Computer Science Department

Software Engineering & Business Analysis

Bachelor's Thesis

Procedural generation of game maps using graph rewriting

Marko-Oleksandr Hlibovytskyy

Supervisor
KSE, Oleksandr Syrotiuk, PhD, osyrotiuk@kse.org.ua

Submission date
16 June 2026

TODOS

Academic Integrity Statement

I, undersigned, hereby declare that this capstone project is the result of my own work.

- All ideas, data, figures and text from other authors have been clearly cited and listed in the bibliography.
- No part of this project has been submitted previously for academic credit in this or any other institution.
- All code, diagrams, and third-party materials are either my original work or are used with permission and properly referenced.
- I have not engaged in plagiarism or any form of academic dishonesty.
- Any assistance received (e.g. from peers, tutors, or online forums) is acknowledged in the acknowledgements section.

I understand that failure to comply with these declarations constitutes academic misconduct and may lead to disciplinary action.

Place, date Kyiv, 04.06.2026

Signature _____

Contents

Acknowledgements

Abstract

Procedural generation is widely used in games to reduce reliance on manually created content and to increase replayability. However, generated content can become predictable when players begin to recognize repeated structures or generation patterns. This thesis focuses on procedural generation of game maps composed of hexagonal cells in Unity and explores whether a graph rewriting approach can provide a configurable framework for generating varied maps.

The objective of the thesis is to design and implement a proof-of-concept framework that represents a map as a graph of hexagonal cells and allows generation rules to transform this graph. The research phase explores existing procedural generation tools and selected game examples to identify available capabilities in solutions and gaps.

The framework is capable of generating hexagonal maps with meaningful terrain relationships and repeatable outputs from fixed settings and seeds. At the same time, the prototype remains limited to small and medium-sized maps, existing rule components, and conceptual validation through generated examples rather than a complete playable game or formal user study.

1 Introduction

1.1 Background and motivation

Game studios face a persistent challenge: manually created content is expensive, time-consuming, and finite, while the casual player base expects large, expansive worlds and replayability [1].

Although, game developers have found a way to compensate for this problem. Procedural generation addresses this issue by allowing developers to create content via algorithms rather than relying solely on asset designers. Most obvious cases are generated game worlds.

A few prominent examples:

- No Man's Sky
- Minecraft
- Civilization series
- Heroes of Might and Magic series (HoMM)
- Age of Wonders
- Borderlands series
- Deep Rock Galactic
- Don't Starve Together

1.2 Problem statement

In practice, Procedural Content Generation (PCG) has solved only part of the problem. After repeated play, players may begin to recognize generation patterns. The moment the player gains a working understanding of the generation system is pivotal to their perception of the game. The interest that draws the players in weakens as the sense of novelty fades.

Procedural generation increases replayability, but it does not guarantee meaningful variety. Problems leading to predictable generation can be succinctly summarized into two:

Flatness: lack of depth, interactions, and connectivity between features.

Monotony: insufficient variance in generation caused by weak randomization or a limited set of statically-made features.

1.3 Research objective and scope

The scope of this thesis was deliberately limited to procedural generation of maps using grids made of Hexagonal grids (Hex grids). This limitation exists to provide a well-defined area of practical application for procedural generation. As mentioned above, procedural generation covers many possible applications, therefore attempting to support all of them would make the project difficult to explain, design, and evaluate.

This thesis narrows procedural generation to a specific technical setting – hexagonal game maps generated inside Unity using graph rewriting. The detailed reasoning behind these choices is discussed later in this section and other chapters, but they are introduced here because they define the object of research and the boundaries of the prototype.

Object of research – procedural generation of game maps composed of hexagonal cells, implemented in Unity through graph rewriting.

In this Proof of concept (Proof of concept) a Graph (Graph) is defined as:
- **Vertex (Vertex)** – hexagonal cell.
- **Neighbors** – vertices connected via Edge (Edge).
- **Graph structure** – grid gapless hexagonal tiles.

Generation process:
1. Generation rules look for patterns defined by designers
2. If pattern matches, the graph is rewritten according to the designer-defined active rule

Reasoning behind Hexagonal cells (Hexagonal cells):
1. Enough examples exist for evaluation in strategy/tactical/puzzle games.
2. Map type restriction keeps the topic from becoming too broad.
3. Hexagon-shaped cells allow for more complexity than square cells, which are more common in video games.
4. Player cell-to-cell movement is equidistant in all directions to neighboring cells, creating more even movement than square cells.
5. Hexagonal cells require a more complex implementation than square cells, making future support for square or other cell types simpler than implementing a new framework from the ground up.

In addition, this limited scope gives clear scenarios to benchmark against. Maps made up of hexagons are not that common in video games, mostly appearing in strategy/tactical, puzzle genres of games. For the most part, strategy games will be used as benchmarks.

1.4 Aim and Objectives

The aim of this thesis is to design and implement a proof-of-concept procedural generation framework for hexagonal game maps in Unity.

Objectives:
1. Analyze existing uses of procedural generation in games.
2. Identify causes of predictable or repetitive map generation.

3. Define requirements for a designer-friendly generation framework.
4. Design a configurable model for hexagonal map generation.
5. Implement the framework as a Unity prototype.
6. Validate the prototype through generated map examples for example game concepts.

The framework aims to expose designers to features enabling them to configure generation in a flexible way. Generation set up should happen entirely within the engine editor, requiring no code. As a result, designers should be empowered to create more meaningful and varied generation without the prerequisite of having a lot of technical knowledge.

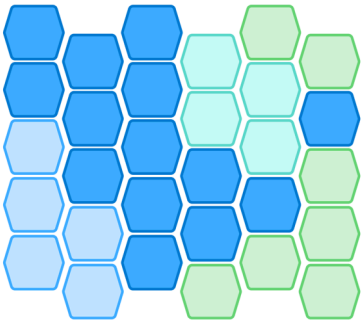

Figure 1: Sample square hexagonal grid

Framework uses a hexagonal cell grid to build game worlds

Aim: proof-of-concept Unity framework thatprovides game designers with flexibility in generationdesign

Main context for benchmarking – strategy games

1.5 Methodology

This thesis follows an applied engineering approach.

At first, existing procedural generation approaches and relevant game examples are reviewed. Following that, requirements are defined for a configurable proof-of-concept framework. After requirements are defined, the system is designed and built in Unity. Lastly, the prototype is evaluated using example game concepts and comparison against existing games.

1.6 Structure of the thesis

- **Chapter 2** – Research of related work and approaches to procedural generation.
- **Chapter 3** – design of the proposed framework.
- **Chapter 4** – implementation in Unity.
- **Chapter 5** – validation of the framework through selected generation scenarios.
- **Chapter 6** – summary of the results and thesis.

2 Research

2.1 Research questions and initial functional requirements

In this chapter, existing procedural generation tools, frameworks, and game examples will be examined in order to identify existing capabilities and which gaps are left open. The research focuses on map generation workflows, designer control, tool accessibility, and support for hexagonal grid-based generation. Observations from the research will be used to derive requirements for the prototype framework.

Research questions:

1. What existing procedural generation tools and frameworks are available for game map generation, especially in Unity?
2. How do existing tools balance designer control, ease of use, flexibility, and technical complexity?
3. To what extent do existing tools support hexagonal grid-based map generation?
4. What gaps in existing tools justify the development of a focused Unity framework for hexagonal map generation using graph rewriting?
5. How do existing video games use procedural generation for map variety and feature-richness and what generation options are exposed to players or designers?

Based on the thesis scope defined in Section 1 and the following research, the framework should satisfy the following functional requirements:

ID	Requirement
1	The framework shall produce a game map as a graph of hexagonal cells.
2	The framework shall allow designers to define map generation rules based on global or local cell patterns.
3	The framework shall allow designers to define what pattern matches are considered valid candidates for rule transformations.
4	The framework shall allow rules with matches that are considered valid to transform the grid during map generation.
5	The framework shall expose generation configuration inside the Unity editor Inspector (Inspector) window.
6	The framework shall allow map generation without requiring designers to write little to no code.
7	The framework shall support repeatable and stable generation through configurable parameters and Seeds (Seeds), which will guarantee a deterministic result across configurations.
8	The framework shall provide visual feedback for generated hexagonal maps inside the Unity editor scene view.

Table 1: Functional requirements for the proposed framework

Important note: Although hexagonal grids are common in strategy/tactical, puzzle games, the Proof of concept framework is not limited strictly to these genres. Genre compatibility serves us in the context of the defined strategy-centric scope rather than being a strict requirement.

2.2 Candidate solutions

The following items will be inspected in the reviewed solutions:
- Flexibility in generation setup and settings.
- Designer-frendliness/code use for setup.
- Unity integration.
- Cost and accessibility.
- Learning curve.
- How wide or narrow the scope of existing solutions is.

Since the framework will be developed using the Unity engine, soultion exploration will begin with the Unity Asset Store.

Packages were found by searching “procedural generation”, “procedural map generation”, “procedural world generation” and other search terms with minimal changes. Most of the reviewed packages were categorized under **Tools/Terrain**, **Tools/Level Design** or **Tools/Utilities**

Valid candidates for evaluation found in the asset store:
1. DunGen
2. RoomGen - Procedural Generator
3. Edgar Pro - Procedural Dungeon Generator
4. Dungeon Architect
5. Procedural Generation Grid

2.2.1 Unity Asset Store map generators

Important note: Unity Asset Store packages often provide limited public documentation before purchase. As a result, the evaluation relies on store descriptions, available documentation, public reviews, and community discussions where official documentation is unavailable.

The 5 listed candidates can be categorized into 2 categories:

Category	Reviewed solutions
Room/prefab-based dungeon generators	DunGen, RoomGen, Edgar Pro, Procedural Generation Grid
Broad procedural generation frameworks	Dungeon Architect

Table 2: Categories of reviewed procedural generation solutions

To make the comparison easier to follow, candidates will be compared together by category. First, candidates will be reviewed according to categories listed in the previous subsection. Then, in the Candidate evaluation subsection, packages will be compared to one another.

Room/prefab-based dungeon generators:

1. DunGen: [2], [3]

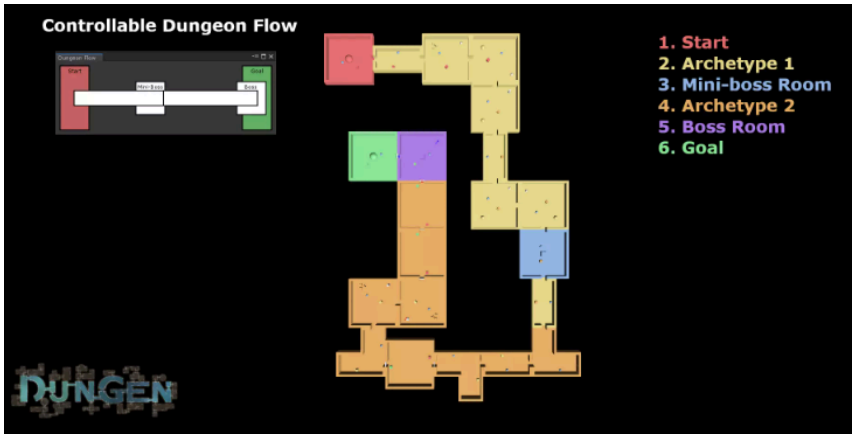

Figure 2: DunGen map view during the generation process, from the Unity Asset Store page [2].

1. Flexibility in setup and settings:
   - Node-based room relationship setup allows for easy generation customization. User defines dungeon flows, tile sets, branching paths, locks, keys, object placement rules.
   - Setup focuses on dungeon-style layouts composed of room prefabs.
   - Uses a visual dungeon flow graph.
   - Archetypes consist of one or more tile sets and are used to define behavior for dungeon sections (branching, path straightening, branch caps and branch pruning and so on).
2. Designer-friendliness:
   - The package is designed around Unity editor assets and visual configuration. Little to no coding is required for basic use.
   - Coding is required when the user wants behavior that cannot be expressed through its inspector assets and built-in rules.
3. Unity integration:
   - Developed as a Unity Asset Store package
   - Workflow closely tied to Unity concepts.
4. Cost and accessibility:
   - Cost: listed for **`$79.99`**.
   - Listed as a paid Asset Store package.
   - More accessible than large professional tools, which will be considered later, but still commercial.
5. Learning curve:

- Basic setup is approachable, however dungeon flow concepts and tile rules require some learning.

6. Scope of the solution:
   - The package primarily focuses on procedural dungeon generation using connected rooms/tile prefabs.
   - The scope is relatively narrow. The package aims to provide designers with a flexible dungeon generator, but it also includes assets for generation.
   - Markets itself as a one-stop-shop for dungeon generation since it includes a generation framework and assets for map generation.

2. RoomGen: [4]

1. Flexibility in setup and settings:
   - RoomGen focuses on generating rooms, buildings, modular interiors, dungeons and dense landscapes from modular tiles and presets which the user provides.
   - Setup is based around configurable presets. A designer creates a preset, assigns tiles and decorative objects, adjusts generation settings, and generates the result inside the Unity editor.
   - Package allows depth in dungeons via multi-floor setups, where each floor can use different presets and settings.
   - RoomGen exposes many customization options for map generation like tile weights, object placement probabilities, object spacing and other spacial placement settings.
   - It does not expose an explicit graph-based workflow.

Figure 3: Dungeon screenshot from the RoomGen Unity Asset Store page [4].

Figure 4: Landscape screenshot from the RoomGen Unity Asset Store page [4].

1. Designer-friendliness:
   - Package explicitly designed around ease of use. Package advertises a workflow consisting of preset creation, dragging and dropping tiles and objects within the Unity editor and generating inside the Unity editor.
   - Basic use does not require coding. Most features are set up inside the Unity editor.
   - Code required when using RoomGen at runtime. If designers choose to generate dungeons during gameplay, then scripts need to be used to trigger map generation.
2. Unity integration:
   - Developed as a Unity Asset Store package
   - Workflow closely tied to Unity concepts.
3. Cost and accessibility:
   - Cost: **$39.99**.
   - Listed as a paid Asset Store package.
   - Compared to other candidate solutions and larger tools which will be inspected later, RoomGen is relatively accessible in cost and ease-of-use.
4. Learning curve:
   - Basic workflow for map generation setup is simple because it is based on presets, modular tiles and object placement rules.
   - Without the use of scripts, maps can only be generated in the Unity editor. For dynamic runtime generation, coding is required.
5. Scope of the solution:
   - Broader scope than pure dungeon map generation. Aside from dungeon generation, rooms, buildings and landscapes can be generated.
   - The package is centered around modular room, dungeon, building, landscape creation and variation. Allows designers to create varied sets of assets from prefabs.

3. Edgar Pro: [5], [6]

1. Flexibility in setup and settings:
   - Edgar Pro focuses on generating 2D dungeons.
   - Its generation model consists of handlade room templates and a graph containing the level structure description.
   - The level graph is the central structure. Nodes represent rooms and edges represent necessary connections between rooms. The graph describes which rooms must exist and how rooms should be interconnected.
   - Rooms can be connected directly or via corridors.
   - The level graph provides explicit control over dungeon structure.
   - The package stitches rooms together, but does not modify the rooms themselves.

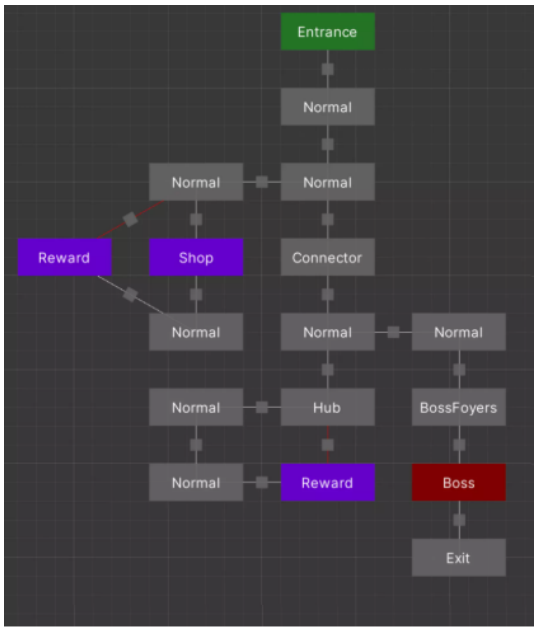

Figure 5: Level graph screenshot from the Edgar Pro Unity Asset Store page [5].

1. Designer-friendliness:
   - Visual workflows inside the Unity editor are used to configure level graphs and room templates.
   - Basic generation setup is performed entirey in the Unity editor.
2. Unity integration:
   - Developed as a Unity Asset Store package.
   - Workflow closely tied to Unity concepts.
3. Cost and accessibility:
   - Cost: **$55**.
   - Edgar Pro is a paid Unity Asset Store package.
4. Learning curve:

- Edgar Pro makes basic setup approachable. However, a lot of details need to be set up for dungeon generation. For example, room templates, tilemaps, room outlines, door positions, corridor templates, etc.
- The level graph facilitates flexible setup, but it requires the user to think in terms of room relationships to generate interesting layouts.

5. Scope of the solution:
   - Edgar Pro is narrow in scope, but provides user with deep flexibility in its niche.

Procedural Generation Grid: [7], [8]

1. Flexibility in setup and settings:
   - Procedural Generation Grid uses grids to define fields and paths for procedural generation.
   - First grid size and prefabs are defined, then the designer sets up generation rules.
   - Generation rules handle mutation of the grid via grid modifiers. Designers define rule conditions which have to be true for grid mutation to take place.
   - Includes a Node Graph system, allowing designers to combine multiple field setups into a larger generation workflow. This makes it possible to compose several grid generation stages into a more complex map generator.

Figure 6: Generated dungeon screenshot from the Procedural Generation Grid Unity Asset Store page [7].

1. Designer-friendliness:
   - Package provides visual scripting-like tools for defining its generation workflows. Basic setup is done entirely within the Unity editor with the help of the package's visual tools.
   - Coding is not required for basic use, but designers are still required to understand rule-based generation, grid modifiers, and other concepts to set up map generation.
2. Unity integration:
   - Developed as a Unity Asset Store package.
   - Workflow closely tied to Unity concepts.
3. Cost and accessibility:
   - Cost: **`$45.99`**.
   - Procedural Generation Grid is a paid Unity Asset Store package.
   - Price falls within expected range for narrow scope packages which is based on previously reviewed candidates.

4. Learning curve:
   - For the most part the package avoids coding for its workflows. Despite that, the package requires the designer to learn the concepts in its generation model to utilize the flexibility of the map generator.
5. Scope of the solution:
   - Scope is closer to this thesis as it utilizes grid cells and rule-based generation.
   - Fairly narrow scope, focusing on dungeon-like generation and prefab placement.

Broad procedural generation frameworks:

Dungeon Architect: [9], [10]

1. Flexibility in setup and settings:
   - The package includes several builder types like, including Grid Flow Builder, Snap Builder, Grid Builder, Snap Grid Flow Builder and City Builder.
   - The Grid Flow Builder allows designers to create node based dungeon flows with cyclic paths, a variety of ways to connect rooms like teleporters and one-way doors, tilemap-based layouts.
   - Dungeon Architect provides very high flexibility. Its flexibility comes from supporting many different procedural workflows.

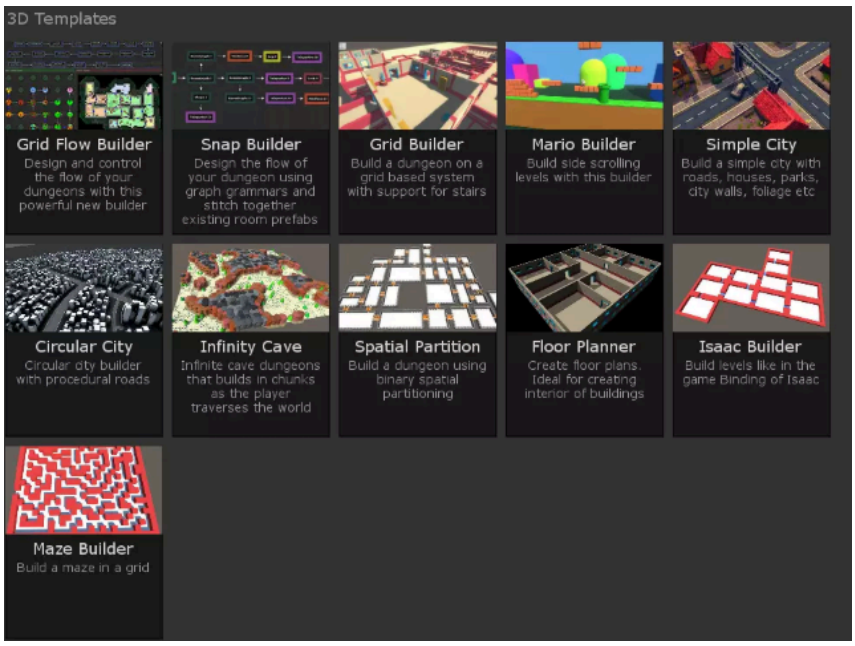

Figure 7: 3D templates screenshot from the Dungeon Architect Unity Asset Store page [9].

1. Designer-friendliness:
   - The package provides visual Unity editor tools for most of its systems.
   - Package comes with many specialized systems
   - Coding required for advanced customization.
2. Unity integration:
   - Developed as a Unity Asset Store package.
   - Workflow closely tied to Unity concepts.

3. Cost and accessibility:
   - Cost: **$300**.
   - Dungeon Architect is a paid Unity Asset Store package.
   - Comes at a significantly higher price than previous packages but is more feature-rich. At this price point, the package is less accessible for small teams, hobbyists and students.
4. Learning curve:
   - Due to the sheer amount of features and options presented to the designer, the learning curve is high.
   - Designers have to get aquainted with generation worflows and different builders offered by the package.
5. Scope of the solution:
   - Dungeon Architect's scope is quite wide and feature-rich.
   - This package could be considered a full procedural generation suite, rather than a map, dungeon or landscape builder.
   - For this thesis, Dungeon Architect is relevant because it demonstrates mature visual procedural generation in Unity. This package establishes what a wider scope includes.

2.2.2 Non-Unity generators

Unreal Engine 5:

Unreal Engine Procedural Content Generation Framework: [11], [12]

1. Flexibility in setup and settings:
   - Unreal Engine's PCG Framework is a built-in procedural generation toolset for creating procedural content inside Unreal Engine.
   - Utilizes procedural node graphs that can generate content in the editor and at runtime.
   - Flexibility stems from a general-purpose graph workflow. Designers express generation rules through connected nodes in a visual graph.
   - Connections between nodes define order and relationships between operations.
   - Node operations cover a wide array of features. For example, nodes can generate sets of points, use attributes for filtering, transform points in numerous ways, spawn actors or static meshes and combine these operations in a wide range of scenarios.
   - This allows the framework to support a wide range of procedural tasks, such as environment population, foliage placement, biome generation, object scattering, spline-based generation, and runtime procedural content.
   - This framework is tied to Unreal Engine's ecosystem and does not support Unity.
2. Designer-friendliness:
   - PCG provides visual graph tools, making it accessible without traditional scripting for many procedural tasks.
   - However, designers need to be familiar with Unreal Engine 5, PCG graphs, runtime generation behavior, spatial constraints and the other many concepts that the generation features are composed of.

3. Unity integration:
   - Unreal PCG is not integrated with Unity. It is part of Unreal Engine.
   - Because this thesis targets a Unity-based framework, Unreal PCG is useful as a point of comparison but not as a direct candidate solution.
4. Cost and accessibility:
   - Free.
   - The framework is available inside Unreal Engine 5.
   - Access to the framework comes with adopting Unreal Engine 5.
5. Learning curve:
   - The learning curve is significant for users who are not already familiar with Unreal Engine 5.
   - For teams already using Unreal, PCG is powerful and accessible. In the context of this thesis, it is used as a point of reference.
6. Scope of the solution:
   - Unreal PCG has a broad scope, providing a solution for many procedural generation-centric problems.
   - PCG is a demonstration of what mature procedural generation tooling can provide.

Houdini FX:

Houdini FX and Houdini Engine: [13], [14], [15]

1. Flexibility in setup and settings:
   - Houdini FX is a professional procedural content creation tool based on node networks.
   - Houdini possesses the most expansive set of feature out of all the candidates. It can be used for procedural modeling, terrain generation, destruction, fluids, cloth, particles, simulations, asset generation, and game content pipelines.
   - Houdini workflows can be packaged as Houdini Digital Assets, which expose selected parameters to artists or designers.
   - Through the Houdini Engine, digital assets created inside the engine can be loaded into game engines such as Unity and Unreal Engine, allowing procedural controls to be used inside the engine editors.
   - This makes Houdini one of most flexible reviewed solution, but its flexibility comes from being a full procedural Digital Content Creation (DCC) and technical art platform rather than a focused game asset generator, let alone a map or dungeon generator.
2. Designer-friendliness:
   - Houdini can expose simplified controls through Houdini Digital Assets (HDAs), but creating those assets requires significant technical knowledge.
   - To create a procedural system, a technical artist must understand Houdini's node workflow, geometry processing, parameters, asset packaging, and possibly scripting.
   - Therefore Houdini can be designer-friendly for the end user of a prepared asset, but not necessarily for the person building the generator asset itself.
3. Unity integration:

- Houdini Engine provides a Unity plugin that allows Houdini Digital Assets to be used inside the Unity editor.
- The Unity plug-in does not replace Houdini itself. A Houdini Engine, Houdini Core, or Houdini FX license is required to run procedural assets through the plugin.

4. Cost and accessibility:
   - Houdini has free or lower-cost options for non-commercial or indie use, but production use depends on SideFX licensing.
   - Compared with Unity Asset Store tools, Houdini is less accessible for students or small teams if the goal is only a focused hexagonal map generator.
   - It also requires learning an additional professional tool outside Unity.
5. Learning curve:
   - The learning curve is very high compared with the reviewed Unity packages.
   - Houdini is powerful because it exposes low level procedural generation control, but this also means users must understand Houdini Engine concepts.
6. Scope of the solution:
   - Houdini has the broadest scope of all reviewed solutions.
   - It is capable of producing many types of procedural content, but this makes it excessive for the kinds of relatively specific problems that most reviewed Unity packages solve.

2.3 Candidate evaluation

Unity packages:

Most reviewed candidate Uinty packages provided solutions for reasonably narrow-scoped problems, with the exception of Dungeon Architect. DunGen, RoomGen, Edgar Pro, Procedural Generation Grid are focused on allowing game designers to create a map/dungeon for their game. Some offer slighly deeper functionality like multi-layer dungeons or object generation. The scope of these packages loosely resembles the scope of the proof-of-concept framework of this thesis.

Most notably, a relationship can be traced in setup generation among the packages. Despite all of the candidates being little-to-no-code solutions, designers still face the prerequisite of learning generation models and which components implement it. Coding is traded for workflows with visual setup inside the Unity editor, but learning the system is required in every case to a varying degree. For the game designers, this means that they still need to familiarize with technical aspects of the system.

Dungeon Architect encompasses significantly more map generation options than the rest of the packages. While it has deep functionality, its feature-richness goes far beyond the scope of the proof-of-concept framework of this thesis. For that reason, it does not serve as a good comparison. But it is worth restating that Dungeon Architect actively uses node-based setup for map/dungeon generation with the ability to have directed graphs, cyclical sections and a plethora of unique ways to connect nodes. The graph structure Dungeon Architect Templates provide is quite flexible. Still, it is limited in expressing the structure hex cell grid generation needs.

Procedural Generation Grid possesses qualities that are very similar to what my requirements describe. Though it uses square cell grids, the generation model is conceptually similar. The designer provides prefabs the dungeon generator can use during generation. Then, generation rules are defined by the designer. In short, rules consist of defining how cells can be placed in relation to others. The generation model is flexible, making it a suitable example to compare against.

Explicit hexagonal cell support is not mentioned explicitly anywhere. Most packages outright do not support them.

Non-Unity solutions:

Unreal Engine PCG:

This framework has a broad scope, allowing for versatile generation and covering different use cases. It does not support Unity, so it is not a valid candidate for proof-of-concept implementation. But it serves as a great reference for a full suite of procedural generation tools.

It has the capability to generate hexagonal cell grids, but the functionality of the node operations can still fall short of a standalone solution aimed specifically at hex cell maps.

Unlike solutions offered by the Unity Asset Store, Unreal Engine's PCG required deeper fundamental knowledge of the game engine and generation tools.

Houdini FX:

Since Houdini is a Digital Content Creation and technical art platform, this solution has applications ranging outside editor tools and game asset generation. While its wide scope sets Houdini apart from other reviewed solutions, it also sets itself apart due to sheer complexity of the tool. Houdini Assets can be more user friendly and centered around a specific concern, the creation of said assets still requires a lot of technical knowledge. Because of Houdini's complexity and lisence pricing, it is the least accessible solution out of all candidates.

Like Unreal Engine's PCG, Houdini FX is not a suitable platform for the creation of a proof-of-concept framework. The exact same reasons apply – the platform's feature set is far too broad for my requirements and the system is not accessible enough.

Solution	Platform	Main scope	Role in this thesis
DunGen	Unity	Dungeon / room-based generation	Narrow benchmark
Edgar Pro	Unity	Dungeon / level generation	Narrow benchmark
Dungeon Architect	Unity / Unreal	Broad dungeon and level generation framework	Strong but oversized benchmark
Procedural Generation Grid	Unity	Grid-based procedural generation	Closest Unity benchmark
Unreal PCG	Unreal Engine	General procedural content framework	Reference system
Houdini / Houdini Engine	External tool / Unity integration	General procedural asset generation	Reference system

Table 3: Positioning of reviewed procedural generation solutions

Solution	Unity integration	Hex grid support	Designer-facing workflow	No-code setup
DunGen	Yes	No / unclear	Yes	Partial
Edgar Pro	Yes	No / unclear	Yes	Partial
Dungeon Architect	Yes	Possible	Yes	Partial
Procedural Generation Grid	Yes	Partial / closest	Yes	Partial
Unreal PCG	No	Possible with custom work	Yes	Partial
Houdini Engine	Partial	Possible with custom work	Partial	No / technical

Table 4: Feature fit of reviewed tools against the thesis scope

2.4 Gaps and missing features

Some Unity packages contained node-based logic and workflows and some also used grids for prefab arrangement and map generation. Unity candidates provide flexible, relatively designer-friendly and relatively accessible solutions. But not a single one of them satisfies the requirements for the generation of a grid of gapless hexagonal tiles.

Unreal Engine PCG and Houdini FX offer capabilities to implement similar generation models. For the most part, they do not offer Unity support, require significantly more technical knowledge and are less accessible, with the exception of Unreal Engine's PCG being free. These solutions have a significantly smaller gap in missing features for the

satisfaction of my requirements, but PCG and Houdini will serve only as reference in this thesis.

Capability	Found in reviewed tools?	Research gap
First-class hexagonal cell grid generation	No	Reviewed tools do not directly target the exact map type used in this thesis.
Unity native hex cell grid map generation workflow	No / limited	Unity tools mostly focus on dungeons, rooms, terrain, or general grid workflows.
Designer defined graph rewriting rules	No	Reviewed tools do not expose generation as designer defined pattern matching and graph transformation.
Focused hex cell grid map scope	No	Broader systems can theoretically support this, but they are not designed around this specific use case.
No-code configuration inside Unity Inspector	Partial	Some tools expose settings visually, but not for configurable hex cell graph rewriting.
Deterministic hex-map generation for comparison	Partial	Repeatable generation may exist, but not in combination with the specific hex-grid rule system proposed here.

Table 5: Missing features related to the proposed hexagonal map generation scope

2.5 Video game examples

Understanding how games use procedural generation is vital to understanding what solutions developers may run into. The video games that will be review below serve as design benchmarks rather than technical competitors. The point is not to answer questions like "Which game has the best generator?"

For this section, the following questions will be used:

1. What do successful games expose to users?
2. What kinds of variety do their generators create?
3. Do generated features interact meaningfully?
4. Do any of them use hex cells in a way relevant to this thesis?

2.5.1 Games with hexagonal cell grid maps

Civilization V:

Civilization V is a turn-based 4X strategy game and one of the most relevant examples for this thesis because its main game map is built from hexagonal tiles. Civilization V introduced hexagonal tiles to the series. This makes the game a useful reference point for evaluating how hexagonal maps support:

1. strategic movement
2. expansion
3. terrain evaluation
4. replayability

Civilization V provides a notable assortment of player-facing setup options. It does not offer low level control over generation. Before starting a match, players can choose parameters such as map type, map size, world age, temperature, rainfall, sea level and resource abundance. These options allow the player to influence the generated world without needing to understand the internal generation algorithm.

The generator produces variety in generated maps with the help of interactions of several map features. Terrain, water, mountains, resources, natural wonders, city-state placement, civilization starting positions and expansion space availability all affect the gameplay value of a generated map. This depicts that map variety isn't simply tied to changes like map size, shape of the land or adding more features. Generation variety stems from how generated features are arranged and interact with one another. Civilization V game mechanics utilize different possible arrangements of features to create different gameplay scenarios for players, inciting replayability.

A little more about the gameplay – power dynamics between civilizations can vary, sometimes greatly, because of the map settings. Especially if one or several of the players may get lucky with luxury resource and natural wonder generation. Generation of these strategic points of interest may depend on one another and can be less or more likely depending on the settings. On top of that, the individual abilities civilizations have may allow them to harness such an advantage gained from the map generation to an even further extent.

Civilization V is a prominent example as its hexagonal map is not only a visual grid. Hex cells are central to movement, combat positioning, city placement, resource access, and territorial expansion. The hex grid is part of the game logic rather than just a display format. This supports the thesis assumption that hexagonal cells are a suitable structure for procedural map generation that creates flexibility in design and gameplay.

However, the game does not provide a reusable generation framework. Its generator is tightly connected to the rules and specific balance of the game. Players can configure high level parameters, but they cannot define their own generation rules or graph transformations.

In the context of this thesis, Civilization V serves as a design benchmark as it demonstrates the value of hexagonal maps and interacting generated features, but it does not provide the kind of designer configurable Unity framework proposed in this work.

Figure 8: Civilization V gameplay screenshot captured by the author.

Age of Wonders 4:

Age of Wonders 4 (AoW 4) is a turn-based 4X strategy game with tactical combat. It is relevant to this thesis because its strategic map uses hexagonal tiles and because its world generation exposes a large number of high level configuration options to the player.

In AoW 4, generated maps and gameplay scenarios are called realms. AoW 4 allows players to shape the generated world through realm traits. These traits can affect geography, climate, inhabitants, special world conditions, and the presence of powerful factions or enemies. The resulting generated map is not only varied spatially, but also thematically and mechanically.

The game creates variety in generated maps through interacting systems. Geography affects movement, expansion, and access to resources. Climate and terrain influence the usability and strategic value of different regions. Inhabitants, hostile enemies, free cities which the player can claim, and special realm modifiers can change how the player approaches the main game mechanics – exploration, combat, expansion, and diplomacy. Replayability is not created merely by rearranging terrain, enemy placement, free city placement and other features. The game combines different map layout with gameplay systems that respond to the features of the generated world.

The strength of AoW 4 as an example is that its generation is configurable while remaining easy to understand for players. Realm setup options allow players to influence the type of experience they want without making low level generation changes. This is relevant to the proposed framework for the reason that it reinforces the value of exposing generation control through readable, designer facing options.

Hexagonal cells are also important to the gameplay of AoW 4. The hex grid structures movement, exploration, territorial expansion, army positioning, and tactical decision making. Similar to Civilization V, the hexagonal map is not merely a visual representation. Hex tiles directly influence the game's strategic logic and affect how players understand and influence power dynamics between them and their rivals.

Age of Wonders 4 does not provide a reusable procedural generation framework, like Civilization V. Its generation system is deeply tied to the game setting, faction design and various systems. Players have high level control over map properties, but they cannot define custom generation rules or reusable map generation logic.

In the context of this thesis, Age of Wonders 4 serves as a design benchmark for configurable, feature-rich hexagonal map generation. It demonstrates that meaningful procedural variety can come from the interaction of geography, world traits, factions, terrain, and gameplay rules.

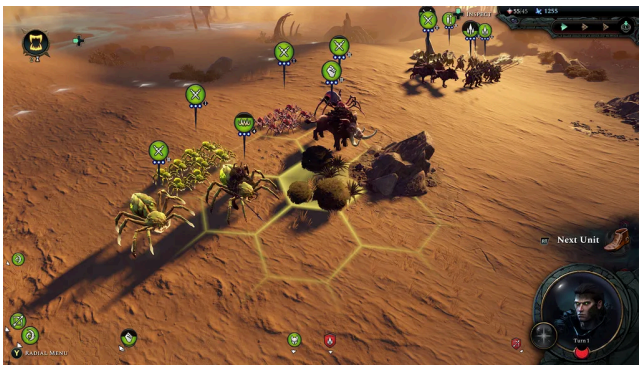

Figure 9: Age of Wonders 4 gameplay screenshot from Gideon's Gaming [16].

Heroes of Might and Magic III:

Heroes of Might and Magic III (HoMM III) is a turn-based strategy game with exploration, resource collection, town development, and tactical combat. It serves this thesis as a comparison point since it separates two different grid use cases:

1. overworld exploration takes place on a square tile adventure map
2. combat takes place on a hexagonal combat grid

The game shows that grid choice can depend on the type of interaction being represented. On the adventure map, the player navigates terrain, visits buildings, collects resources and interacts with various map objects. These objects are often visually and spatially aligned with a square map structure. Square tiles have the upside of making the adventure map readable and practical for placing buildings, roads, obstacles, and other designer-made objects.

Combat, however, uses a hexagonal grid. Hexagonal cells suit this context well. Tactical positioning, movement range, and attack distance are central to gameplay. As stated in Section 1, hexagonal tiles provide more neighboring directions than square tiles without introducing diagonal distance inconsistencies. In the context of HoMM III, hexagonal tiles facilitate more complexity and tactical expression in the combat system. Compared with square cells, hexagons avoid diagonal movement inconsistencies and provide uniform distances to neighboring tiles. This makes them useful for battle systems where relative position and distance need to be clear.

This distinction is relevant to the thesis as it shows that hexagonal cells are not automatically the best choice for every game map. They are particularly useful when movement, adjacency, and tactical positioning are important for the quality of gameplay. At the same

time, square grids may be easier for arranging buildings, rectangular objects, roads, and other various map structures. This supports the thesis decision to focus specifically on hexagonal game maps where the hex grid has gameplay meaning, rather than treating hexes as a universal end-all-be-all replacement for all map types.

In the context of this thesis, HoMM III serves as a contrast benchmark. It shows a successful strategy game that uses hexagonal cells for tactical combat, but not for its main adventure map. This reinforces the idea that the proposed framework should be evaluated in scenarios where hexagonal cells provide clear design value, such as movement, positioning, neighborhood relationships, and tactical or strategic map logic.

Figure 10: Heroes of Might and Magic III combat gameplay screenshot from MobyGames [17].

Example	Relevant observation	Value for this thesis
Civilization V	High-level map settings influence terrain, resources, climate, sea level, world shape, and starting conditions.	Shows how player-facing generation settings can create varied strategic scenarios on a hexagonal map.
Age of Wonders 4	Realm traits expose high-level control over geography, climate, inhabitants, world conditions, and major powers.	Shows how layered generation settings can create thematic and mechanical variety without exposing low-level algorithms.
Heroes of Might and Magic III	The game uses a square adventure map but switches to a hexagonal grid for tactical combat.	Shows that hexagonal cells are especially valuable when movement, adjacency, positioning, and attack distance are central to gameplay.

Table 6: Value of selected game examples for the thesis

3 Design

3.1 Functional and non-functional requirements

The research section identified a gap for a focused Unity framework for designer-configurable hexagonal map generation. This section translates that gap into functional and non-functional requirements for the proposed proof-of-concept.

Functional requirements describe the behavior the framework must provide. They define the core capabilities needed for hexagonal map generation, Graph rewriting (Graph rewriting), designer configuration and deterministic output.

Non-functional requirements describe quality constraints that influence the design of the system, such as usability, maintainability, performance, extensibility and scope control.

Because the project is a proof of concept, the requirements prioritize configurability and demonstrability over production-ready completeness. The goal is not to build a universal procedural generation package, but to design a focused framework that proves the feasibility of designer-configurable hexagonal map generation in Unity.

ID	Requirement
1	The framework shall produce a game map as a Graph of Hexagonal cells.
2	The framework shall allow designers to define map generation rules based on global or local cell patterns.
3	The framework shall allow designers to define which pattern matches are considered valid candidates for rule transformations.
4	The framework shall allow rules with valid matches to transform the grid during map generation.
5	The framework shall expose generation configuration inside the Unity editor Inspector window.
6	The framework shall allow map generation without requiring designers to write code.
7	The framework shall support repeatable generation by producing the same result from the same configuration and seed.
8	The framework shall provide visual feedback for generated hexagonal maps inside the Unity editor scene view.

Table 7: Functional requirements for the proposed framework

ID	Requirement
1	The framework should be understandable for users with limited programming knowledge.
2	The framework should keep generation configuration visible and editable through Unity editor interfaces where possible.
3	The framework should be modular enough to separate grid representation, rule definition, rule matching, rule application, and visualization.
4	The framework should be maintainable by keeping generation logic independent from presentation logic.
5	The framework should be extensible enough to support additional rule types, cell properties, or map features in future work.
6	The framework should generate small and medium-sized proof-of-concept maps within an acceptable time for interactive editor use.
7	The framework should make generated results reproducible for testing, debugging, and comparison.
8	The framework should remain focused on hexagonal grid-based map generation and avoid expanding into a general purpose procedural generation system.
9	The framework should rely on Unity native features where possible to reduce external dependencies and improve accessibility.

Table 8: Non-functional requirements for the proposed framework

3.2 Architecture

The functional and non-functional requirements led to an architecture centered on a modular Unity framework rather than a standalone application or external generation service. Since the goal of the project is to demonstrate designer-configurable hexagonal map generation inside Unity, the system was designed as a framework integrated into the editor, composed of:

- Unity Unity MonoBehaviour (MonoBehaviour) orchestration scripts.
- ScriptableObject configuration assets.
- Graph-based generation core.
- Rule rewriting subsystem.
- Materialization (Materialization) layer that converts abstract generated data into visible scene objects.

The requirements influenced the architecture in four main ways. First, the need to represent maps as hexagonal cell graphs required a dedicated graph core that stores grid structure, cell identity, adjacency and properties separately from visual objects. Second, designer-configurable generation required rule structure, definitions for how rules find valid matches for execution, prerequisite conditions for rule and rewrite actions that mutate the grid have a need to be represented as Unity assets editable in the Inspector. Third, the need for reproducible generation required a deterministic seed and random

stream subsystem. Lastly, the need for visual feedback inside Unity required a separate materialization layer that translates the generated graph into prefab instances in the scene view.

3.2.1 Framework-specific terminology

Before discussing how the requirements influenced the architecture, it is useful to define several terms used throughout the framework. The system represents a generated map as an abstract grid of hexagonal cells. Each cell has a position in the grid and an associated property, such as a terrain or region type. The generation process modifies these properties over time, and the final grid is later converted into visible Unity objects.

Rulebook:

A Rulebook (Rulebook) is the main designer-facing generation configuration. It contains an ordered list of rule entries that are executed during generation. Each Rule entry (Rule entry) describes one possible transformation of the grid. A rule entry does not directly contain all generation logic itself. Instead it combines smaller configurable parts such as a match definition, prerequisite conditions, a match selection policy and a rewrite action.

Match definition:

A Match definition (Match definition) describes the cell pattern that a rule is looking for. Pattern matching is the process of scanning the grid and finding cells or regions that satisfy this definition. For example, a match may require an Anchor cell (Anchor cell) with a specific property, a cell adjacent to another property or a cell inside an active subgraph. The result of this process is a set of candidate matches.

Match selection policy:

A match selection policy determines which valid candidate matches are used by a rule entry. The framework supports policies such as selecting the first valid anchor, selecting a random valid anchor or applying the rule to all valid anchors. This makes rule behavior configurable without changing code and allows designers to choose between deterministic, random and expansive transformations.

Prerequisite:

A Prerequisite (Prerequisite) is an additional condition that must be satisfied before a rule entry can apply its transformation. While a match definition usually describes a local or structural pattern, prerequisites can express broader conditions, such as counts for properties present on the grid, property grid coverage, adjacent property pairs or subgraph selection constraints. This separation allows the framework to distinguish between finding possible locations and deciding whether the rule should run at all.

Rewrite action:

A Rewrite action (Rewrite action) performs the actual modification once a valid match has been selected. Actions can replace an anchor property, place a shapes at an anchor, draw a path between properties, flood fill a subgraph or modify a subgraph in other ways. Actions do not directly manipulate the grid data structure. They use a mutation gateway, which provides controlled operations for painting cells, shapes, paths and subgraph regions.

Subgraph:

A Subgraph (Subgraph) is a region of the grid used to scope generation. Some rules operate over the entire grid, while others operate only within an active subgraph which represents only a part of the whole grid. Subgraphs allow the framework to express higher-level map structures and then apply local transformations inside those structures.

Materialization:

Materialization is the process of converting the abstract generated grid into visible Unity scene objects. The rewrite system operates on cell properties rather than prefabs. After generation is complete, the materialization layer reads the grid and instantiates the appropriate tile Unity prefabs (Prefabs) based on the generated properties. This keeps procedural logic separate from visual presentation.

Requirement driver	Architectural response	Reason
Maps must be represented as hexagonalcell graphs.	A graph core stores grid dimensions, cells, adjacency and cell properties.	This keeps generated maps independent from Unity scene objects.
Designers must define generation rules based on patterns.	Rulebooks contain rule entries composed of match definitions, prerequisites and rewrite actions.	Generation behavior can be configured as data instead of being hard-coded.
Designers must control which matches are valid.	Matching, prerequisite condition for rule evaluation and match selection are separate stages.	This makes candidate selection explicit and easier to extend.
Rules must transform the grid.	All changes pass through a mutation gateway and specialized grid mutation services.	Centralized mutation prevents uncontrolled grid changes and gives actions a stable API.
Configuration must be available in the Unity Inspector.	Topology, properties, rulebooks, match definitions, actions and materialization settings are Unity assets.	Unity native assets are familiar to designers and avoid custom external tools.
Designers should not need to write code.	Generation behavior is assembled from reusable rule, match, prerequisite and action assets.	Designers can combine existing components through the Inspector.
Output must be repeatable from the same seed.	The framework uses deterministic seed normalization and separate random streams.	This makes generated results reproducible for testing and debugging.
Generated maps need visual feedback.	A materialization layer converts the generated graph into prefab instances.	The same abstract map can be inspected visually without coupling rules to rendering.
The system should be understandable to non-programmers.	The architecture favors visible assets, Inspector fields and named rule components.	Users can reason about configured generation steps instead of source code.
Generation logic should remain separate from presentation logic.	The graph and rewrite layers are separate from the materialization layer.	Rules operate on abstract properties while visualization remains replaceable.
The framework should be extensible.	Core behavior is split into match definitions, prerequisites, rewrite actions, contexts and mutation services.	New rule types or map features can be added without rewriting the engine.
Small and medium maps should generate interactively.	Generation runs in-process using arrays, dictionaries and bounded rewrite counts.	Simple data structures support acceptable proof-of-concept editor performance.

Requirement driver	Architectural response	Reason
The project should remain focused.	The architecture targets hexagonal grid rewriting only.	This avoids unnecessary complexity from a general-purpose procedural generation system.
Unity native features should be used.	The framework uses `C#`, MonoBehaviours, ScriptableObjects, prefabs, the Inspector and Unity logging.	This reduces dependencies and keeps the workflow accessible inside Unity.

3.3 System diagrams

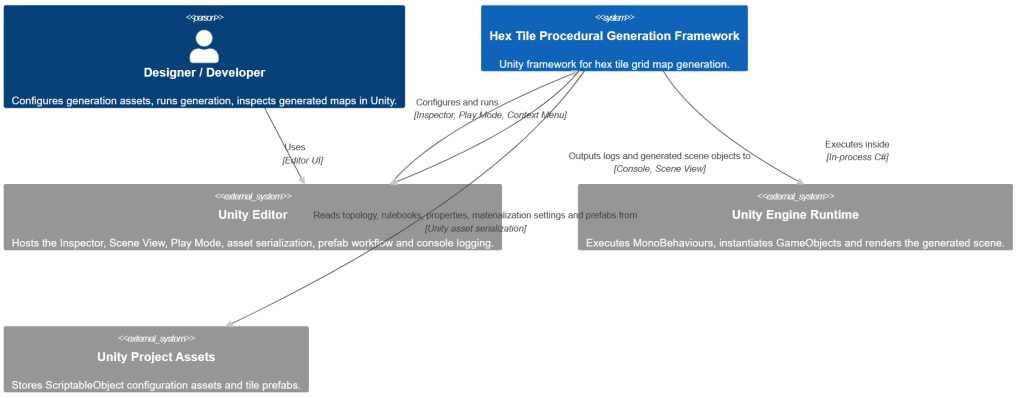

Figure 11: C4 Level 1 system context.

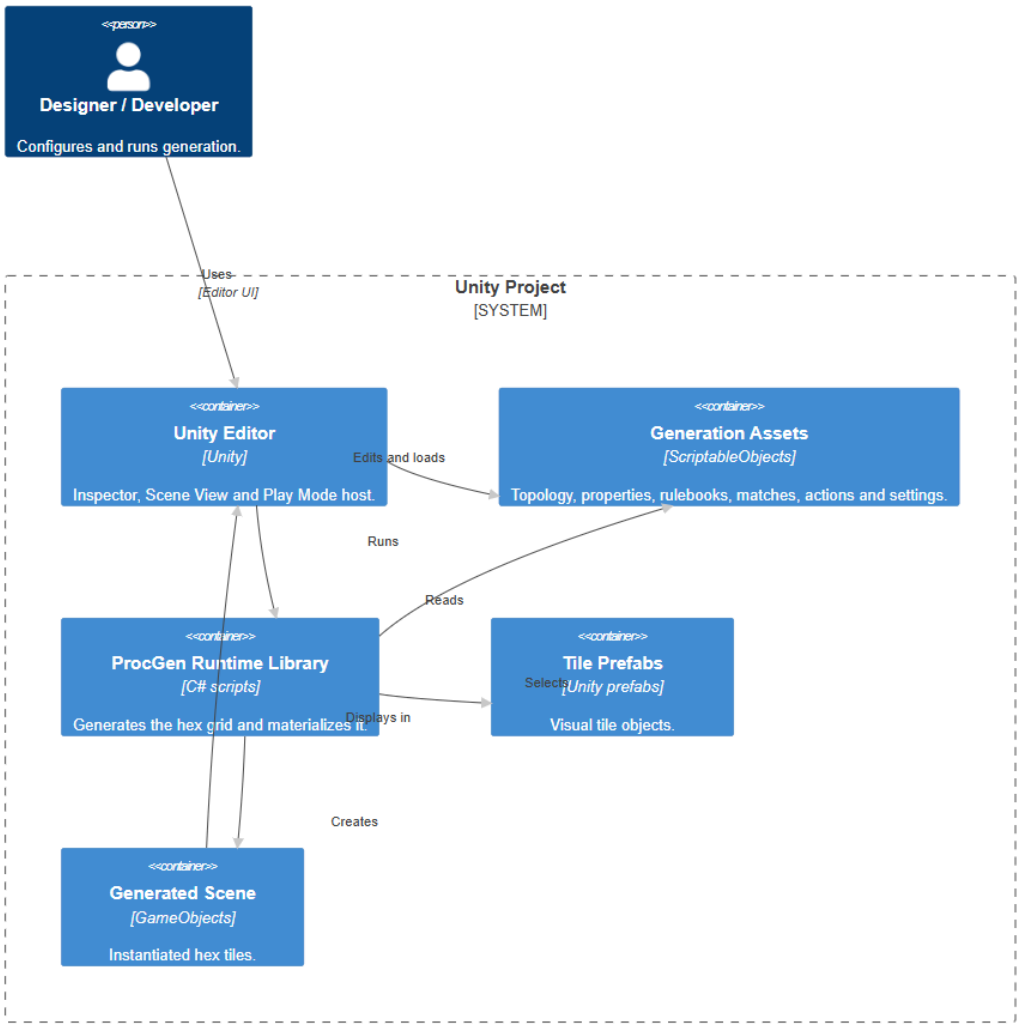

Figure 12: C4 Level 2 container diagram.

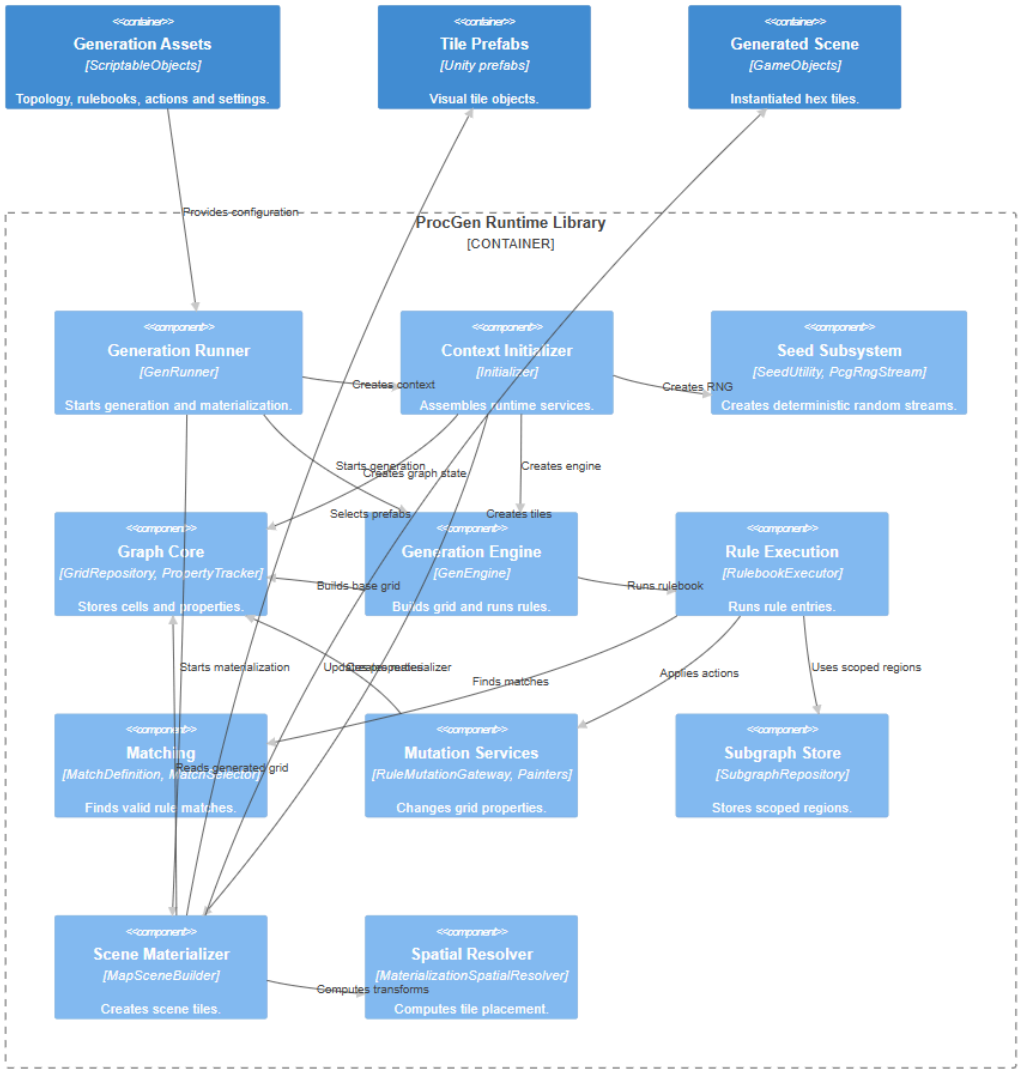

Figure 13: C4 Level 3 component diagram.

3.4 Technology stack

The framework is built as a Unity-native proof of concept. The main programming language is **C#**, and the system runs inside the Unity Editor and Unity Engine runtime. Unity MonoBehaviour scripts are used for scene-level orchestration, Unity ScriptableObject assets are used to store designer-editable configuration such as graph topology, cell properties, rulebooks, match definitions, rewrite actions and materialization settings.

The generated map is represented with custom in-memory data structures, mainly grid arrays, dictionaries and lightweight cell reference objects. The framework does not use an external database, server backend or messaging system. Visual output is produced through Unity prefabs, which are instantiated into the scene by the materialization layer. Deterministic generation is supported by a custom seed utility and PCG-based random number stream implementation.

3.5 Security

Because the framework is a local Unity editor tool and does not expose a network API, store user accounts or process sensitive data, possible security threats are limited. The main relevant concern is supply-chain risk. The project depends on Unity and Unity packages imported into the project. A compromised package, asset, editor extension or dependency could introduce malicious code into the Unity project or affect generated outputs.

This risk can be reduced by keeping dependencies minimal, preferring Unity native functionality where possible, importing packages only from trusted sources, reviewing third-party code before use and keeping Unity/editor dependencies under version control where appropriate. Since the framework currently relies mostly on custom `C#` code and Unity native features, its supply-chain exposure is lower than a system that depends on many external libraries or services.

4 Implementation

4.1 Methodology

The implementation followed an iterative sprint-based development process. The work was divided into four month long sprints.

4.2 Prototyping

Prototyping was used to gradually test the main ideas behind the framework before expanding them into the final implementation. The process began with finding a practical way to represent Hexagonal cells and store them in memory. This led to the use of a two-dimensional grid structure with stable cell identifiers and a separate tracker for cell properties. This was followed by another prototype focused on applying changes to them. Simple transformations were tested first, such as replacing the property of a selected cell. After that, sequential changes to the grid were possible.

The following prototypes tested different kinds of generation rules. The first rules were simple and made trivial local changes, while later rules introduced match definitions, candidate selection and rewrite actions as separate concepts that together compose rules. As the rule system became more stable, additional complexity was added through features that operate on parts of the grid rather than only individual cells.

4.3 Project standards

The implementation follows Unity-oriented **C#** project standards. The codebase is organized around clear component responsibilities, Unity serialization conventions and a modular rule execution architecture. These standards make the framework easier to inspect in the Unity editor, easier to extend with new rule types, and easier to maintain as the proof of concept grows.

4.3.1 Code organization

Area	Namespace	Responsibility
Graph core	`ProcGen.Graph.Core`	Stores the abstract hex grid, cell references, topology data and cell properties.
Generation orchestration	`ProcGen.Graph.Run`	Creates the generation context, starts generation, materializes results and connects the framework to Unity scene execution.
Rule rewriting	`ProcGen.Graph.Rewriting`	Contains Rulebooks, Rule entrys, Match definitions, Prerequisites, Rewrite actions,

Area	Namespace	Responsibility
		rule execution and mutation services.
Materialization	`ProcGen.Materialization`	Converts the generated abstract grid into Unity prefab instances and computes tile placement.
Seed handling	`ProcGen.Seed`	Normalizes user seeds and creates deterministic random number streams.

4.3.2 Naming conventions

Element	Convention	Example
Classes and structs	PascalCase names that describe the component responsibility.	`GridRepository`, `RulebookExecutor`, `MaterializationSpatialResolver`
Interfaces	PascalCase names with the `I` prefix.	`IRngStream`, `IGridQuery`, `IRuleMutationGateway`
Methods and properties	PascalCase names describing the operation or exposed value.	`TryGetCell()`, `Materialize()`, `Generate()`
Private serialized fields	Private fields marked with `SerializeField`, commonly using a leading underscore.	`_rulebook`, `_topology`, `_numOfRewrites`
Boolean values	Names indicate state or intent.	`HasSubgraphs`, `HasActiveSubgraph`
Try-pattern methods	Methods returning success use the `Try` prefix and output resolved values through `out` parameters.	`TryGetProperty`, `TryResolveOutputProperty`, `TryMatch`
Unity asset classes	Unity ScriptableObject (ScriptableObject) classes use names that describe the asset role.	`GraphTopology`, `HexProperty`, `Rulebook`, `RewriteAction`

4.3.3 Architectural patterns

Pattern	Implementation in the framework
Repository pattern	`GridRepository` stores the hex cell grid, while `PropertyTracker` stores generated properties.

Pattern	Implementation in the framework
Query and mutation separation	`GridQuery` exposes read access, while `RuleMutationGateway` controls write operations.
Pipeline pattern	Rule execution proceeds through matching, prerequisite evaluation, match selection and action application.
Factory-style initialization	`Initializer` and `RewriteContextFactory` assemble repositories, contexts, random streams, rule services and mutation gateways.
Strategy-style extension	Different match definitions, prerequisites and rewrite actions are implemented as separate components.
Facade pattern	`GridModifierFacade` and `RuleMutationGateway` expose simplified mutation APIs over lower-level painter services.
Separation of generation and presentation	The rewrite system modifies abstract cell properties, while `MapSceneBuilder` handles prefab instantiation.

4.3.4 Implementation style

The implementation prefers small components with explicit responsibilities over large procedural scripts. Runtime state is passed through context objects such, containing the necessary data for each respective operation. This makes dependencies visible and reduces direct coupling between rule components.

The code also uses checks around missing Unity assets and invalid configuration. Components log errors or warnings when required assets, properties, rule definitions, random streams or materialization settings are missing.

The public extension points of the framework are mainly interfaces and abstract base classes. Examples include:

- match definitions
- prerequisites
- rewrite actions
- grid queries
- mutation gateways and random streams.

This supports future expansion while keeping the current proof-of-concept implementation focused on hexagonal grid rewriting.

4.3.5 Core component implementation

The component design is implemented through small extension points. The most important extension points are match definitions and rewrite actions. Match definitions decide

whether a rule can target a cell or region. Rewrite actions perform the transformation once a match has been selected.

The grid component stores the hexagonal map in a two-dimensional array. Although a Hex grid is not geometrically square, the framework uses offset coordinates so that each cell can still be addressed with an **`(x, y)`** pair and stored in a regular matrix. In this representation, every other row is horizontally shifted when the grid is materialized, producing the visual hex layout while preserving simple array-based storage.

This approach follows the implementation ideas described in Red Blob Games' hexagonal grid articles, especially the use of offset coordinates for map storage and cube-coordinate conversion for algorithms such as range and distance calculations [18]. The framework uses offset coordinates for storage because they map naturally to a rectangular two-dimensional array, while helper methods can convert to cube coordinates when hex-specific calculations are needed.

In the implementation, **`GridRepository`** owns the matrix of cells:

```
private Hex[,] _grid;

public Hex GetNode(int x, int y)
{
    EnsureInBounds(x, y);
    return _grid[x, y];
}

public void SetNode(int x, int y, Hex node)
{
    EnsureInBounds(x, y);
    _grid[x, y] = node;
}
```

Each cell also receives a stable node identifier derived from its matrix position:

```
public int GetNodeId(int x, int y)
{
    EnsureInBounds(x, y);
    return x + y * Width;
}
```

Neighbor lookup accounts for the row offset. Even and odd rows use different neighbor offsets because adjacent cells are shifted differently depending on the row:

```
public Vector2Int[] GetNeighborOffsets(int x, int y, int distance)
{
    bool isEven = (y & 1) == 0;

    if (isEven)
    {
        return new Vector2Int[]
        {
            new Vector2Int(-1, -1) * distance,
```

```
            new Vector2Int(0, -1) * distance,
            new Vector2Int(-1, 0) * distance,
            new Vector2Int(1, 0) * distance,
            new Vector2Int(-1, 1) * distance,
            new Vector2Int(0, 1) * distance
        };
    }

    return new Vector2Int[]
    {
        new Vector2Int(0, -1) * distance,
        new Vector2Int(1, -1) * distance,
        new Vector2Int(-1, 0) * distance,
        new Vector2Int(1, 0) * distance,
        new Vector2Int(0, 1) * distance,
        new Vector2Int(1, 1) * distance
    };
}
```

Component	Implementation role	Extension mechanism
Match definition	Finds valid rule candidates in the grid.	New match types inherit from **`MatchDefinition`** and implement **`TryMatch`**.
Rewrite action	Applies a transformation to a selected match.	New actions inherit from **`RewriteAction`** and implement **`Apply`**.
Mutation gateway	Provides controlled write operations to the grid.	Actions call mutation methods instead of directly editing grid storage.
Rule entry	Combines match, prerequisite, selection and action configuration.	Designers assemble rule behavior through serialized Unity assets.

The base match definition exposes a small contract. Given a query context and an Anchor cell, it either produces a valid **`RuleMatch`** or rejects the anchor. The default **`FindMatches`** implementation scans the provided anchor search space and calls **`TryMatch`** for each possible anchor.

```
public abstract class MatchDefinition : ScriptableObject, IMatchDefinition
{
    public virtual string MatchKind => GetType().Name;

    public abstract bool TryMatch(
        IMatchQueryContext context,
        HexCellRef anchor,
        out RuleMatch match);
```

```
    public virtual List<RuleMatch> FindMatches(
        IMatchQueryContext context,
        AnchorSearchSpace anchorSearchSpace)
    {
        var matches = new List<RuleMatch>();

        foreach (HexCellRef anchor in anchorSearchSpace.EnumerateAnchors())
        {
            if (TryMatch(context, anchor, out RuleMatch match))
            {
                matches.Add(match);
            }
        }

        return matches;
    }
}
```

This structure makes pattern matching extensible. A new local or global pattern can be added by creating another **`ScriptableObject`** match definition without changing the rulebook executor.

Rewrite actions use a similar pattern. Each action receives a rule action context, the rule entry that invoked it and the selected match. The action returns whether it actually changed the grid.

```
public abstract class RewriteAction : ScriptableObject, IRewriteAction
{
    public virtual string ActionKind => GetType().Name;

    public abstract bool Apply(
        RuleActionContext context,
        RuleEntry entry,
        RuleMatch match);
}
```

Actions do not directly modify the grid repository. Instead, they use the mutation gateway exposed by the action context. Because of this approach low-level grid updates are centralized and gives all actions the same controlled API for modifying cells.

The following action shows how an individual rule action is implemented. It resolves the output property from the rule entry, then delegates the actual grid modification to the mutation gateway.

```
[CreateAssetMenu(
    menuName = "ProcGen/Rewriting/Actions/Draw Path Between Properties",
    fileName = "DrawPathBetweenPropertiesAction")]
public sealed class DrawPathBetweenPropertiesAction : RewriteAction
{
    [SerializeField]
    private PathBetweenPropertiesSettings _settings = new();
```

```
    public override bool Apply(
        RuleActionContext context,
        RuleEntry entry,
        RuleMatch match)
    {
        if (!context.Values.TryResolveOutputProperty(
            entry,
            context.ActionRng,
            out HexProperty pathProperty))
        {
            return false;
        }

        return context.Mutation.PaintPathBetweenProperties(
            _settings,
            pathProperty,
            entry.OutputProperties,
            entry.OutputPropertySelectionMode,
            entry.MatchSelectionPolicy);
    }
}
```

This implementation style keeps the rule system open for extension. New actions can be added as new assets, while the executor, rulebook structure and grid storage remain unchanged.

4.4 Testing

4.4.1 Manual testing

Testing for the proof of concept was performed manually inside the Unity editor. This approach was suitable for the project because the framework is primarily editor-facing and many of its requirements involve designer configuration, visual feedback and interaction with Unity assets. The goal of testing was to confirm that the implemented components worked together correctly across the full generation pipeline. Manual testing focused on feature validation rather than automated coverage metrics. Test cases were created by preparing different topology, property, rulebook and materialization configurations, running generation in the editor and checking both the abstract generation result and the materialized scene output.

Visual inspection was an important part of testing because one of the framework requirements is to provide feedback inside the Unity editor scene view. After generation, the materialized map was inspected to confirm that tile placement, prefab selection, paths, shapes and larger generated regions appeared as intended.

Repeatable seeds were also used as a manual quality assurance tool. By keeping the seed and configuration fixed, the same generated output could be reproduced between runs. This made it easier to compare the effect of individual rule or parameter changes and to confirm that unrelated edits did not unexpectedly change the generation result.

4.4.2 Recommended testing strategy

A more complete version of the framework should include automated unit and integration tests. Below is an explanation for the testing strategy for more complete version of the framework.

Test level	Purpose	Examples
Unit tests	Verify individual classes and algorithms in isolation.	Hex neighbor lookup, seed hashing, random stream reproducibility, property tracking, value resolution, prerequisite comparisons.
Integration tests	Verify that multiple components work together correctly.	Run a fixed topology and rulebook with a fixed seed, then compare the generated grid properties against an expected result.
Editor tests	Verify Unity-specific asset and Inspector workflows.	Check that ScriptableObject assets can be created, serialized, assigned and used by the generation runner.
Play mode tests	Verify runtime behavior inside Unity scene execution.	Run generation in a test scene and confirm that the materialized grid contains the expected number of tile objects.
Regression tests	Protect previously validated generation behavior from accidental changes.	Store known seeds and rulebooks as test fixtures and compare later outputs to approved snapshots.

A useful testing technique for this framework is direct grid-state comparison. Since the generated map is stored as a two-dimensional grid with a separate property tracker, tests can compare expected and actual grid states cell by cell. A test fixture can define an expected matrix of property keys, run generation with a fixed topology, rulebook and seed, and then assert that each coordinate contains the expected property. For example, a small expected grid can be represented conceptually as:

```
water water land  land
water land  land  hill
land  land  hill  hill
```

The most important unit tests should target deterministic and algorithmic behavior. Hexagonal grid neighbor lookup should be tested for even rows, odd rows, corners, edges and interior cells. Seed handling should be tested to confirm that the same normalized seed produces the same random sequence and that different salt values produce independent

streams. Prerequisites should be tested with controlled grid states to verify property counts, coverage thresholds, distinct property counts and adjacency checks.

Rule matching should also be covered by unit tests. Each match definition should be tested with small hand-built grids where the expected anchors are known. This would confirm that local pattern matching, neighbor-based matching, subgraph matching and global matching return the correct candidates.

Integration tests should cover the full rewriting pipeline. A test should create a topology, initialize a grid, run a small rulebook with a fixed seed and then compare the final property layout against an expected grid. This would verify that the rulebook executor, match resolver, prerequisite evaluator, match selector, rewrite action and mutation gateway interact correctly with one another.

Coverage targets should prioritize the framework core rather than Unity boilerplate. The graph model, seed subsystem, rule execution pipeline, match definitions, prerequisites and mutation services should receive the highest coverage. MonoBehaviour orchestration and purely editor-facing code can be covered with lighter editor or play mode tests.

4.5 Performance bottlenecks and optimizations

The framework is intended for small and medium-sized maps, so its implementation favors clarity, configurability and editor usability over heavy optimization. For this scope, direct array access, dictionary-based property tracking and bounded rewrite counts are sufficient. However, several parts of the system would become bottlenecks if the framework were used for larger maps, more complex rulebooks or frequent runtime regeneration.

small maps – approximately 30 by 30 cells **medium maps** – approximately 50 by 50 cells

Bottleneck	**Cause**	**Possible optimization**
Repeated full-grid scans	Many match definitions, prerequisite checks and property counts iterate over all valid cells.	Maintain indexes of cells by property, cache frequently used counts, or restrict search spaces to affected regions.
Rulebook complexity	Each rule entry may perform matching, prerequisite evaluation and action application. Large rulebooks multiply the number of grid scans.	Group related rules, short-circuit earlier, cache match results where valid.
Apply all valid anchor rules	Rules that apply to every valid match can produce many mutations in one pass.	Limit the number of selected matches, process matches in batches, or add stronger candidate filters.
Path generation	Path drawing may inspect neighbors repeatedly and use fallback breadth-first search	Use reusable pathfinding buffers, cap path attempts, or

Bottleneck	Cause	Possible optimization
	when direct path construction fails.	precompute distance fields for repeated path queries.
Subgraph operations	Subgraph growth, shrinkage and boundary calculations use sets, neighbor checks and bounds recalculation.	Batch subgraph changes, reuse collections, and update boundary data incrementally.
Materialization	The materializer destroys and recreates the generated scene hierarchy when rebuilding the map.	Use object pooling, update changed tiles only, or separate generation preview from final scene construction.
Prefab instantiation	Instantiating many Unity GameObjects is expensive compared with updating data structures.	Use pooled prefabs, Unity Tilemap-style rendering, GPU instancing, or mesh batching for larger maps.

The most important bottleneck is repeated grid traversal. The current implementation often favors simple iteration over the full grid because this is easy to understand and reliable for small maps. For example, property counts, coverage checks, match finding and candidate searches can all scan the grid. This is acceptable for proof-of-concept usage.

4.6 Deployment

4.6.1 CI and configuration management

4.6.2 Deployment and CI

The project includes a GitHub Actions release pipeline for packaging the framework as a Unity Package Manager (UPM) package. The workflow runs when a version tag is pushed, using either the `*.*.*` or `*.*.*` format. It can also be started manually through GitHub Actions.

Stage	Purpose
Trigger	Runs on pushed version tags such as **`v0.1.0`** or **`0.1.0`**, and through manual workflow dispatch.
Checkout	Downloads the repository contents into the CI runner.
Package validation	Runs the PowerShell packaging script and validates the UPM package manifest.
Package assembly	Copies runtime scripts, demo assets, sample scene and package metadata into a clean UPM folder structure.
Assembly definition	Generates a **`ProcGen.asmdef`** file for the runtime package.
Artifact upload	Creates and uploads a zipped package artifact.

Stage	Purpose
UPM branch publication	Force-pushes the generated package contents to the **`upm-release`** branch.
GitHub Release	Attaches the generated zip file to the GitHub Release for the pushed tag.

The package can be added through the Unity Asset Store. Alternatively, users can add the package inside the Unity editor with the project github url:

```
Window -> Package Management -> Package Manager -> Install package from git
url
```

4.7 Documentation

The documentation for this framework explains how to set up:

- hex grid generator
- configure topology
- properties, rulebooks
- rules
- matches
- actions
- prerequisites
- subgraphs
- path drawing
- repeat modes
- materialization

It also includes common troubleshooting cases.

The documentation also covers extension points for developers. In particular, it explains how custom behavior can be added by implementing new match definitions, prerequisites or rewrite actions in code.

5 Validation

5.1 Validation test

5.1.1 Validation scenario

To validate the framework, an example for a game using this framework was created. The example game emphasises the infuence generation has on gameplay decisions made by players.

The validation scenario is a strategy game played on a procedurally generated hex grid map. The player must defend ancient towers while an expanding desert gradually spreads from existing sand regions. The quality of the generated map directly affects the quality of the game, because terrain properties determine where sand can spread, where the player can move, where resources can be collected, and how defensible each tower is.

This example game for validation will be reffered to as “It consumes.”

Game loop description:

1. At the start of each turn, the player may place a limited number of defensive cells on eligible map cells. While a defensive cell is active, sand cannot spread onto that cell. Defensive cells are temporary and expire after a small number of turns. The player may place new defensive cells each turn, but the total number of active defensive cells cannot exceed a predefined limit.
2. After the player has placed all available defensive cells or chooses to advance the turn early, the sand spread simulation progresses. Sand expands according to Cellular automata (Cellular automata) rules that depend on neighboring sand cells and the type of target cell. Different terrain types therefore affect how quickly the threat spreads, which routes it can take, and which towers are easiest or hardest to defend.
3. The objective is to keep at least one tower uncorrupted for a defined number of turns. The game is lost if all towers are corrupted. The game is also lost if sand covers more than a defined percentage of the map.

Because of the limited availability of defensive cells, the player is forced to consider map features to use defensive cells most effectively.

Different cell types from which the map grid is composed influence gameplay with their individual characteristics.

Below are cell types present in It consumes:

Game cell type	Basic description
Grass	Vulnerable to sand spread. Supports player defensive cell placement.

Game cell type	Basic description
Dark grass	Less vulnerable to sand spread than grass. Supports player defensive cell placement.
Forest	Slows sand spread more than dark grass, can be cleared or protected.
Frozen land	Naturally resistant terrain. Sand spreads into frozen land slowly, making it a defensive buffer. Supports player defensive cell placement.
Mountain barrier	Blocks sand spread. Creates natural chokepoints.
Desert / active threat	Starting source of the spreading threat. Consumes livable terrain if not contained.
Shallow water	Sand spreads slower than on dark grass. Supports player defensive cell placement.
Deep water	Blocks direct sand spread. Defensive cells cannot be placed on it. It creates permanent natural channels that shape sand routes and tower defensibility.
Defense objective / tower	Primary structure the player must defend. Losing all towers represent a loss.

In the following images, cell types that were described in the table above can be seen.

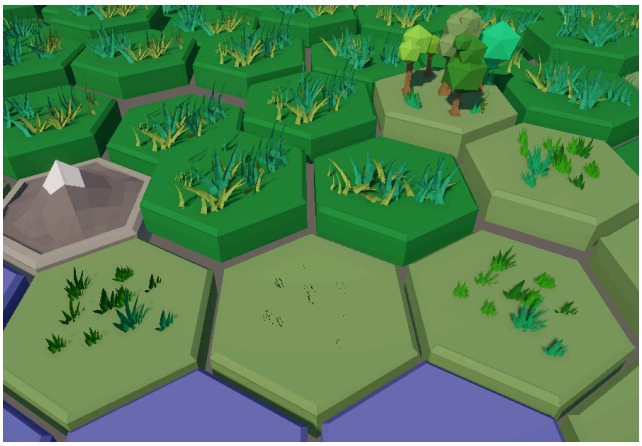

Figure 14: Screenshot of a generated map containing grass, dark grass, a mountain and a forest cell.

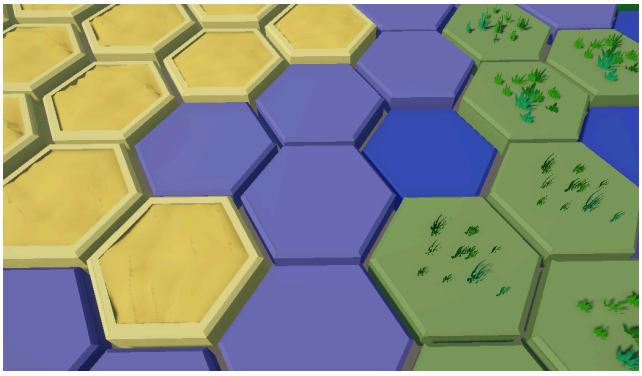

Figure 15: Screenshot of a generated map containing sand cells, shallow and deep water, grass.

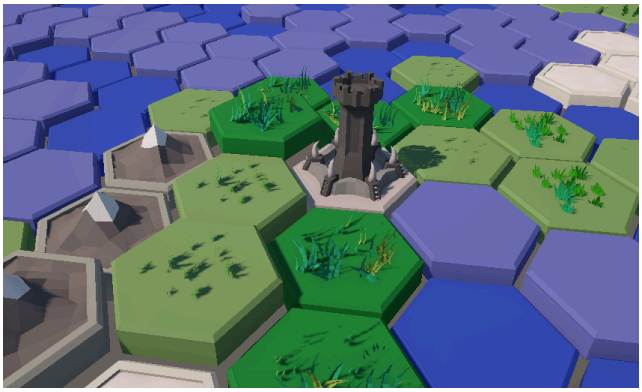

Figure 16: Screenshot of a generated map containing a tower, regular and dark grass, a few mountains and snow.

The towers that need to be defended by players have a distinct visual appearance, being the only man-made structure in the generated map.

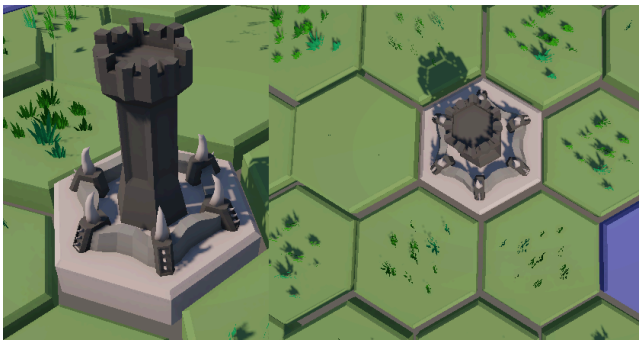

Figure 17: Screenshot of a tower from two angles.

5.1.2 Generated examples for validation

The validation is not based only on whether the generated maps look visually varied. The important question is whether the generated terrain creates meaningful gameplay conditions for the Desert tower defense scenario.

A useful generated map should contain sand sources that threaten the playable land, towers that can be defended through different terrain strategies, resistant regions that slow the spread, and barriers or channels that shape the direction of expansion.

The following generated map examples are evaluated with these criteria in mind. Each example is treated as a potential level for the game. The analysis focuses on how the generated cell properties influence tower defensibility, sand spread routes, use of defensive cells and overall difficulty.

Example game rules:

- Defensive cells last 7 turns
- Player can have 13 defensive cells active at once
- Player loses if 60% of the map is covered in sand
- Player has to keep lose conditions from triggering for 80 turns

Map size – 30 by 30 cells

All provided example maps were generated within 1-2 seconds. Execution timer starts when the generation engine begins processing rule entries and ends when materialization is complete.

All examples were generated with the same generation settings.

Example 1 analysis:

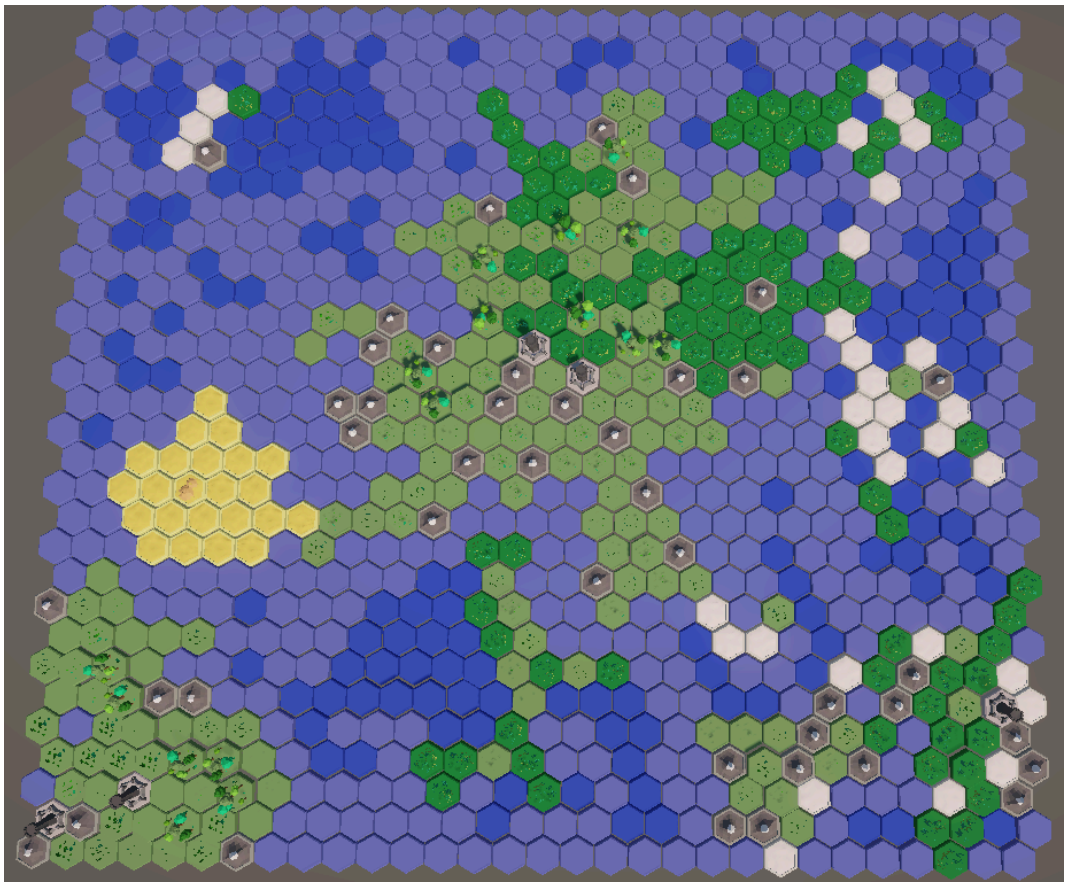

Figure 18: Screenshot of the generated map example 1 produced by the prototype.

The first generated map contains multiple tower objectives distributed across separate land regions. The initial sand region is located in the lower left part of the map, away from most towers but close enough to connected land that it can become a long term threat.

A notable feature of this map is the placement of mountains around several tower areas. These mountain cells create partial barriers that can slow sand expansion and produce natural chokepoints. Because defensive cells are limited and temporary, the player does not need to protect every neighboring cell equally. Instead, the player can identify narrow routes where a small number of defensive cells can delay sand for several turns.

The shallow water and deep water regions also influence the defensive problem. Sand cannot expand through the map uniformly. It must follow available land routes or spread slowly through more resistant and therefore less ideal terrain. This creates a juggling act between closing a chokepoint early, allowing sand to advance temporarily, or saving defensive cells for tower adjacent positions.

This map successfully demonstrates terrain properties can create meaningful defensive decisions. The player is not only reacting to the sand source, but also interpreting the generated terrain structure.

Example 2 analysis:

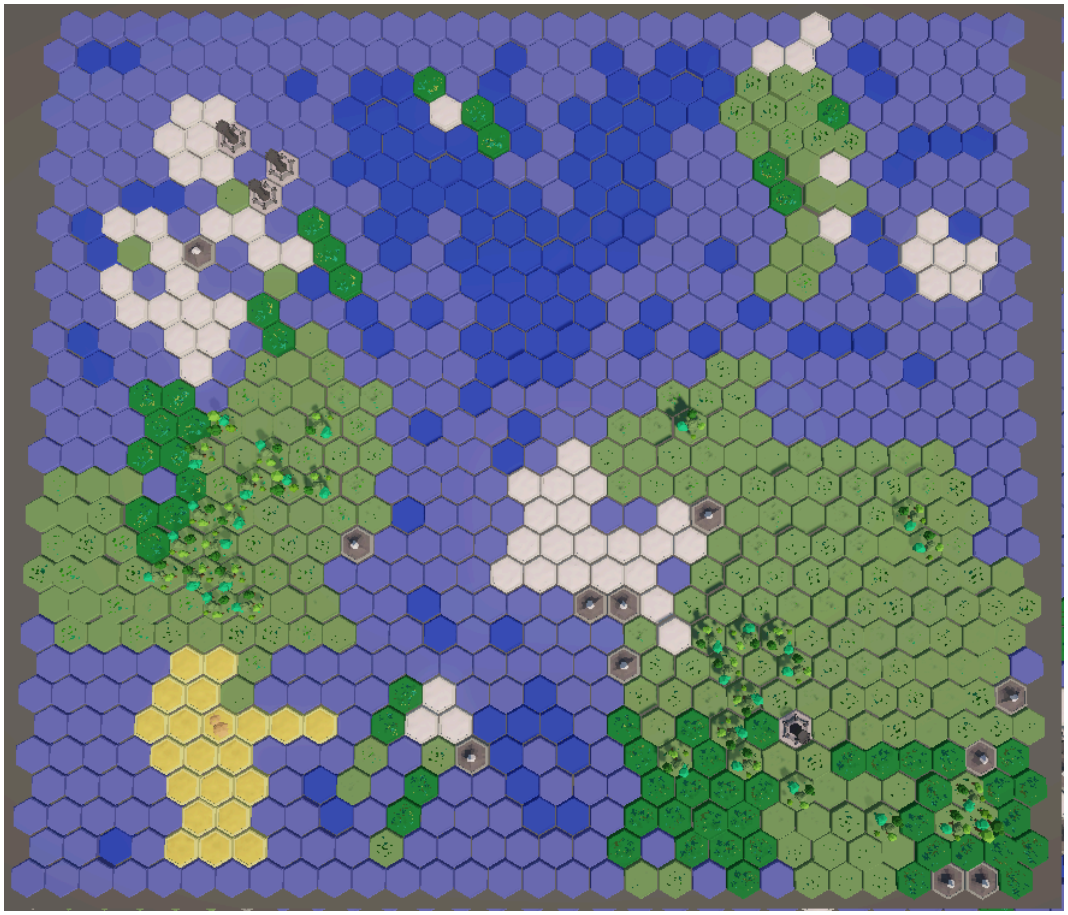

Figure 19: Screenshot of the generated map example 2 produced by the prototype.

The second generated map contains fewer tower objectives than the first example, and the towers are positioned farther apart from one another. Instead of forming one dense defensive region, the towers in separate parts of the map. This changes the defensive problem since the player cannot protect all towers through one shared defensive front.

The map also contains fewer mountain barriers near the main tower areas. As a result there are fewer reliable natural chokepoints that completely block or strongly restrict sand spread. The player must rely more heavily on temporary defensive cells and must choose carefully where to place them each turn.

Snow and forest regions still provide partial defensive value. Snow slows sand spread and can function as a resistant buffer, while forested or dark grass regions can delay expansion more than ordinary grass. These terrain types are still less reliable than mountains because they slow the threat rather than fully blocking it, therefore putting the player in a more precarious position than in example 1. This creates a more dynamic defensive situation where the player can delay sand, but cannot depend on permanent barriers.

The initial sand region begins in the lower left part of the map. From there, it can move toward nearby land routes and eventually threaten separated tower areas if not contained early enough. Because the towers are spread out, the player must decide whether to contain the sand near its source, reinforce the most exposed tower, or use defensive cells to preserve important terrain chokepoints.

Unlike the first map, which offers many natural chokepoints this map creates pressure through openness and distance between objectives. Despite having been generated with the exact same rules as the first example, the current example demonstrates varied topology while still satisfying the same rules.

Example 3 analysis:

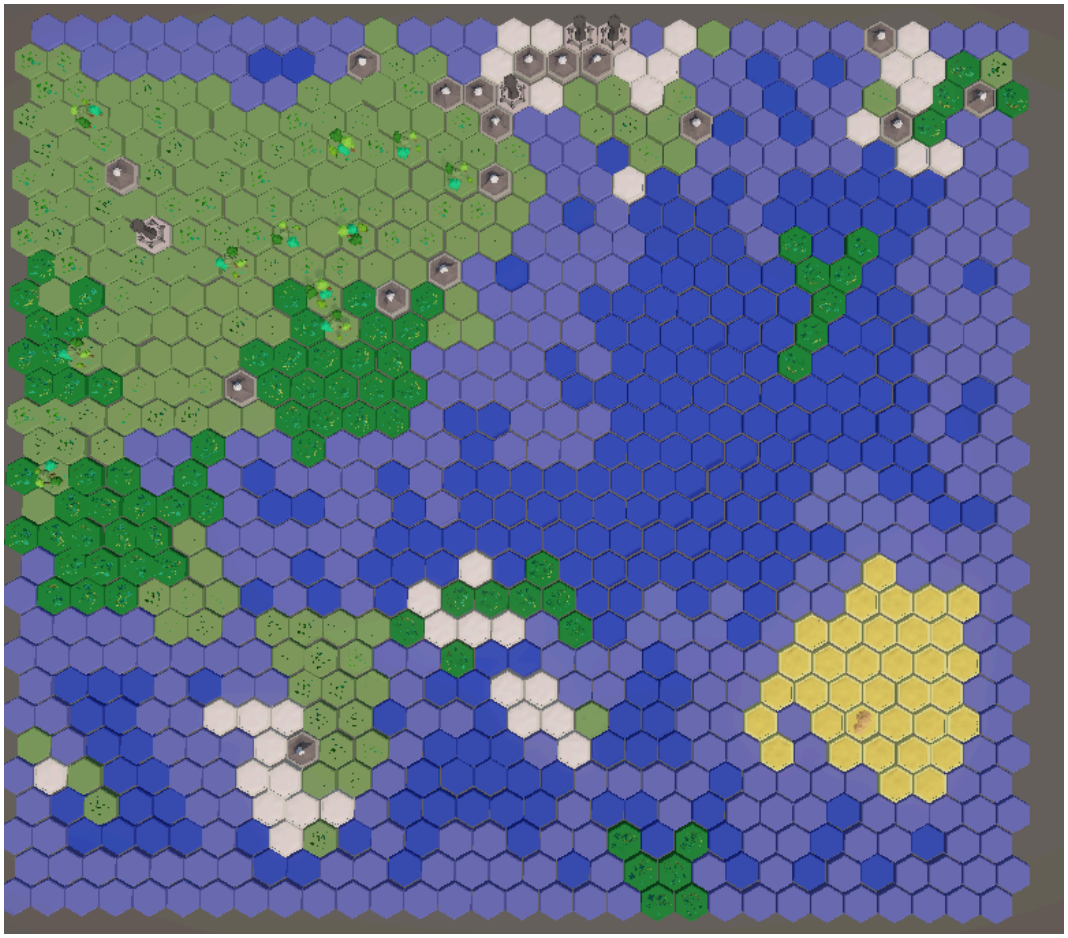

Figure 20: Screenshot of the generated map example 3 produced by the prototype.

The third generated map contains four tower objectives in a closer arrangement than in example 2. Several towers can potentially be protected through overlapping defensive choices instead of being treated as completely separate parts of the map.

The initial sand region is located in the lower right part of the map. It starts far from the main tower cluster, giving the player time to respond. Deep water creates a natural barries for sand progression. However land paths for spand spreading still remain opened.

Compared with example 1, mountains are less consistently useful as defensive barriers. Some mountain cells appear near tower regions, but they do not form reliable chokepoints, thus the player cannot rely solely on mountains. Mountains can be used in conjunction with forests and snow cells to create chokepoints and slow sand spread.

5.1.3 Satisfaction of requirements

The It consumes validation scenario shows how the framework satisfies the functional and non-functional requirements defined in the design section. The validation does not claim that the framework is a complete production-ready game system. Instead, it demonstrates that the generated maps can support a concrete gameplay scenario where generated cell properties affect player decisions.

ID	Requirement	Validation evidence
1	Produce a game map as a graph of hexagonal cells.	The generated 30 by 30 maps are composed of hexagonal cells with distinct terrain and objective properties. These cells form the playable space for It consumes.
2	Allow designers to define generation rules based on global or local cell patterns.	The examples contain structured terrain relationships. For example, mountain barrier spawn rate is directly tied to number of towers successfully added. Forest formations rely on the shape and size of grass land regions.
3	Allow designers to define valid candidates for rule transformations.	The generated maps show controlled placement of features such as towers, sand regions and terrain clusters rather than purely uniform random placement. This indicates that rule matches and candidate selection influence where transformations occur.
4	Allow rules with valid matches to transform the grid.	The resulting maps contain transformed cell properties which depend on one another in various ways. For example, dark grass appears inside of grass regions, given the region has enough grass cells and forest cells.
5	Expose generation configuration inside the Unity editor Inspector window.	The validation maps were produced by configuring generation assets and materialization settings in Unity, then running the generator inside the editor.
6	Allow map generation without requiring designers to write code.	The It consumes examples use existing configured properties, rule assets and materialization assets. The validation scenario relies entirely on generated outputs rather than scripted, static content.
7	Support repeatable generation from the same configuration and seed.	The example maps were generated from fixed generation settings. This allows specific maps to be regenerated and compared when validating the same gameplay scenario.
8	Provide visual feedback inside the Unity editor scene view.	The tile examples and full map screenshots show that generated cell properties are materialized into visible Unity scene objects that can be inspected directly.

Table 9: Functional requirement satisfaction demonstrated through the It consumes validation scenario

5.1.4 Validation summary

The validation was successful as a proof-of-concept demonstration. The generated maps could be interpreted as playable levels for It consumes, and the generated cell properties had clear gameplay meaning. Towers, sand, water, forests, snow and mountains created different defensive situations, showing that the framework can produce maps where generation affects player decisions rather than only visual appearance. Reviewed examples also outline how properties can be dependant on one another and how they create different gameplay interactions.

But this validation methodology is limited. The game scenario was analyzed conceptually through generated examples, not through a completed playable prototype or formal user study. Because of this, the validation can show that the maps appear suitable for the proposed mechanics, but it cannot fully prove that the gameplay would be balanced, fun or understandable to players.

A stronger validation method involves implementing the game rules as a playable prototype and testing generated maps with developers or players. The validation supports the feasibility of the framework, but should be treated as preliminary.

5.2 System limitations

The framework has several limitations that reflect its proof-of-concept scope. It demonstrates designer-configurable hex map generation, but it is not a complete production-ready procedural generation package.

The system is intended for small and medium sized maps. Larger maps or more complex rulebooks may suffer from repeated full grid scans, expensive path and subgraph operations, and costly Unity prefab instantiation during materialization.

Examples reviewed in this section took 1-2 seconds to generate. The time it takes to generate larger maps scales fast. The same examples generated on a larger map size, 80 by 80 for instance, with a larger number of rewrites take around 50 seconds.

Current rule system is flexible but still requires technical work to add new rule behavior. Designers can configure existing match definitions and rewrite actions, but programmers are needed to implement new kinds of matches, prerequisites or actions.

The visual materialization layer serves demonstration and inspection purposes. It is not optimized for a shipped game.

6 Conclusion

6.1 Thesis summary

This thesis explored procedural generation as a way to create varied game maps while reducing reliance on manually created, static content. The work focused on a specific problem: procedural generation can become predictable when players recognize repeated structures or generation patterns. To keep the project concrete, the thesis narrowed the topic to hexagonal game maps generated inside Unity through a graph rewriting approach.

The research phase reviewed existing procedural generation tools, Unity Asset Store packages, Unreal PCG, Houdini, and game examples using procedural or hexagonal maps. This comparison showed that existing solutions provide useful procedural workflows. They are usually focused on prefab based dungeons, broad procedural content pipelines, or tools outside Unity. Based on these findings, functional and non-functional requirements were defined for a proof-of-concept Unity framework.

The final prototype represents a map as a graph of hexagonal cells, exposes generation configuration through Unity assets and Inspector workflows, and uses rulebooks, match definitions, prerequisites, match selection policies, and rewrite actions to transform the grid. The framework was validated through the It consumes scenario, where generated 30 by 30 maps were analyzed as playable levels for a strategy game. The validation showed that generated terrain properties can influence player decisions, tower defensibility, sand spread routes, and overall map difficulty.

6.2 Alignment with objectives

Thesis objectives were met through the research, design, implementation, and validation stages. The research chapter identified that many available tools either operate on prefab rooms, provide broad procedural suites, or require workflows outside Unity. This supported the need for a narrower Unity native framework focused on hexagonal map generation.

The design chapter translated this gap into functional and non-functional requirements. The implementation then addressed these requirements through a modular architecture with a graph core, rule rewriting subsystem, deterministic seed handling, Unity asset based configuration, and a materialization layer for visual feedback.

The validation chapter depicted that the generated maps satisfy the main requirements:

- Maps are composed of hexagonal cells.
- Maps are produced through configurable rules.
- Maps can be generated from repeatable settings and seeds.
- Maps are visible inside the Unity editor.

6.3 Challenges

One major challenge was controlling the scope of procedural generation. Procedural generation can include terrain, dungeons, object scattering, simulations, large worlds, or full content pipelines. Limiting the thesis to hexagonal map generation made the project more focused and allowed the framework to be evaluated through concrete examples.

Another challenge was balancing designer-friendliness with generation flexibility. The framework needed to expose rule configuration through Unity assets and Inspector fields, while still supporting meaningful map transformations. This led to a modular rule structure where rulebooks combine match definitions, prerequisites, match selection policies, and rewrite actions. The implementation also introduced technical challenges that become a problem when using larger maps or a lot of generation rules.

6.4 Limitations

The framework remains a proof of concept rather than a production ready procedural generation package. It focuses on small and medium sized hexagonal maps and does not attempt to replace broad tools such as Dungeon Architect, Unreal PCG, or Houdini.

The current implementation favors configurability and editor usability over heavy optimization. Larger maps or complex rulebooks can suffer from repeated full grid scans, expensive path and subgraph operations, and costly Unity prefab instantiation during materialization. The validation examples used 30 by 30 maps that generated quickly, while larger maps such as 80 by 80 can take significantly longer.

The rule system is configurable, but not fully designer-extensible. Designers can configure existing match definitions, prerequisites, rewrite actions, rulebooks, seeds, and materialization settings, but programmers are still needed to implement new kinds of rule behavior. The validation is also limited in some ways. The It consumes game scenario was analyzed through generated examples, not through a completed playable prototype or formal user study.

6.5 Future work

Future work should improve usability, tesing and validation. The editor workflow could be extended with richer rule editing tools, clearer visualization of pattern matches, step by step generation debugging, and better feedback when configurations are invalid. These additions would make the framework easier for designers to understand and to adjust generation according to their needs.

The framework could also be optimized for larger maps and more complex rulebooks. Possible improvements include caching property counts, limiting search spaces to affected regions, batching rule applications.

Further development could expand the rule system with additional match definitions, prerequisites, rewrite actions, and support for other map structures such as square grids or irregular graphs. Validation could also be strengthened by implementing It consumes as a playable prototype and testing generated maps with developers or players. This would

make it possible to evaluate not only whether the maps appear structurally suitable, but also whether they are balanced, understandable, and engaging in actual play.

Glossary

External tools

DCC – **Digital Content Creation**: Professional software used to create digital assets for games, animation, film, or visualization. 16

HDA – **Houdini Digital Asset**: A packaged Houdini procedural asset that exposes selected parameters for reuse in other workflows or tools. 16

General

PCG – **Procedural Content Generation**: Algorithmic creation or placement of game content, such as maps, terrain, objects, rooms, or levels. 3, 15

Proof of concept – **Proof of concept**: An implementation built to demonstrate feasibility rather than production readiness. 4, 8

Generation Mmdel

Rewrite action – **Rewrite action**: The component that performs the actual grid transformation after a valid match has been selected. 28, 34

Generation model

Anchor cell – **Anchor cell**: The cell currently being considered as the starting point or reference point for a rule match. 28, 38

Graph rewriting – **Graph rewriting**: A rule-based process where parts of a graph are matched and transformed. In this thesis, graph rewriting changes properties of hexagonal map cells. 26

Match definition – **Match definition**: A configurable rule component that describes what cell pattern or region a rule searches for. 28, 34

Prerequisite – **Prerequisite**: A condition that must be satisfied before a rule entry can apply its transformation. 28, 34

Rule entry – **Rule entry**: One generation step inside a rulebook. It combines a match definition, prerequisites, match selection, and a rewrite action. 28, 34

Rulebook – **Rulebook**: The main designer-facing generation configuration. It contains ordered rule entries used during map generation. 28, 34

Seed – **Seed**: A value used to initialize random generation so that the same configuration can produce repeatable results. 7

Subgraph – **Subgraph**: A selected region of the map graph used to restrict or guide generation. 29

Map representation

Edge **– Edge**: A connection between two vertices. In this thesis, an edge connection means that two hexagonal cells are neighbors. 4

Graph **– Graph**: A structure made of vertices and edges. In this thesis, vertices represent hexagonal cells and edges represent neighbor relationships. 4, 26

Hexagonal cell **– Hexagonal cell**: A single hexagon-shaped tile in the generated map. In this thesis, each cell is represented as a graph vertex. 4, 26, 34

Hex grid **– Hexagonal grid**: A map structure composed of connected hexagonal cells. 4, 37

Vertex **– Vertex**: A node in a graph. In this thesis, a vertex corresponds to one hexagonal map cell. 4

Unity Implementation

MonoBehaviour **– Unity MonoBehaviour**: A Unity base class used for scripts that are attached to GameObjects and participate in Unity scene execution. 27

Unity implementation

Inspector **– Unity editor Inspector**: The Unity editor panel used to view and edit component or asset properties. 7

Materialization **– Materialization**: The process of converting the abstract generated grid into visible Unity scene objects. 27

Prefab **– Unity prefab**: A reusable Unity asset that stores a configured GameObject hierarchy. 29

ScriptableObject **– Unity ScriptableObject**: A Unity asset type used to store reusable data and configuration outside scene objects. 35

UPM **– Unity Package Manager**: Unity's package system for installing and managing reusable Unity packages. 43

Validation

Cellular automata **– Cellular automata**: A rule based simulation model where cell states change over time based on neighboring cell states. 45

Bibliography

[1] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural Content Generation for Games: A Survey," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, pp. 1–22, 2013, doi: 10.1145/2422956.2422957.

[2] Aegon Games Ltd, "DunGen." Accessed: June 04, 2026. [Online]. Available: https://assetstore.unity.com/packages/tools/level-design/dungen-15682

[3] Aegon Games Ltd, "DunGen Documentation." Accessed: June 04, 2026. [Online]. Available: https://dungen-docs.aegongames.com/latest/

[4] Angular Fox Dev, "RoomGen - Procedural Generator." Accessed: June 04, 2026. [Online]. Available: https://assetstore.unity.com/packages/tools/level-design/roomgen-procedural-generator-215804

[5] Ondřej Nepožitek, "Edgar Pro - Procedural Dungeon Generator." Accessed: June 04, 2026. [Online]. Available: https://assetstore.unity.com/packages/tools/utilities/edgar-pro-procedural-dungeon-generator-212735

[6] Ondřej Nepožitek, "Edgar - Unity Documentation." Accessed: June 04, 2026. [Online]. Available: https://ondrejnepozitek.github.io/Edgar-Unity/docs/introduction/

[7] FImpossible Creations, "Procedural Generation Grid (Beta)." Accessed: June 04, 2026. [Online]. Available: https://assetstore.unity.com/packages/tools/utilities/procedural-generation-grid-beta-195535

[8] FImpossible Creations, "Procedural Generation Grid - User Manual." Accessed: June 04, 2026. [Online]. Available: https://filipmoeglich.pl/download/Procedural%20Generation%20Grid%20-%20User%20Manual.pdf

[9] Code Respawn, "Dungeon Architect." Accessed: June 04, 2026. [Online]. Available: https://assetstore.unity.com/packages/tools/utilities/dungeon-architect-53895

[10] Code Respawn, "Dungeon Architect for Unity Documentation." Accessed: June 04, 2026. [Online]. Available: https://docs.dungeonarchitect.dev/unity/unity-overview/

[11] Epic Games, "Procedural Content Generation Framework in Unreal Engine." Accessed: June 04, 2026. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-framework-in-unreal-engine

[12] Epic Games, "Procedural Content Generation Framework Node Reference in Unreal Engine." Accessed: June 04, 2026. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/procedural-content-generation-framework-node-reference-in-unreal-engine

[13] SideFX, "Houdini." Accessed: June 04, 2026. [Online]. Available: https://www.sidefx.com/products/houdini/

[14] SideFX, “Houdini Engine Unity Plug-In.” Accessed: June 04, 2026. [Online]. Available: https://www.sidefx.com/products/houdini-engine/plug-ins/unity-plug-in/

[15] SideFX, “Houdini Store.” Accessed: June 04, 2026. [Online]. Available: https://www.sidefx.com/buy/

[16] Gideon's Gaming, “Age of Wonders 4 Review.” Accessed: June 04, 2026. [Online]. Available: https://gideonsgaming.com/age-of-wonders-4-review/

[17] MobyGames, “Screenshot of Heroes of Might and Magic III: The Restoration of Erathia.” Accessed: June 04, 2026. [Online]. Available: https://www.mobygames.com/game/1494/heroes-of-might-and-magic-iii-the-restoration-of-erathia/screenshots/windows/73550/

[18] A. Patel, “Hexagonal Grids.” Accessed: June 04, 2026. [Online]. Available: https://www.redblobgames.com/grids/hexagons/