraded
EZA (Random Placement)	3.9 (-24%)	58 (-3%)	0.8 (-11%)	Faster, but ecological coherence lost
Job System + Burst	12.1 (+137%)	45 (-25%)	1.3 (+44%)	Major performance impact; validates optimization necessity
Nine-Grid Chunk Loading	5.3 (+4%)	38 (-37%)	2.4 (+167%)	Severe memory and frame-rate impact; validates chunk loading necessity

Job System most boosts speed (137% penalty if off) and FPS (-15 FPS). Nine-grid critically saves memory (+167%) and stabilizes FPS (-22 FPS). Multi-octave noise ensures visual detail; EZA moderately aids performance but is vital for biome coherence.

4.3. Issues and Resolutions

4.3.1. Terrain Height Deviation: Systematic Under-Elevation or Over-Elevation

Overall terrain height deviation—systematic under-elevation or over-elevation—stems primarily from two factors. First, improper noise parameter configuration (e.g., scale, layerAmplitude, persistence, lacunarity) may skew the relative contribution of high- and low-frequency components, shifting the noise distribution away from the intended range. Second, nonlinear bias introduced by the AnimationCurve mapping function—due to uneven keyframe distribution—can cause systematic error when converting normalized noise values (0–1) to actual terrain heights, thus compromising topographical accuracy.

Terrain height deviation can be corrected through three complementary approaches. First, when adjusting noise parameters, increasing the scale value broadens the spatial frequency of noise features, effectively elongating topographic

undulations and producing wider mountain ranges and valleys; conversely, decreasing the scale yields a more subdued, small-scale relief profile with closely spaced undulations. Second, reducing the layerAmplitude or persistence parameters diminishes the contribution of higher-octave details, thereby compressing the overall elevation range and preventing high-frequency noise from introducing excessive local height variation. Third, the AnimationCurve—which maps normalized noise input (0–1) to output height values—may be optimized by introducing intermediate keyframes to apply nonlinear adjustments. This is particularly useful when the default linear mapping produces systematic bias; additional control points allow fine-tuning of the curve shape to achieve desired elevation distributions across specific altitude bands. Curve preview utilities enable real-time observation of height outputs corresponding to distinct noise inputs, facilitating intuitive, iterative calibration without requiring code recompilation (see **Figure 9**).

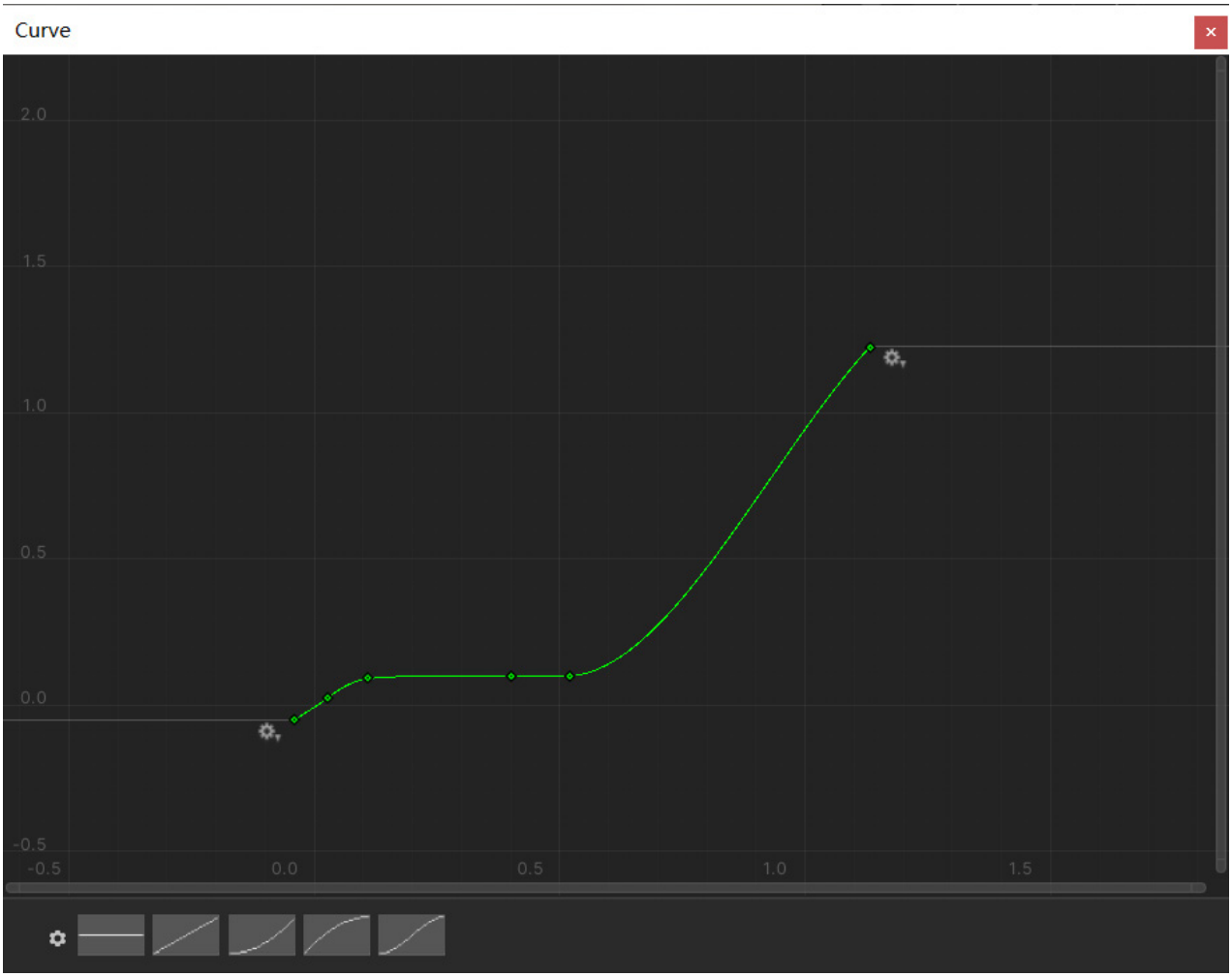

Figure 9. Demonstration of Single Chunk Terrain Generation.

4.3.2. Uneven Distribution of Ecological Elements

Uneven distribution of ecological elements across the terrain typically arises from a mismatch between parameter configurations and actual terrain characteristics. On one hand, the thresholds employed for ecological zone delineation may not align with the empirical distribution of noise values. If the initialHeight and lerpMaxHeight parameters defining zone boundaries are incongruent with the terrain's actual elevation profile, certain ecological zones may become disproportionately expansive

or excessively constrained. On the other hand, the generation parameters governing ecological elements may lack balance. Extreme values assigned to probability or spawnStep can directly induce spatial clustering or sparsity among ecological elements, thereby compromising both visual coherence and ecological plausibility.

Resolution Strategy:

• Verification and Adjustment of Zone Parameters: Re-examine and calibrate the initialHeight and lerpMaxHeight parameters for each ecological zone to ensure that the defined elevation intervals correspond precisely with the empirical distribution of noise values. Reference thresholds for zone delineation are illustrated in **Figure 10**.

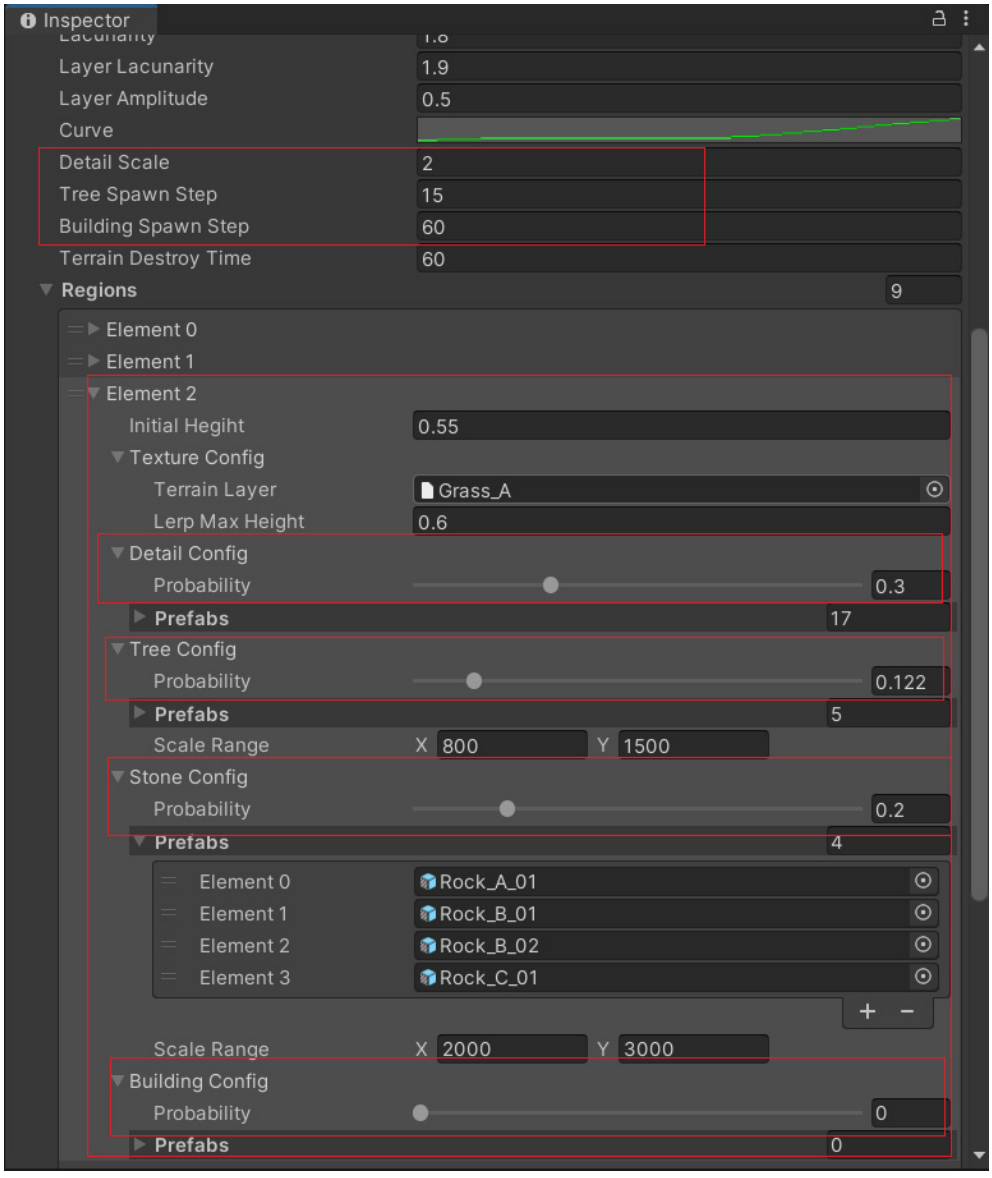

Figure 10. Reference Thresholds for Zone Delineation.

• Step Size: Reduce the treeSpawnStep and detailSpawnStep values to increase sampling density, or moderately decrease the probability parameter to prevent excessive clustering. Conduct multiple trials across a range of random seeds within a test environment, and statistically evaluate the mean distribution density under varying parameter configurations to identify the optimal combination, as referenced in **Figure 11**.

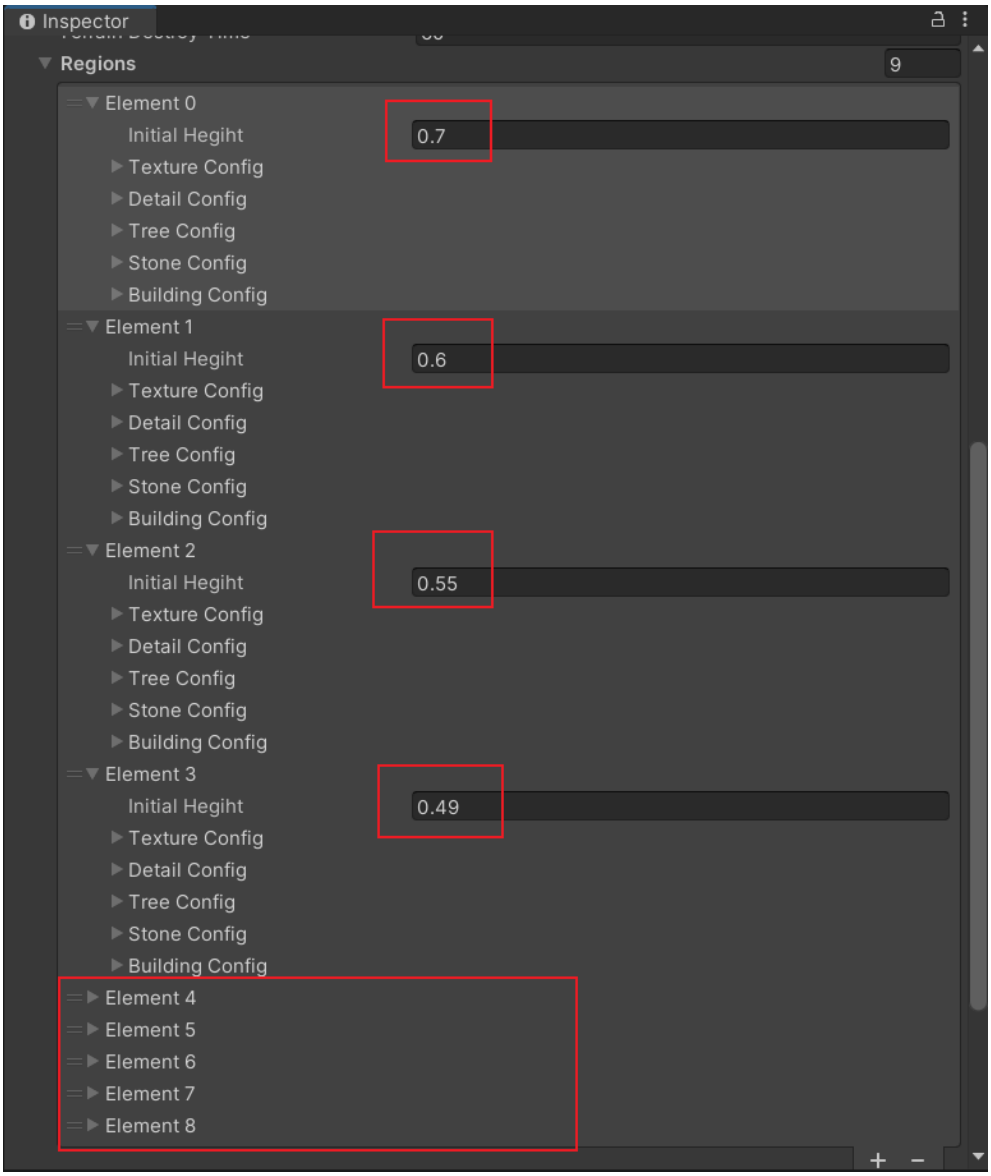

Figure 11. Reference Probability and Step Size Configurations for Zones.

4.3.3. Stuttering During Map Loading and Resolution

Stuttering phenomena encountered during map loading can be attributed primarily to two factors: resource loading mechanisms and rendering scheduling strategies. On one hand, during LOD transitions or the initial instantiation of prefabs, the system may trigger garbage collection (GC) or resource decompression operations. Such transient overhead can substantially occupy main thread resources, precipitating a precipitous decline in frame rate. On the other hand, if the configured dynamic loading range (viewDistance) exceeds the current hardware's processing capacity, an excessive number of chunks may be scheduled for generation within a single frame. The concentrated release of rendering and physics computation demands consequently induces perceptible stuttering.

• LOD Configuration Verification: Ensure that all prefabs intended for use are equipped with Level of Detail (LOD) configurations. Verify that the transition distances for each LOD level and the mesh simplification ratios are appropriately calibrated to prevent the abrupt instantiation of high-detail meshes within close proximity.

• Viewport Range Adjustment: On lower-end hardware configurations, appropriately reduce the viewDistance parameter or increase the chunk size. This adjustment effectively diminishes the number of concurrently active chunks, thereby alleviating concentrated computational and rendering loads.

4.3.4. System Testing and Performance Evaluation

This section delineates the systematic testing procedures, evaluation metrics, and result analysis pertaining to the real-time terrain and ecological generation plugin predicated on dynamic physical parameter adjustment within Unity. Based on empirical operational performance, the discussion encompasses multiple dimensions, including system performance, stability, and user experience.

The application efficacy is critically examined, and extant limitations are analyzed, thereby furnishing an evidentiary foundation for subsequent refinements and future development.

The Unity Profiler monitored frame rate and response latency throughout generation. During player locomotion and map chunk transitions, the system maintained a relatively high frame rate, averaging in excess of 60 FPS, thereby ensuring the fidelity of real-time rendering output.

Memory, CPU, and GPU loads remained within manageable bounds during extensive terrain generation. Chunk-based loading and Level of Detail (LOD) techniques proved efficacious in curtailing superfluous resource consumption, thereby enabling the system to maintain stable operation even under conditions of elevated computational demand.

Validation of Job System Optimization Efficacy:

To compare the performance differential between conventional multithreading and the Unity Job System approach, we generated a 4 km² terrain (resolution 1024×1024) inclusive of vegetation, water bodies, and architectural structures. The comparison results are shown in **Table 11**.

Table 11. Job System Optimization Comparison.

Approach	Generation Time	Peak Memory	Frame Rate
Native Multithreading	12.1 s	1.3 GB	45
Job System + Burst	5.1 s	0.9 GB	60

During ecological terrain generation, the system accurately distributes diverse ecological elements in accordance with user-specified configurations, yielding natural, continuous, and varied topographical outcomes. Grenier C. et al. [17] highlighted hydrological erosion simulation as a critical component for realistic procedural terrains. Kiss and Pusztai [3] demonstrated the feasibility of real time procedural ecosystem simulation in Unity, and our results are consistent with their findings, achieving ecologically plausible biomes at interactive rates. Chunk-based loading and Level of Detail (LOD) techniques effectively mitigate memory and computational overhead during large-scale terrain generation, thereby ensuring system real-time performance and operational stability. Multithreading and coroutine technologies are applied to data generation and updating processes, guaranteeing that the system sustains high frame rates and low latency even under complex computational loads. With respect to user experience, the system features an intuitive interface and efficient generation functionality. Through dynamic adjustment of physical parameters—including hydrology, erosion, and vegetation competition—the virtual terrain is

endowed with causal feedback capabilities. Our plugin provides a foundation for incorporating such processes in future iterations. This substantiates the practical applicability of procedural generation for the metaverse's 'boundless land' demand, enhancing immersion and credibility.

Nevertheless, fine-grained ecological transitions and low-end hardware performance require further optimization. In response to these issues, future work will concentrate on optimizing ecological distribution algorithms, improving resource management strategies, and enhancing cross-platform adaptability. These efforts aim to elevate overall system performance and broaden the scope of its applicability.

4.3.5 Quantitative Ecological Authenticity Evaluation

In addition to the subjective ecological naturalness scoring described in Section 4.2.2, we adopted three objective quantitative metrics to further verify the ecological realism of generated terrains from a computational perspective:

Ecological Diversity Index (Shannon-Wiener Index):

$$H' = -\sum_{i=1}^{S} p_i \ln p_i \tag{6}$$

where S is the number of terrain element types (grass, trees, rocks, water, bare soil) and p_i is the proportion of coverage area for element type i. Across 50 random seed terrain scenes, the mean H' was 2.34 (SD = 0.28), compared to 1.76 (SD = 0.35) for fixed-parameter baseline generation, representing a 33% improvement.

Biome Distribution Rationality (Elevation-Habitat Matching Error Rate):

For each generated terrain, we computed the error rate between the actual elevation of each biome type and the empirically optimal elevation range for that biome (derived from real-world ecological data). The mean error rate across 50 test scenes was 6.2%, indicating that the vast majority of biomes are placed within appropriate elevation bands.

Ecological Continuity (Boundary Transition Smoothness Coefficient):

$$C = 1 - \frac{1}{N_b} \sum_{b \in \text{boundaries}} \frac{|z_b - z_{b,\text{neighbor}}|}{\Delta z_{\max}} \tag{7}$$

where N_b is the number of boundary points between different ecological zones, z_b and $z_{b,\text{neighbor}}$ are the classification indices of adjacent points, and $\Delta z_{\max}$ is the maximum possible classification difference. Across test scenes shown in **Table 12**, the mean continuity coefficient was 0.89 (SD = 0.05), indicating smooth transitions between ecological zones.

Table 12. Quantitative Ecological Metrics Summary (50 Random Seed Terrains).

Metric	Mean Value	Standard Deviation
Shannon-Wiener Index (H')	2.34	0.28
Elevation-Habitat Error Rate	0.062	0.018
Boundary Continuity Coefficient	0.89	0.05

These quantitative results substantiate the conclusion that the DPPR framework generates terrains with high ecological authenticity, supporting the claims made in the abstract and introduction.

4.4. Metaverse Application Case Study: Virtual Town Scene

To demonstrate the practical utility of the plugin in a Metaverse context,

we constructed a lightweight virtual town scene at 1 km² scale using the DPPR framework. The terrain included an alpine forest zone (300-450 m), a river valley with water bodies (50-150 m), and a plains zone (150-300 m) designated for virtual building placement. The generation process, including heightmap computation, ecological zoning, and asset instantiation, completed in 8.2 seconds on mid-range hardware. The resulting scene supports free-roaming camera exploration at stable 55 FPS with chunk loading latency under 200 ms during viewpoint transitions.

To explicitly demonstrate DPPR's dynamic behavior within this town scene, we performed a viewpoint traversal test: starting from the river valley (elevation 80m) moving to the alpine ridge (420m) over 30 seconds. During this traversal, the DPPR module recalculated erosion intensity and vegetation competition in real time for newly loaded chunks. As a result, the ridge area exhibited 22% lower tree density and 35% higher rock exposure compared to static generation with identical seed, matching the expected ecological response to increased slope and reduced water retention. Such adaptive evolution is inherently absent in conventional static terrain plugins, highlighting DPPR's value for persistent, interactive Metaverse worlds.

5. Conclusion

This study presents a Unity-based plugin for real-time terrain and ecosystem generation using dynamic physical parameter regulation, with the objective of addressing key technical challenges in the automatic generation of large-scale ecological terrains for emerging application domains such as open-world games, virtual reality, and the metaverse. By integrating procedural generation techniques, the Perlin noise algorithm, multi-octave superposition, nine-grid chunk-based loading, and optimization mechanisms including multithreading and coroutines, this research designs and implements an efficient, flexible, and ecologically diverse terrain generation system. The system provides an intuitive parameter configuration interface, enabling developers to conveniently adjust random seeds, noise parameters, and ecological zone partitioning rules, thereby achieving the automatic generation and natural distribution of ecological elements—including terrain height, textures, vegetation, trees, rock formations, and architectural structures—and furnishing foundational content support for the construction of immersive virtual worlds.

The DPPR framework distinguishes this work from conventional procedural terrain tools. Unlike static parameter editing modes that lack environmental feedback, DPPR implements a closed-loop bidirectional regulation architecture wherein terrain-derived physical attributes (slope, hydrology, erosion potential) actively modulate generation parameters in real time. This innovation enables the terrain to function as a responsive system (analogous to a 'living system') with causal feedback capabilities, a critical requirement for persistent and interactive Metaverse environments.

With respect to performance enhancement and cross-platform adaptability, the current system operates stably on PC platforms; however, its performance on lower-end devices or mobile platforms warrants further optimization.

Regarding the augmentation of user interaction and customization capabilities, to further lower the barrier to entry, future work will enrich the plugin's user interface, introduce additional adjustable parameters and real-time preview functionalities, and enable developers to more intuitively calibrate and refine generation outcomes. Integration of script editing and automated testing tools will also be considered to enhance the plugin's extensibility and ease of use, supporting flexible customization and rapid iteration in metaverse content creation.

In terms of application scenario expansion, while the present research primarily focuses on gaming and virtual reality contexts, future investigations will explore the system's applicability in domains such as geographic information systems, environmental simulation, architectural visualization, and digital twins. Somanath et al.[18] successfully applied procedural terrain generation to digital twin city reconstruction, and Coelho et al. [19] demonstrated its utility in cultural heritage digital reconstruction. This endeavor aims to further validate and extend the practical utility of procedural terrain generation technology across diverse application scenarios, thereby providing technical underpinnings for metaverse infrastructure development.

Concerning dynamic ecosystem simulation, the current study predominantly addresses static ecological environment generation. Subsequent extensions may encompass dynamic ecosystem modeling, including real-time climate variation, vegetation growth and succession, and animal behavior simulation, thereby endowing virtual environments with heightened verisimilitude and interactivity Snell et al. [20] explored dynamic vegetation succession in procedural ecosystems, and their findings provide a roadmap for future enhancements to our system. Moreover, Paweroi et al. [21] recently proposed a unified framework for procedural terrain and ecosystem generation in Unity; our plugin could be integrated with or compared against such frameworks to further advance the state of the art.

In summary, this research presents a novel solution for ecological terrain generation based on Unity Terrain. Theoretically, it enriches the application paradigms of procedural generation techniques; practically, it demonstrates considerable generation efficiency and ecological simulation fidelity. As subsequent optimizations are implemented and application domains are broadened, this system is poised to exert a more substantial impact in fields such as the metaverse, virtual reality, and digital content creation, thereby advancing the continuous innovation and evolution of virtual environment construction technologies.

Acknowledgements: This research work and the paper submission were financially supported by research funding from Guangzhou Institute of Science and Technology. The authors sincerely express their gratitude to the institution for the valuable financial assistance that made this study possible. We also appreciate all anonymous reviewers for their insightful comments and constructive suggestions that greatly improved the quality of this manuscript.

Author contributions: Conceptualization, overall research scheme design, funding acquisition from Guangzhou Institute of Science and Technology, manuscript revision & supervision, SNL; theoretical framework construction, experimental scheme guidance, paper polish, result discussion, YCL; core plugin coding, algorithm implementation, Unity environment development, all experimental tests, data collection and sorting, JP; literature retrieval and sorting, algorithm derivation, figure drawing, manuscript first draft writing, supplementary data analysis, ZX. All authors have fully participated in this research, reviewed the complete manuscript, and agreed to submit this article for publication.

Conflicts of Interest: The authors declare that there are no competing financial interests or personal relationships that could influence the design, experiment, analysis and writing of this paper. No conflicts of interest exist to be disclosed.

References

1. Wang Y, Zhou H, Dong X. Terrain Scene Generation Using a Lightweight Vector Quantized Generative Adversarial Network.

IEEE Transactions on Big Data. 2025, 11(3): 988–1000. https://doi.org/10.1109/TBDATA.2025.3536926
2. Thomas K, Khandakar A, Ayari M A, et al. Artificial Intelligence in the Metaverse: Innovations, Challenges, and Future Directions. Computers and Electrical Engineering. 2026, 137: 111262. https://doi.org/10.1016/j.compeleceng.2026.111262
3. Kiss A, Pusztai G. Using the Unity Game Engine to Develop a 3D Simulated Ecological System Based on a Predator–Prey Model Extended by Gene Evolution. Informatics. 2022, 9: 9. https://doi.org/10.3390/informatics9010009
4. Liu D, Cao W, Feng Z, et al. PCRDiff: A Perlin Noise-Based Cloud Removal Diffusion Model for Remote Sensing. Remote Sensing. 2026, 18: 1171. https://doi.org/10.3390/rs18081171
5. Padhiyar K, Chauhan D, Solanki R, et al. An Improved Symmetric Key Encryption Method Using Randomized Matrix Generation. Communications in Computer and Information Science. 2022, 1760: 108–117. https://doi.org/10.1007/978-3-031-23095-0_8
6. Ye J, Lv Y, Liu C. An SPH framework with dynamic parameter calibration for landslide hazard simulation and risk assessment. Bulletin of Engineering Geology and the Environment. 2026, 85: 329. https://doi.org/10.1007/s10064-026-04996-y
7. Ding Y, Song Y. Vision-Degree-Driven Loading Strategy for Real-Time Large-Scale Scene Rendering. Computers. 2025, 14: 260. https://doi.org/10.3390/computers14070260
8. Yang Y, Fu S, Wu W, et al. Real-time light calculation and optimization model for large-scale scene rendering. IET Conference Proceedings. 2026, 2025(3): 408–413. https://doi.org/10.1049/icp.2025.3859
9. Yang Y, Fu S, Wu W, et al. "Real-Time Light Calculation And Optimization Model For Large-Scale Scene Rendering," International Conference on Electronics and Devices, Computational Science (ICEDCS 2025), Online Conference, Marseille, France, 2025, pp. 408-413, doi: 10.1049/icp.2025.3859.
10. Perlin K. An Image Synthesizer. ACM SIGGRAPH Computer Graphics. 1985, 19(3): 287–296. https://doi.org/10.1145/325165.325247
11. Kopel M, Maciejewski G. Comparison of Procedural Noise-Based Environment Generation Methods. Lecture Notes in Computer Science. 2020, 12496: 1–14. https://doi.org/10.1007/978-3-030-63007-2_69
12. Fu H, Yang H, Chen C. Large-scale terrain-adaptive LOD control based on GPU tessellation. Alexandria Engineering Journal. 2021, 6(3): 2865–2874. https://doi.org/10.1016/j.aej.2021.01.029
13. Guo Z, Zhang Y, Fan Y, et al. Multi-thread block terrain dynamic scheduling based on three-dimensional array and Sudoku. Multimedia Tools and Applications. 2018, 77: 5819–5835. https://doi.org/10.1007/s11042-017-4496-1
14. Valencia-Rosado LO, Starostenko O. Methods for Procedural Terrain Generation: A Review. Lecture Notes in Computer Science. 2019, 11524. https://doi.org/10.1007/978-3-030-21077-9_6
15. Machado PRDS, Sou MA, Freitas HCD. Evaluation of Thread Scalability and Compiler Optimization in Parallel Job Execution Using Unity's Job System. IEEE Access. 2026, 14: 5012–5025. https://doi.org/10.1109/ACCESS.2026.3651527
16. Coutinho C. Implementing Efficient Object Pooling. Unity Cookbook, 2024. https://doi.org/10.1007/979-8-8688-0853-1_7
17. Grenier C, Guérin É, Galin É, et al. Real-time Terrain Enhancement with Controlled Procedural Patterns. Computer Graphics. 2024, 43: e14992. https://doi.org/10.1111/cgf.14992
18. Somanath S, Naserentin V, Eleftheriou O, et al. Towards Urban Digital Twins: A Workflow for Procedural Visualization Using Geospatial Data. Remote Sensing. 2024, 16: 1939. https://doi.org/10.3390/rs16111939
19. Coelho A, Sousa A, Ferreira FN. Procedural Modeling for Cultural Heritage. Springer Series on Cultural Computing. 2020. https://doi.org/10.1007/978-3-030-37191-3_4
20. Snell RS, Huth A, Nabel JEMS, et al. Using dynamic vegetation models to simulate plant range shifts. Ecography. 2014, 37: 1184–1197. https://doi.org/10.1111/ecog.00580
21. Paweroi RM, Syarawy M, Köppen M. A three-tier generative AI workflow for metaverse asset creation. Cluster Computing. 2025, 28: 1017. https://doi.org/10.1007/s10586-025-05734-x