

# Omnigenesys: A Universal Framework for Procedural Map Generation in Digital Games

Rodrigo Dubeux Mendes de Oliveira<sup>1</sup>, Erick Simões de Matos<sup>1</sup>

<sup>1</sup>Centro de Estudos e Sistemas Avançados do Recife (CESAR School)  
Recife – PE – Brasil

digodubeux@gmail.com, esm@cesar.org.br

**Abstract. Introduction:** *The creation of digital maps and scenarios is one of the most costly processes in game development, becoming a bottleneck especially in genres that rely on replayability and scalability. Procedural Content Generation (PCG) emerges as a promising alternative, yet current tools such as Houdini Engine, Unreal PCG Framework, and Dungeon Architect operate within closed ecosystems and lack standard interoperability between game engines, forcing developers to rebuild procedural content from scratch when migrating platforms. Objective:* This paper presents Omnigenesys, a universal and modular framework for procedural map generation, independent of any specific game engine, designed to address the fragmentation and lack of portability in existing PCG tools. **Steps:** *The framework was developed following the Design Science Research (DSR) methodology through iterative cycles of problem identification, requirements definition, artifact construction, and demonstration. The core is implemented in Go, exporting maps as structured JSON consumed by engine-specific adapters, with working integrations delivered for both Unity and Unreal Engine. Expected Results:* End-to-end pipelines have been demonstrated in both Unity and Unreal Engine, generating maps deterministically from a seed and assembling them at runtime. Formal evaluation of cross-engine reuse metrics remains as future work. **Keywords** Procedural Content Generation, Game Engine, Interoperability, Framework, Map Generation.

## 1. Introduction

Modern game genres such as roguelikes, sandbox, and open-world titles demand vast, varied environments that are impractical to craft entirely by hand [Shaker et al. 2016]. Procedural Content Generation (PCG) addresses this bottleneck by automating parts of the map and scenario creation process, reducing dependence on manual work and increasing variety and unpredictability [Hendrikx et al. 2013].

Despite these advances, the current landscape still presents significant challenges. The diversity of tools, methodologies, and game engines has resulted in a fragmented ecosystem in which interoperability between systems is limited [Hasnain et al. 2022]. The absence of a universal framework that enables the full integration of PCG techniques across multiple development contexts limits their potential for large-scale adoption. Currently, each solution is frequently restricted to a specific ecosystem or a particular domain. Commercial solutions such as Houdini Engine, Unreal PCG Framework, and Dungeon Architect are powerful but operate in silos: Houdini Engine is restricted

to editor pipelines and requires SideFX licensing; the Unreal PCG Framework is tightly coupled to Unreal Engine and offers no support for exporting content to other platforms; and Dungeon Architect, while compatible with Unity and Unreal, lacks a standard intermediate format that would allow portability to other engines such as Godot [Zamorano et al. 2024].

This lack of interoperability is the core problem. A studio that begins prototyping a game in Unity but later decides to migrate to Unreal for its advanced graphics capabilities must manually rebuild all procedural content, resulting in weeks of rework and loss of consistency between project versions [Zamorano et al. 2024]. Stefan et al. [Stefan et al. 2013] highlight that fragmented ecosystems and low cross-platform content reuse are recurring problems that limit collaboration and increase maintenance costs. This work addresses the following research question: *How can a universal framework for procedural content generation in digital games be developed that is engine-independent, modular, and extensible?*

## 2. Methodology and Development Steps

The methodology adopted in this work is based on Design Science Research (DSR), an approach that guides the development and evaluation of technological artifacts through iterative cycles. This method organizes the investigative process into stages that feed back into each other: problem identification, requirements definition, artifact construction, demonstration, evaluation, and communication, allowing the proposed solution to be continuously refined as new evidence or limitations emerge [Hevner et al. 2004]. The iterative nature of DSR is especially suitable for this study, which involves the creation of an intermediate model capable of standardizing procedural generation rules and its subsequent validation in distinct game engines.

The research is applied in nature, exploratory in scope, and descriptive in approach, analyzing how different engines structure and process PCG operations and documenting their data models, APIs, and pipelines.

### 2.1. Development Activities

The activities follow the DSR iterative cycle through six stages: (1) **Bibliographic review** survey of PCG literature and technical documentation of relevant engines; (2) **Requirements definition** specification of functional operators, the intermediate JSON format, and module boundaries; (3) **Core architecture design** design of data structures, operator composition strategy, and abstraction layers; (4) **Incremental core implementation** construction of the Go-based core with continuous consistency testing; (5) **Adapter and prototype development** implementation of engine-specific adapters and end-to-end validation prototypes; (6) **Technical evaluation** assessment using objective metrics aligned to the research problem.

### 2.2. Framework Architecture Omnigenesys

The resulting artifact, named **Omnigenesys**, is implemented in Go and structured around three core abstractions: *Grid2D* (the spatial data structure), *Pipeline* (a composable sequence of operators), and *Operators* (stateless, atomic transformations applied to the grid). Although the current implementation targets 2D grids, the operator and

pipeline abstractions are grid-agnostic and can be extended to 3D structures such as voxel grids without changes to the JSON contract or the adapter layer. The framework exports the generated map as structured JSON via `core/export`, which serves as the engine-agnostic intermediate format consumed by all adapters. Unlike scene-description standards such as USD or glTF, which encode renderable assets and spatial hierarchies, the Omnigenesys JSON encodes procedural rules, operator pipelines, and tile-level cell data information that has no representation in those formats. The main modules are organized as follows:

- `core/grid` `Grid2D`, `Layer`, and `Cell` structures
- `core/pipeline` `Pipeline`, `Operator`, `Context`, and composable `Condition` predicates
- `core/generators/noise` deterministic Perlin noise keyed by seed
- `core/generators/pathfinding` A\* with organic noise-based traversal cost
- `core/operators` terrain fill, noise scatter, placement, and path operators
- `core/export` JSON serialization to file or stdout
- `cmd/mapgen` CLI entry point accepting `-seed`, `-width`, `-height` flags

The noise module implements a deterministic variant of Perlin noise [Perlin 1985], guaranteeing that the same seed always produces the same output regardless of execution environment. Procedural rules are expressed as composable *Conditions*, cell-level predicates that can be combined freely to encode placement constraints without coupling game-specific logic to the operators. This design allows the same pipeline definition to run identically regardless of the target engine.

### 2.3. Engine Adapters

Both adapters follow the same integration strategy: Unity or Unreal invokes the compiled `mapgen` binary as a subprocess at runtime, receives the JSON map via stdout, and assembles the scene through engine-specific components, keeping all procedural logic inside the Go core.

The **Unity adapter** (installable via Package Manager) deserializes the JSON in C# using Newtonsoft JSON and assembles the scene through a `MapBuilder` component. Each JSON layer maps to a separate Unity Tilemap. Tile and prefab bindings are configured via a `TileRegistry` `ScriptableObject`, keeping all engine-specific configuration outside the core.

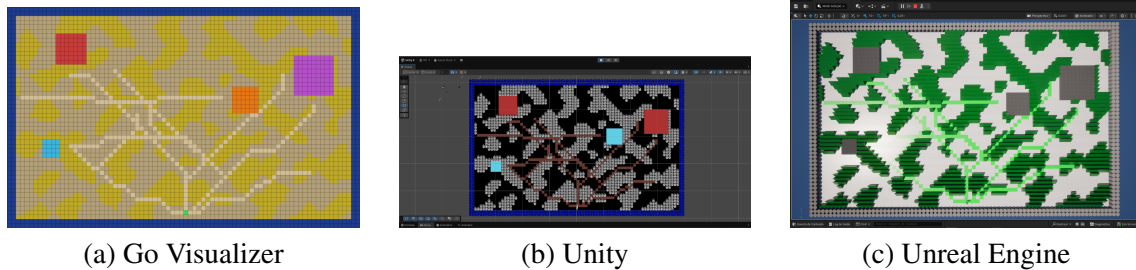
The **Unreal Engine plugin** (`unreal-plugin/`) exposes an `AMapGeneratorRuntime` Actor and a `TileRegistry` Data Asset. Generation can be triggered from Blueprints (`MapGeneratorRuntime` → `Generate`) or from C++ (`MapGenRuntime->GenerateWithSeed(42)`). Static Meshes and Actor Blueprints are mapped per tile type through the registry, preserving the same layer-based separation used in the Unity adapter.

### 2.4. Cornfield Prototype

To validate the end-to-end pipeline on both engines, a map called *Cornfield* was implemented as a reusable package (`maps/cornfield`), serving as the procedural map generation prototype for *Lost Fields*, a roguelike horror game under development. The

map is organized in four layers: *terrain* (floor, border vegetation, paths), *vegetation* (tall grass distributed by Perlin noise), *structures* (shop, challenge, arena, item buildings), and *entities* (player spawn marker). All placement rules are expressed as Conditions with no game-specific logic embedded in the operators, demonstrating that the same pipeline definition produces consistent output across both Unity and Unreal Engine.

Figure 1 shows the same map generated with seed 512 as rendered by the Go visualizer, Unity, and Unreal Engine, confirming output consistency across all three environments.



**Figure 1. *Cornfield* map generated with seed 512 across three environments. The same JSON output from the Go core is consumed by each adapter, producing identical layouts.**

### 3. Preliminary Results

The *Cornfield* prototype validated the following behaviors across both engine integrations: (i) **deterministic generation** identical seeds always produce the same map; (ii) **multi-layer composition** independent layers assembled and rendered separately with no coupling between generation logic and engine rendering; (iii) **composable placement rules** Condition predicates enforced without any engine-side logic; and (iv) **runtime generation** maps produced at game startup via subprocess with no bake step required in either Unity or Unreal Engine.

#### 3.1. Code Isolation

To quantify architectural isolation, lines of source code were counted with *cloc* (excluding comments and blanks). Table 1 shows that all PCG algorithms reside exclusively in the Go core: the adapters contain zero lines of generation logic no A\*, no Perlin noise, no scatter or fill operators. Their sole responsibility is IPC, JSON deserialization, and asset mapping.

**Table 1. Lines of code and responsibility per component.**

| Component                              | LOC          | PCG logic           |
|----------------------------------------|--------------|---------------------|
| Go core ( <i>core/</i> + <i>cmd/</i> ) | 1 333        | All (100%)          |
| Unity adapter (C#)                     | 250          | None (I/O + assets) |
| Unreal adapter (C++)                   | 503          | None (I/O + assets) |
| <b>Total</b>                           | <b>2 086</b> |                     |

The size difference between adapters (250 vs. 503 lines) reflects platform verbosity C++ requires separate header files and the Unreal adapter uses Instanced

Static Meshes with BFS flood-fill not additional procedural complexity. Both adapters share the same three-class structure (`MapGeneratorRuntime`, `TileRegistry`, `MapBuilder`) with identical responsibilities. Adding new operators to the Go core requires zero changes in any adapter, confirming that the architecture scales without coupling.

### 3.2. Generation Latency

To measure the overhead of the subprocess strategy, the `mapgen` binary was executed 51 consecutive times on an  $80 \times 50$  tile grid (seed 512, Cornfield pipeline) under Windows 10 22H2 (x86-64), timed with PowerShell `Measure-Command`. Table 2 summarizes the results.

**Table 2. Subprocess generation latency ( $80 \times 50$  grid, 51 runs).**

| Metric               | Value (ms) |
|----------------------|------------|
| Cold start (1st run) | 73.9       |
| Mean ( $n = 50$ )    | 35.5       |
| Std. deviation       | 9.9        |
| Min                  | 29.6       |
| Max                  | 94.3       |

The cold-start latency of 73.9 ms reflects OS binary loading and occurs once per game session. After warm-up, generation averages 35.5 ms with low variance, well within acceptable bounds for point-in-time scenarios such as level loading or area transitions. The single 94.3 ms outlier is attributed to OS scheduling rather than generation logic. These latencies are adequate for discrete generation but would constitute a bottleneck for continuous real-time streaming. The ongoing Rust rewrite targets native library compilation (`.dll/.so`), eliminating subprocess overhead while also improving console platform compatibility.

### 3.3. Integration Effort and Future Work

Both adapters share the same three-class structure subprocess launcher, asset registry, and scene builder with identical responsibilities. Integrating a new engine only requires reimplementing these three I/O and asset-mapping classes; all PCG algorithms remain untouched in the Go core. By contrast, without the framework a developer would need to rewrite the full generation pipeline grid structures, Perlin noise, A\* pathfinding, operators, and conditions in the target engine’s language.

These results address the core objectives of this work: mapping existing PCG tool limitations, designing an engine-agnostic architecture with proven modularity, delivering working adapters for Unity and Unreal Engine, and evaluating the framework through code isolation and latency metrics. The remaining objective releasing the framework as an open tool is fulfilled by the public repositories: the Go implementation used in this evaluation is available at <https://github.com/Cenafowzin/Omnigenesys-legacy>, and a Rust rewrite under a fully open-source license is in progress at <https://github.com/Cenafowzin/Omnigenesys>. Future work includes additional generator modules (cellular automata, Voronoi, BSP, Wave Function Collapse) [Freiknecht e Effelsberg 2017].

## References

- Freiknecht, J. e Effelsberg, W. (2017). A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4):27.
- Hasnain, S., Togelius, J., e Risi, S. (2022). A survey of procedural content generation for games. In *IEEE Conference on Games*.
- Hendrikx, M., Meijer, S., Velden, J. V. D., e Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications and Applications*, 9(1).
- Hevner, A. R., March, S. T., Park, J., e Ram, S. (2004). Design science in information systems research. volume 28, pages 75–105.
- Perlin, K. (1985). An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296.
- Shaker, N., Togelius, J., e Nelson, M. J. (2016). *Procedural Content Generation in Games*. Springer.
- Stefan, I. A., Warmelink, H. J. G., Arnab, S., Dagnino, F., Lo, J., e Mooney, J. (2013). Accessibility, reusability and interoperability in the european serious game community. In *9th International Conference eLearning and Software for Education*, volume 2, pages 55–60.
- Zamorano, M., Zaragoza, U. S. J., e Cetina, C. (2024). Game software engineering: A controlled experiment comparing automated content generation techniques. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24)*.