Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Procedural Content Generation in Unreal Engine

Lukáš Jůza

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2025

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jůza Lukáš** Personal ID number: **518306**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Procedural Content Generation in Unreal Engine

Bachelor’s thesis title in Czech:

Procedurální generování obsahu v Unreal Enginu

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **08.02.2025** Deadline for bachelor thesis submission: ____________

Assignment valid until: **20.09.2026**

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student’s signature

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jůza Lukáš** Personal ID number: **518306**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Procedural Content Generation in Unreal Engine

Bachelor’s thesis title in Czech:

Procedurální generování obsahu v Unreal Enginu

Guidelines:

Review procedural generation methods for video game environments. Focus on the support of procedural content generation (PCG) in Unreal Engine 5. Describe the PCG tools available in UE 5 and try to connect them with procedural methods known from the literature.
Create practical examples of procedural generation of different environments (e.g., forests, meadows, mountains, cities) and procedural creation of individual objects (e.g., houses and building elements). Integrate the partial outputs into a project demonstrating the procedural generation of an infinite open world. Focus on the efficiency and smoothness of the PCG generation. Minimize visual artifacts when moving in the generated scenes and focus on maintaining constant rendering times.
Conduct performance tests that evaluate the dependence of visual quality and rendering speed on the important parameters of the PCG setup, such as scene partitioning density, minimal generation distance, and memory allocation limits.

Bibliography / sources:

[1] Short, T., Adams, T. (2017). Procedural Generation in Game Design. CRC Press.
[2] Gregory, J. (2017). Game Engine Architecture. 3rd ed. CRC Press.
[3] Togelius, J., Yannakakis, G. N., Stanley, K. O., Browne, C. (2011). A Survey of Procedural Content Generation. IEEE Transactions on Computational Intelligence and AI in Games, 3(3), 166-184.
[4] Togelius, J., Shaker, N., Nelson, M. J. (2016). Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer.
[5] Epic Games (2024). Procedural Generation in Unreal Engine 5. Available at: https://dev.epicgames.com/ (Accessed: 7 October 2024).

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Jůza Lukáš
Personal number:	518306
Programme name:	Open Informatics

declare that I have elaborated the bachelor’s thesis entitled

Procedural Content Generation in Unreal Engine

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor’s and Continuing Master’s Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 13.05.2025

Lukáš Jůza
....
student's signature

Acknowledgements

I would like to express my sincere gratitude to Mr. J. Bittner, Ph.D. for his invaluable guidance and for granting me the freedom to choose and develop this topic in my own way.

Declaration

I declare that I have completed the submitted thesis independently and that I have cited all sources used.

In Prague, 2 May 2025

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 2. května 2025

Abstract

This thesis introduces a cohesive system for infinite, on-the-fly world generation in Unreal Engine 5.5. We present a multi-threaded C++ terrain generator leveraging custom noise functions to stream diverse landscapes without main-thread stalls. In parallel, we develop modular PCG Graphs for procedurally placing natural elements (biomes, vegetation) and man-made structures (roads, fences, buildings, villages). Custom Blueprint and C++ nodes enable advanced filtering criteria—such as biome rules—and the combination of world partitioning with background-task scheduling preserves constant frame rates.

Keywords: procedural generation, Unreal Engine 5.5, C++, terrain generator, PCG Graph, infinite world, biome, chunk

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Technická 2,
166 27 Praha 6

Abstrakt

Tato práce představuje soudržný systém pro nekonečnou generaci světů za běhu v Unreal Engine 5.5. Předkládáme vícevazebný C++ generátor terénu založený na vlastních šumových funkcích, který dokáže plynule streamovat různorodé krajiny bez blokování hlavního vlákna. Současně jsme navrhli modulární PCG grafy pro procedurální rozmístění přírodních prvků (biomy, vegetace) a lidských staveb (silnice, ploty, budovy, vesnice). Vlastní Blueprint a C++ uzly umožňují pokročilé filtrování dle biomu a díky dělení světa na zóny a zpracování úloh v pozadí zachovává systém konstantní snímkovou frekvenci.

Klíčová slova: procedurální generování, Unreal Engine 5.5, C++, generátor terénu, PCG graf, nekonečný svět, biotopy, sekce

Překlad názvu: Procedurální vytváření obsahu v Unreal Engine

Contents

Figures

Tables

Chapter 1

Introduction

In recent years, the scale and ambition of game worlds have grown dramatically. Open-world and live-service titles demand vast, varied landscapes that remain engaging over hundreds of hours of play. Traditionally, such environments required enormous manual effort from artists and level designers—painstakingly placing each rock, tree, and building by hand. Procedural Content Generation (PCG) offers a compelling alternative: algorithmic workflows that can spawn terrain, vegetation, structures, and even entire cities with minimal human intervention. By encoding rules and randomness into a generation pipeline, developers can achieve both diversity and consistency at a fraction of the cost.

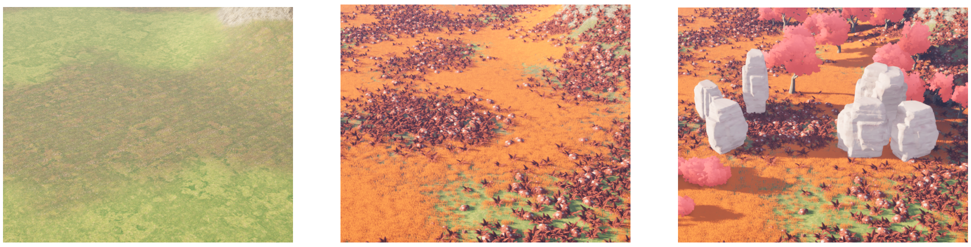
The aim of this work is to explore and analyze the possibilities of PCG in Unreal Engine 5 and subsequently create a functional prototype of a procedurally generated infinite world.

A key focus is to determine the feasibility of utilizing PCG in runtime environments—where content must be generated on the fly as the player moves—without introducing visible artifacts or stutters. Throughout this work, we will measure performance and discuss trade-offs between quality and speed. The outcome is both a theoretical framework relating PCG theory to UE5's toolset and a concrete prototype that demonstrates an end-to-end pipeline for infinite procedural worlds.

Chapter 2

Foundations and Design

This chapter brings together the theoretical foundations of Procedural Content Generation (PCG) with the design of our solution. We begin by reviewing core PCG techniques and algorithms from the literature, then present the high–level architecture and key components of our prototype.



(a) : Terrain before PCG run.

(b) : Plants spawned procedurally.

(c) : Cliffs, trees, and bushes added.

Figure 2.1: Motivational demonstration of PCG in a stylized “red” biome: (a) begins with empty terrain, (b) adds procedurally spawned vegetation, and (c) completes the scene with cliffs, trees, and bushes—all generated at the click of a button using PCG.

2.1 Procedural Content Generation: Concepts and Algorithms

Procedural Content Generation (PCG) refers to the algorithmic creation of game assets—terrain, vegetation, structures, levels—either at build time or dynamically at runtime. Common methods include:

- **Noise Functions:** Perlin, Simplex, and Value noise generate smoothly varying fields used for terrain heightmaps, texture masks, and variation in object placement. More discused later in 4.2.1.
- **Lindenmayer Systems (L-systems):** L-systems are formal grammars originally developed by Aristid Lindenmayer for modeling the growth processes of plants and multicellular organisms. An L-system begins

with an initial "axiom" string and applies production rules iteratively to replace symbols, yielding complex, self-similar branching structures such as trees, ferns, and coral. This approach underpins many procedural botany algorithms and is explored in depth by Prusinkiewicz and Lindenmayer [27].

- **Flow Fields:** Flow fields are continuous vector fields that guide the placement of paths, roads, rivers, or crowd movement by indicating a preferred direction at each point in space. By sampling the field and following the vectors, one can generate natural, winding trajectories that mimic fluid dynamics or erosion patterns. A practical introduction to flow-field pathfinding is provided by Patel [28].

- **Tile-based Rules:** Tile-based methods use a set of discrete tiles and metadata-driven adjacency constraints (e.g., Wang tiles) to assemble large environments without visible seams or repetitive patterns. Each tile encodes edge compatibility, and procedural assemblers select tiles based on neighborhood rules. This technique was pioneered by Wang [30] and later applied to texture synthesis and environment tiling by Cohen et al. [31].

- **Shape Grammars:** Shape grammars extend the idea of string grammars to geometric forms: starting from an initial shape, production rules replace shapes with arrangements of sub-shapes according to recursive rules. This hierarchical rewriting process can generate architectural elements—walls, windows, roofs—and complex procedural meshes. The foundational work on shape grammars was published by Stiny and Gips [32].

- **Cellular Automata:** Cellular automata consist of regular grids of cells that update their states simultaneously according to simple local rules based on neighbor values. Despite their simplicity, they can produce elaborate patterns used for cave and dungeon layouts, terrain erosion, or vegetation spread. Wolfram's survey of cellular automata highlights their complexity and applications [33], and they are often employed in voxel-based cave generation systems [22].

Pragmatic Runtime Constraints. Generating tens of thousands of meshes on the fly leaves no budget for heavy simulations or costly computations. Instead, our system relies on lightweight, rule–based workflows—chaining simple sampling, filtering, and transformation nodes, with Perlin-noise perturbations—to introduce variation and break up repetition in both terrain and object placement. While techniques such as L-systems, flow fields, tile-based adjacency, shape grammars, and cellular automata offer rich, biologically inspired patterns, they were not directly implemented here; rather, their core ideas motivated our streamlined PCG graphs, where basic operations and sequenced noise applications deliver real-time performance alongside believable procedural detail.

2.2 Related Work

Below are additional noteworthy theses in related areas of procedural generation:

- **Tile-based Procedural Generation (Rudolf Líbal, 2023):** Analyzes existing solutions for tile-based PCG, proposing a generic library that supports multiple terrain layers, metadata-driven adjacency rules, and weighted random tile selection using noise to minimize incompatible placements [1].
- **Procedural Generation of Voxel Worlds (Lukáš Hepner, 2021):** Develops a voxel rendering engine with deterministic noise-based terrain and L-system vegetation placement, emphasizing reproducibility via fixed seeds [2].
- **3D Modeling and Visualization of Underground Structures (Martin Hudeček, 2021):** Presents methods for creating and preserving accurate 3D models of underground spaces (e.g., the Johannes adit) using mobile laser scanning and point-cloud processing in VR [3].
- **Procedural Terrain Generation Using GPU Acceleration (Jakub Navrátil, 2016):** Explores GPU-accelerated terrain generation with CUDA-driven Perlin noise heightmaps, demonstrating performance improvements over CPU implementations [4].

2.3 Project Architecture and Design

The system design unfolds in two main phases, each focusing on building and integrating core functionalities.

Phase 0: Conceptual Foundations of PCG. Before any implementation, we establish the core concepts and workflows of Procedural Content Generation in UE5:

Phase 1: Component Development.

- **Environment:** PCG Graphs for forests, meadows, and mountains and more are created.
- **Infrastructure:** PCG Graphs for roads, fences, houses, and villages are created and discussed.
- **Procedural Material:** A unified landscape material blends texture layers based on elevation, slope, curvature, and noise masks to produce seamless, context-aware terrain texturing.

Note: A PCG Graph is an object that spawns meshes based on its node logic.

Phase 2: Integration into Infinite World.

- **Terrain Streaming:** A multithreaded C++ generator creates chunked terrain and handles dynamic loading and unloading of chunks.
- **PCG Population:** Terrain is being dynamically populated with meshes using PCG Graphs at runtime.
- **Performance Tuning:** Key parameters—chunk size, generation radius, LOD thresholds—are adjusted to maintain consistent frame rates and minimize runtime stalls.

Note: A chunk is a square section of terrain used to optimize loading and processing.

2.3.1 Dual Visual Style

The prototype supports two distinct visual styles:

- **Low-Poly Style:** Prioritizes minimal geometry and simple materials to maximize performance on lower-end hardware, while maintaining clear, stylized visuals.
- **Photorealistic Style:** Employs high-resolution meshes, detailed PBR materials, and advanced lighting to achieve cinematic-quality visuals.

This dual-mode design allows us to:

- Evaluate how easily the system's graphical pipeline (terrain generation, PCG Graphs, materials) adapts to different art directions.
- Measure performance differences—frame rates.
- Provide flexibility for target platforms, enabling a single codebase to serve both mobile/VR (low-poly) and AAA desktop/console (photorealistic) builds.

These capabilities inform our performance tuning strategy and highlight the modularity of the PCG pipeline.

2.3.2 Design Goal

The primary goal of our system is to generate rich, varied open-world environments in real time, while maintaining smooth, consistent performance without frame-rate drops or loading hitches.

Figure 2.2: Inspirational open-world vistas from *Breath of the Wild*, *Genshin Impact*, *Red Dead Redemption 2*, and *Just Cause*.

Chapter 3

Procedural Content Generation in Unreal Engine 5

Procedural Content Generation (PCG) has emerged as a cornerstone of modern game development, enabling artists and designers to create vast, varied worlds with minimal manual effort. In Unreal Engine 5.5, Epic Games has refined its PCG framework into a robust, node-based system that lets you craft intricate procedural workflows entirely within the editor—no C++ required. Whether you're populating rolling meadows, dense forests, or craggy mountain passes, PCG Graphs offer the flexibility to sample, filter, transform, and spawn thousands of models at runtime or during level build.

This chapter serves as your one-stop reference and hands-on guide to UE5's PCG. We'll begin by exploring the core concepts of the PCG Graph—how metadata, points, and attributes power every node—and then walk through the most commonly used node types: samplers, filters, transformers, and spawners. From there, you'll learn best practices for integrating PCG assets into your scenes, organizing complex graphs with subgraphs, and optimizing performance using World Partition, Nanite, and LOD techniques.

Finally, we'll dive deeper into biome-specific examples (meadows, forests, mountains), advanced custom logic (Blueprint and C++ integration), and the newest UE 5.5 features for grammar-based rules and procedural mesh editing. By the end of this chapter, you'll have a complete, reusable toolkit for building your own advanced PCG graphs—and the confidence to push them even further.

3.1 PCG Graph and Its Functionality

In this section, we will walk through the inner workings of the PCG Graph in Unreal Engine 5.5. First, we'll introduce the concept of metadata—collections of points carrying transforms and custom attributes—and show how nodes exchange and modify this data. Next, we'll categorize and describe the primary node types (input/output, point generation, processing, Blueprint/C++ injection, and mesh/actor spawners) and illustrate how chaining them builds powerful procedural workflows.

Note: This project was developed in Unreal Engine 5.5, which introduced changes compared to previous versions. Some tools and concepts have different names or functionality, requiring adaptation to the new environment and workflow.

Basic Concept of Nodes

Nodes in the PCG Graph operate on **metadata** consisting of collections of individual points. These points are not merely 3D coordinates but also include **attributes** that define their properties. Each point contains information about its transform (position, rotation, scale, bounds, color, density, seed atc...). Additionally, user-defined **attributes** can be assigned to points, making the system highly flexible and customizable.

We can think of a point as a structure containing all relevant data, with the metadata represented as arrays of these points.

From a higher-level perspective, nodes in a PCG Graph can generally be divided into several categories based on their primary function:

- **Input/Output Nodes** – These nodes handle importing or exporting metadata (collections of points) into or out of the graph.
- **Point-Generating Nodes** – These nodes create new points (for example, by sampling a surface or duplicating existing points).
- **Point-Processing Nodes** – These nodes modify existing points (e.g., applying transformations or filtering based on attributes).
- **Blueprint Nodes** – These nodes allow custom logic to be integrated into the PCG Graph through Blueprints or C++.
- **Final Nodes (Mesh/Actor Spawners)** – These nodes use the points defined in the graph to instantiate static meshes, actors, or other objects in the scene. They represent the final step in the procedural generation workflow.

3.1.1 Commonly Used Nodes

For illustration of how PCG nodes work, here are some examples of commonly used nodes in the project along with brief descriptions of their functionality:

- **Get Landscape Data:** Retrieves data directly from Unreal's built-in Landscape system. This node provides essential information for various samplers but is limited to use with the `Landscape` actor only.
- **World Ray Hit Query:** Performs a raycast in the world to retrieve surface data from any object that supports collision. Unlike the previous node, this method can sample both terrain and static meshes, making it more versatile for runtime procedural generation.
- **Surface Sampler:** Samples points on the surface of input geometry (e.g., landscape, mesh) based on specified density and noise functions.

- **Spline Sampler:** Generates points along splines or between sequence of guiding points, useful for creating paths, fences, or other linear features.

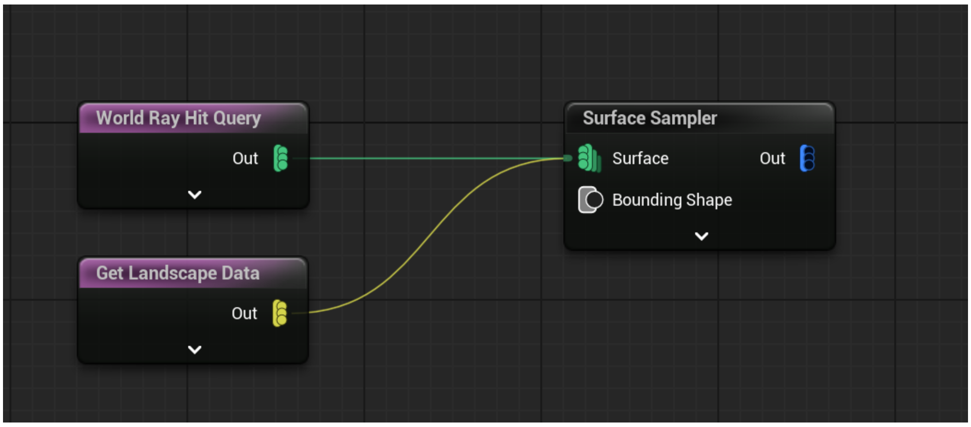
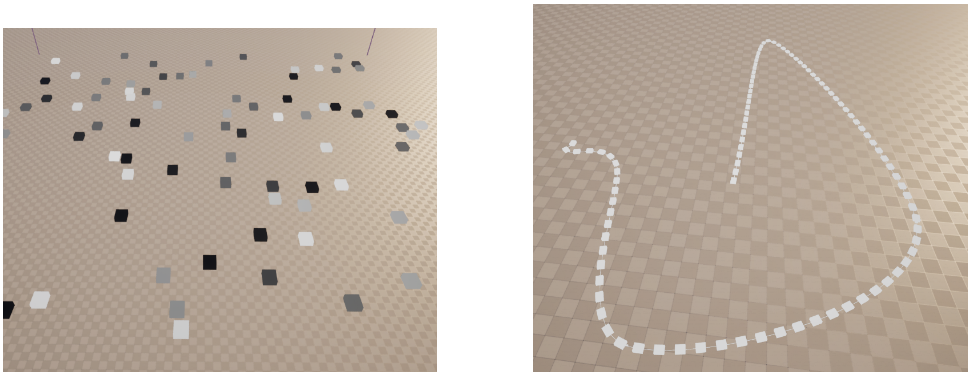


Figure 3.1: Example of how to use the Surface Sampler with different input sources. The node can accept either collision data from `World Ray Hit Query` or heightmap data from `Get Landscape Data`.

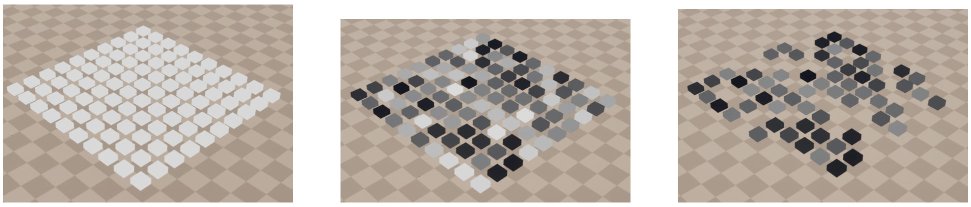


(a) : Surface sampler.

(b) : Spline sampler.

Figure 3.2: Showcase of points being created using different samplers.

- **Copy Points:** Duplicates existing points at specified locations, allowing for replication of objects or data.
- **Create Points Grid:** Creates a grid of points, often used for structured layouts or uniform distributions.
- **Attribute Noise:** Adds variation to attributes like position, density, or color using noise functions.
- **Attribute Filter:** Filters points based on specific attribute values or conditions.



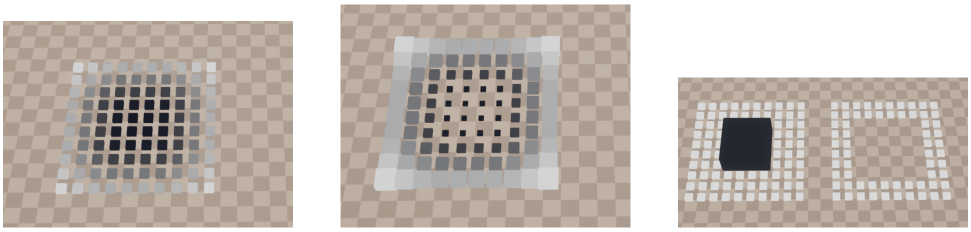
(a) : Create points grid and copy points.

(b) : Attribute noise.

(c) : Attribute filter.

Figure 3.3: Sequence of point manipulation using PCG nodes: (a) grid and duplication, grid of point is being created around the central one, (b) noise-based variation, (c) density-based filtering (shown in grayscale).

- **Distance:** Measures the distance between points or objects, often used for spatial filtering.
- **Difference:** Calculates the difference between attributes, enabling comparisons or conditional logic.
- **Scale by Density:** Adjusts the scale of objects based on their density, creating natural variation. (This node is accessed by executing a Blueprint node and selecting "Scale by Density.")



(a) : Distance from center

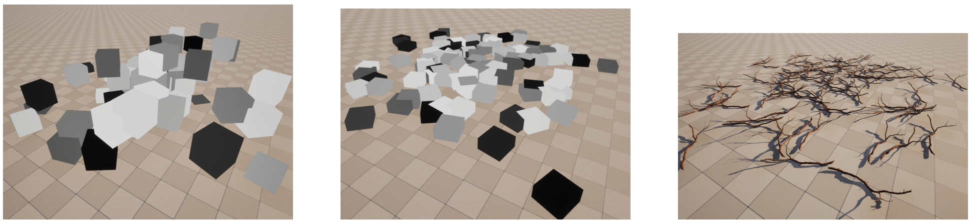
(b) : Scale by density

(c) : Difference

Figure 3.4: Showcase of points manipulation using nodes in (a), (b) and (c).

- **Transform Points:** Modifies the position, rotation, or scale of points either randomly within a given range or to a specified value.
- **Projection:** Projects points onto a surface or plane, aligning objects with terrain or other geometry.

  Note: Always use the Projection node when manipulating points that are meant to be on the ground to ensure they do not end up floating above or below the surface.
- **Static Mesh Spawner:** Instantiates static meshes at the locations defined by points, forming the final objects in the scene.



(a) : Transform Points **(b)** : Projection **(c)** : Static Mesh Spawner

Figure 3.5: Example of points being moved, projected to landscape, and finally used as spawning point for a branch mesh.

Workflow and Scene Integration

Integrating the PCG framework into a project involves several key steps:

1. **Activating the PCG Plugin:** We enable the PCG plugin in Unreal Engine to access PCG features.
2. **Creating the PCG Graph:** We generate a new PCG Graph asset in the Unreal Engine Editor. This graph can be opened in the PCG Editor, where procedural logic is defined by adding and connecting nodes.
3. **Defining and Sampling Points:** Input data is supplied to the graph or generated using nodes such as *Surface Sampler* or *Spline Sampler*. These points act as anchors for object placement.
4. **Transforming Points:** We modify sampled points using nodes like *Transform Points* (e.g., change scale, rotation, position) for specific use cases.
5. **Debugging the Workflow:** We can visualize the current state of points and their attributes at various stages in the PCG Graph. This includes position in the scene, metadata values (e.g., density, color), and any user-defined attributes. Such visualization aids in refining logic and ensuring correctness.
6. **Generating Objects:** We instantiate objects (e.g., static meshes) in the scene using nodes like *Static Mesh Spawner* or *Blueprint Spawner*. Additional logic can refine placement and behavior, such as procedural building or urban layout generation.
7. **Iterative Refinement:** We expand the graph with additional nodes, subgraphs, and conditional logic to create sophisticated and adaptive procedural systems.

Subgraphs and Modular Design

Complex PCG Graphs can grow large quickly, as shown in Figure 3.6. To manage this complexity, we can use **PCG Subgraphs**, similarly to functions in programming, allowing us to package a specific subset of the logic (e.g., repetitive generation of rock formations or rules for creating fences) into a separate file. This approach keeps graphs more organized and facilitates reusability.

However, **well-commented and clearly structured graphs** can remain readable even without subgraphs. In this project, we mostly preferred this approach to reduce fragmentation and keep the logic visible in a single view when possible.

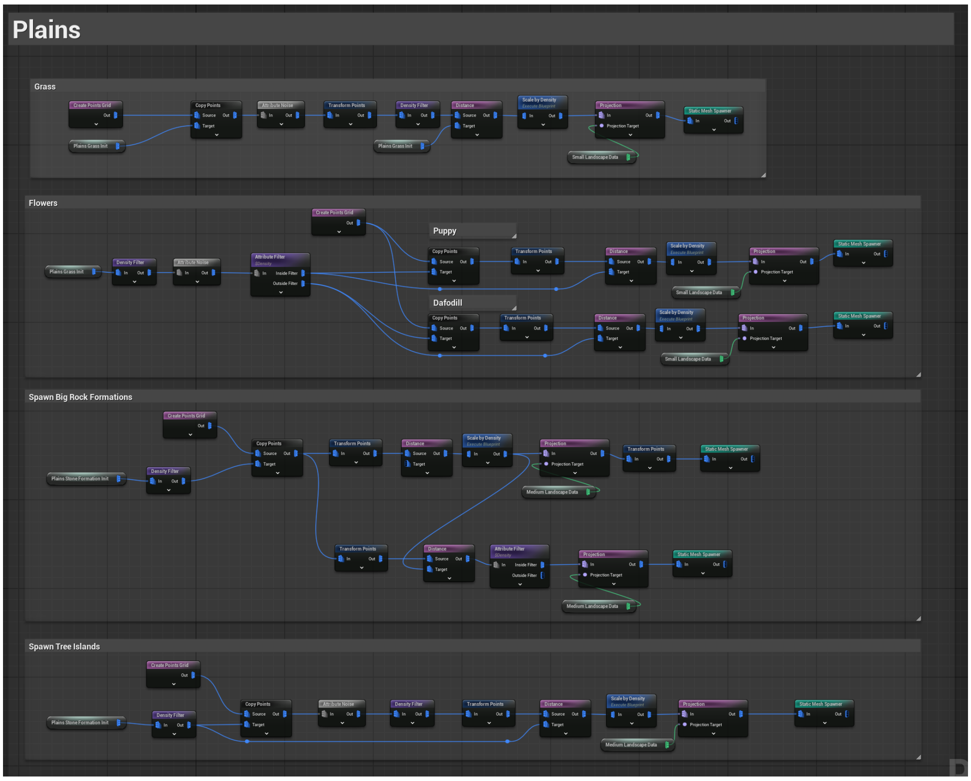


Figure 3.6: Example of complex PCG graph used to generate plains: 3.16.

Differences Compared to Previous Versions of Unreal Engine

- **Terminology and Selected Nodes:** In versions 5.0–5.2, the PCG Graph was primarily experimental. Certain nodes or features had different names or limited functionality. In version 5.5, several components were renamed or refined; thus, when migrating older projects, we must adapt to the new nomenclature and improved nodes.

- **Performance and Debugging:** Version 5.5 enhanced runtime generation, thanks in part to better collaboration with *Nanite* and newly introduced Partitioning options. However, in complex scenes featuring many generated objects, parameters such as generation distances, instance types, and object details still require careful tuning.

- **New Features in 5.5:** Unreal Engine 5.5 introduced **Grammar** and

Procedural Geometry systems, offering advanced tools for procedural rule definition and geometry generation. We discuss these functionalities in later sections of this document.

Extending Logic and Implementing Custom Features

Although the PCG Graph covers most common scenarios, there are instances where extending the logic is appropriate:

- **Blueprint Integration:** Within the PCG Graph, special nodes (*Blueprint Injection*) can execute custom Blueprint logic, enabling any functionality we might need. Examples of custom blueprint is provided in section: 3.4.1.
- **C++ Code:** For highly demanding operations, complex logic, or optimization, we can create custom C++ classes that extend the existing PCG modules. More technical details and example will be provided in section: 4.2.7.

3.2 Procedural Meshes and Grammar in UE 5.5

This section discusses the enhanced tools in Unreal Engine 5.5 that integrate Procedural Content Generation (PCG) with advanced 3D geometry editing or creation (often referred to as *Procedural Meshes*) and with *Grammar-based* (rule-driven) systems. The main goal is to simplify the development of dynamic, rule-defined objects or entire scenes without the need for manual modeling.

Procedural Meshes.

- Tools for **creating or modifying 3D models directly in the engine** – for instance, generating non-standard shapes, adjusting vertices, or applying procedural noise.
- In UE 5.5, one might implement a simple mesh-generation process using the *Geometry Script Plugin* in combination with Blueprint nodes. For example:

  ```
   [GenerateBaseMesh] -> SubdivideMesh(2x) -> ApplyNoise(Amplitude=20)
  -> PlaceUVs -> [FinalizeMesh]
  ```

  This rule-based pipeline first creates a base mesh, then subdivides it twice for additional detail, applies a noise modifier to achieve a natural, irregular surface, and finally assigns UV coordinates before finalizing the mesh. Each of these steps can be controlled via Blueprint logic or PCG Graph nodes.

  Additional details on implementing this workflow can be found in official documentation [13].

Grammar-based Approaches.

- These approaches draw on *shape grammars* or L-systems, where **rule sets define the placement** and transformation of components (e.g., walls, doors, or tree branches).
- In UE 5.5, such logic can be implemented within a *PCG Graph* using the *Execute Blueprint* node. For instance, you might define a simple rule that says:

  ```
  Place two walls, then place a door.  Next, again place
  two walls, then place a window.  Repeat this sequence
  until there is no more available space.
  ```

  By applying this rule repeatedly, the PCG Graph can automatically generate a layout where walls, doors, and windows follow the specified pattern.
- **Iterative development**: Grammar-based systems grow quickly in complexity, so it is advisable to start with simpler rules (e.g., generating basic walls) and gradually add more (windows, doors, or specialized roof shapes).
- For more details on using shape grammar with PCG in Unreal Engine, refer to official documentation [14] or watch those tutorial videos [15], [16].

Although official documentation for these features remains somewhat limited. While still evolving, these procedural mesh and grammar-based features in UE 5.5 hint at powerful, flexible pipelines for creating complex worlds. Early adopters often rely on custom Blueprint logic, partial documentation, and experiments shared in community channels (forums, Discord, GitHub). For large-scale projects or intricate designs, continuing to refine these tools, especially in combination with PCG, can significantly reduce manual work and foster rapid iteration on asset layouts.

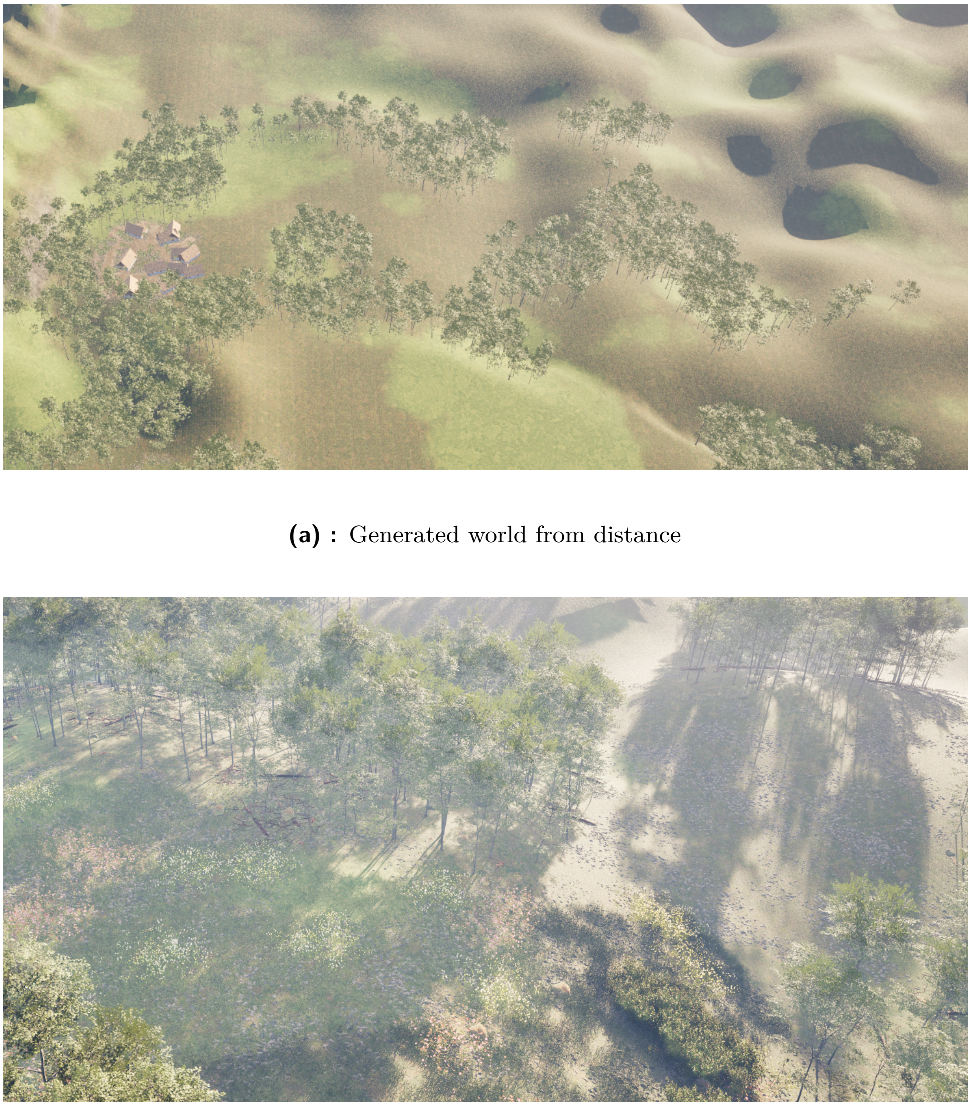
3.3 Optimization

Effective optimization is crucial for maintaining performance in procedurally generated worlds. Whether our project involves vast landscapes, dense forests, or large urban scenes, the following techniques help ensure smooth gameplay and efficient resource usage.

Hierarchical Generation and Partitioning.

- **Hierarchical Generation:** This approach splits content generation into smaller segments and processes them hierarchically (one by one based on distance from player). By using *World Partition*, we divide the game world into chunks, and the PCG Graph generates content only for the currently loaded chunks. Although node logic can run in parallel, *spawning meshes* must still occur on the main thread, which can cause brief lags when many meshes spawn at once.

- **Partitioning (is partitioned = true):** Enabling this feature allows the PCG Graph to execute in *sections* (partitions). If we also set `Generation Trigger` to `Generated at Runtime`, objects can be generated or unloaded dynamically during gameplay a crucial feature for **infinite open worlds**.



(a) : Generated world from distance

(b) : Generated world from upclose

Figure 3.7: Hierarchical generation with World Partition: terrain is divided into chunks of varying sizes that stream in and out at different distances (e.g., grass appears only near the camera).

Draw Distance, Collisions, and LOD/Nanite. In addition to partitioning, several other optimizations help maintain high framerates and smooth gameplay:

- **Custom Render Distances:** We can control the visibility and spawning distance of objects by assigning them to different grid sizes. For instance, grass is generated using smaller grid cells and only appears close to the player, while large objects like trees or rocks can be spawned at greater distances using larger grids. This method improves performance by avoiding unnecessary generation or rendering of distant objects. When combined with `World Partition`, this system dynamically loads and processes only the relevant sections of the world.

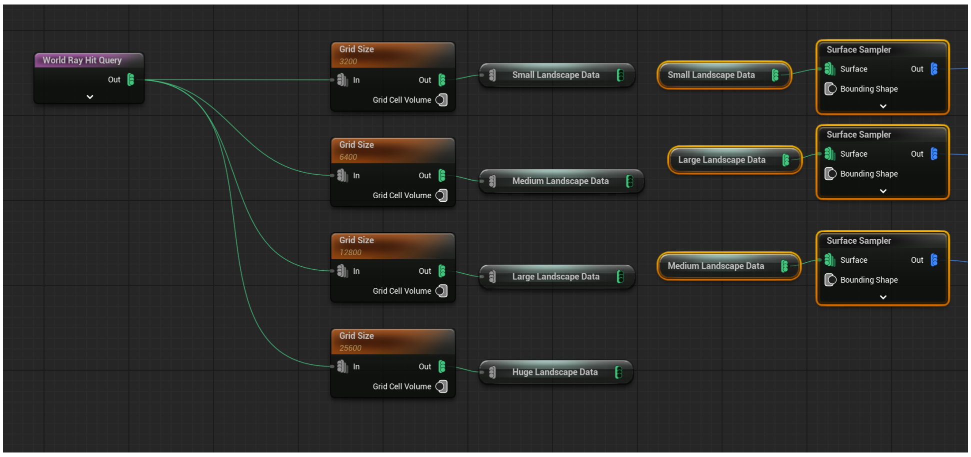


Figure 3.8: Example of dividing the world into multiple `Grid Size` layers for custom render distances. Smaller grids result in higher density and closer spawning, while larger grids are used for distant, low-detail elements. Enabled through the World Partition system.

- **Selective Collisions and Hitboxes:** We should limit collision checks to meshes where they are genuinely needed (e.g., rocks, buildings, interactive props). Decorative or distant meshes can have simplified or no collision bounds, reducing physics overhead.
- **Using LOD or Nanite:** By employing Level of Detail (LOD) models, we dynamically reduce polygon counts for distant objects. In Unreal Engine 5 and above, *Nanite* can render high-poly models efficiently without the need for multiple LODs. This keeps rendering resources focused where they are needed most.

Summary of PCG Graph Benefits

- **Visualization and Easy Modification:** The node-based interface quickly reveals how data flows and transforms, allowing immediate rule and parameter adjustments in the editor.

- **Modularity and Reusability:** PCG Subgraphs, hierarchical generation, and partitioning simplify work on large projects and enable reuse of previously developed components.
- **Extensibility and Performance:** Integration with Blueprints or C++ code, along with *Nanite* and *World Partition*, makes the PCG Graph suitable for generating large open worlds—both static and dynamic.

3.4 PCG Graphs in the Project

In this chapter we present the key PCG Graphs developed for our prototype. We begin with simple environmental graphs—meadows, forests, and mountains—then proceed to more complex systems such as roads, villages, and urban layouts. Each example highlights how node sequences translate design goals into procedural workflows.

All photorealistic models used in this work are free assets from Quixel on the Fab Marketplace [10] (To see all free meshes from Quixel you need to open Fab through Unreal Engine), and the low-poly models come from the "Low Poly Starter Pack" bundle, which was previously available free on Unreal Engine Marketplace [11].

3.4.1 Forests, Meadows, and Mountains

We start by creating basic environmental graphs that:

- Sample points on the terrain (surface or spline).
- Apply filters and noise to control density and variation.
- Transform point attributes (scale, rotation) for natural randomness.
- Spawn meshes (grass, trees, rocks) via static-mesh spawners.

These simplified graphs demonstrate the core PCG concepts—sampling, processing, and spawning—before moving on to more intricate generation scenarios.

Meadows

Meadows are open, grassy areas that provide a sense of calm and spaciousness. They are often characterized by a uniform grass layer interspersed with patches of flowers, small plants, and occasional rocks maybe some bushes. The goal is to create a natural yet visually appealing meadow that looks dynamic and alive without being overly cluttered.

Implementation.

- **Grass Layer:** We aim to make grass evenly distributed across the meadow. To achieve this, we can simply use the *Surface Sampler* node with a high probability density for points. Adding some randomness is always beneficial; this can be achieved by using *Transform Nodes* set to random rotation along the vertical axis, with slight random offsets and scaling. Next, we project our points onto the landscape to ensure they remain aligned with the terrain. Finally, we use the *Static Mesh Spawner* with several different types of grass to create variation.

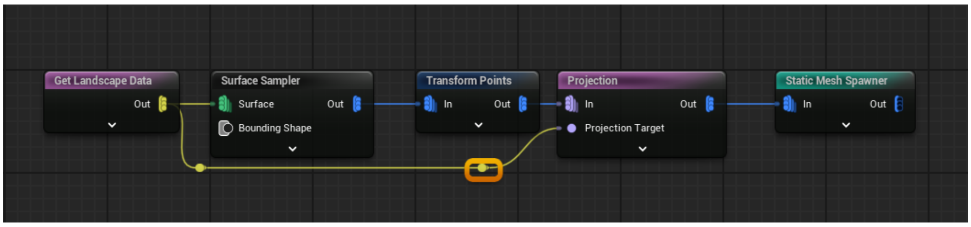


Figure 3.9: Showcase of entire grass logic

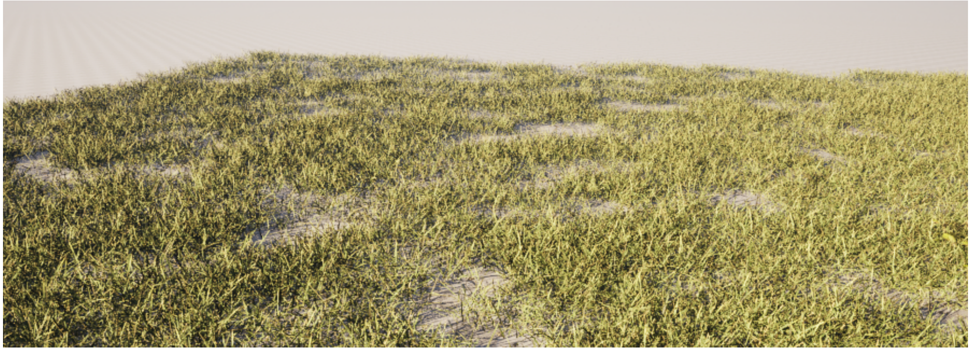


Figure 3.10: Showcase of grass spawning in scene

- **Flower Layer:** We aim to create clusters of flowers. To get this effect, we first generate starting points spaced far apart using the *Surface Sampler* node. Next, we create a grid of points at each of these locations, which forms the basis of a cluster. To add natural randomness, we use the *Transform Points* node to slightly offset the flowers within the cluster.

  At this stage, we have flower clusters, but to make them more visually interesting, we can vary the flower size within each cluster. Specifically, flowers on the edges can be made smaller, while those in the center appear larger. To achieve this, we use the *Distance* node to calculate the distance between the cluster's origin and each point, modifying the *density* attribute accordingly.

  Important: In the *Distance* node, make sure to enable the `Set Density` option—this stores the calculated distance into the point's `density` attribute, which is required for subsequent operations.

The updated density is then used by the *Scale by Density* node to scale the individual flowers based on their position in the cluster.

Finally, we use the *Static Mesh Spawner* to instantiate flowers at these points, completing the flower layer. To add versatility to the scene, we reuse this logic for two different flower species.

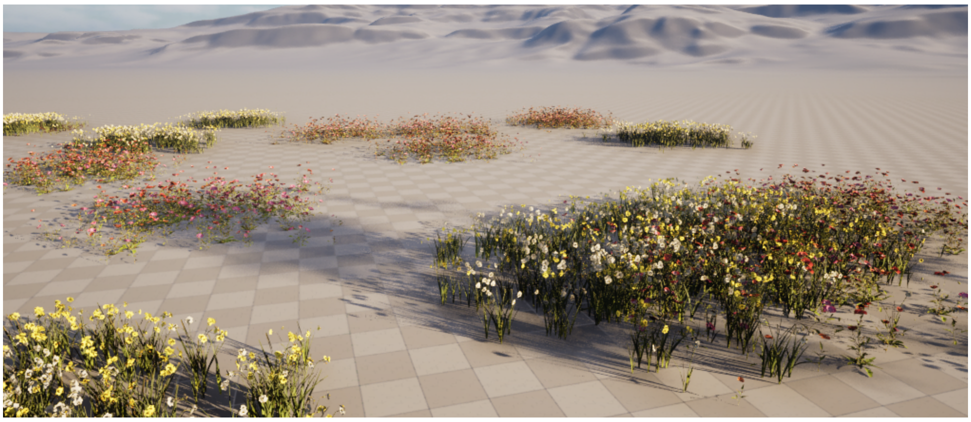


Figure 3.11: Showcase of flowers clusters being spawned

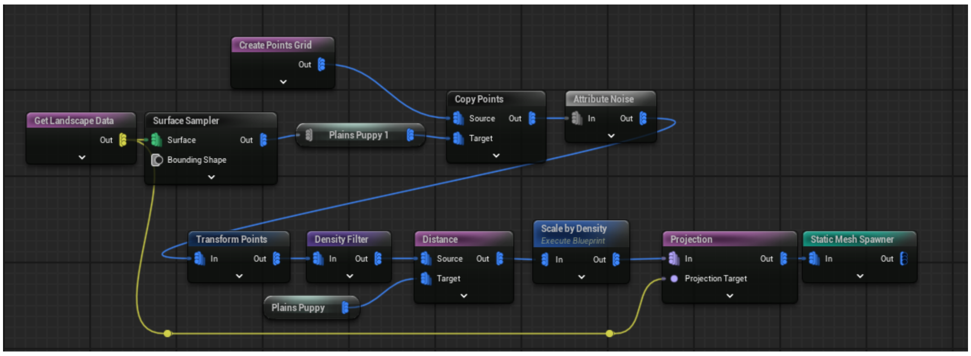


Figure 3.12: Showcase of flower spawning logic

- **Small Stones and Rocks:** To spawn stones, we can reuse the same cluster logic as we used for flowers. For larger rocks, there is no need to over-complicate the process; a simple *Surface Sampler* node with a very low density of points, combined with the *Transform Points* node to add some randomness, is ideal.

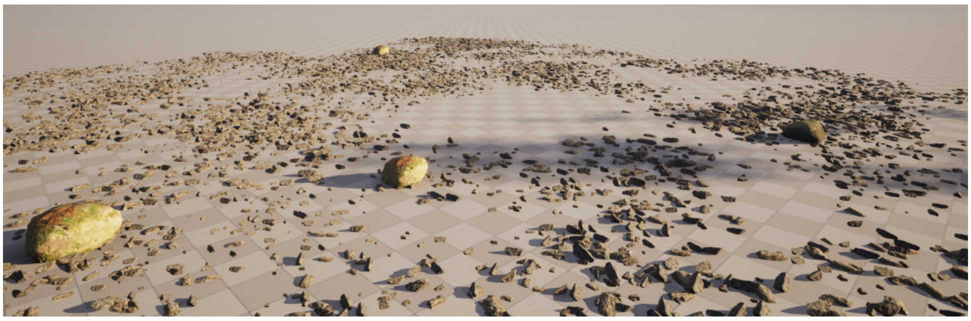


Figure 3.13: Showcase of stones being spawn

- **Bushes:** Once again, we can reuse the cluster logic from creating flowers. However, in this case, we ensure that bushes appear only occasionally by setting the *Surface Sampler* node to a very low probability of points. Additionally, I discovered that creating two clusters at the same location and slightly offsetting them still gives the appearance of a single bush while adding more unique shapes. To duplicate the starting point for the cluster, we use the *Duplicate Points* node and apply the *Transform Points* node for the offset.

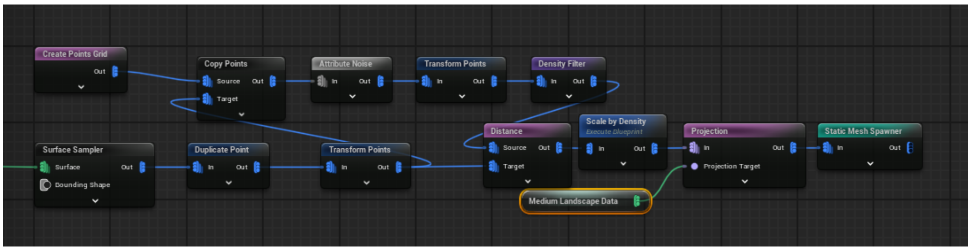


Figure 3.14: Showcase of bush spawning logic

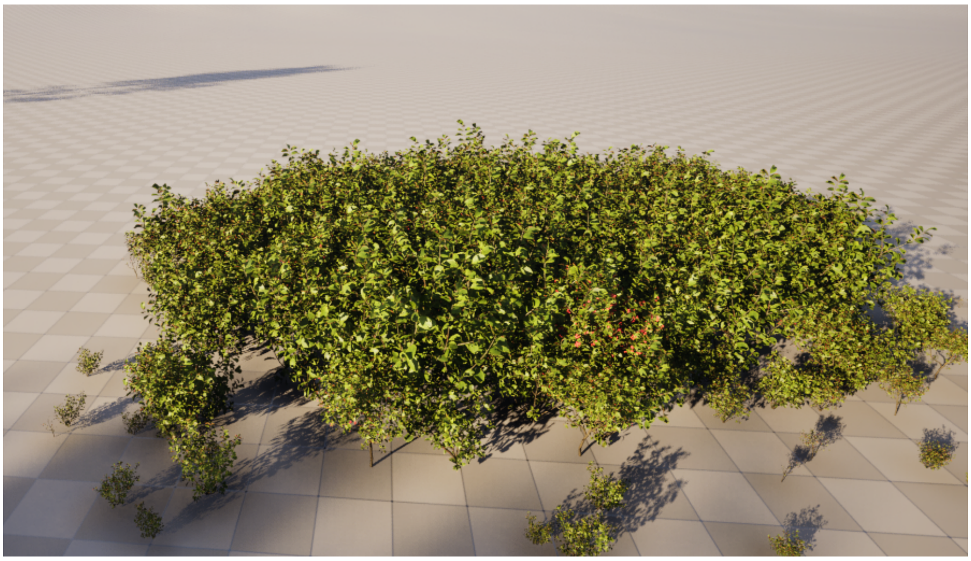


Figure 3.15: Showcase of bush

- **Putting It All Together:** Now we will combine all the individual layers into a single PCG graph to create dynamic and visually appealing meadows. The goal is to integrate the grass, flowers, stones, and bushes in a way that feels cohesive and natural while preserving the unique characteristics of each layer.

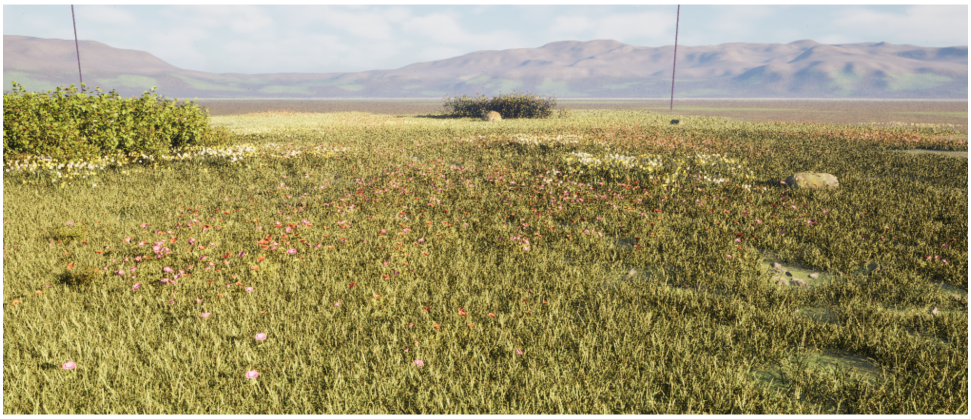


Figure 3.16: Final result of the Plains biome, populated with grass, flowers, bushes, and rocks.

This demonstrates the power of PCG graphs. With just a few meshes, we can create expansive grassy areas containing thousands of objects. While designing each type of PCG may initially require extensive experimentation and time, the resulting graphs can be reused across multiple projects, saving significant effort in the long term.

Forests

We aim to create a dense European forest featuring tall trees, scattered stones and rocks, fallen branches, and additional elements such as ferns and mushrooms.

We spawn multiple varieties of trees uniformly throughout the forest. Under the canopy, grass growth is more limited due to reduced sunlight. Occasionally, we include fallen trees (with branches) to diversify the environment visually. Mushrooms appear primarily in shaded, moist areas close to the trunks.

Implementation.

- **Tree Placement:** Smaller trees often appear at forest edges in nature; however, our forest lacks strict borders. To mimic this distribution, we use a *Surface Sampler* with a *Transform Points* node.

  To achieve a dense forest, we set a suitably high sampling density and apply randomness via *Transform Points*, ensuring `Absolute Rotation` is set to `true` so that trees remain upright. Finally, we use the *Static Mesh Spawner* to place a small selection of different tree meshes with varied weights, achieving natural diversity.

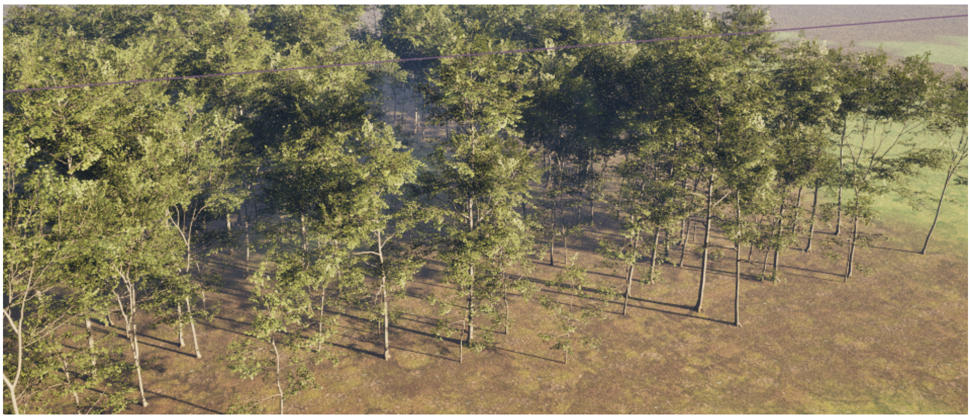


Figure 3.17: Tree layer example.

- **Ferns and Grass:** Ferns typically grow in clusters, which can be produced with the same "cluster logic" used for meadows.

  Grass requires a slightly more nuanced approach. First, we distribute points across the area with a *Surface Sampler*. Next, we use the *Distance* node to measure how close each point is to a tree. We then apply *Scale by Density* so grass appears smaller near trunks, simulating the reduced sunlight under a canopy.

  In the end, we randomly offset points again with *Transform Points* to avoid unnatural radial patterns around the trees.

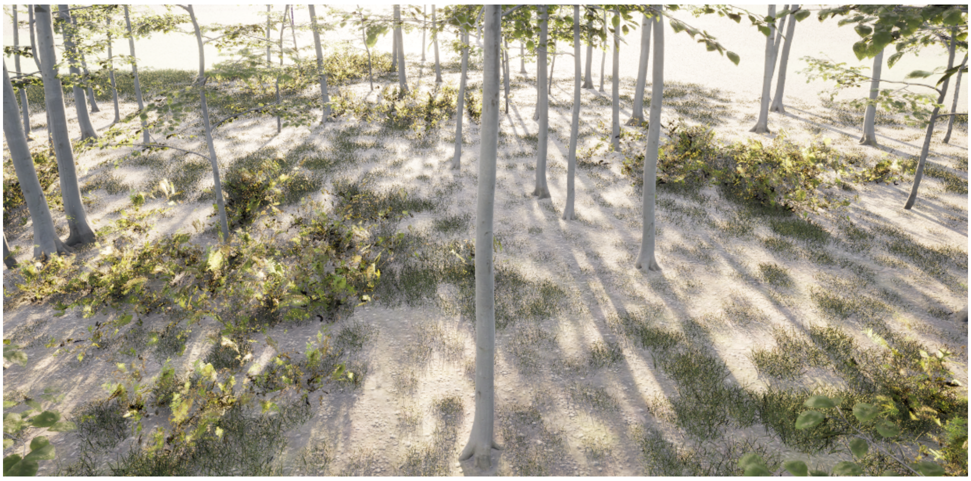


Figure 3.18: Grass and ferns in the forest environment.

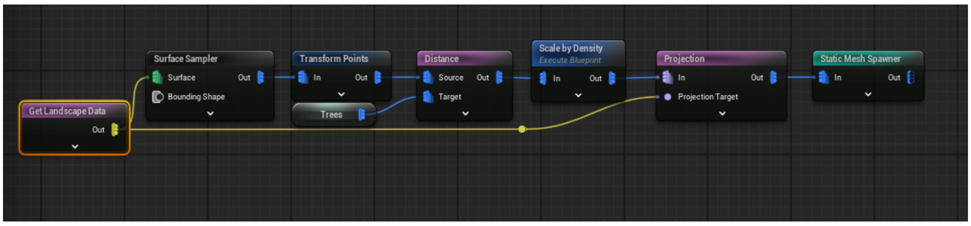


Figure 3.19: Code snippet for spawning grasss.

- **Fallen Trees:** To represent fallen logs, we use another *Surface Sampler* at a very low density. We also enable debugging on the *Projection* node and adjust `Point Extent` to match log length. Once these points are spawned, we place the fallen trees.

  Branches are then spawned around each log by generating a small grid of points (*Create Grid Points* + *Copy Points*) and applying *Transform Points* for randomness. We prevent overlap with the fallen tree using the *Distance* + *Density Filter* nodes to remove them, and finally scale branches near the log to enhance realism.

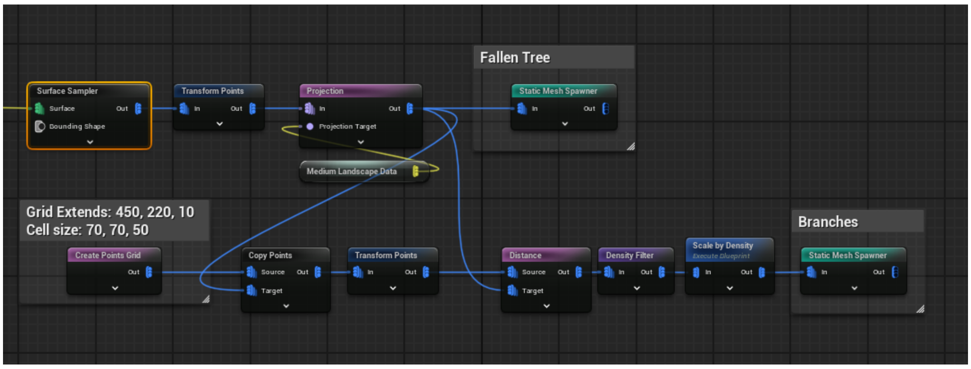


Figure 3.20: Sample logic for generating fallen trees.

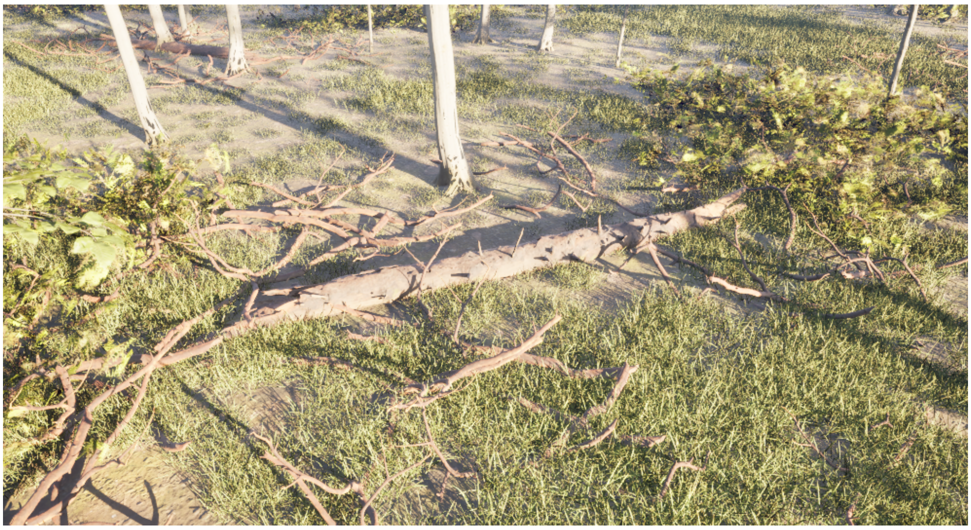
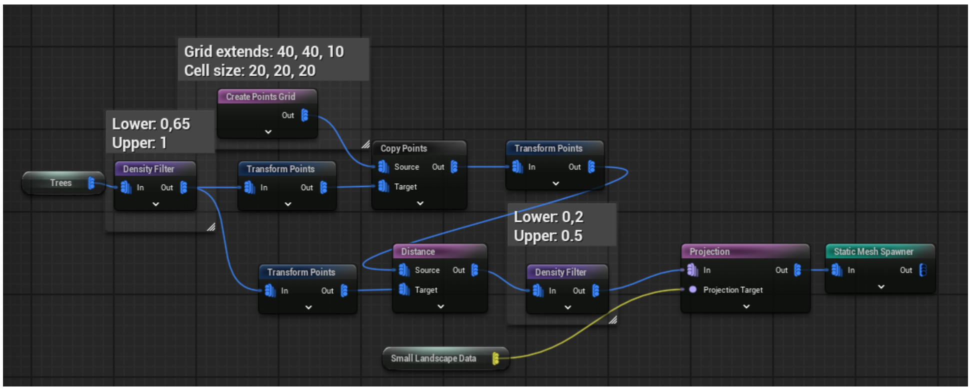


Figure 3.21: Fallen logs added to the forest.

- **Adding Mushrooms:** We utilize the random density attribute on existing tree points to decide where mushrooms appear. For instance, we might remove a portion of these points with the *Density Filter*. The remaining points are slightly shifted using *Transform Points* and expanded into a small grid (via *Create Grid Points* and *Copy Points*).

  Next, an *Attribute Filter* combined with *Distance* ensures mushrooms spawn only within a suitable radius of tree trunks. After finalizing the filtered points, we apply the *Static Mesh Spawner* for mushroom placement.



Grid extends: 40, 40, 10
Cell size: 20, 20, 20

Lower: 0,65
Upper: 1

Lower: 0,2
Upper: 0.5

Figure 3.22: Example blueprint for spawning mushrooms.

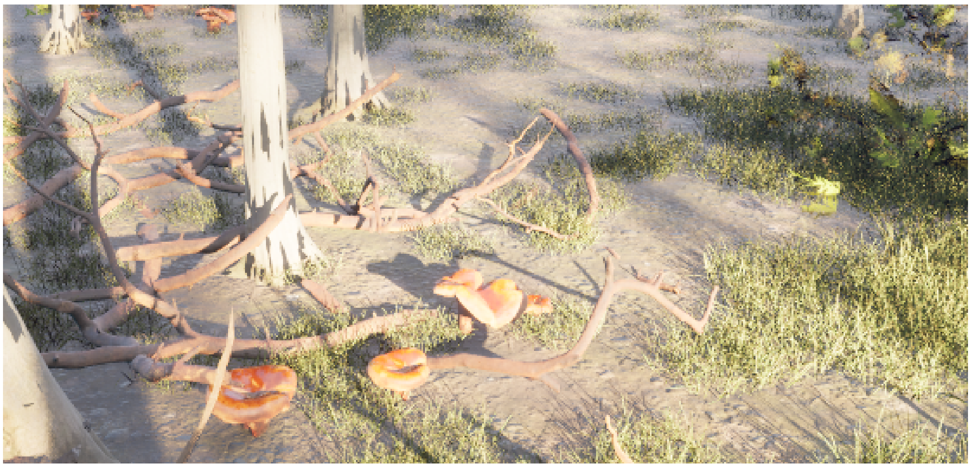


Figure 3.23: In-game mushroom distribution showcase.

- **Result:** We can reuse logic from meadows (e.g., spawning rocks and stones) to add more detail to forest floors. As a final step, we might apply a *Difference* node to ensure these objects do not overlap trees or fallen logs.

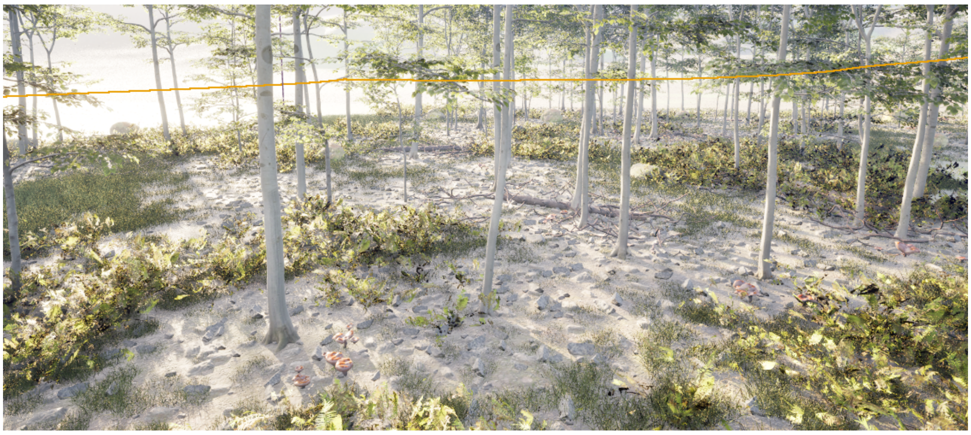


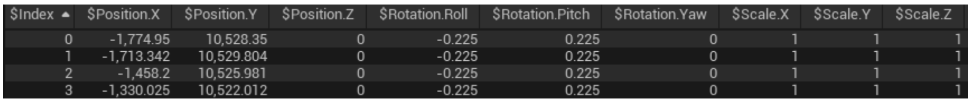
Figure 3.24: Example of a complete forest environment.

Once again, we manage to create a rich environment with just a few well-structured PCG rules.

Mountains

Focus on Height and Steepness. For mountainous terrain, our primary challenge is identifying areas based on altitude and slope, rather than repeating the standard point-distribution logic used in other biomes (e.g., forests or meadows). In this section, we concentrate on how to filter points by *height* and *normal direction* to determine where to spawn relevant assets such as rocks, sparse vegetation, or cliffs.

Filtering Based on Attributes. When we create or sample a point, Unreal Engine automatically assigns several attributes (see Figure 3.35). We can view these attributes at any node by pressing `A`; they appear at the bottom of the PCG graph window. Many of these attributes (prefixed with a dollar sign, `$`) are predefined, but we can also define custom ones.



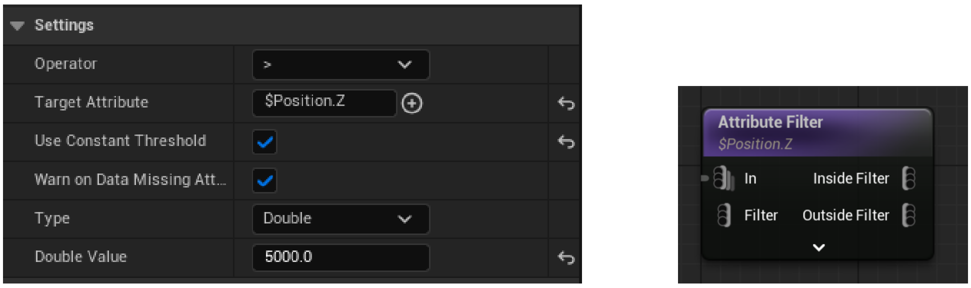
$Index ▲	$Position.X	$Position.Y	$Position.Z	$Rotation.Roll	$Rotation.Pitch	$Rotation.Yaw	$Scale.X	$Scale.Y	$Scale.Z
0	-1,774.95	10,528.35	0	-0.225	0.225	0	1	1	1
1	-1,713.342	10,529.804	0	-0.225	0.225	0	1	1	1
2	-1,458.2	10,525.981	0	-0.225	0.225	0	1	1	1
3	-1,330.025	10,522.012	0	-0.225	0.225	0	1	1	1

Figure 3.25: Predefined attributes assigned to points.

To distinguish mountain regions, we often use:

- **Height (Altitude):** Filter out points below a certain threshold to focus on higher elevations.
- **Normal:** Identify points that lie on steep slopes or cliffs.

We can perform these operations via the *Attribute Filter* node. Simply select the `Target Attribute` (e.g., `$height`) or type the name of a custom attribute. Figure 3.26 shows an example of filtering points based on height.



Settings

Operator	>
Target Attribute	$Position.Z
Use Constant Threshold	☑
Warn on Data Missing Att...	☑
Type	Double
Double Value	5000.0

(a) : Filtering points by height, settings.

(b) : Attribute filter node.

Figure 3.26: Example how to divide points based on height with threshold being 5000

Custom Node for Calculating Steepness from Normal. Although a predefined `$steepness` attribute exists, it serves a different purpose. Instead, we can create a new *Execute Blueprint* node to measure slope more precisely.

1. In the PCG Graph, add an *Execute Blueprint* node.
2. In the `Blueprint Element Type` field, search for `BoundsModifier`. Locate its Blueprint and duplicate it (e.g., "Up Vector to Density"), then open the duplicated Blueprint.
3. We recommend reviewing the complete `BoundsModifier` Blueprint that you just duplicated.

4. In the `PointLoopBody` function, remove the default logic and replace it with:

   a. Break the point data to access its transform and rotation.

   b. Use *Get Up Vector* to extract the vector perpendicular to the surface.

   c. Assign the `Z` component of that up vector to the point's `density`. Optionally square this value to emphasize steepness.

Figure 3.27 illustrates this custom Blueprint logic.

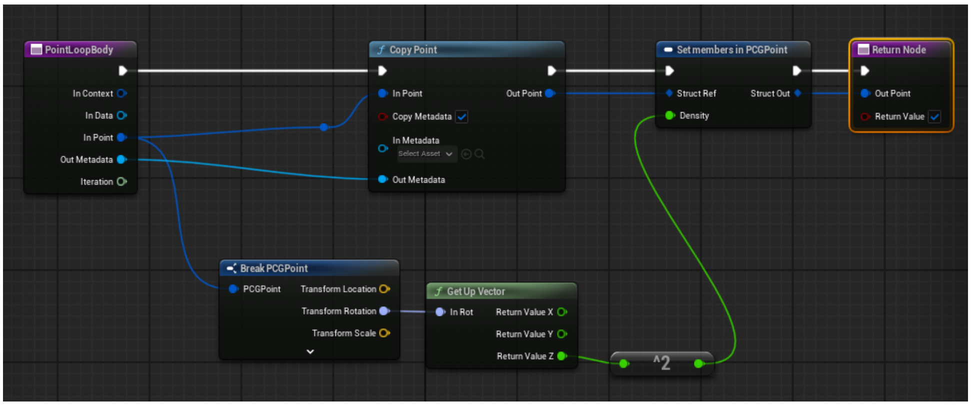


Figure 3.27: Blueprint logic for converting an up vector to density.

By modifying the `density` attribute according to slope, we can easily visualize or filter steep areas in the PCG Graph. For debugging, enabling color visualization (`Show Debug` in the PCG Graph) reveals how density correlates with slope (see Figure 3.28).

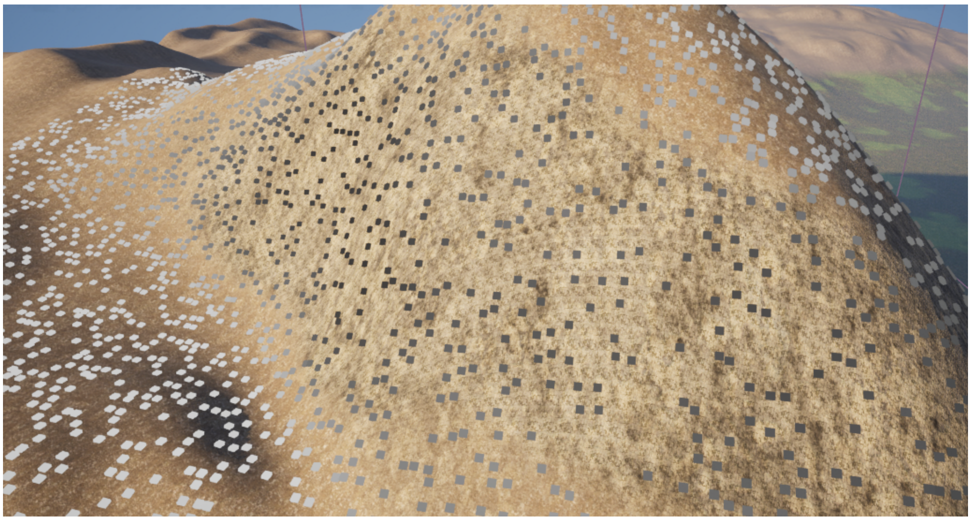


Figure 3.28: Visualization of steepness values assigned to points.

Applying the Filters. Once we have robust height and steepness metrics, we can decide whether (and what) to spawn on each point:

- **Steep Slopes:** May host only rocks or no vegetation at all.
- **High Altitude:** Could have sparse trees or shrubs.
- **Gentler Areas:** Transition zones where we can blend forest or meadow logic.

This targeted filtering enables dynamic biome transitions. We can, for instance, spawn different static meshes, change foliage density, or apply custom materials based on altitude or slope, thus forming more convincing mountain landscapes without redoing standard distribution steps.

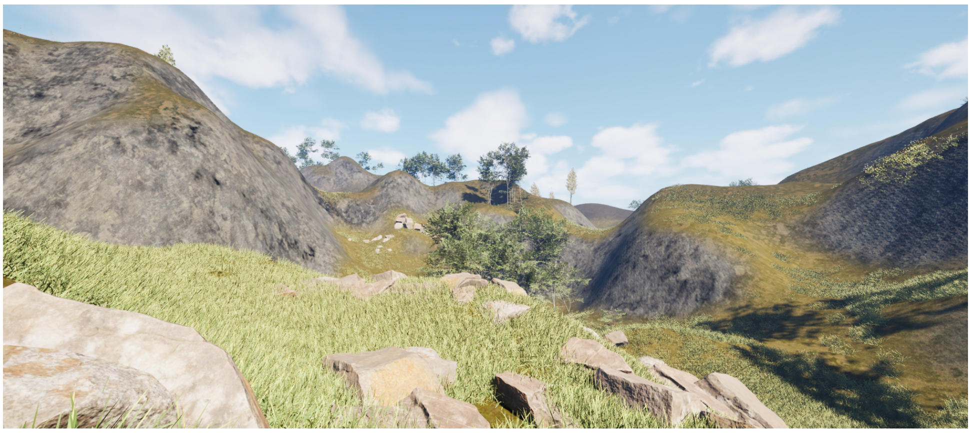


Figure 3.29: PCG for mountains.

3.4.2 Roads and Urban Structures

This section explains how we generate roads, houses, and villages using PCG graphs, which in this context become significantly more complex. Rather than offering a comprehensive, step-by-step tutorial for every scenario, we provide references to relevant external guides where available or you can download our entire project from git [17] for topics that lack existing resources.

Roads and Paths

In Unreal Engine, there is already a simple logic to create paths, as shown in *3.30a*. Unfortunately, this method is limited to the editor and is not usable within PCG graphs, requiring us to develop custom logic. While we can use tiles to create roads, this approach does not support arbitrary shapes. Ideally, we would have a tool capable of generating visually appealing roads along a spline, with the spline itself being procedurally generated in a Blueprint.

To generate splines procedurally, there are several methods:

- **Noise-based Paths:** Use noise to create natural, winding paths.

- **Flow Field Methods:** Simulate water flow or erosion patterns to define realistic path trajectories.
- **A* or Pathfinding:** Create roads between defined start and end points, avoiding obstacles via pathfinding algorithms.
- **Anchor-Based Splines via PCG:** Approach used in this project. Generate a set of anchor (control) points directly with PCG logic (e.g., distributing them based on terrain conditions or random noise). These points then form the spline's control vertices, allowing roads or paths to adapt to local constraints. This approach can partially randomize road layout while still conforming to desired terrain features.

Below are examples of different types of roads created using PCG.

Note: There is no official documentation or tutorial on creating PCG-based roads I could find. If you are interested, you can explore the *PCG_Village_Creation* PCG graph file in my project, where I demonstrate the process of creating an entire village.

Cobblestone Roads. This is a type of road where PCG excels. Cobblestone roads can adapt to any terrain or shape, which would be labor-intensive to create manually by placing each brick individually. To achieve this:

- Densely sample the spline using a *Spline Sampler*.
- Create a grid of points for each brick using the *Point Grid* node.
- Use the PCG logic to spawn individual cobblestones at these points.

This method ensures flexibility and seamless integration with uneven terrain, as shown in Figure 3.30b.

Asphalt Roads. Creating asphalt roads is more challenging due to the need for continuous textures and the visibility of seams. At the time of implementation, geometry-based PCG was not supported, limiting available options. The simplest solution was to use a tile-based system:

- Combine different types of tiles to create a road.
- This works well for grid-patterned roads, such as those in towns.
- However, this approach struggles with uneven terrain and can appear repetitive.

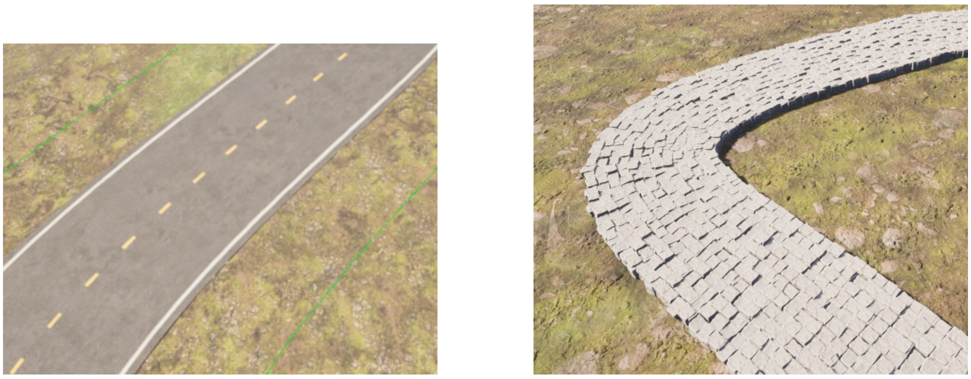
Dirt Roads. Dirt roads are simpler to create. One approach is to spawn decals (explained below) along a spline or create paths within procedural landscape te, but decals can alter the textures of objects placed on top of them. Instead, a more reliable method involves using path tiles:

- Dirt path tiles lack distinct lines, making connections less noticeable.
- However, this method may require a relatively flat surface for best results.

A dirt road created with decals and tiles is shown in Figures 3.31a and Figure 3.31b.

A decal. in Unreal Engine 5 is a material applied at runtime to project textures onto existing geometry (e.g., bullet holes, stains, or dirt). Under the hood, a Decal Actor uses a deferred decal pass: it renders the decal's material onto a frustum volume that overlaps scene meshes, blending its textures (color, normal, roughness, etc.) with the underlying surface based on projection parameters and the decal's sort order. This lets you add localized detail without modifying base materials or meshes.

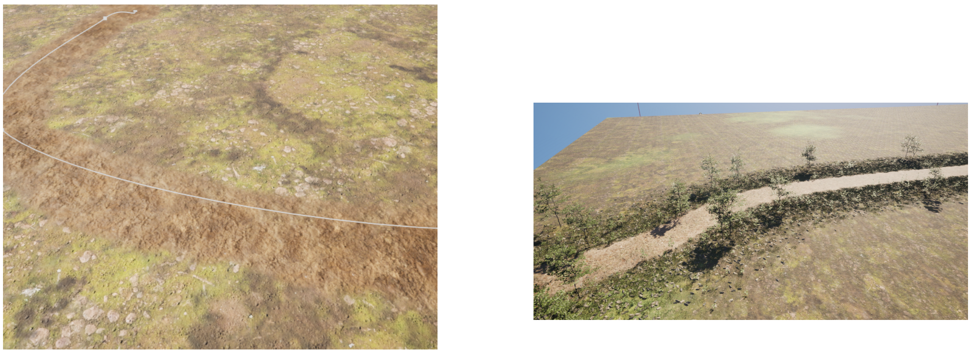
These examples demonstrate the potential and limitations of PCG when generating roads and paths. While some road types can be handled efficiently, others require workarounds or additional tools for more polished results.



(a) : Road created in landscape mode.

(b) : PCG cobblestone road.

Figure 3.30: Example of editor only (a) and PCG approach of creating roads (b).



(a) : Decal path.

(b) : Tile based path entirely created by PCG graph.

Figure 3.31: Two examples of creating road along spline using PCG.

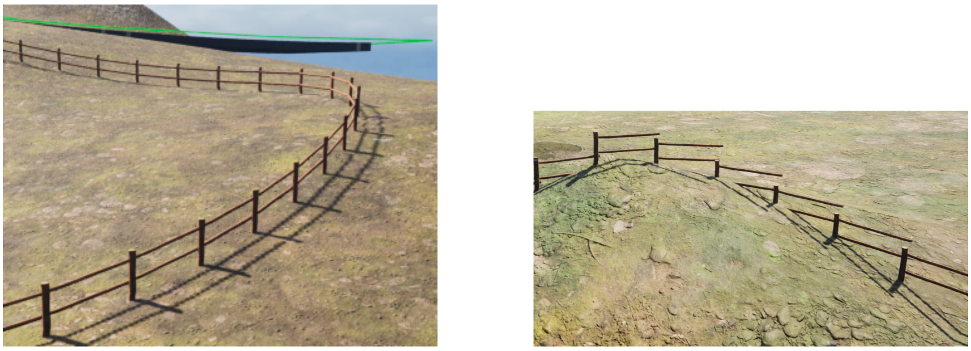
Fences and Buildings

In Unreal Engine, creating procedural fences and buildings using PCG is both promising and challenging. For fences, the idea is relatively straightforward and works well even on uneven terrain, while procedural buildings face more significant hurdles, particularly for non-standard shapes like pointed roofs.

Fences. Procedural fences can adapt seamlessly to terrain and splines, making them an excellent candidate for PCG. To create fences:

- Use a *Spline Sampler* to generate points along the spline.
- Spawn fence posts and rails using PCG logic, ensuring that the posts follow the terrain using projection nodes.
- Adjust connections between fence sections by implementing custom logic to align cross-pieces correctly.

For a more advanced tutorial, refer to my project (*PCG_Fence* and *BP_SplinePointOrientation* node), where I have extended the logic shown in a video tutorial by Unreal Engine [8]. The most challenging part of this process was ensuring proper alignment of cross-sections on complex terrain.

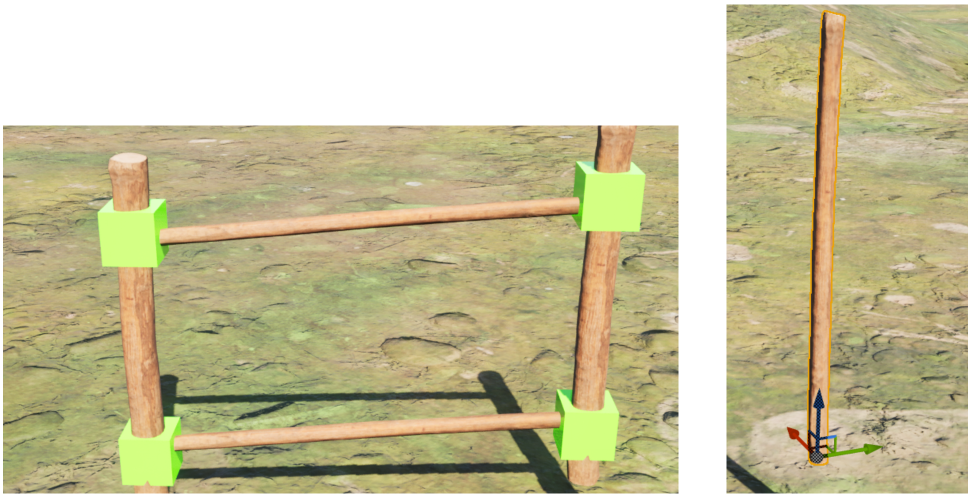


(a) : Fence being generated.

(b) : Problem with fence alignment.

Figure 3.32: Showcase of fence being correctly generated in (a) and problem of aligning fances on (b).

Fence Rail Alignment. The tutorial [8] does not address rail alignment, so we devised a solution. In the Static Mesh Editor, we add sockets at the base and top of each fence post ("BottomSocket" and "TopSocket," highlighted by green boxes in Figure 3.33a) and position the rail's pivot at one end (Figure 3.33b). At runtime, each rail segment is spawned at a post's BottomSocket (or TopSocket) and oriented toward the corresponding socket on the next post (Figure 3.33a). This ensures that rails connect cleanly and align precisely between adjacent posts.



(a) : Rails correctly spanning between sockets on adjacent posts.

(b) : Rail pivot at mesh end.

Figure 3.33: Correct fence generation: (a) rails spawned between matching sockets, (b) rail pivot placement ensures accurate positioning.

Buildings. Procedural building generation is more complex, especially when dealing with irregular shapes such as pointed roofs that must interact with arbitrary floor plans. We tried to make the roof system work for any house shape, but after few unsuccessful days, we had to move on. While creating modular apartment buildings is achievable with PCG, as demonstrated in a tutorial by Procedural Minds [12], more intricate structures require additional effort.

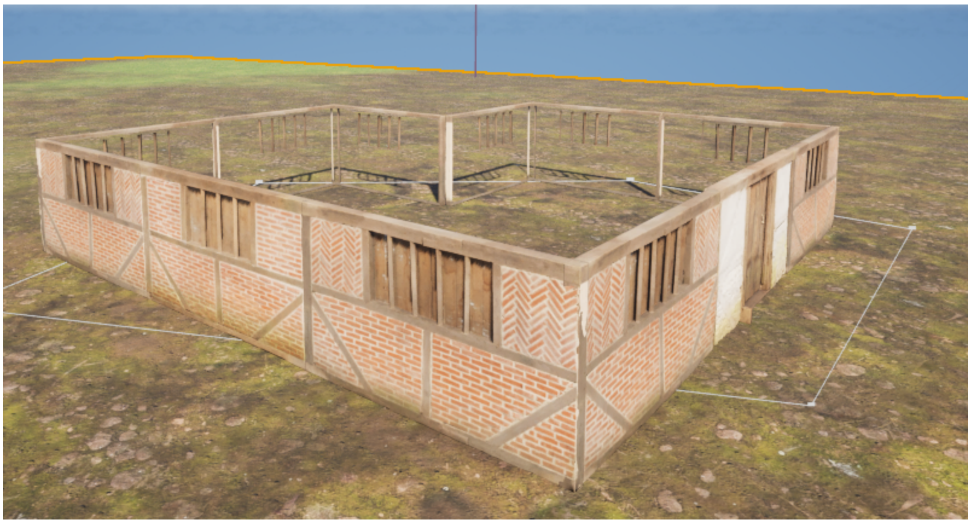


Figure 3.34: Showcase of fence being created along spline

Straight Skeleton. To solve our roof-generation challenge—deriving consistent ridge lines and slopes from arbitrary floor plans—we employ the straight skeleton algorithm (Angular Bisector Network). By offsetting each polygon edge inward at a constant rate and tracing where adjacent edges collide, it naturally produces the internal network of roof ridges and facets needed for any house footprint. A more detailed description of the algorithm and its implementation can be found in [29].

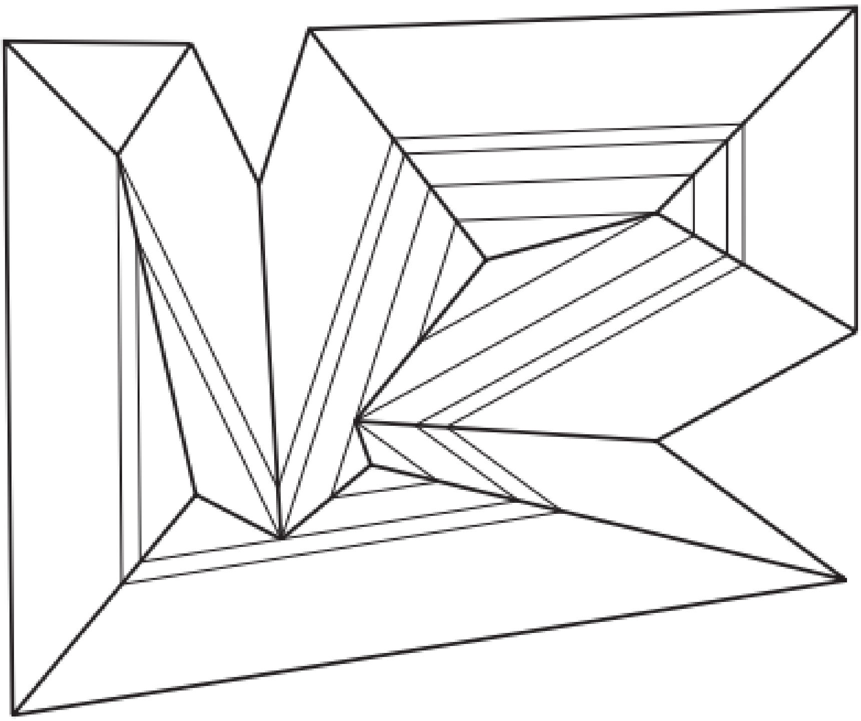


Figure 3.35: Geometry for irregular roof using Straight Skeleton

3.4.3 Village Generator

This section presents a procedural approach to generating a network of roads within a village. The method is based on creating an initial grid of intersections, refining it, and finally generating roads through an iterative connectivity process.

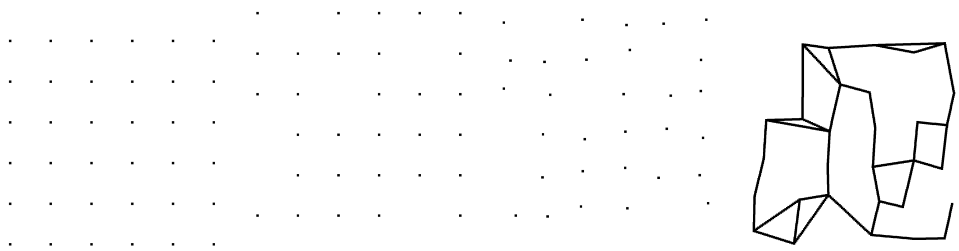
This road-network algorithm was developed entirely in isolation; only later did I realize it closely parallels the core method presented by Parish and Müller in their seminal SIGGRAPH paper on procedural city modeling [34].

Generating the Road Network. The road network is generated through the following steps:

1. **Grid Initialization:** A regular grid of points is created with a fixed spacing, representing road intersections (See Figure 3.36a).

2. **Random Point Removal:** Some intersections are randomly removed to introduce irregularity (See Figure 3.36b).

3. **Point Perturbation:** The remaining intersections are slightly displaced randomly to break the uniformity (See Figure 3.36c).
4. **Connecting Intersections:** Roads are generated by connecting points in such a way that all intersections remain accessible.
5. **Adding Additional Roads:** Extra connections are introduced to create a denser city layout (See Figure 3.36d).
6. **Applying Noise-based Displacement:** Perlin noise is used to displace the roads, making them more natural and winding.



(a) : Step 1, initialization.

(b) : Step 2, removing points randomly.

(c) : Step 3, moving points.

(d) : Step 5, creating roads.

Figure 3.36: Simplified walk-trough of main steps in creating road points for a village

Results. The final city road network follows an organic layout with a balanced mix of structure and randomness. The generated roads serve as the foundation for placing houses, green areas, and other village features. By leveraging Perlin noise, the roads appear naturally curved, avoiding artificial grid-like patterns commonly found in procedural generation approaches (See Figure 3.37).

The full implementation of the procedural road generation system can be found on GitHub [17]. in the file `ProceduralRoads.cpp`

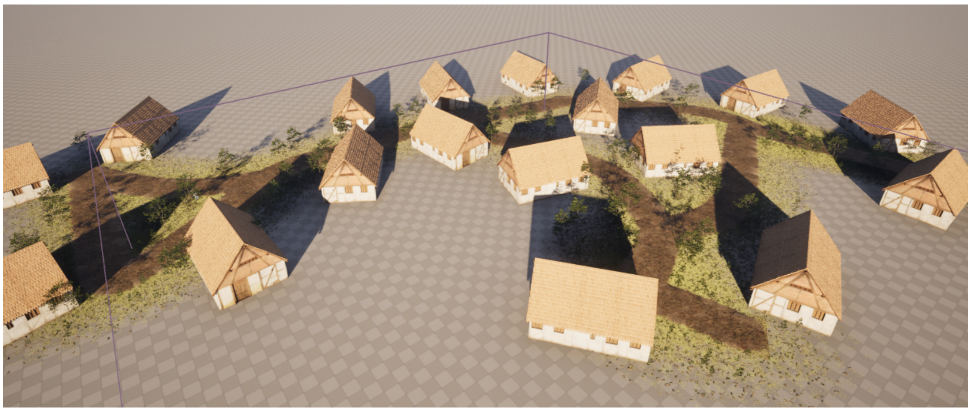


Figure 3.37: Final result of procedural Village

Chapter 4
Implementation

In this chapter we describe the design and construction of or an infinite, procedurally generated world in Unreal Engine 5. First, we present the approach to terrain creation and real-time rendering, including mesh data generation, noise-based height synthesis, and chunked streaming. We then explain how the raw terrain is segmented into distinct biomes—meadows, forests, mountains—using attribute filters on height and slope. Next, we cover the application of both manual landscape materials and rule-driven auto-materials to achieve seamless, non-repetitive texturing. Finally, we demonstrate how PCG Graphs are integrated to populate each biome with foliage, rocks, and other environmental actors, balancing visual richness with performance through World Partition, Nanite, and LOD optimizations.

4.1 Landscape Materials and Landscape Auto Materials

To ensure our infinite world is not only functional but also visually compelling, we begin with a discussion of landscape materials and the creation of a universal procedural material that adapts to any terrain.

In Unreal Engine, terrain texturing can be approached in two complementary ways:

Landscape Materials Traditional materials composed of one or more layers (e.g., grass, rock, sand, snow), manually painted onto the terrain using the Landscape Editor. This approach offers maximum artistic control over texture placement and is ideal for key areas that require precise attention.

Landscape Auto Materials A single, rule-driven material applied across the entire terrain. Within its material graph, parameters such as elevation, slope angle, surface curvature, or procedural noise determine where each texture appears. This method automates large-scale texturing and ensures consistency across chunks in an infinite world.

To build a universal procedural landscape material, we proceed as follows:

- Define core texture layers and create noise-driven masks for subtle surface variation.
- Configure height- and slope-based blending to achieve smooth transitions between biomes.
- Expose editable parameters (e.g., seam intensity, noise contrast, tile scale) for real-time tuning of the material's appearance.

This foundational material serves both manual and auto-material workflows, ensuring a harmonious look as the PCG system populates the environment.

4.1.1 Landscape Materials in General

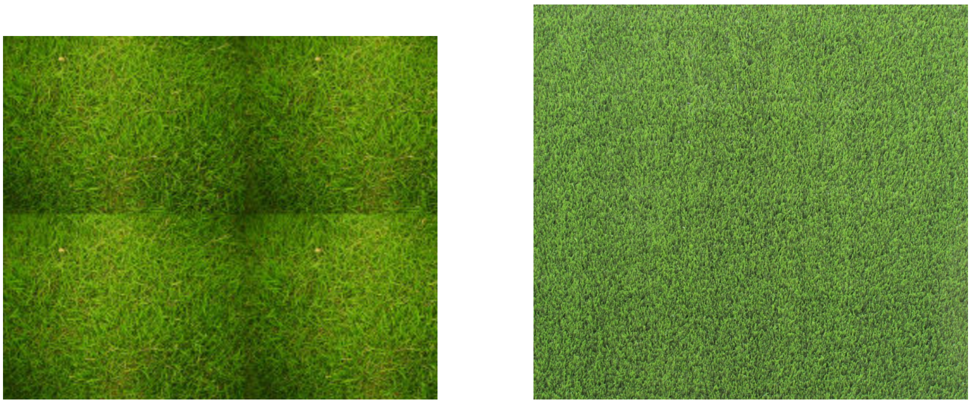
Landscape materials in Unreal Engine operate similarly to standard materials but are optimized for large-scale terrains. These materials use textures, which are mapped onto the surface of the terrain using UV coordinates (UV mapping defines how a 2D texture is applied to a 3D surface).

UV Mapping and Tiling

Textures are mapped onto the terrain through UV calculations, which determine how the image wraps around the surface. To cover large areas, textures are typically tiled, meaning they repeat across the terrain. While tiling is essential for performance and scalability, it introduces challenges such as noticeable seams and repetitive patterns.

Seamless Textures

These challenges can be mitigated through the use of seamlessly designed textures. Seamless textures are carefully designed to repeat without visible seams or artifacts, but creating them is non-trivial, as the human eye is particularly adept at detecting patterns and inconsistencies. Careful attention must be paid to blending edges and ensuring uniformity.



(a) : Seam grass texture tiles.

(b) : Great grass texture tiles.

Figure 4.1: Example of wrong and good tiling on grass texture.

Proposed Solution: Breaking Repetitive Patterns. To address the issue of repetitive patterns in terrain textures, a combination of texture blending and noise variation was implemented. The approach involves the following steps:

1. **Creating a Mask with Grayscale Noise:** A mask is generated by blending two grayscale noise textures of different scales (e.g., *T_Default_MicroVariation* from the Unreal Engine library). This mask is then used to modulate the base texture, introducing subtle variations across the terrain. The result is a more natural look that helps break up noticeable repetition in surface patterns. See Figure 4.3.

2. **Interpolating the Original Texture with a Color Variation:** The base texture is linearly blended with a color-modified version of itself, using a larger tiling scale. The blend factor (alpha) is driven by a noise map, creating smooth color transitions across the surface. This technique introduces color diversity over a wide area and effectively reduces visible tiling. See Figure 4.4.

3. **Dynamic Adjustments with Parameters:** Parameters such as tiling, alpha blending, and contrast can be adjusted to allow dynamic adjustments and fine-tuning based on specific terrain requirements.

This implementation is highly inspired by the techniques demonstrated in a video tutorial by Game Dev Academy [6].

Note: The base color for final texture is computed by multiplying the results from Figure 4.4 and Figure 4.3.

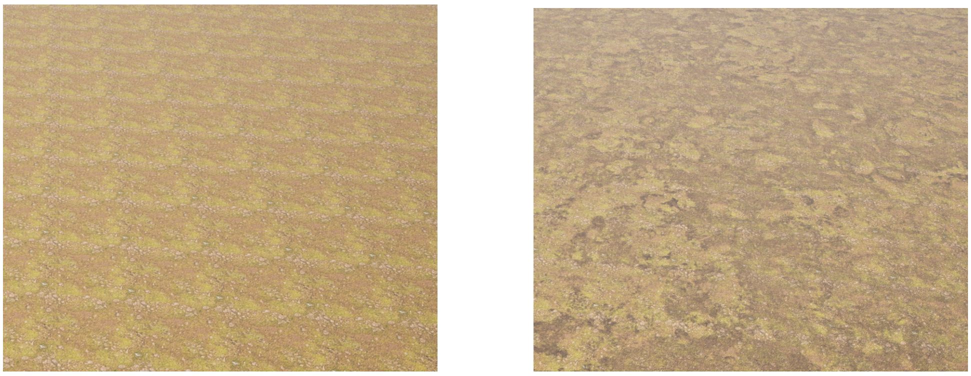


Figure 4.2: Comparison of incorrect and improved tiling. First illustrates repetitive tiling, while second presents the final result generated by the code from Figure 4.4 and Figure 4.3.

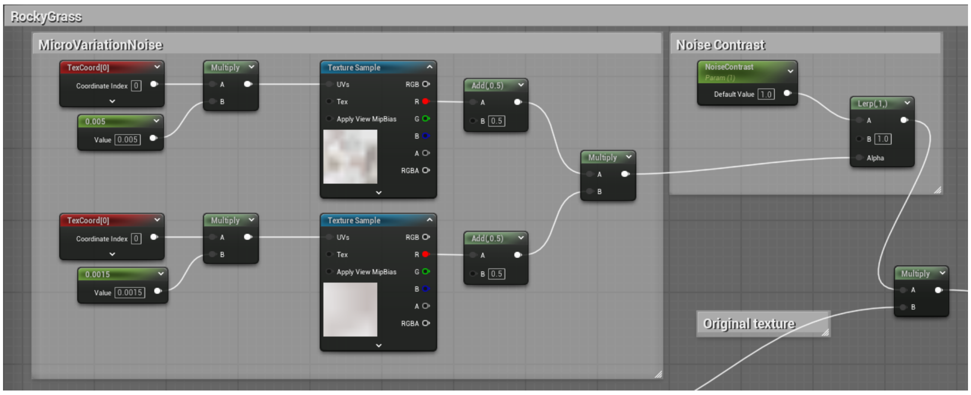


Figure 4.3: Reducing texture repetition by multiplying the base texture with grayscale noise maps of different scales.

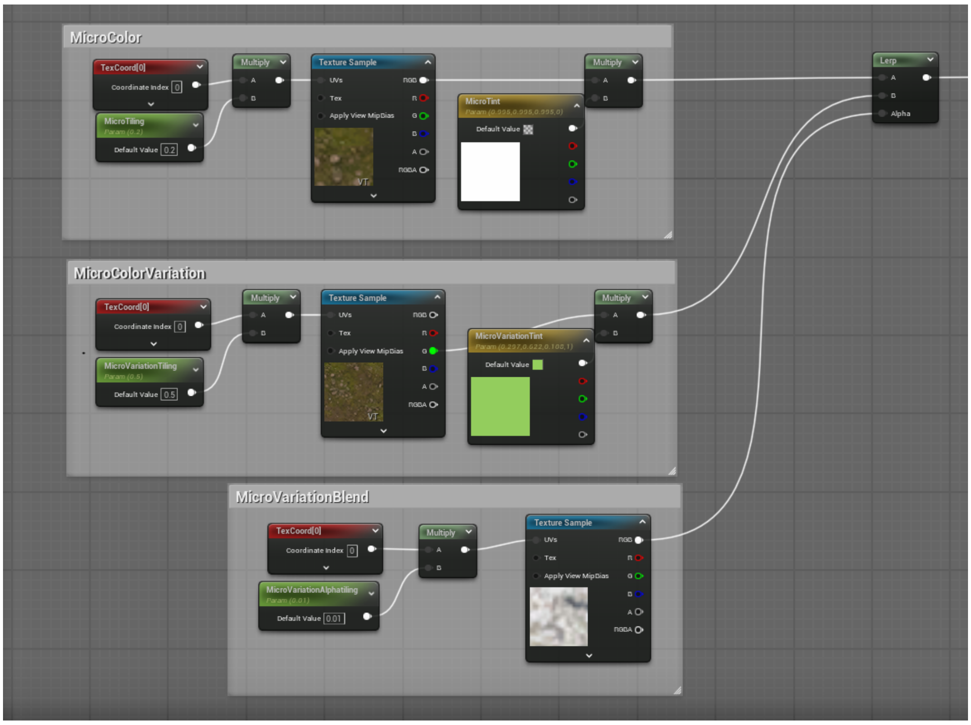


Figure 4.4: Blending the base texture with a color-variant version using a noise-driven alpha. This interpolation introduces large-scale variation and helps to hide obvious tiling.

Landscape Materials

Landscape materials in Unreal Engine function like standard materials but are optimized for large-scale terrain. They consist of multiple texture layers—such as grass, rock, sand, and snow—blended together to cover vast areas seamlessly. Core material nodes include:

- `Landscape Layer Blend` – for combining texture layers based on defined weights or masks.
- `Landscape Coordinate` – for controlling UV tiling, offset, and scale across the terrain.

Once the material graph is configured, you can apply the material to your landscape actor and begin painting directly in the editor.

Painting Landscape Materials in the Editor. To paint landscape materials in Unreal Engine, first define your texture layers and blend them with a `Landscape Layer Blend` node. Next, use the `Landscape Coordinate` node to adjust UV tiling and scale. Finally, switch to the Landscape Paint tool in the editor and assign each layer to a paint channel—now you can paint grass, rock, sand, or snow directly onto your terrain. For a detailed, step-by-step walkthrough, see the video tutorial by Aziel Arts [7].

Landscape Auto Materials

Landscape Auto Materials streamline terrain texturing by:

- Automatically assigning textures based on parameters such as height, slope, or terrain layers. For instance, we use noise maps to determine the type of grass placed on "grassy areas."
- Reducing manual effort, particularly for large or procedurally generated terrains.
- Recording the current texture (mapped onto a given UV) into the physical material for later use (e.g., specialized collision handling and usage as attribute in PCG graphs).
- Reflecting changes in real time after any modifications to the terrain.

While efficient, this approach may limit artistic flexibility compared to manually painted landscape materials. For a detailed tutorial on creating Landscape Auto Materials, refer to the video by Aziel Arts [9], which also inspired parts of this project.

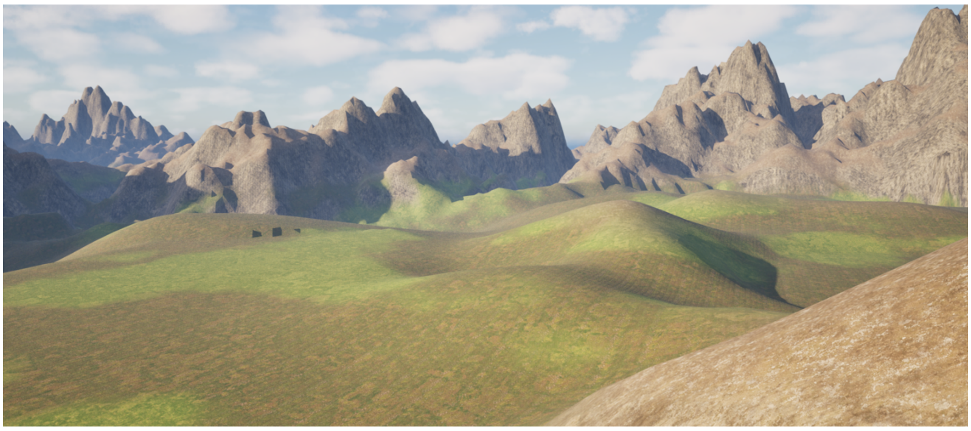


Figure 4.5: Example of the auto-material usage in large landscape.

4.2 Infinite Procedurally Generated World

In this section, we describe how an infinite, procedurally generated world is produced and rendered at runtime. We also show how to adapt both terrain texturing and our PCG graphs so that they fulfill the needs of a fully dynamic, infinite environment. All mesh parameters—vertex positions, normals, UVs, and other per-vertex attributes—are computed in C++ to fully leverage multithreaded processing and minimize game-thread overhead.

Initially, we prototyped terrain streaming using Unreal Engine's **Procedural Mesh Component**, which offers rapid iteration via Blueprint or C++ APIs [**?**]. However, as the number of active chunks grew, the per-chunk calls to `CreateMeshSection` introduced noticeable frame-rate hitches.

To achieve steady-state performance, we replaced this with a fully C++ pipeline: each terrain chunk is generated once as a **UStaticMesh** [24] asset (see `ProceduralWorldGenerator.cpp` in our GitHub repository [17]), thus eliminating the runtime overhead of the Procedural Mesh Component.

4.2.1 Generating and Computing Terrain Data

Before rendering either a Procedural Mesh or a Static Mesh, we generate all the mesh data needed to represent the terrain. In our system, the terrain is modeled as a continuous 3D surface composed of triangular polygons, whose union forms the landscape the player traverses.

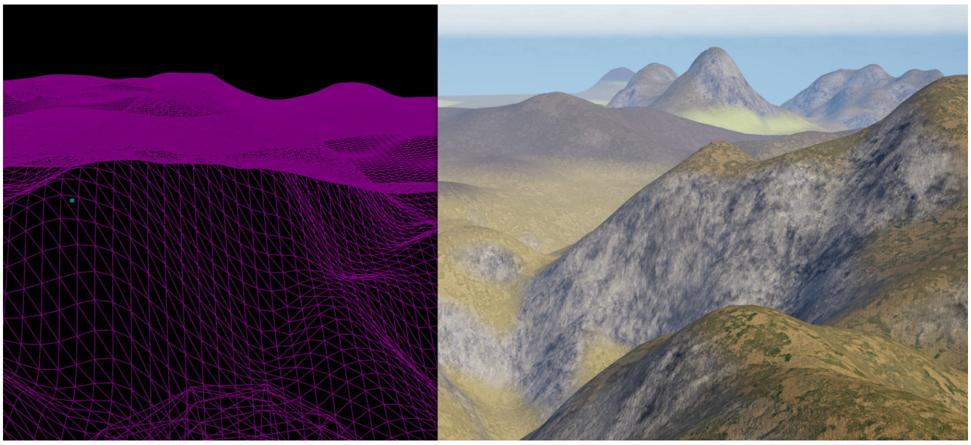


Figure 4.6: How triangular polygons combine to form the terrain mesh.

Computing Polygon Data

For each triangle, we compute and store three per-vertex attributes: vertex positions, normals, and UV coordinates.

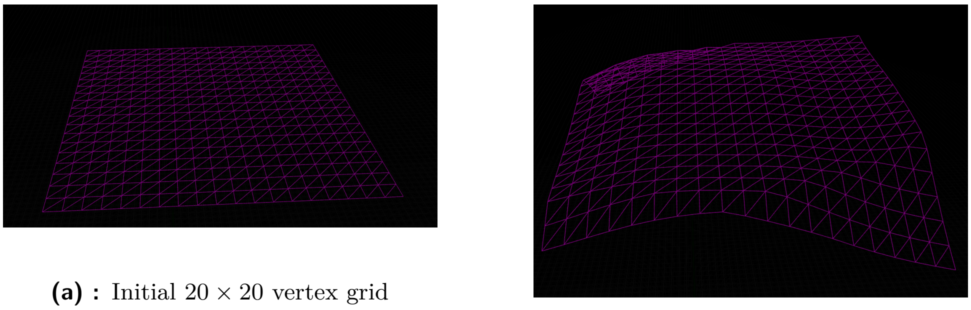
Vertex Positions Vertices lie on a uniform (x, y) grid (typically at $(i\,\Delta x,\ j\,\Delta y)$ for integers i, j). We compute the height z at each grid point by blending multiple continuous noise functions

$$n_k\colon \mathbb{R}^2 \to \mathbb{R}, \quad (x, y) \mapsto n_k(x, y),$$

(e.g., Perlin, Simplex, or Value noise) and optionally adding a constant offset c. Concretely,

$$z = w_c\, c + \sum_{k=1}^{K} w_k\, n_k(x, y),$$

where each weight w_k controls the amplitude of its corresponding layer. We will discuss noise maps in more detail in Section 4.2.1.



(a) : Initial 20 × 20 vertex grid

(b) : After applying noise to the z-coordinates

Figure 4.7: Vertices displaced along the z-axis by noise functions.

UV Coordinates For each vertex, we compute UV coordinates and save them in an array at the same index as the vertex positions. In our implementation, we tile the texture over the world grid by dividing the (x, y) position by a `TextureTileSize` parameter:

```
// Inside the vertex-generation loops:
float U = WorldX / TextureTileSize;
float V = WorldY / TextureTileSize;
UVs.Add(FVector2D(U, V));
```

- `WorldX`, `WorldY` are the world-space coordinates of the vertex (including any chunk offset).
- `TextureTileSize` defines how many Unreal units correspond to one full texture repetition.
- UV values outside $[0, 1]$ automatically wrap in the material, so tiles repeat seamlessly across chunks.

Normal Vectors We adopt smooth shading by averaging adjacent face normals using Unreal's `UKismetProceduralMeshLibrary::CalculateTangentsForMesh()` function, which:

1. Computes each triangle's face normal via the cross product of two edges.
2. Averages and normalizes those normals at each vertex.
3. Generates tangent vectors for correct normal-mapped lighting.

Border issue: Edge triangles on an $X \times Y$ grid lack a full set of neighbors, so their averaged normals would be incorrect. To address this, we:

1. Expand our iteration to $x \in [-1..X]$, $y \in [-1..Y]$, creating a $(X + 2) \times (Y + 2)$ vertex array.
2. Build triangles (inner and border) into an extended index list.

3. Call `CalculateTangentsForMesh` on the extended mesh.
4. Discard the outer-row/column triangles, preserving correct normals on the original $X \times Y$ region.

Function signature:

```
void UKismetProceduralMeshLibrary::CalculateTangentsForMesh(
    Vertices,           // TArray<FVector>
    ExtendedTriangles, // TArray<int32> index list
    UVs,                // TArray<FVector2D>
    Normals,            // TArray<FVector>
    Tangents            // TArray<FProcMeshTangent>
);
```

Noise Maps for Height Generation

A *noise map* is a 2D field of smoothly varying pseudo-random values—sampled by coordinate—that we use to modulate terrain height.

Procedurally generated terrain relies on these noise functions to introduce natural, pseudo-random variation in height while remaining fully reproducible via a fixed seed.

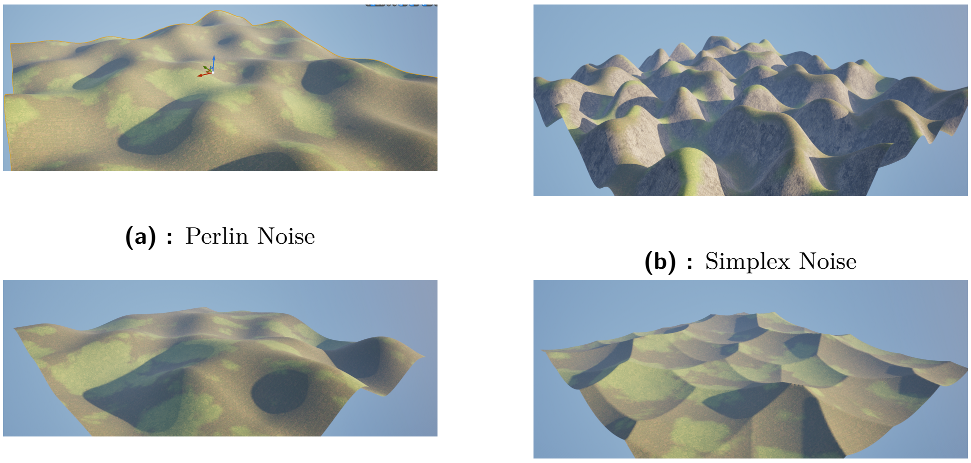
- **Motivation:** Noise provides controllable randomness, so the same seed always produces the same terrain.
- **Core Noise Types:**

  Perlin Noise [19]: smooth, gradient-based noise ideal for rolling hills.

  Simplex Noise [20]: computationally efficient, with fewer directional artifacts.

  Value Noise interpolates random grid values; simpler but less natural.

  Cellular Noise produces cell-like patterns, useful for stylized features.



(a) : Perlin Noise

(b) : Simplex Noise

(c) : Value Noise

(d) : Cellular Noise

Figure 4.8: Examples of different noise types used for terrain height.

- **Frequency & Amplitude:**
  - *Frequency* controls how often peaks and valleys occur per unit distance (higher frequency → finer detail).
  - *Amplitude* scales the output height range (higher amplitude → taller features).
  - Most noise functions return values in $[-1, 1]$; you multiply by the amplitude to map into world-space heights.
- **Fractal Noise (Octaves):** Rather than a single layer, we sum several "octaves" of noise at increasing frequencies and decreasing amplitudes:

  $$H(x, y) = \sum_{i=0}^{N-1} A_i \, n(f_i \, x, \, f_i \, y), \quad f_i = 2^i f_0, \; A_i = p^i A_0,$$

  where $p \in (0, 1)$ is the persistence. In FastNoiseLite:

  ```
  if (octaves > 1) {
    NoiseGen.SetFractalType(FastNoiseLite::FractalType_FBm);
    NoiseGen.SetFractalOctaves(octaves);
    NoiseGen.SetFractalLacunarity(2.0f); // freq  2  each octave
    NoiseGen.SetFractalGain(0.5f);       // amp  0 .5 each octave
  }
  ```

  By tuning:

  - the number of octaves N (e.g. 2–6) for desired complexity; $N = 3$ often provides a good trade-off between computation time and visual detail,
  - lacunarity (≈ 2.0) to control the frequency multiplier,
  - gain/persistence (0.4–0.7) to set the amplitude falloff,

you adjust terrain roughness versus performance.

- **3D Noise for Caves and Overhangs:** By sampling a 3D noise field $N(x, y, z)$ and applying an isosurface threshold, one can carve out caverns and natural overhangs [22]. This is not used in the present project.

- **Implementation with FastNoiseLite:** We use the FastNoiseLite library in C++ to generate noise at runtime [21]. Example:

```
FastNoiseLite Noise;
Noise.SetSeed(Seed);
Noise.SetNoiseType(FastNoiseLite::Simplex);
Noise.SetFrequency(0.002f);
float raw    = Noise.GetNoise(x, y); // [-1,1]
float height = raw * Amplitude;
```

Note: The complete source code for this chapter—including all C++ classes, PCG Graph assets, and materials—is available in the GitHub repository [17].

4.2.2 Rendering Terrain with the Procedural Mesh Component

To display our dynamically generated terrain, we need a system that takes raw polygon data and renders it at runtime while loading new chunks as the player moves and unloading those that fall out of view. Unreal Engine's **Procedural Mesh Component** exposes C++ APIs (and Blueprint nodes) for creating, updating, and destroying mesh sections on the fly.

Full source code for this project is available online [18].

Setup Overview

In this setup, each terrain chunk is represented by its own Blueprint actor (`BP_TerrainActor`). The workflow is as follows:

1. **C++ Actor:** Implement `AWorldGenerator` and expose key properties in the header with `UPROPERTY(EditAnywhere, BlueprintReadWrite)`, for example:

   - `Vector2D ChunkSizes` — Size of each chunk.
   - `TSubclassOf<AActor> TerrainActorClass` — the Blueprint class containing a `UProceduralMeshComponent`.
   - `UMaterialInterface* TerrainMaterial` — optional material to assign to each mesh.

2. **Instantiate Generator:** Create a Blueprint subclass of `AWorldGenerator`, configure defaults in the Details panel, and place one instance in your level.

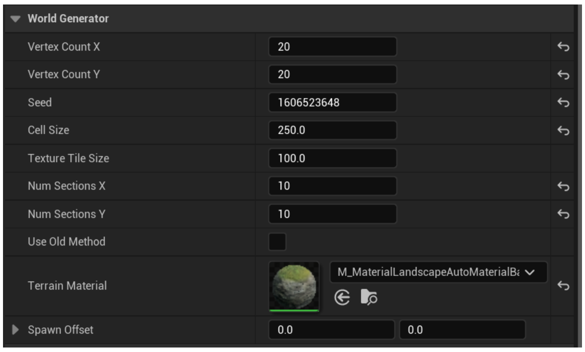
3. **Terrain Actor:** Create `BP_TerrainActor` with a `UProceduralMeshComponent`.

4. **Terrain Actor reference:** pass a reference of BP_TerrainActor to AWorldGenerator.

5. **Runtime Spawning:** On each tick, process up to ChunksPerTick chunks:

```
AActor* NewActor = GetWorld()->SpawnActor<AActor>(
    TerrainActorClass, Position, Params);

if (NewActor)
{
    // Create mesh section
    if (UProceduralMeshComponent* MeshComp =
        NewActor->FindComponentByClass<...>())
    {
        MeshComp->CreateMeshSection(
            0,
            MeshData->Vertices,
            MeshData->Triangles,
            MeshData->Normals,
            MeshData->UVs,
            MeshData->VertexColors,
            MeshData->Tangents,
            true
        );

        MeshComp->SetMaterial(0, TerrainMaterial);

    }

    // Track this chunk for later updates/removal
    ActiveChunks.Emplace(NewActor,
                         MeshData->SectionIndex,
                         MeshData->SectorPosition);
}
```

6. **Chunk Management:** We store each spawned actor and its section index in ActiveChunks, allowing us to clear or update individual chunks later (e.g. via ClearMeshSection or respawn).



World Generator

Vertex Count X	20
Vertex Count Y	20
Seed	1606523648
Cell Size	250.0
Texture Tile Size	100.0
Num Sections X	10
Num Sections Y	10
Use Old Method	□
Terrain Material	M_MaterialLandscapeAutoMaterialB
Spawn Offset	0.0	0.0

Figure 4.9: Details panel for `AWorldGenerator`, showing exposed parameters.

Throttled Chunk Submission

In `Tick()`, we process up to `ChunksPerTick` ready chunks per frame (we found 1 section/frame is ideal to maximize FPS while still loading chunks quickly enough):

```
void AWorldGenerator::ProcessRenderQueue(float DeltaTime)
{
    int32 Count = 0;
    Generating = true;
    while (Count < ChunksPerTick)
    {
        // chunk spawning logic
    }
    Generating = false;
}
```

By offloading data creation to worker threads and throttling section creation on the game thread, we maintain smooth, consistent frame rates. For the raw array-generation code, see Section 4.2.1.

Common Issues with Procedural Mesh Component

- **Array length mismatches:** All input arrays (`Vertices`, `Triangles`, `Normals`, etc.) must match expected sizes or `CreateMeshSection` will assert.

- **Duplicate indices:** Recreating a section under the same index without clearing it first may cause flicker or stalls.

- **Thread safety:** Only call `CreateMeshSection` (and other component APIs) on the game thread—Unreal Engine will not allow calls from background threads.

- **Memory leaks:** Always `delete` your `FMeshData` after use, and clear inactive sections to free GPU buffers.

Limitations of `UProceduralMeshComponent`

`UProceduralMeshComponent` provides only one extra per-vertex data channel—*vertex color* (4×8-bit)—and does not expose any UV channels. By contrast, `UStaticMesh` (and plugin `URealtimeMesh`) support up to eight UV sets (UV0–UV7) in addition to the standard vertex attributes (position, normal, tangent, color). In practice, UV0 is usually reserved for texture coordinates and UV1 for lightmaps, leaving UV2–UV7 (each two 16-bit floats) available for custom data.

4.2.3 Rendering Terrain with Static Meshes

While the *Procedural Mesh Component* provides a straightforward API for dynamic geometry, it only exposes a single extra data channel (vertex color) and does not support multiple UV sets, severely limiting the amount of per-vertex data available to the vertex shader. To work around these constraints, we implemented a pipeline that converts each chunk into a `UStaticMesh` asset. Unfortunately, the Static Mesh build library is only accessible within the Unreal Editor and cannot be invoked at runtime in packaged (shipping) builds. Consequently, runtime generation of `UStaticMesh` assets is not supported in shipped games without either pre-cooking the meshes or integrating a custom runtime mesh plugin.

4.2.4 Rendering Terrain with Static Meshes

While the *Procedural Mesh Component* provides a straightforward API for dynamic geometry, it only exposes a single extra data channel (vertex color) and does not support multiple UV sets, severely limiting the amount of per-vertex data available to the vertex shader. To work around these constraints, we implemented a pipeline that converts each chunk into a `UStaticMesh` asset. Unfortunately, the Static Mesh build library is only accessible within the Unreal Editor and cannot be invoked at runtime in packaged (shipping) builds. Consequently, runtime generation of `UStaticMesh` assets is not supported in shipped games without either pre-cooking the meshes or integrating a custom runtime mesh plugin.

We use the same `AWorldGenerator` actor (Section 4.2.2) and data-generation code (Section 4.2.1), but instead of feeding a `UProceduralMeshComponent`, we:

- Convert each `FMeshData` into a `UStaticMesh` using `CreateStaticMeshFromMeshData()`.
- Spawn an `AStaticMeshActor` and assign that mesh at the correct world position.
- Track spawned actors (or their mesh components) for later removal.

Key Steps in CreateStaticMeshFromMeshData

In Unreal Engine, a UStaticMesh is a fully optimized, GPU-friendly mesh asset. Internally, you build it by populating an editable FMeshDescription—which holds vertices, faces, UVs, normals, and other per-vertex attributes—before baking it into the final static mesh format. Our function CreateStaticMeshFromMeshData takes the runtime-generated FMeshData arrays, fills a FMeshDescription, and produces a ready-to-spawn UStaticMesh.

Rather than include every line here, the essential operations are:

1. **Instantiate a new UStaticMesh:**

   ```
   UStaticMesh* NewSM = NewObject<UStaticMesh>(this);
   if (!NewSM) return nullptr;
   ```

2. **Prepare the mesh description:**

   - Allocate one source model: NewSM->SetNumSourceModels(1);
   - Create a FMeshDescription and register FStaticMeshAttributes.

3. **Populate vertices and polygons:**

   ```
   // For each vertex: CreateVertex() + set its position
   // For each triangle: CreateVertexInstance(), set UV, normal, color
   MeshDescription.CreatePolygon(PolygonGroup, VertexInstanceIDs);
   ```

4. **Build and finalize the mesh:**

   ```
   NewSM->CreateMeshDescription(0, MeshDescription);
   NewSM->CommitMeshDescription(0);
   // Configure BuildSettings (disable recompute normals/tangents)
   NewSM->Build(true);
   ```

5. **Set up collision and bounds:**

   ```
   NewSM->CalculateExtendedBounds();
   NewSM->GetBodySetup()->CollisionTraceFlag = CTF_UseComplexAsSimple;
   ```

These steps convert the raw FMeshData buffers into a fully configured UStaticMesh asset at runtime, ready for spawning in the level. The full implementation is in WorldGenerator.cpp under CreateStaticMeshFromMeshData(FMeshData* MeshData) [18]. For details on the UStaticMesh class, see the Unreal Engine 5.5 API Reference [24].

Spawning and Removing Static Mesh Chunks

Once a UStaticMesh has been created, we need to spawn it into the world and later remove it when it is out of view.

Spawning a Static Mesh Chunk. Unreal Engine requires that you spawn actors on the game thread. For each ready `FMeshData* MD`:

1. **Spawn an `AStaticMeshActor` and attach the mesh:**

```
// Spawn actor (must be on game thread)
auto* SMActor = GetWorld()
    ->SpawnActor<AStaticMeshActor>();

// Configure its component
auto* SMC = SMActor->GetStaticMeshComponent();
SMC->SetMobility(EComponentMobility::Static);
SMC->SetStaticMesh(NewSM);
if (TerrainMaterial)
    SMC->SetMaterial(0, TerrainMaterial);

// We offset vertices when building the mesh, so we spawn
// at world origin
SMActor->SetActorLocation(FVector::ZeroVector);
// (Alternatively, you could leave vertices un-offset and
// spawn at MD->SectorPosition)
```

2. **Record it for culling:**

```
// Track for later removal
ActiveChunks.Add(FGeneratedChunk(SMC, MD->SectorPosition));
```

3. **Free the CPU data:**

```
// Clean up
delete MD;
```

Removing Distant Chunks. Each frame in `Tick()`, we compute the distance from the player and destroy any chunk beyond `MaxDistance`:

```
for (int i = ActiveChunks.Num() - 1; i >= 0; --i)
{
    float Dist2 = FVector::DistSquared(
        FVector(ActiveChunks[i].SectorPosition, 0),
        PlayerLocation);
    if (Dist2 > FMath::Square(MaxDistance))
    {
        // Destroy the actor owning this component
        ActiveChunks[i].MeshComponent->GetOwner()->Destroy();
        ActiveChunks.RemoveAt(i);
    }
}
```

This ensures that only nearby static-mesh chunks remain in the scene, keeping memory and draw calls under control.

Issues of UStatiMesh

- **Lightmap UV Preservation:** Unreal Engine will regenerate lightmap UVs if `TexCoord[1]` is modified or left uninitialized at runtime. Reserve `TexCoord[1]` exclusively for your lightmap coordinates—and never overwrite it—to avoid costly UV rebuilds.
- **Editor-Only API:** The static mesh build functionality is exposed only in the Unreal Editor. Consequently, `CreateStaticMeshFromMeshData()` cannot be called in packaged (shipping) builds without integrating a custom runtime mesh plugin or pre-cooking assets.

Alternative: `URealtimeMesh` Plugin

As an alternative to the editor-only static mesh build pipeline, the `URealtimeMesh` plugin enables runtime mesh creation in shipped builds and provides full control over vertex formats and updates. However, its current release is not fully compatible with Unreal Engine 5.5, and we were unable to fully integrate it—particularly for generating collision geometry at runtime.

4.2.5 Dividing Terrain into Biomes

In large, varied worlds it is essential to group regions with similar environmental characteristics into *biomes*. A **biome** is a contiguous area classified by its dominant terrain features, climate, vegetation type, and appearance—examples include forests, mountains, plains, deserts, and wetlands.

Why divide into biomes?

- *Visual variety and realism:* Each biome can use its own textures, foliage rules, and object placement logic to evoke a distinct look and feel.
- *Modular PCG logic:* By tagging terrain points with a biome identifier, we can apply different PCG graphs or subgraphs (e.g. forest spawner vs. mountain spawner) without mixing rules.
- *Modular Terrain logic:* By tagging terrain points with a biome identifier, we can create its own local generation rules.
- *Gameplay mechanics:* Biomes often drive game systems (weather, AI behavior, resource distribution), so a clear biome division supports higher-level design.

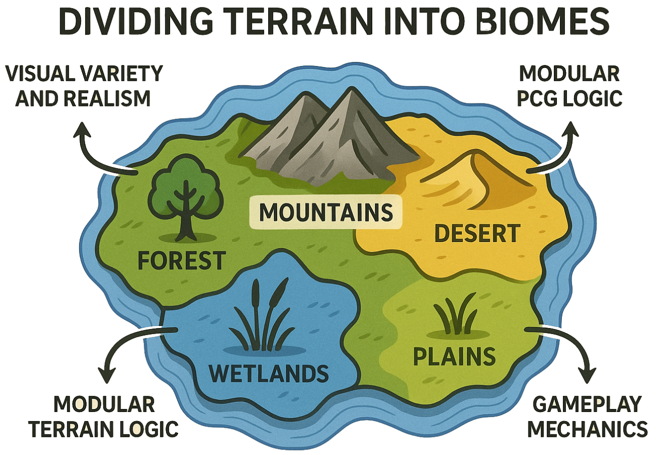


Figure 4.10: Overview of biome partitioning: distinct regions (Forest, Mountains, Desert, Plains, Wetlands) highlight visual variety, modular PCG logic, modular terrain rules, and gameplay mechanics.

How do we divide terrain into biomes?

Noise maps are ideally suited for biome classification, as they provide smoothly varying, seedable values across the world. In this work we explore two complementary approaches:

1. **Thresholded Perlin-noise biome partitioning (novel contribution):** We devised a recursive Perlin-noise splitting algorithm in this work. At each level, a 2D Perlin noise field $n_i(x, y)$ is thresholded to divide the current region into two subregions, and we recurse until exactly N terminal regions (biomes) remain. To our knowledge, this specific recursive-thresholding method has not been published elsewhere.

2. **Humidity–Temperature lookup (widely used):** We generate two independent 1D Perlin noise values per point—one for humidity (H), one for temperature (T)—and then classify the resulting (H, T) pair via a predefined biome table (e.g. desert, grassland, rainforest). This humidity–temperature method is a commonly used approach in procedural world generation.

Thresholded Perlin-noise Biome Partitioning

This method guarantees exactly N biomes by recursively splitting the terrain using successive threshold tests on Perlin noise fields.

Algorithm. Starting with the full region R and a target of k biomes:

1. Evaluate the 2D Perlin noise field

$$n_i(x, y)\colon \mathbb{R}^2 \to \mathbb{R}$$

at frequency f_i.

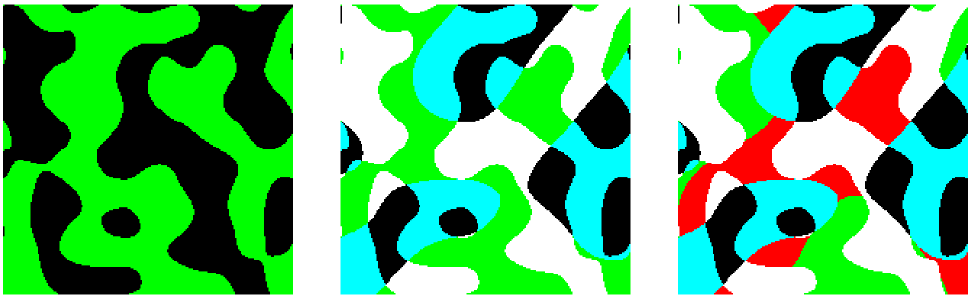
2. Partition R into two subsets via threshold T_i:

$$R_{\text{left}} = \{(x,y) \in R \mid n_i(x,y) > T_i\}, \quad R_{\text{right}} = \{(x,y) \in R \mid n_i(x,y) \leq T_i\}.$$

3. Allocate the remaining k leaves between left and right:

$$L = \lceil k/2 \rceil, \quad R = \lfloor k/2 \rfloor.$$

4. Recurse on (R_{left}, L) using noise n_{i+1} (typically $f_{i+1} = f_i/2$), and on (R_{right}, R) likewise, until each region becomes a final biome.



(a) : First division to left and right

(b) : Second division to left and right

(c) : Result after final iteration

Figure 4.11: Showcases how terrain is recursively split into five biomes

Example for Five Biomes. Suppose we need exactly five biomes: *Forest*, *Mountains*, *Meadow*, *Desert*, and *Tundra*. We proceed as follows:

1. **Level 1** ($k = 5$) Compute $n_1(x,y)$ at f_1, choose threshold T_1. Split into

$$R_{\text{left}} = \{n_1 > T_1\}, \quad R_{\text{right}} = \{n_1 \leq T_1\},$$

then assign $L = \lceil 5/2 \rceil = 3$ leaves to the left, and $R = \lfloor 5/2 \rfloor = 2$ leaves to the right.

2. **Level 2**

   - On R_{left} with $k = 3$: Compute n_2 at $f_2 = f_1/2$, threshold T_2, split into

   $$R_{2L} = \{n_2 > T_2\},\ k_{2L} = \lceil 3/2 \rceil = 2, \quad R_{2R} = \{n_2 \leq T_2\},\ k_{2R} = \lfloor 3/2 \rfloor = 1.$$

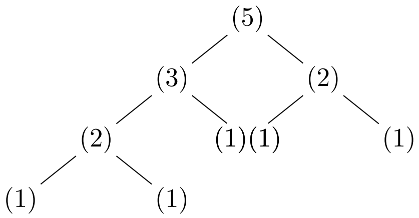
   Then recurse on R_{2L} ($k = 2$) into two single-leaf biomes, and declare R_{2R} a single biome.

   - On R_{right} with $k = 2$: Split by n_2' at $f_2' = f_1/2$, threshold T_2', into two single-leaf biomes.

3. Label the five leaves (in left-to-right order):

$$\text{Forest, Mountains, Meadow, Desert, Tundra.}$$

Visualizing the Split Tree. The binary-split process for $k = 5$ biomes can be illustrated as follows. Starting from the root "(5)", we split into a left child "(3)" and a right child "(2)". The left "(3)" node then splits into "(2)" and "(1)", where that "(2)" further splits into two leaf nodes. The right "(2)" node likewise splits into two leaf nodes. In total we obtain five leaves, which we label (left to right) as Forest, Mountains, Meadow, Desert, and Tundra.



Here each "(1)" is one final biome. By tuning each f_i (e.g. halving at each level) and threshold T_i, you control both large-scale placement and relative area (rarity) of the five biomes.

Calculate biome's height. Once each point (x, y) has been assigned a final biome i, we compute its height by blending per-biome noise functions $n_j(x, y)$. For example:

$$z(x,y) \; = b_i(x,y) = \; C_i \; + \sum_{j=1}^{K_i} n_j(x,y)\,,$$

where

- C_i is a constant base-height offset,
- $n_j(x, y)$ is the noise function (height contribution) for biome j.
- $b_i(x, y)$ is the height function of i-th biome.
- K_i is number for noise base height functions for i-th biome.

Biome blending issue. If we now generate the terrain with hard thresholds, there will be noticeable jumps between each biome.

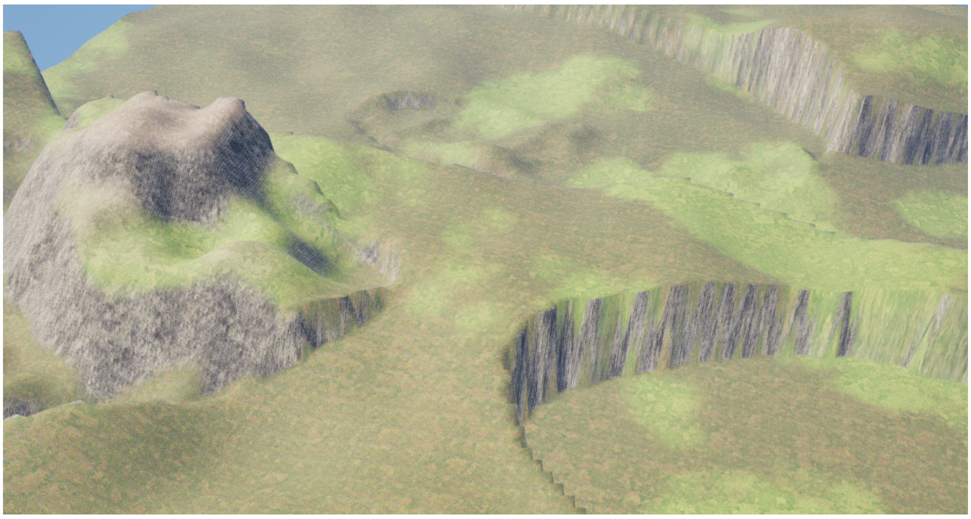


Figure 4.12: Example of abrupt seams between biomes when no smoothing is applied.

We address abrupt seams by introducing a transition band of half-width B around each split threshold T. Consider a split of region R into two child regions—left and right—and let

$$n_i = n_i(x, y), \quad T = T_i, \quad B \geq 0.$$

We compute provisional weights (w_L, w_R) at this split as

$$(w_L, w_R) = \begin{cases} (w_{\text{prev}}, 0), & n_i \leq T - B, \\ \left(w_{\text{prev}} \dfrac{T + B - n_i}{2B},\ w_{\text{prev}} \dfrac{n_i - (T - B)}{2B}\right), & T - B < n_i < T + B, \\ (0, w_{\text{prev}}), & n_i \geq T + B. \end{cases}$$

At the root we initialize

$$w_{\text{prev}} = 1.$$

At each split we ensure

$$w_L + w_R = w_{\text{prev}},$$

and propagate each branch's weight as the new w_{prev} for subsequent splits. Upon reaching the N leaf regions, the resulting weights

$$\alpha_i = w_i, \quad i = 1, \ldots, N,$$

automatically satisfy

$$\sum_{i=1}^{N} \alpha_i = 1$$

by construction, so no further normalization is required.

Finally, the terrain height at each point (x, y) is computed as a weighted sum of the per-biome height functions $b_i(x, y)$:

$$z(x,y) \;=\; \sum_{i=1}^{N} \alpha_i \, b_i(x,y), \qquad b_i(x,y) = C_i \;+\; \sum_{j=1}^{K_i} n_j(x,y).$$

Here:

- α_i is the blending weight for biome i, with $\sum_{i=1}^{N} \alpha_i = 1$.

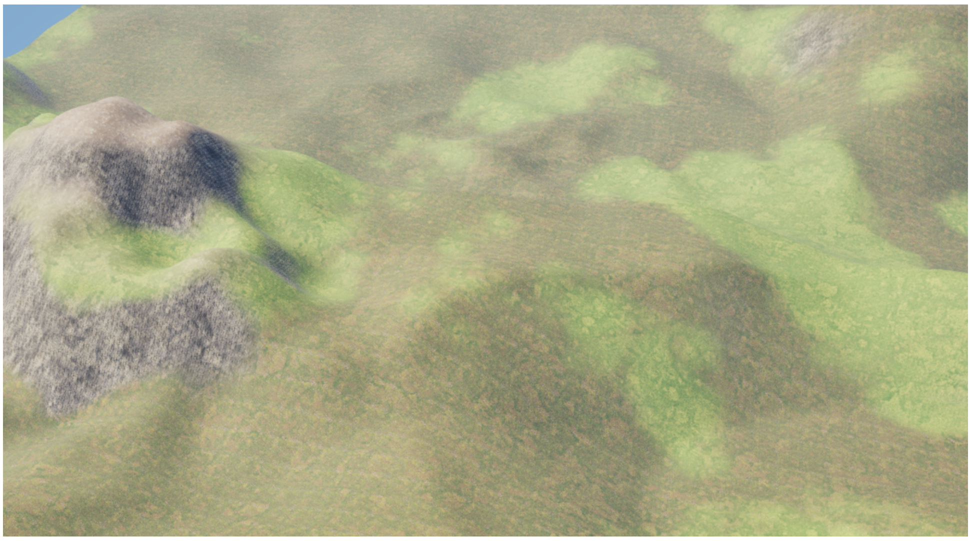


Figure 4.13: Blending between biomes after applying blending method

Summary. The thresholded Perlin-noise partitioning lets us generate exactly N biomes of arbitrary count, each covering an area proportional to our chosen thresholds. By fine-tuning each threshold T_i and corresponding noise frequency f_i, we control both the size and relative rarity of every biome while preserving smooth blends along their boundaries. Once segmented, each biome can be treated independently—applying bespoke height functions, textures, and object placements—rather than overlaying global modifiers across the entire world. This yields coherent, visually seamless results with minimal manual effort.

Humidity–Temperature lookup

The humidity–temperature lookup method classifies each terrain point into a biome by sampling two independent 1D noise functions—one controlling humidity H, the other controlling temperature T—and then consulting a predefined *biome table*. In nature, biomes occupy characteristic regions in an H–T diagram (e.g. tundra at low H, low T; tropical rainforest at high H, high T), as shown in Figure 4.14.

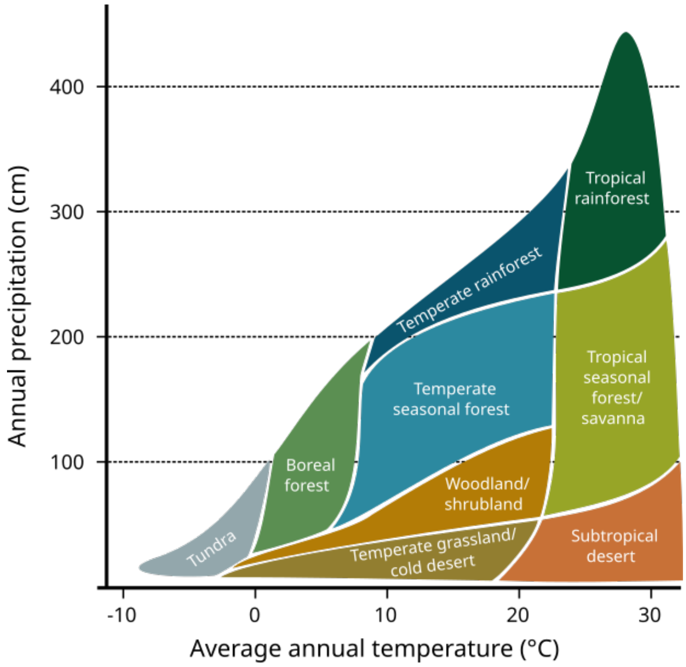


Figure 4.14: Real-world biome distribution in humidity–temperature space (after Whittaker [26]).

In our implementation we map each sampled pair $(H, T) \in [0,1]^2$ directly into a continuous 2D biome graph, avoiding a large discrete table. A minimal discrete example would be

	$T < 0.5$	$T \geq 0.5$
$H < 0.5$	Tundra	Grassland
$H \geq 0.5$	Swamp	Rainforest

Algorithm.

1. Compute humidity
$$H = n_H(x) \colon \mathbb{R} \to [0,1],$$
where n_H is a 1D Perlin (or other) noise at frequency f_H.

2. Compute temperature
$$T = n_T(y) \colon \mathbb{R} \to [0,1],$$
via an independent 1D noise at frequency f_T.

3. Classify (H, T) by locating it in the 2D biome graph (or lookup table).

Smooth-blend weights (abbreviated). To avoid hard cell boundaries, we blend the four neighboring (H, T) cells. Let

$$\delta_H = \{H \bmod 1\}, \quad \delta_T = \{T \bmod 1\},$$

and define

$$d_H = \min(\delta_H,\ 1-\delta_H,\ B), \quad d_T = \min(\delta_T,\ 1-\delta_T,\ B), \quad N = \frac{0.25}{B^2}.$$

Then the bilinear weights are

$$\begin{aligned} w_{00} &= (B+d_T)\,(B+d_H)\,N, \\ w_{10} &= (B-d_T)\,(B+d_H)\,N, \\ w_{01} &= (B+d_T)\,(B-d_H)\,N, \\ w_{11} &= (B-d_T)\,(B-d_H)\,N, \end{aligned}$$

which satisfy $w_{00} + w_{10} + w_{01} + w_{11} = 1$. These weights blend the four candidate biomes around (H, T).

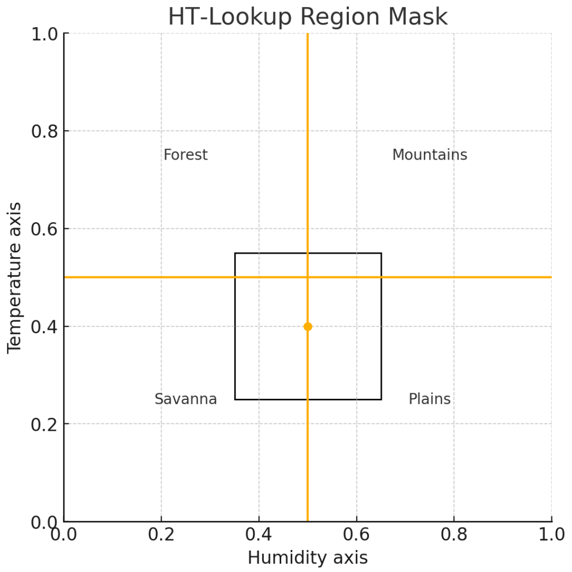


Figure 4.15: 2×2 HT-lookup mask. The sample point at $(0.5, 0.4)$ is the center of a square of side-length $2B$.

As shown in Figure 4.15, we compute blending weights for the four surrounding biomes, with each biome's weight proportional to the fraction of the square mask's area that lies within that biome.

Summary. The humidity–temperature lookup method assigns biomes by sampling two independent 1D noise fields—humidity H and temperature T—and mapping each (H, T) pair into a 2D biome diagram. This yields immediately clear adjacency relationships and allows designers to create biome palate via a simple table, without touching code. Per-cell blend weights can be precomputed for runtime efficiency, and each biome retains its own isolated parameters for height, textures, and gameplay rules. The resulting distribution closely mirrors real-world climate zones, and with proper choice of noise frequencies (f_H, f_T) delivers visual fidelity and performance comparable to recursive splitting—though it offers less fine-grained control over individual biome threshold shapes than the thresholded Perlin-noise method.

4.2.6 Auto-Material for Procedural Terrain using UStaticMesh

In modern real-time 3D engines, every rendered triangle carries per-vertex *attributes*—position, normal, UV coordinates, vertex colors, and so on—which the GPU interpolates across the face and hands off to the *fragment shader* (the little "pixel program" that actually writes out your screen image). By stuffing our biome-blend data into those per-vertex channels, we can drive entirely custom, smoothly blended terrain materials.

How Texturing Normally Works

UV Mapping. Each mesh vertex carries a pair of texture coordinates $(u, v) \in [0, 1]^2$. When the triangle is rasterized, the GPU linearly interpolates these UVs across the surface and looks up a color from your base terrain texture (grass, rock, etc.) at each pixel.

Vertex Colors. Most engines also let you paint an RGBA color on each vertex. These per-vertex colors are interpolated over the triangle and exposed in the material as a four-component input. In a conventional workflow you might use this to tint or blend textures, but since our terrain's base color comes entirely from texture sampling, we repurpose the Vertex Color channel purely as compact storage for per-vertex metadata (blend weights, biome IDs, etc.).

Multiple UV Channels. Unreal Engine UStaticMesh supports up to eight UV sets (UV0–UV7). By convention:

- UV0 holds the main UV coordinates.
- UV1 is reserved for lightmaps.
- UV2–UV7 are available for any additional two-component data you need per vertex (e.g. further blend weights, biome masks, procedural parameters).

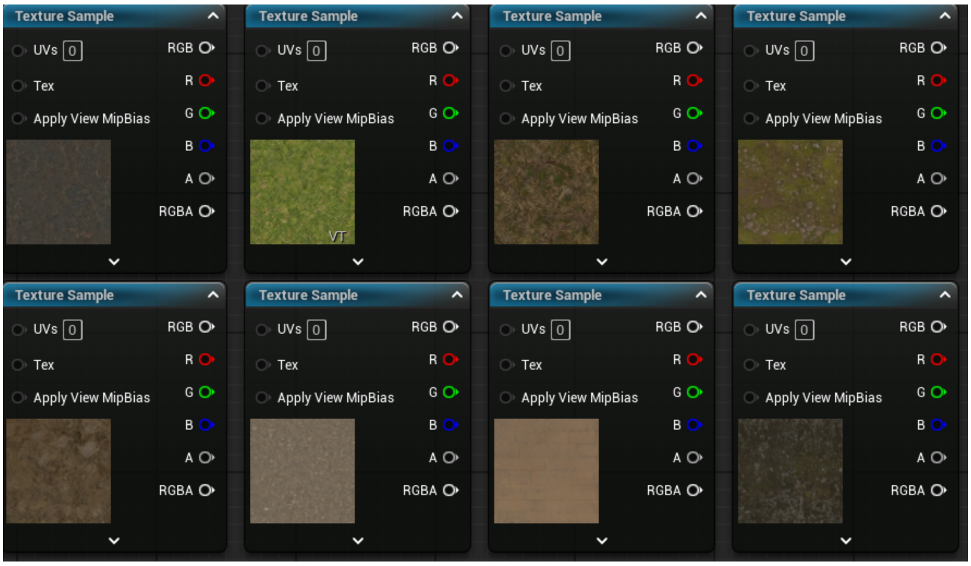


Figure 4.16: Example of texture palate.

What Do We Want to Achieve

Our goal is for each biome to use its own palette of three textures—*lowland*, *highland*, and *slope* (see Fig. 4.16)—and to blend smoothly all of them between neighboring biomes and within each biome's internal bands. In total we support eight textures (enough to cover all six biomes and their transitions). To accomplish this:

1. **Compute per-vertex biome weights.** We reuse the thresholded Perlin-noise partitioning or H-T lookup table (see Section 4.2.1), but for more natural look we tighten the blend threshold from B to $B' = B/4$. We get
$$\{w_{\text{forest}}, w_{\text{desert}}, \dots\}, \quad \sum_i w_i = 1,$$
so that each vertex smoothly interpolates with adjacent biomes.

2. **Compute three intra-biome band weights.** Within each biome k, we split into:
   - *Lowland vs. highland*, based on height z, using a center threshold $T_{\ell\to h}$ and blend half-width $\Delta_{\ell\to h}$;
   - *Slope*, based on steepness σ, with angle threshold Θ_s and blend half-width Δ_s.

   Concretely, define two "alpha" factors via clamped linear ramps:
$$\alpha_h = \text{clamp}\Big(\tfrac{z+\Delta_{\ell\to h}-T_{\ell\to h}}{2\,\Delta_{\ell\to h}},\ 0,\ 1\Big), \quad \alpha_s = \text{clamp}\Big(\tfrac{\sigma+\Delta_s-\Theta_s}{2\,\Delta_s},\ 0,\ 1\Big).$$
   Then:
$$b_s = \alpha_s, \quad b'_\ell = 1-\alpha_h, \quad b'_h = \alpha_h,$$

and we scale the lowland/highland pair to occupy the $(1 - b_s)$ remainder:

$$b_\ell = (1 - b_s)\, b'_\ell, \quad b_h = (1 - b_s)\, b'_h.$$

By construction $b_\ell + b_h + b_s = 1$.

3. **Combine biome and band weights.** Each biome k now contributes three texture weights; we multiply by its biome weight w_k:

$$w_{k,\ell} = w_k\, b_\ell, \quad w_{k,h} = w_k\, b_h, \quad w_{k,s} = w_k\, b_s.$$

Across all biomes this produces up to eight nonzero per-vertex weights, whose sum remains $\sum_{k,\ell,h,s} w_{k,*} = 1$.

4. **Pack and blend in the shader.** We encode the eight $[0, 1]$ weights into the vertex's RGBA color and extra UV channels (two floats each). Generated code when creating mesh sections might look like this:

```
for (int i = 0; i < MeshData->Triangles.Num(); i += 3) {
    // Texture weights 0 4        VertexColor (RGBA)
    Attributes.GetVertexInstanceColors()
        .Set(VtxInstID, weights0_4);

    // Texture weights 5 6        UV channel 2
    Attributes.GetVertexInstanceUVs()
        .Set(VtxInstID, 2, weights5_6);

    // Texture weights 7 8        UV channel 3
    Attributes.GetVertexInstanceUVs()
        .Set(VtxInstID, 3, weights7_8);
}
```

$$\text{FinalColor} = \sum_{i=1}^{8} w_i \ \text{Texture}_i$$

5. **Accessing weights in Material Blueprint.** In your material, add a `VertexColor` node to read weights 0–4, then two `TexCoord` nodes with "Coordinate Index" set to 2 and 3. Pass each through a `ComponentMask` (R,G) to extract weights 5–6 and 7–8 respectively, as illustrated in Fig. 4.17.

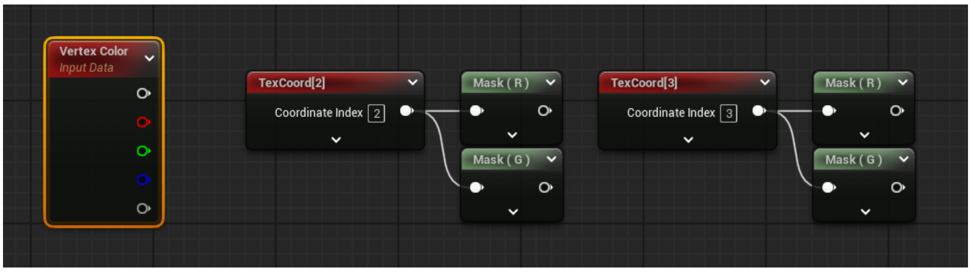


Figure 4.17: Material Blueprint setup for unpacking per-vertex texture weights.

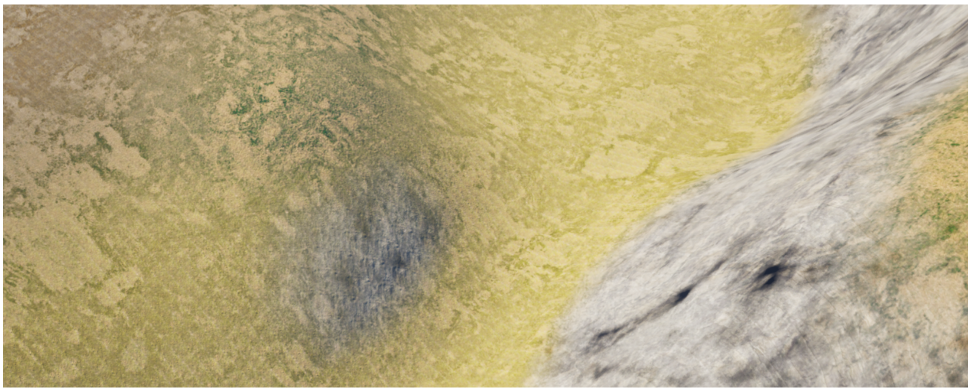


Figure 4.18: Final terrain rendering showing smooth, per-vertex blended transitions between biomes and elevation bands, driven entirely by the custom VertexColor and UV-packed weights.

Note: This scheme lets us pass up to 16 independent 8-bit values per vertex into the material:

- **VertexColor** (RGBA) provides 4 channels.
- **TexCoord**[**2**–**7**] each carry two 8-bit values, for a total of $6 \times 2 = 12$ channels.

All 16 channels can store blend factors or other metadata, giving us ample bandwidth for complex, per-vertex control of our procedural terrain shader.

Why This Matters

- **Familiar pipeline.** We never stray from standard UVs and vertex channels—no engine hacks required.
- **Smooth transitions.** Because the GPU linearly blends vertex attributes, neighboring biomes dissolve seamlessly.
- **Per-vertex control.** You can mix any combination of up to eight textures on every triangle—perfect for edge cases where multiple habitats meet.

4.2.7 Adapting PCG for a Fully Procedural World

Our original PCG graphs assumed a static Landscape asset. For an infinite, runtime-generated world, we must revise them as follows:

Replacing Landscape Data and Integrating World Partition

In our updated PCG graphs, each `Get Landscape Data` node is replaced by a `World Ray Hit Query`, allowing us to sample height, normal, and layer data from any collision-enabled geometry—not just Landscape actors. We enable World Partition's `Runtime Generation` so that PCG graphs execute dynamically as new chunks appear, and set `Partitioned` to split each graph into per-chunk sections for optimized, staggered execution. These combined settings let the system sample arbitrary terrain at runtime, spawn objects on freshly streamed chunks, and maintain high performance by processing only the chunks currently loaded.

Filtering Points by Biome

To correctly assign each point to its biome, we create a custom PCG node that divides points into their respective biomes (see Figure 4.19).

Creating a Custom PCG Node: `BP_SplitPointsToBiomes`. We want a single PCG node that takes an incoming point list and dispatches each point into exactly one of six biome-specific output pins (Forest, Plains, Desert, Taiga, Mountains, Dunes). Here's how to implement it in Blueprints:

1. **Define the new node class.**
   Duplicate `GetSteepness` and rename it to `BP_SplitPointsToBiomes`.
2. **Expose six output execution pins.**
   In `BP_SplitPointsToBiomes`'s PCG graph:
   - Drag off the `Execute` pin and select **Add Output Exec** six times.
   - Rename each pin to `Forest`, `Plains`, `Desert`, `Taiga`, `Mountains`, and `Dunes`.
   - Set each pin's data type to "Point Data" (or your PCG metadata type).
3. **Loop over incoming points.**
   - Pull in the `In` point-array pin and feed it into a `ForEach` loop.
   - Each iteration yields one `Point` structure.
4. **Classify each point's biome.**
   - Call your Blueprint-exposed function `GetBiomeAtLocation(FVector Position)` on `Point.Transform.Location`.
   - This returns an `EBiome` enum value (Forest, Plains, . . .).
5. **Accumulate points per biome** (see Figure 4.19).
   - After the `GetBiomeAtLocation` node, add a `Switch on EBiome`.
   - For each case, `Add` the current `Point` to a dedicated local array: `ForestPoints`, `PlainsPoints`, etc.

6. **Dispatch grouped arrays to outputs.**

   - After the loop completes, wire each biome's array into its matching Exec-output pin.
   - Downstream PCG subgraphs now receive exactly those points belonging to each biome.

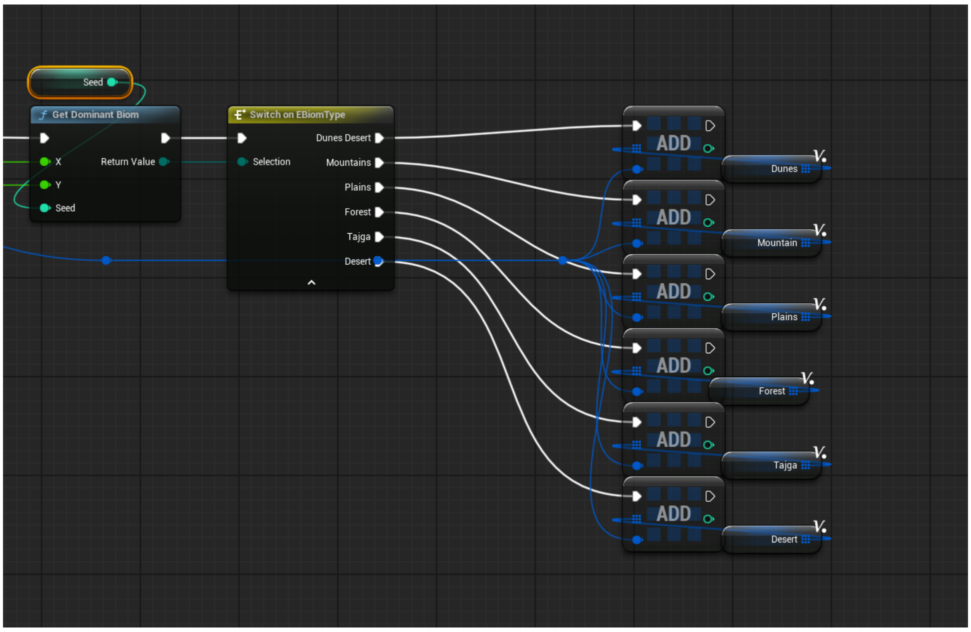


Figure 4.19: Core logic inside our new `BP_SplitPointsToBiomes` node

Now, in your main PCG graph you can hook the `Forest` pin into one subtree (e.g. tree spawners), the `Desert` pin into another (cacti), and so on, all driven by the procedural terrain.

Optimization and Distance Culling

To maintain high performance in a large, procedurally generated world, we employ three complementary strategies:

- **Selective Collision (Hitboxes)**
  Only enable collision where it's truly needed—rocks, buildings, and interactive props—while disabling physics on purely decorative elements (grass, flowers, distant foliage). This dramatically reduces the per-frame physics workload.

- **Multi-scale Sampling Sections**
  We partition terrain sampling into several nested grid sizes, each driving a different level of detail as shown in Figure 4.20:

- *Small cells* (e.g. 3 200 UU): precise data for near-ground foliage (grass, weeds).
- *Medium cells* (e.g. 6 400 UU): low bushes and small props.
- *Large cells* (e.g. 12 800 UU): trees, rocks.
- *Huge cells* (e.g. 25 600 UU): large landscape features (mountains, cliffs).

In our PCG graph, a single `World Ray Hit Query` feeds four `Grid Sampler` nodes—one per cell size—which then dispatch to separate subgraphs (`SmallLandscapeData`, `MediumLandscapeData`, etc.). This ensures each feature uses just the resolution it requires.

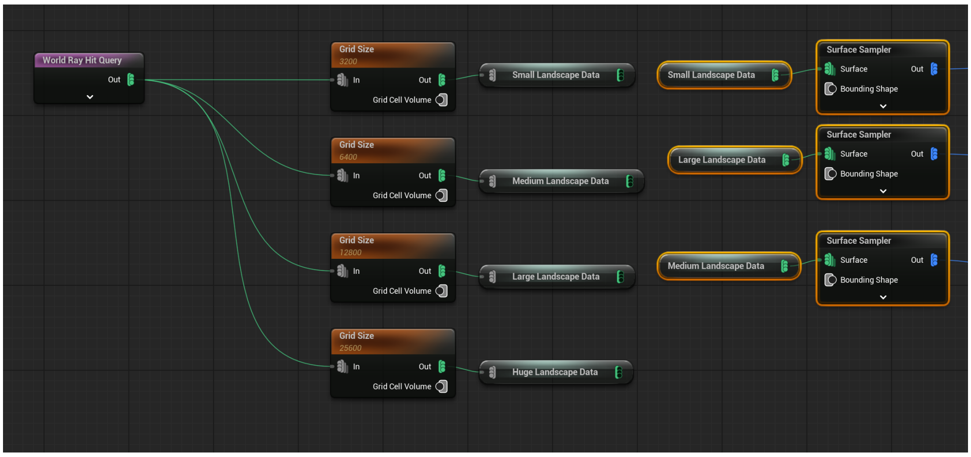


Figure 4.20: Terrain is partitioned into multiple sampling scales to optimize detail and performance.

Combining selective hitboxes, multi-scale sampling, and distance culling focuses computation and rendering on the player's vicinity, while distant regions incur minimal overhead."'

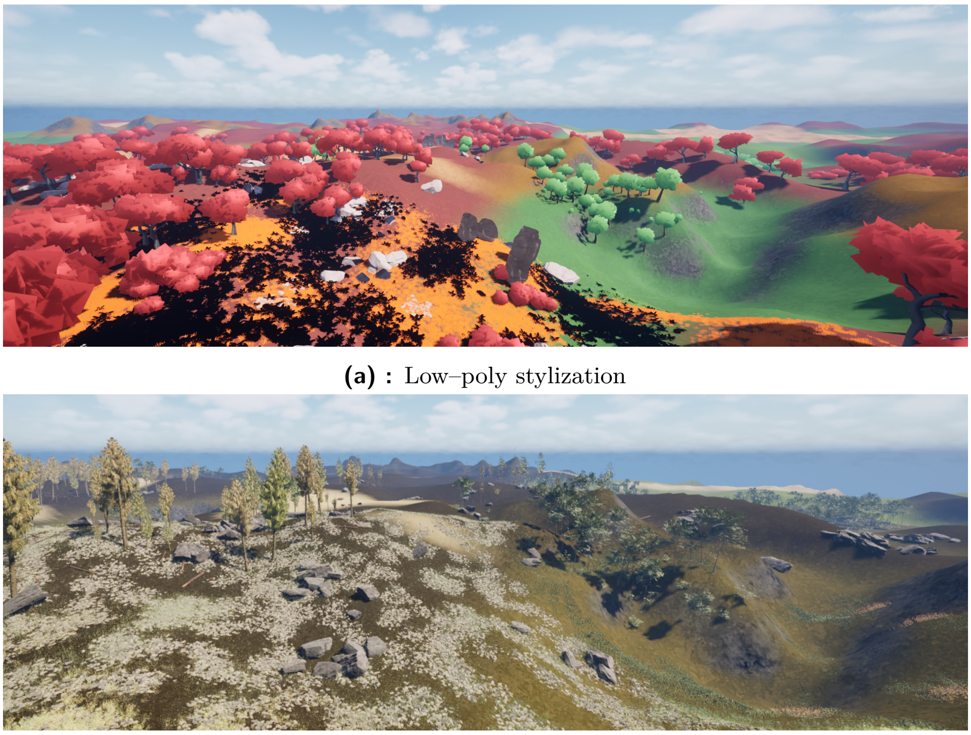
4.2.8 Results and Stylizations

With the PCG pipeline fully integrated into a runtime-generated world (including biome splitting via `BP_SplitPointsToBiomes`, see Section 4.2.7), creating the final environment is now trivial: you simply assign each PCD biome discussed in Section 3.4 to its corresponding biom pin from `BP_SplitPointsToBiomes` or you create new one to suit your theme.

Beyond functional correctness, the same system supports multiple artistic styles by swapping only the terrain textures and meshes in PCG graphs. For example, Figures 4.21a and 4.21b show the exact same world geometry and PCG logic rendered in:

- **Low–Poly Style:** flat colors, sharp edges, minimal texture detail.

- **Photorealistic Style:** high-resolution albedo, normal and roughness maps for lifelike terrain.



(a) : Low–poly stylization

(b) : Photorealistic stylization

Figure 4.21: Final procedural world rendered in two distinct visual styles.

Chapter 5

Results

The procedural world generation runs smoothly and delivers ample visual detail to engage players, while maintaining a stable frame rate suitable for performance-sensitive games.

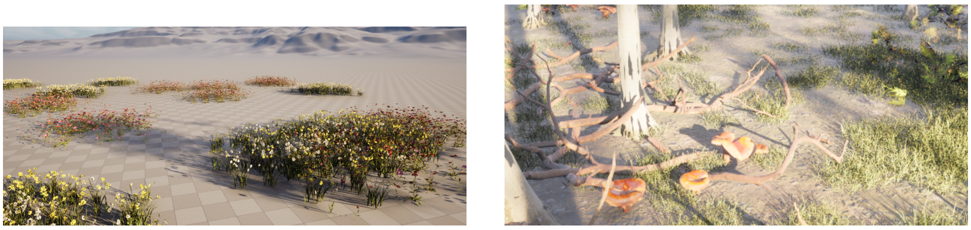
5.1 Procedural Content Generation

We implemented a flexible PCG (Procedural Content Generation) framework in Unreal Engine 5.5 that unifies terrain shaping and runtime asset placement. The core system consists of custom PCG Graphs featuring:

- **On-Demand Sampling:** Real-time noise evaluation and environmental queries to drive dynamic spawning of meshes and foliage.
- **Rule-Based Filters:** Configurable attribute tests (e.g. slope, altitude, biome weight) to constrain placement to valid regions.
- **Hierarchical Density Control:** Multi-tier density settings allowing coarse feature zoning and fine-grained detail distribution.
- **Streaming Integration:** Seamless triggering of PCG tasks via Unreal's World Partition as the player traverses streaming cells.

5.1.1 Flora PCGs

The following PCG graphs focus on vegetation placement across various biomes, using density controls and environmental filters to create natural plant distributions. More detailed flora configurations for each biome are provided in the "Biomes" subsection of the next chapter.



(a) : Wildflowers distributed across landscape.

(b) : Mushroom clusters occasionally growing under the trees.

Figure 5.1: Point out details for plains PCG shown in 5.1a and Forest in 5.1b.

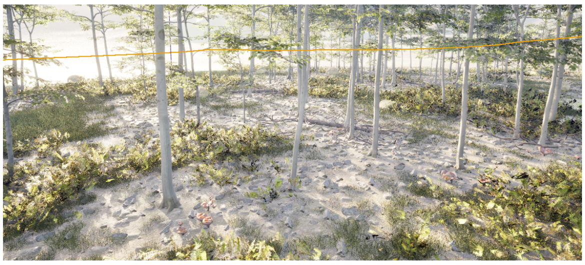
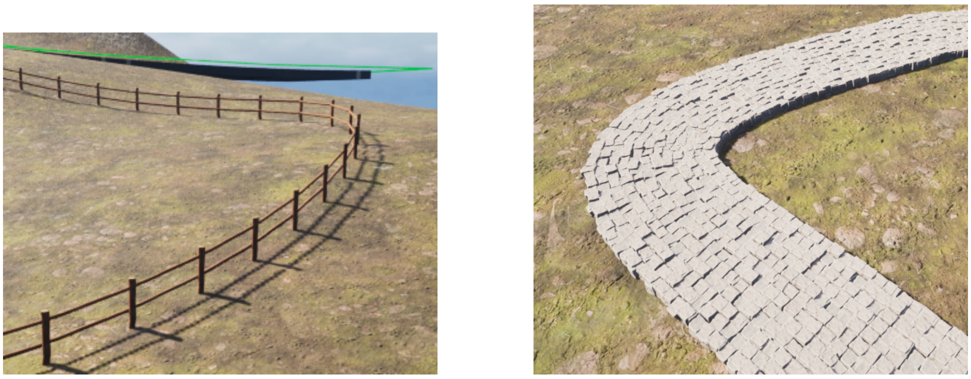


Figure 5.2: Forest biome with trees, undergrowth based on sun access, fallen logs, mushrooms, small and large rocks, and fallen branches.

5.1.2 Infrastructure PCGs

These PCG graphs automate the placement of man-made elements—fences, roads, and village layouts—by adapting to terrain contours and applying rule-based filters. Further implementation details and parameters are discussed in the "Biomes" subsection of the next chapter 5.2.2.



(a) : Fence generation following terrain contours and slope constraints.

(b) : Dynamic road placement adapting to landscape features.

Figure 5.3: Examples of infrastructure PCGs for fences and roads.

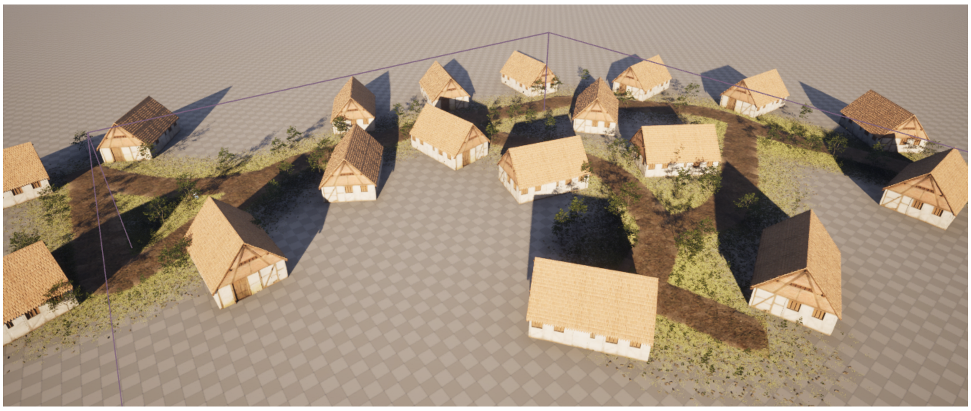


Figure 5.4: Village PCG showcasing procedural street networks and building placement.

5.2 World Generation

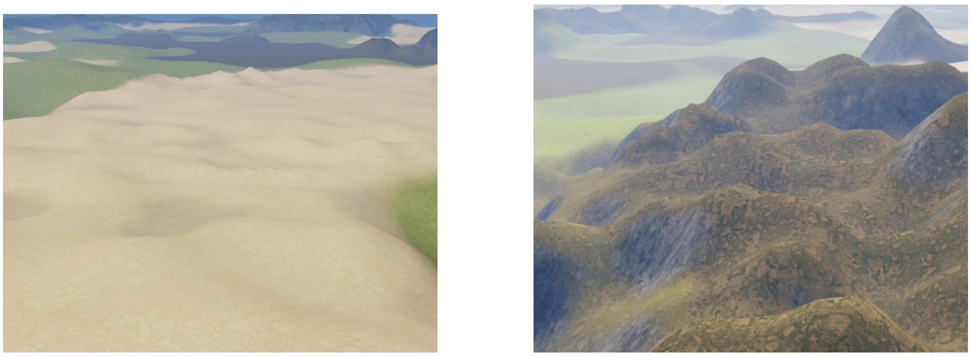
Our pipeline procedurally generates an expansive, voxel-inspired landscape at runtime and populates it using the PCG framework. To enhance visual interest and ecological variety, the world is divided into six distinct biomes in 2 different artstyles:

- **Desert:** Sparse vegetation and undulating dunes sculpted by low-frequency noise.
- **Dunes:** Sharply contoured sand hills with wind-driven ripple patterns.
- **Mountains:** Rugged peaks generated via fractal Brownian motion, supporting rocky outcrops and sparse conifers.
- **Tundra:** Flat, frost-covered plains populated with low-lying shrubs and periodic ice patches.
- **Forest:** Dense groves of procedurally scaled trees, undergrowth, and fallen logs.
- **Meadows:** Rolling grasslands with wildflowers and scattered rock formations.

Each biome's logic is implemented in its own PCG subgraph, and adjacent zones blend smoothly using linear interpolation of biome weights, ensuring natural transitions without abrupt borders.

5.2.1 Terrain Generation Results

Our refined runtime terrain generator supersedes the initial Procedural Mesh Component prototype, overcoming its performance and streaming constraints. By dividing the world into biome-specific zones—each governed by its own noise-driven height formula and tailored texture set—we achieve richly varied landscapes at scale. Large areas are generated in seconds, thanks to multi-threaded mesh construction and World Partition–based streaming, all while maintaining smooth frame rates.



(a) : Desert biome: undulating dunes and sandy textures.

(b) : Mountain biome: rugged peaks and rocky outcrops.

Figure 5.5: High-detail terrain renderings for two distinct biomes.

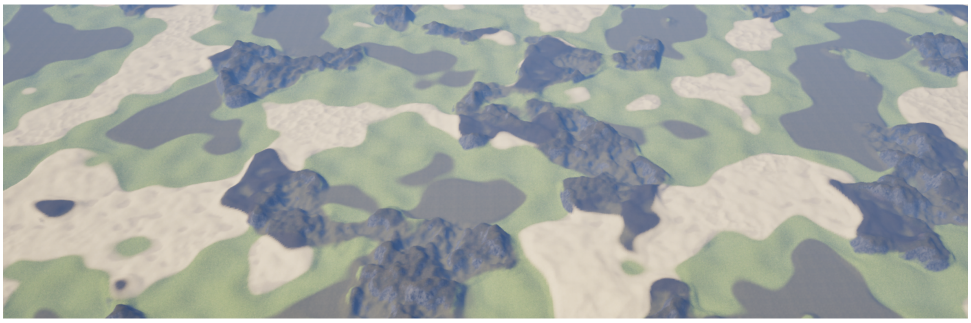
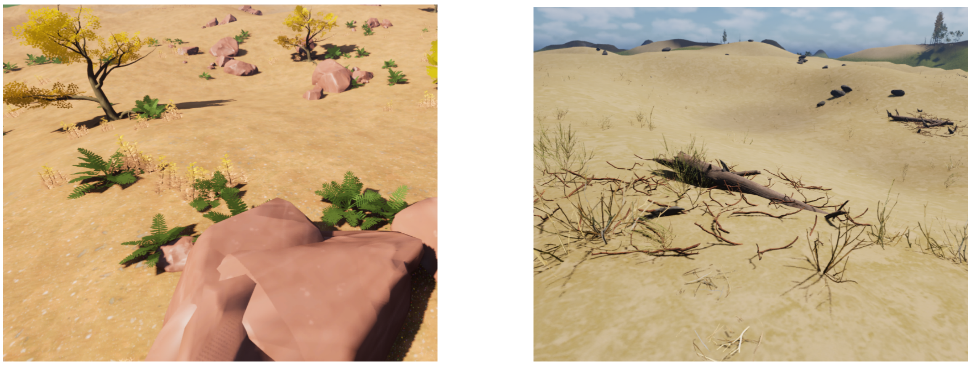


Figure 5.6: Panoramic view of the procedurally generated world, showcasing seamless streaming and biome transitions.

5.2.2 Biome Results

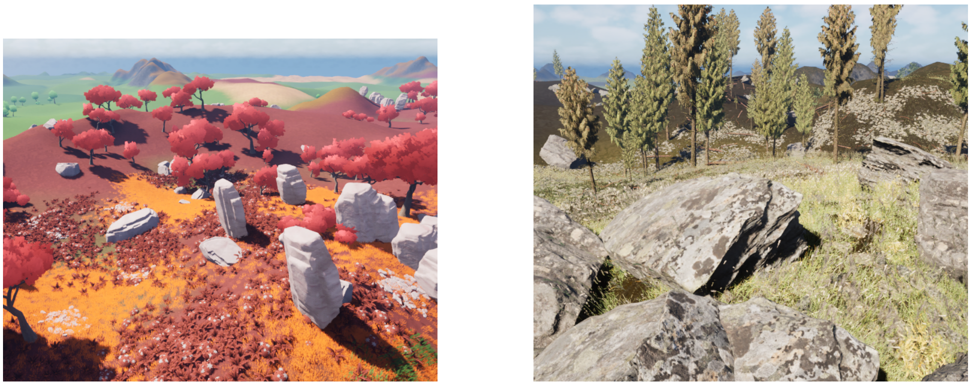
Below are examples of several biomes presented in low-poly and photo-real variants.



(a) : Low-poly Desert style.

(b) : High-fidelity Desert terrain with photoreal textures.

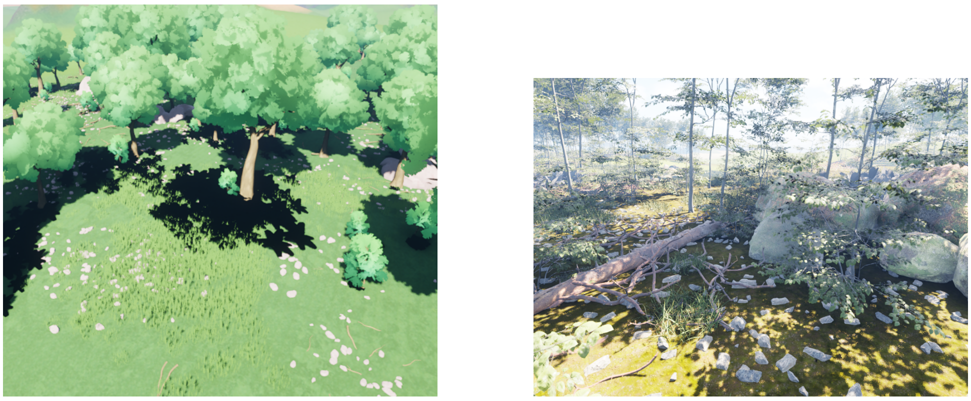
Figure 5.7: Comparison of Desert biome in low-poly vs. photorealistic styles.



(a) : Fictional low-poly Red Forest biome.

(b) : Photorealistic Tundra terrain with detailed snow and ice.

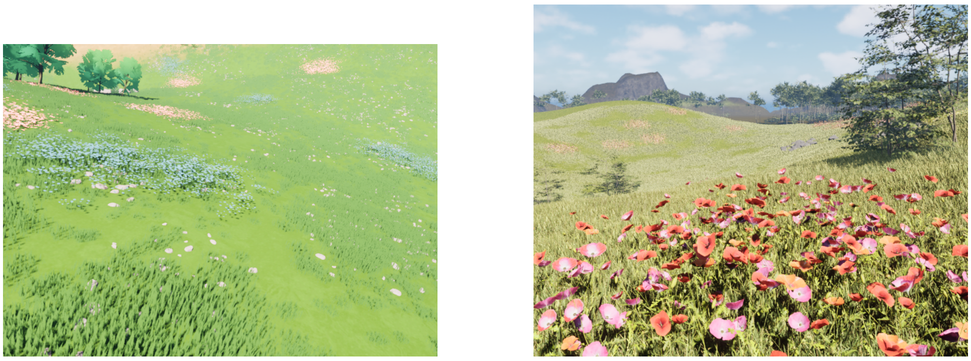
Figure 5.8: Comparison of Red Forest (low-poly) and Tundra (photoreal) biomes.



(a) : Low-poly Forest with simplified geometry and flat shading.

(b) : Photorealistic Forest with high-resolution foliage and lighting.

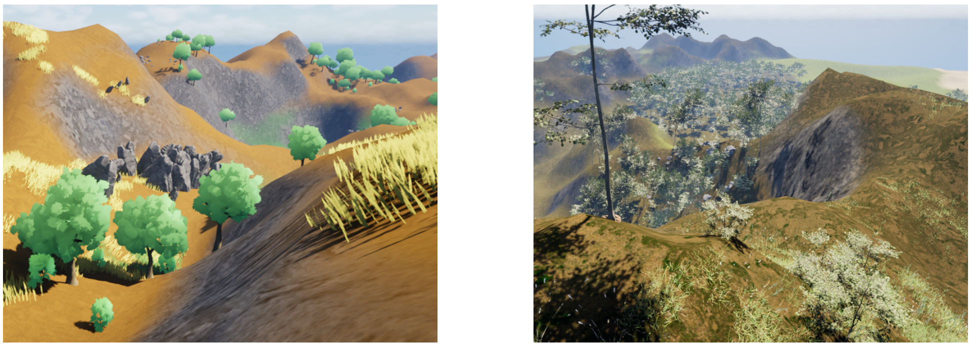
Figure 5.9: Forest biome rendered in both low-poly and photorealistic modes.



(a) : Low-poly Plains with minimal geometry and stylized grass.

(b) : High-detail Plains using photorealistic grass and rocks.

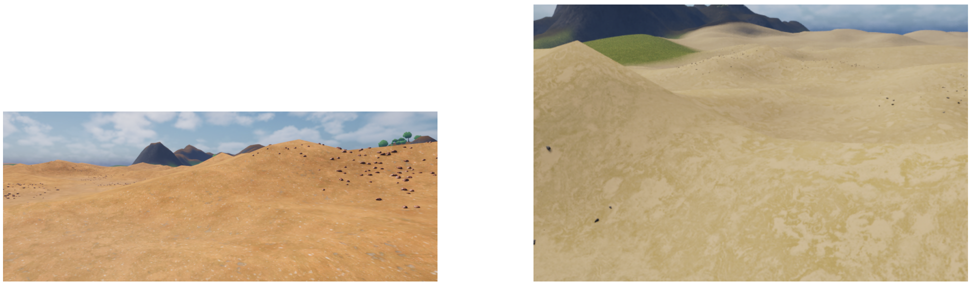
Figure 5.10: Plains biome comparison: low-poly versus high-fidelity rendering.



(a) : Low-poly Mountains with blocky peaks and simplified textures.

(b) : Photorealistic Mountains featuring detailed rock faces and snow caps.

Figure 5.11: Mountains biome shown in low-poly and photorealistic styles.



(a) : Low-poly Dunes.

(b) : Photorealistic Dunes.

Figure 5.12: Desert Dunes biome rendered in low-poly and photorealistic modes.

For hands-on experimentation, you can download the demo implementation of our world-generation system from the project's Git repository [17].

5.3 Performance

Leveraging Unreal Engine 5's built-in streaming, occlusion culling, and Nanite-friendly mesh pipelines, our system maintains consistent frame rates even in extensive worlds. Key optimizations include:

- Multi-threaded terrain mesh construction and PCG computations off the game thread.
- Nanite-enabled assets with automatic LOD scaling, delivering high-detail geometry up close while efficiently culling distant objects.
- Grid-based world partitioning to limit active PCG workloads to nearby cells.

While runtime generation incurs initial CPU overhead, once terrain chunks and assets are instantiated, subsequent navigation through the scene yields consistently high performance, with frame rates comparable to static levels.

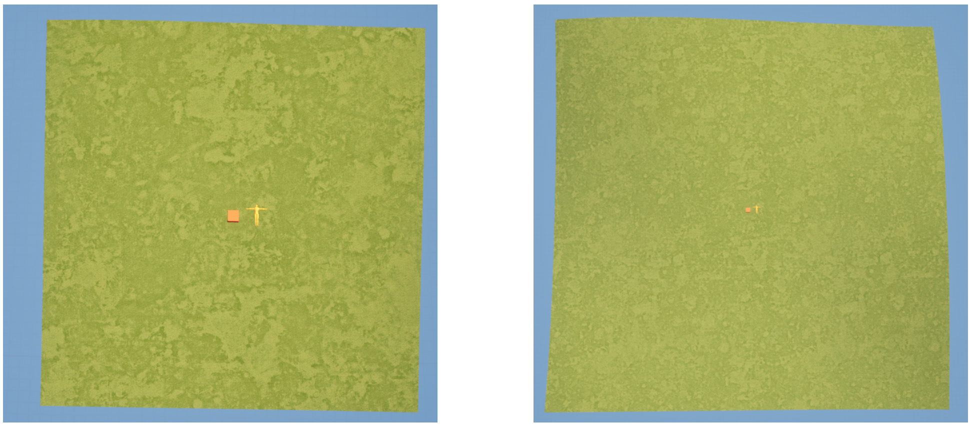
5.3.1 Terrain Generation Performance

For smooth terrain streaming in real-time environments, both generation speed and frame-rate stability are critical. In this section, we compare two approaches used to chunk generation:

- **Procedural Mesh Component**: dynamic mesh creation via `CreateMeshSection()` from chapter 4.2.2.
- **Static Mesh Generation**: runtime construction of `UStaticMesh` assets and their spawning from chapter 4.2.4.

Performance Benchmarks

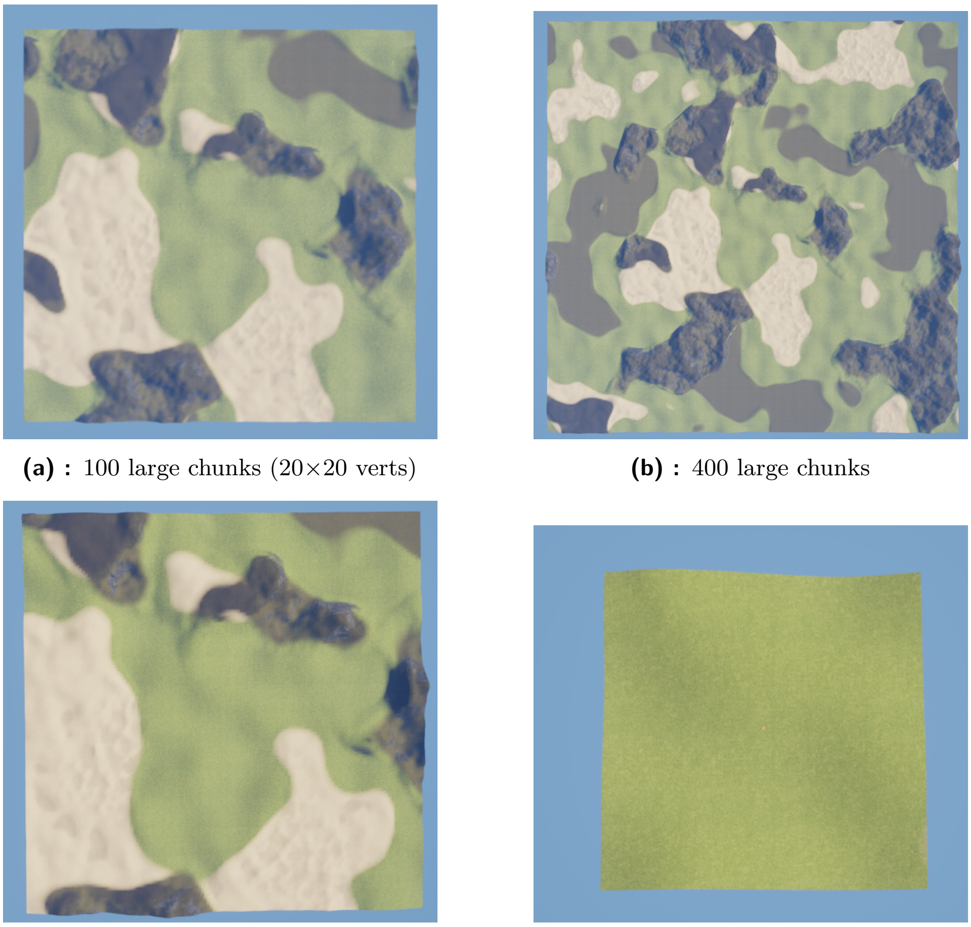
We compared the two streaming approaches—Procedural Mesh Component vs. Static Mesh Generation—by measuring the average build time (total time to generate and submit all chunks) over five runs. Tests cover different chunk resolutions and world sizes.



(a) : Small chunk: 8×8 vertices

(b) : Large chunk: 20×20 vertices

Figure 5.13: Chunk resolutions used in performance tests.



(a) : 100 large chunks (20×20 verts)

(b) : 400 large chunks

(c) : 400 small chunks (8×8 verts)

(d) : 25 small chunks

Figure 5.14: Examples of world configurations used in the build-time measurements. Terrains (a)–(d) correspond to the four test scenarios shown in Table 5.1, in the same order.

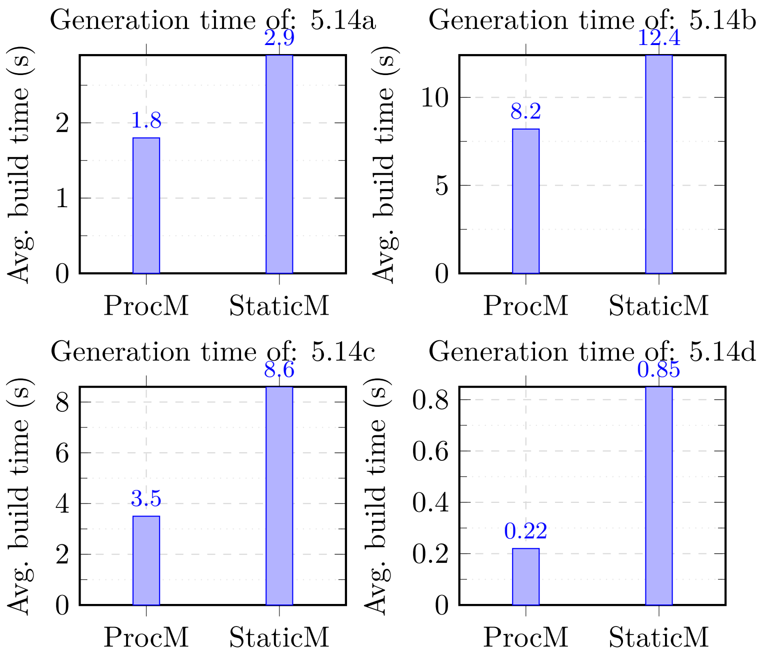


Table 5.1: Average chunk build times for ProceduralMesh vs. Generator across four configurations (verts per chunk × number of chunks).

During these performance tests, terrain is generated on-the-fly as the player moves at high speed. Chunks outside the streaming radius are unloaded, so only the terrain within the active load radius remains around the player:

- **First test:** 20 × 20 chunk radius (see Fig. 5.14a, Table 5.2).
- **Second test:** 5 × 5 chunk radius (see Fig. 5.14d, Table 5.3).
- **Third test:** 8 × 8 chunk radius, covering roughly one-quarter of the large-area scenario (see Fig. 5.14a, Table 5.4).

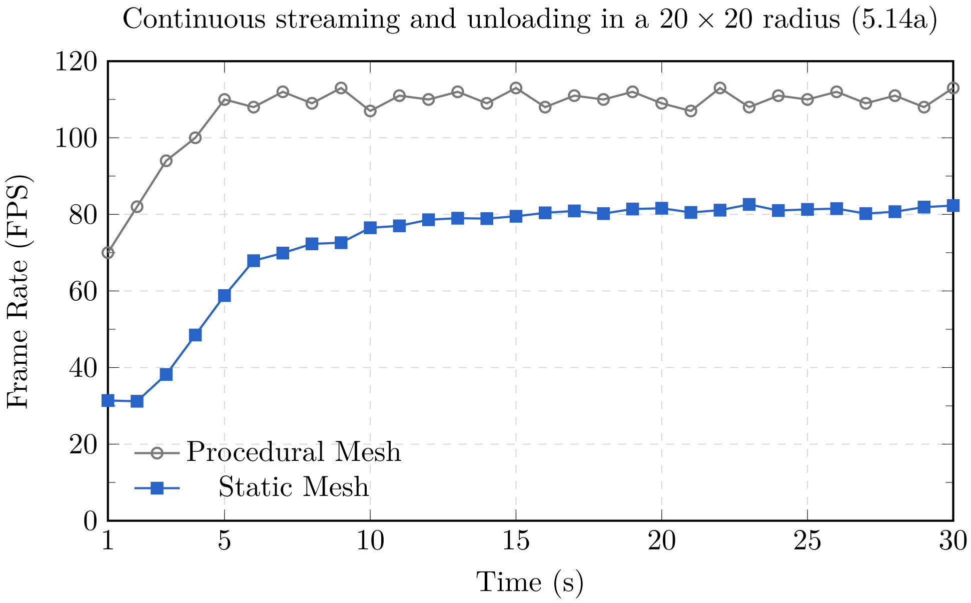


Table 5.2: In the 20 × 20 radius test, the ProceduralMesh approach (Procedural Mesh) ramps from 70 FPS up to 110 FPS within the first 5 s and then oscillates within ±3 FPS around that level, while the Generator method (Static Mesh) quickly stabilizes near 80 FPS with minimal variation.

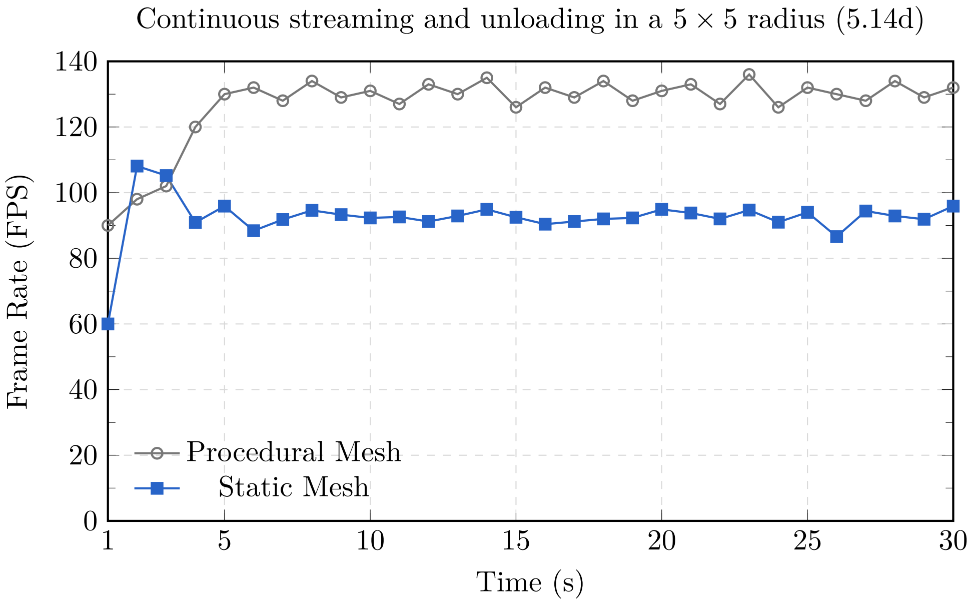


Table 5.3: In the 5 × 5 radius scenario, the ProceduralMesh component (Procedural Mesh) climbs from 90 FPS to about 130 FPS in the first 5 s and then fluctuates within ±5 FPS, while the Generator approach (Static Mesh) gradually rises from 60 FPS to 108 FPS and maintains smooth performance around that level.

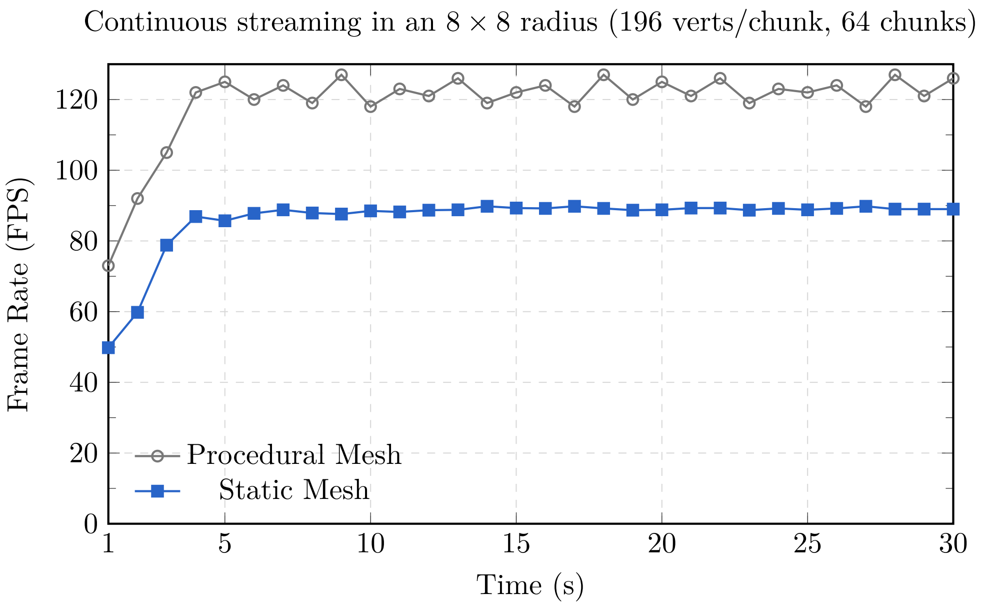


Table 5.4: In the 8 × 8 radius test, ProceduralMesh (Procedural Mesh) ramps from 73 FPS to 122 FPS in the first 4 s and then oscillates within ±5 FPS around 125 FPS, while the Generator (Static Mesh) maintains a steady 90 FPS throughout.

Key observations

- **Procedural Mesh** is faster in all scenarios.
- **Procedural Mesh** performance is more dependent on individual chunk size.
- **Static Mesh** incurs a higher upfront build time per chunk yet stabilizes at roughly $\sim$ 80 FPS regardless of world size.

5.3.2 PCG Performance

We evaluated the runtime cost of our PCG–driven vegetation placement using low-poly assets. Key findings include:

- **Point sampling dominates cost.** Mesh assets are loaded once and cached, so instantiation overhead is minimal—there is virtually no difference between low- and high-poly models during PCG. Instead, the number of spawn points (tens of thousands of grass blades and small props) drives CPU usage.
- **Distance-based spawn radius trade-off.**
  - *Tight radius around the player:* Limits per-frame work, minimizes frame-rate impact and eliminates stalls, but may cause terrain or vegetation pop-in at the edge of the distance threshold.

  - *Wide radius:* Ensures full local scene density, but incurs large CPU spikes and frame-rate drops when many instances spawn simultaneously.

- **Balancing fidelity and performance.** An intermediate spawn radius—wide enough to mask pop-in yet small enough to bound work per frame—offers the best compromise between visual richness and smooth frame rates.

- **Low-poly vs. high-poly.** While PCG instantiation time is similar for low- and high-poly assets, a static low-poly scene sustains significantly higher frame rates due to its lower vertex count.

- **Dominance of grass in PCG cost.** Grass and other abundant plants comprise up to 80% of the PCG computation in lush biomes. Minimizing grass spawn radius is therefore critical. Disabling non-interactive or purely decorative objects can reduce PCG overhead to as little as 20% in tundra or 10% in meadows. Similar optimizations apply to other heavily represented models across biomes.

In practice, tuning the number of sample points and spawn distance per asset type is the most effective way to optimize PCG performance without sacrificing visual fidelity.

Chapter 6

Conclusion

Throughout this thesis, we develop and validate a comprehensive system for infinite, runtime-driven world generation in Unreal Engine 5.5. Our approach seamlessly merges C++-based noise algorithms with the PCG Graph framework to produce richly varied terrains, dynamic biomes, and procedurally placed assets—all delivered in real time at interactive frame rates.

6.1 Key Contributions

- **Comprehensive PCG Documentation:** Delivered clear, example-driven guidance for Unreal Engine's PCG framework, demonstrating core principles and workflows via custom graph examples—providing a concise, practical supplement to the official documentation.
- **Modular PCG Graphs:** Developed reusable, extensible graphs for biome-based vegetation, infrastructure (roads, fences, buildings), and procedural village layouts, facilitating rapid prototyping and iteration.
- **Multi-threaded Terrain Generator:** Implemented a C++ solution that computes mesh data off the game thread and streams new chunks dynamically, eliminating stalls and enabling uninterrupted exploration.
- **Performance-Driven Integration:** Leveraged World Partition, Nanite, and background task scheduling to sustain stable frame rates under heavy load—significantly outperforming the basic Procedural Mesh Component approach across varied world scales.
- **Dynamic Biome Blending:** Employed biome-specific noise formulas and weight-based interpolation to create seamless transitions between adjacent regions, maintaining ecological consistency without visible seams.
- **Scalability and Customization:** Provided configurable noise layers, spawn radii, and LOD settings, allowing developers to balance visual fidelity and performance for targets ranging from high-end PCs to mobile devices.

6.2 Future Directions

To build on this foundation, we plan to:

- **Publish as a C++ Library:** Refine, package, and document the terrain and PCG modules as an Unreal Engine plugin—complete with example projects, API references, and tutorials—to ease adoption by other developers.
- **Mobile Indie Demo:** Create a lightweight, mobile-focused prototype that leverages our library to validate performance and usability on lower-end devices, showcasing commercial viability across hardware tiers.
- **Enhanced Biome Fidelity:** Integrate multi-layer noise blending, procedural erosion, and GPU-accelerated simulation (e.g., hydraulic or thermal processes) to achieve more realistic terrain morphology.
- **Persistent Chunk Caching:** Implement on-disk serialization and differential updates for generated chunks, minimizing recomputation when revisiting areas and further reducing runtime CPU load.

By delivering both a well-documented toolkit and a practical proof-of-concept, we aim to lower the barrier for high-quality procedural generation in Unreal Engine and demonstrate its versatility for games, simulations, and real-time applications.

Bibliography

[1] R. Líbal, “Tile-based Procedural Generation: A Generic Library for Multi-layer Terrain,” Bachelor’s thesis, Czech Technical University in Prague, 2023. [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/108634/F3-BP-2023-Libal-Rudolf-thesis.pdf. [Accessed: May 1, 2025]

[2] L. Hepner, “Procedural Generation of Voxel Worlds,” Master’s thesis, Czech Technical University in Prague, 2021. [Online]. Available: https://dspace.cvut.cz/handle/10467/101198. [Accessed: May 1, 2025]

[3] M. Hudeček, “3D Modeling and Visualization of Underground Structures,” Master’s thesis, Czech Technical University in Prague, 2021. [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/95898/F1-DP-2021-Hudecek-Martin-Martin_Hudecek_dp-final.pdf. [Accessed: May 1, 2025]

[4] J. Navrátil, “Procedural Terrain Generation Using GPU Acceleration,” Bachelor’s thesis, Czech Technical University in Prague, 2016. [Online]. Available: https://theses.cz/id/5req06/12666.pdf. [Accessed: May 1, 2025]

[5] Epic Games, "Procedural Content Generation Framework in Unreal Engine," *Unreal Engine Documentation*, [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation--framework-in-unreal-engine. [Accessed: Jan. 18, 2025].

[6] Game Dev Academy, "The Secret to Hide Texture Repetition in Unreal Engine 5: 4 Pro Tips," YouTube, May 7, 2023. [Online]. Available: https://www.youtube.com/watch?v=zY8AtjM2Jxg. [Accessed: Jan. 18, 2025].

[7] Aziel Arts, "How to Make Landscape Layer Materials with Natural Height Blending in Unreal Engine 5," YouTube, Aug. 29, 2024. [Online]. Available: https://www.youtube.com/watch?v=W-BMbadinPI. [Accessed: Jan. 18, 2025].

[8] Unreal Engine, "Introduction to PCG Workflows in Unreal Engine 5 | Unreal Fest 2023," YouTube, Oct. 26, 2023. [Online]. Available: https://www.youtube.com/watch?v=LMQDCEiLaQY&t=1356s. [Accessed: Jan. 18, 2025].

[9] Aziel Arts, "How to Make an Auto Material in Unreal Engine 5," YouTube, Oct. 25, 2024. [Online]. Available: https://www.youtube.com/watch?v=2h2kzV7_o9c&t=4s. [Accessed: Jan. 18, 2025].

[10] Quixel, “Megascans Free Assets,” FAB Marketplace, [Online]. Available: https://www.fab.com/search/channels/unreal-engine?q=quixel&sellers=o-cbf5bbfdb340d6a839b5520cb31328&listing_types=3d-model&categories=nature-plants. [Accessed: May 1, 2025].

[11] FAB.com, “Low Poly Starter Pack,” FAB Marketplace, [Online]. Available: https://www.fab.com/listings/fbda9e5c-00fe-4667-a2df-5849182512c8. [Accessed: May 1, 2025].

[12] Procedural Minds, "Get Started with PCG 5.4 by Creating a Full Building | UE 5.4 P1," YouTube, Jul. 28, 2024. [Online]. Available: https://www.youtube.com/watch?v=oYNA24tcYc0&t=40s. [Accessed: Jan. 18, 2025].

[13] Epic Games, "Geometry Scripting User’s Guide," *Unreal Engine Documentation*, [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/geometry-scripting-users-guide-in-unreal-engine. [Accessed: Jan. 18, 2025].

[14] Epic Games, "Using Shape Grammar with PCG in Unreal Engine," *Unreal Engine Documentation*, [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/using-shape-grammar-with-pcg-in-unreal-engine. [Accessed: Jan. 18, 2025].

[15] Procedural Minds, "PCG Grammar Is Here and It’s Amazing," YouTube, Oct. 20, 2024. [Online]. Available: https://www.youtube.com/watch?v=4Y7sOTolI-M&t=248s. [Accessed: Jan. 18, 2025].

[16] Yazan Hanna, "The New PCG Grammar Is Too Good | Tutorial | Procedural Building Generator | Unreal Engine 5.5," YouTube, Nov. 26, 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/using-shape-grammar-with-pcg-in-unreal-engine. [Accessed: Jan. 18, 2025].

[17] Lukáš Jůza, "Procedural Road Generation System," GitHub Repository. [Online]. Available: https://gitlab.fel.cvut.cz/juzaluk2/pcg-and-terrain-generation-in-ue5.5. [Accessed: Jan. 18, 2025].

[18] Lukáš Jůza, "Procedural Road Generation System," GitHub Repository. [Online]. Available: `https://gitlab.fel.cvut.cz/juzaluk2/pcg-and-terrain-generation-in-ue5.5`. [Accessed: Jan. 18, 2025].

[19] K. Perlin, “An Image Synthesizer,” *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, July 1985.

[20] S. Gustavson, “Simplex Noise Demystified,” Linköping University, Sweden, Mar. 22, 2005. [Online]. Available: `https://itn-web.it.liu.se/~stegu76/TNM084-2011/simplexnoise-demystified.pdf`. [Accessed: Jan. 20, 2025].

[21] J. Peck, “FastNoiseLite: a portable single-header open source noise generation library,” GitHub, 2020. [Online]. Available: `https://github.com/Auburn/FastNoiseLite`. [Accessed: Jan. 20, 2025].

[22] Dan O., “Voxel cave generation using 3D Perlin noise isosurfaces,” [Online]. Available: `https://blog.danol.cz/voxel-cave-generation-using-3d-perlin-noise-isosurfaces/`. [Accessed: Jan. 20, 2025].

[23] Epic Games, “Programming with C++ | Unreal Engine 5.5 Documentation,” *Unreal Engine Documentation*, [Online]. Available: `https://dev.epicgames.com/documentation/en-us/unreal-engine/API`. [Accessed: Apr. 26, 2025].

[24] Epic Games, “UStaticMesh,” Unreal Engine 5.5 API Reference, [Online]. Available: `https://dev.epicgames.com/documentation/en-us/unreal-engine/API/Runtime/Engine/Engine/UStaticMesh`. [Accessed: Apr. 26, 2025].

[25] P. Felkel and Š. Obdržálek, “Straight Skeleton Implementation,” ResearchGate, 1998. [Online]. Available: `https://www.researchgate.net/publication/2398714_Straight_Skeleton_Implementation`. [Accessed: May 13, 2025].

[26] R. H. Whittaker, *Communities and Ecosystems*, 2nd ed., Macmillan, 1975.

[27] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*, Springer-Verlag, 1990.

[28] A. Patel, “Flow Field Pathfinding,” Red Blob Games Blog, Apr. 27, 2024. [Online]. Available: `https://www.redblobgames.com/blog/2024-04-27-flow-field-pathfinding/` [Accessed: May 1, 2025]

[29] P. Felkel and Š. Obdržálek, “Straight Skeleton Implementation,” in *Proceedings of the Spring Conference on Computer Graphics*, Budmerice, Slovakia, pp. 210–218, 1998. :contentReference[oaicite:0]index=0

[30] H. Wang, "Proving theorems by pattern recognition—II," *Bell System Technical Journal*, vol. 40, no. 1, 1961.

[31] M. F. Cohen and J. R. Wallace, "Wang Tile based Texture Synthesis," in *Proceedings of ACM SIGGRAPH*, 2003.

[32] G. Stiny and J. Gips, "Shape Grammars and the Generative Specification of Painting and Sculpture," in *Proceedings of the IFIP Congress*, 1972.

[33] S. Wolfram, "Cellular Automata as Models of Complexity," *Nature*, vol. 311, pp. 419–424, 1984.

[34] Y. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of ACM SIGGRAPH*, 2001, pp. 301–308.