Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

Bachelor's thesis

Tile-based procedural generation

Rudolf Líbal

Supervisor: Ing. Tomáš Havlík

26 May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Líbal Rudolf** Personal ID number: **498851**

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

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

Study program: **Open Informatics**

Specialisation: **Computer Games and Graphics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Library for procedural generation of tiles in computer games

Bachelor’s thesis title in Czech:

Knihovna pro procedurální generování dlaždic v počítačových hrách

Guidelines:

1. Analyze existing solutions for procedural content generation (PCG) in tile-based worlds (see literature).
2. Design a generic PCG library with support for multiple terrain layers and the ability to define additional metadata and rules for adjacent tiles.
3. Explore the possibilities of tile randomization using a combination of weighted probability and noise while mitigating the likelihood of incompatible placements in larger areas.
4. Propose an algorithm for finding the shortest possible path through the generated environment, taking into account the constraints imposed by terrain specification. Allow the developer to define arbitrary pathfinding rules in order to facilitate game design decisions.
5. Allow for manual specification and editing of predefined map sections.
6. Design user interface for developers in the form of Unity editor scripts. Describe library's API from the perspective of a game developer.
7. Implement aforementioned functionality in the form of a library in the Unity game engine. The implementation should utilize principles of object pooling, tile state should be stored independently of the scene instance. Allow for serialization of generated structure to a file and subsequent state recovery.
8. Demonstrate library usage by the means of a game prototype.

Bibliography / sources:

DYKEMAN, Isaac. Procedural Worlds from Simple Tiles [online] [visited on 2022-07-10]. Available from: https://ijdykeman.github.io/ ml/2017/10/12/wang-tile-procedural-generation.html
MAUNG, David. Tile-based Method for Procedural Content Generation. Thesis. Graduate School, The Ohio State University. 2016.
SCHOLZ, Dominik. Tile-Based Procedural Terrain Generation. Thesis. Faculty of Informatics, TU Wien. 2019.

Name and workplace of bachelor's thesis supervisor:

Ing. Tomáš Havlík Department of Computer Graphics and Interaction FEE

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

Date of bachelor's thesis assignment: **12.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Tomáš Havlík Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

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

.	
Date of assignment receipt	Student's signature

Acknowledgements

I am deeply grateful to my thesis supervisor, Ing. Tomáš Havlík, for their invaluable guidance and support throughout the research and writing of this thesis. Their encouragement and patience helped me through this challenging journey. Thanks is also due for my family, absolute pillars of support, as well as to my friends, co-sufferers in the endeavor of thesis writing. I would also like to thank Ing. Jakub Jirůtka for permission to use a version of his LaTeX template. Further thanks goes to, in no particular order, everyone not mentioned in this acknowledgement.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on 26 May 2023 .....

Czech Technical University in Prague

Faculty of Electrical Engineering

Citation of this thesis

LÍBAL, Rudolf. *Tile-based procedural generation.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2023.

Abstrakt

Cílem této bakalářské práce je prozkoumat možný vývoj generování tzv. dlaždicových systémů v reálném čase. Implementace generátoru na základě návaznosti hranic kachlíků slouží jako podklad k průzkumu možností generace dlaždicových systémů. Referenční implementace zahrnuje širokou škálu funkcí, včetně poolování objektů, generování terénu, vícevrstvého obsahu, manuálního nastavování stavu a pathfindingu. Analýzou výkonnosti různých aplikací této implementace poskytuje tato práce poznatky o možnostech a omezeních generace takovýchto systémů v reálném čase.

Klíčová slova knihovna, videohra, procedurální generace, terén, dlaždice

Abstract

This bachelor thesis explores the feasibility and capabilities of real-time procedural generation for tileable content. By implementing a generator based on tile border connectivity, this research investigates the possibilities and limits of generating tileable content in real time. The reference implementation encompasses a wide range of features, including object pooling, terrain generation, layered content, initial state editing, and pathfinding. Through an analysis of performance and efficiency in various applications, this research provides insights into the possibilities afforded by real-time, tile connectivity-based procedural generation across various domains.

Keywords toolset, video game, procedural generace, terrain, tile

Contents

Introduction

The goal of this work is to explore the options of creating procedural content generators (PCGs), and subsequently creating a toolset allowing for the automatic generation of game systems. The world generator implemented is inherently the basis of the formal system of any game using it, and therefore a special focus will be on flexibility for usage in a number of different scenarios.

The basis of the explored generating logic is best demonstrated using the board game Carcassonne as an example: the world consists of several same-shaped tiles, ordered in a 2D lattice. The tiles have one main rule concerning their generation, that in order for two tiles to be placed next to each other, their near sides (*borders*), must be of the same type[1].

In the following chapters, the possibilities and limitations of generating such tile-border based systems will be explored, as well as formal definitions for several terms including the tiles themselves. This is followed by a description of the implementation of a tool implementing these concepts. The last part of this work details means of usage and usability in 2D and 3D games and presents a concrete example.

[1] This concept shares the same basis as Wang tiles(8)

Overview

This work consists of multiple parts. Starting with a complete overview of relevant theory and definitions of basic terms in chapter 1, and an analysis of relevant research alongside more advanced concepts (2).

Following up with a methodology statement for the implementation of a PCG tool in chapter 3 (also defining several research questions in section 3.2), and a thorough description of the implemented tool itself (chapters 4 to 7), interlaced with relevant theory.

The third part of this thesis consists of the tool's usage and usability from a game designer's perspective (8) and an evaluation of its strengths and limitations (9).

Chapter 1

Tile-based systems

In this chapter, baseline theory is explained. Terms used in further chapters are defined here.

1.1 Prologue: Of Games And Formal Systems

Games as a medium combine the figurative "best of two worlds", by adding together a story and a *formal system*(20). The story has a virtually unlimited range of complexity; ranging from Pac-Man(3) (the story is "stay away from ghosts"), up to Dungeons & Dragons(1), in which players can build up arbitrarily convoluted stories using just a basic rule set.

The distinctive feature for all games, and videogames in particular, is the underlying formal system, more commonly known under terms like "rules" or *game mechanics*(5) in the case of videogames. For example, Chess(11) is still played the same, no matter the design or name of the pieces.

Like with the story, the underlying formal system can be of any complexity, but the defining feature is its integration with the game's story. Typically game designers try to either hide the mechanics as much as possible[2], or expose as much as possible[3]. For the purpose of this work, (audio)visuals will be considered as story, and technical details as part of the formal system.

1.2 Procedural Content Generation

The blunt definition of procedural content is that it is simply anything, that is not hand-crafted. More specifically, it is content generated by a procedure, whether

[2]Typically games that focus heavily on telling a story, like What Remains of Edith Finch(12)

[3]Perhaps the most exemplary is Kittens Game(4), which initially has only one single line of story, but an entire wiki devoted to precise explanations of game mechanics.

entirely random, or deterministic (39). Conversely, the procedure itself is typically hand-crafted.[4] Examples of procedurally generated content include:

- the randomly selected next piece in Tetris(42)
- players are part of the generating procedure in Carcassonne (51)
- a single, but effectively infinite world in No Man's Sky(15)
- the entire sequence of player tasks in Dwarf Quest(25)
- procedural music generation(7): "[A] composition that evolves in real time according to a specific set of rules or control logics".

Procedural content generation is commonly used to either increase variety (Tetris, Catan(41)) and/or create more content, than would be otherwise possible or practical (No Man's Sky, Minecraft(29)). This often comes at the expense of content broadness in all generated aspects[5].

In this thesis, the focus will be on procedural generation as a means of creating virtual game worlds, e.g. away from the game's story and primarily on procedural generation as part of a formal system.

1.3 Tiles

Many PCGs use a simplification in which generated elements are ordered inside a 2D or 3D lattice. Perhaps the most basic example is Minesweeper(18), where a 2D array contains a mix of empty and mined tiles.

For practical purposes, lattice-based systems are very effective, as they simplify the process of definition down to a discrete space, as compared to most physical systems, which have, in essence, a continuous state space. This is the same reason for why most tabletop games include a discrete tile array as part of their playing field, for example Chess(11), or Catan(41).

From a game design perspective, tile-based outputs, especially orthogonal ones, are generally inferior to lattice-less worlds, as they inherently include some degree of repetitiveness. The subsequent goal for a game developer is thus to either capitalize on the advantages of this approach(10)(34) or mitigate the disadvantages (often by trying to hide the lattice itself - typically for terrain(2)).

[4] Interestingly, the definition allows for some broader and less expected processes to be considered as PCG. For example, growing (*procedure*) a tree (*content*) from a seed (*limited input*).

[5] Discussed in section 2.1

1.4 Tile Ordering

Many tile-based systems utilize a square, oblong[6], or cuboid ordering. The main advantage is in simplicity, which again, limits versatility. Other types of ordering include hexagonal (e.g. Catan) and non-uniform shape (e.g. letters in this paragraph). No ordering at all, that is, a continuous state space, is also possible. This is commonly used for example in foliage generation(19).

No ordering is considered superior from a game design perspective, as lattice-based systems limit possibilities. Short and Adams even advise against using, or at the very least, revealing the presence of any lattice system in their publication.(40) I.e., there is an unavoidable trade-off between versatility and simplicity.

For the purposes of this work, only 2D orderings will be explored. The position of a tile is thus encodable into a 2-dimensional integer vector: *coordinates*. Each unique coordinate corresponds to a different position in the world itself. Generally, this is given by a function $\mathbb{N}^2 \rightarrow \mathbb{R}^n$. For example, $(x, y) \rightarrow (x, 0, y)$ could be used for square tiles ordered in a 2D plane within a 3D world, or $(x, y) \rightarrow (\frac{3}{2} \cdot y, 0, \sqrt{3} \cdot (x + \frac{y}{2}))$ for a hexagonal tile system, by using axial coordinates.(16)

1.5 Connectivity

Generating a new tile requires selecting from a pool of different tile types. The pool can be predetermined or also procedurally generated.

Many methods of selection involve choice based on the output of a mathematical function. Examples include terrain generators based on Perlin noise (e.g. Minecraft(29)), and graphical calculators — tiles represent pixels(43). In other words, the generator navigates the state space(32) of the system and selects such tile states that, ideally, align well with already chosen ones.

The requirement of proper alignment is satisfied by considering the mutual compatibility of two different tiles' edges. This is the basis of a class of formal systems, called Wang tiles(8). In Wang tile systems, each tile is a square with a color assigned to each side. Two tiles can be placed next to each other if, and only if, their near edges are the same color.

A notable example of Wang tile appearances is in the board game Carcassonne(51), where players create a world from square tiles with exactly 3 different edge types.

[6]Square and oblong (i.e. rectangular) systems can be considered equivalent in every aspect bar appearance.

In this work, the term *connectivity* will be used to quantify the compatibility of two tiles by their *borders*. Connectivity can operate on Boolean logic (compatible, not compatible), or on fuzzy logic (i.e. extent of connectivity).

1.6 Conflicts

A *conflict* or a *conflicted state* is when a tile should be placed at some coordinate, but there is no tile type that is compatible with all previously placed neighboring tiles. Tile sets that can and cannot tile the entire plane are called *plane-filling* and *non-plane-filling*, respectively.

Avoiding conflicted states is crucial for the viability of a procedural content generator (PCG). Several methods of reducing and avoiding conflicts will be presented in this and following chapters.

1.7 Run Time Generation

Many PCGs serve only as a baseline for a static world, where the world is either generated entirely beforehand (at "development time"), or where the PCG runs once at run time and is then discarded. In the following chapters, this type of generators will be referred to as *static*.

Contrary to this, *run-time* or *real-time* generators aim to generate content within the limited computational margins of a program, usually simultaneously working on several other calculations, like graphics rendering, or game logic(13).

The work presented in this thesis aims to facilitate both - the generator's control over the game world as well as the world's control over the generator. An emphasis will be put on exploration of the limits of how this goal can be achieved with real-time constraints.

1.8 Determinism

There are a few options related to the way random outputs work. In the context of tile PCGs, generators can either be non-deterministic, where generated tiles are completely random, deterministic for a specific loading order, or completely deterministic, where generating the same world twice *always* results in the same tile layout.

Full determinism is not easy to achieve in connectivity-based systems as each tile placed can change the probabilities for other not yet generated tiles in a non-trivial manner.[7]

The only universal way to enforce a completely deterministic behavior in a connectivity-based system is by introducing a constant tile generation order.

[7]Consider one of the many *aperiodic* square tile sets(21).

Chapter 2

Existing Solutions

There are several relevant works including procedural content generators. In this chapter, key features and options of various PCGs are examined in their advantages and disadvantages and their relevancy to this project. These features are then compared across these existing examples, ranging from generic tools to generators directly embedded into a single game. Additionally, followup concepts will be defined side-by-side with the relevancy descriptions.

The following are exemplary works, which are analysed in depth to provide an insight into how tile-based generators function.

2.1 Linden, Lopes & Bidarra, 2013

Linden, Lopes & Bidarra: Designing procedurally generated levels (25)

2.1.1 Description

The paper describes methodology of how to generate entire games solely by using procedural generation. The PCG uses an action-based and then a tile-based grammar, and then uses that theory to describe the PCG implementation in the game Dwarf Quest(50). The game's playing area consists of several rooms in a square lattice. The paper also discusses the limits of automation using PCGs, in a sense as to minimize human input.

2.1.2 Relevance

While the paper showcases the extent in which procedural generation can be used, it also inherently implies the limitations: the mundanity, and crucially, repetitivity, of random selection; inflexibility at run-time; and tedious implementation of abstraction. Most of the described PCG's complexity is not directly about the implementation of a grammar, but on surpassing specific constraints. This leads

to pipeline approach, with a number of specialized (i.e. the opposite of broadly usable) algorithms being employed in series.

The paper also outlines the primary purpose of procedural generation, which is aiding the *developer*, not necessarily benefiting the end user (player). For example, if a game's content can be created faster, or better, without procedural tools, there is no reason for their usage.[8] The described Dwarf Quest is a perfect example of this; it reduces the task of creating a new, unique, never before seen game level down to setting the difficulty level.

The described PCG does make use of connectivity, and manages to avoid conflict states by using the above mentioned pipeline. This approach is effective in avoiding conflicts entirely, but is highly specialized and is a perfect fit only for the underlying grammar system. In other words, the system has a well-defined *purpose*, which a generic PCG does not.

2.2 Maung, 2016

David Maung: Tile-based Method for Procedural Content Generation (27)

2.2.1 Description

The thesis describes the elementary logic behind a static, connectivity-based square tile generator. Unlike the previous example, this generator attempts to offer the broadest possible usability, while remaining within the realm of static 2D tile generators.

2.2.2 Relevance

There are a number of useful notions described in Maung's thesis or otherwise following up from it.

2.2.2.1 Connectivity as a Relation

Allowing arbitrary definitions of connectivity allows for interesting options; a set of one black and one white tile will yield an infinite chessboard, if the connectivity rule is that neighboring tile border colors must be *different*.

Even more generally, connectivity is a relation and can be represented by a undirected graph, where nodes and edges represent border types and compatibilities, respectively. Typically, each node will only share an edge with itself (i.e. tiles can only be placed if their near borders are of the same type).

[8] Interestingly, Grelsson mentions in his thesis(13) that a second purpose of PCGs can be to inspire designers, by generating combinations that are "not constrained by human imagination".

2.2.2.2 Non-Uniform Distribution

Using Perlin noise(36) to form larger-scale structures. A key takeaway from Maung's work is that some more abstract world features cannot be consistently generated with connectivity alone. Examples include "choke points"[9] and "map borders"[10].

2.2.2.3 Weighted Probability

When a tile is being generated at a coordinate, multiple options of tiles can be available. The chance one of the options will be selected can depend on any number of factors: the degree of connectivity, tile frequency, aforementioned noise distribution, etc.

Weights can also be integrated into the connectivity relation graph, by adding weights to each edge. This allows for partially compatible tiles.

The above two concepts alone can lead to interesting generated outputs. Consider a tile set with mountain tiles, ocean tiles and tiles serving as a border between the two. Decreasing the probability of a border tile being placed to almost zero will lead to large, though awkwardly shaped, formations of ocean and mountains. Changing weights based solely on the tiles' positions leads to less random results.

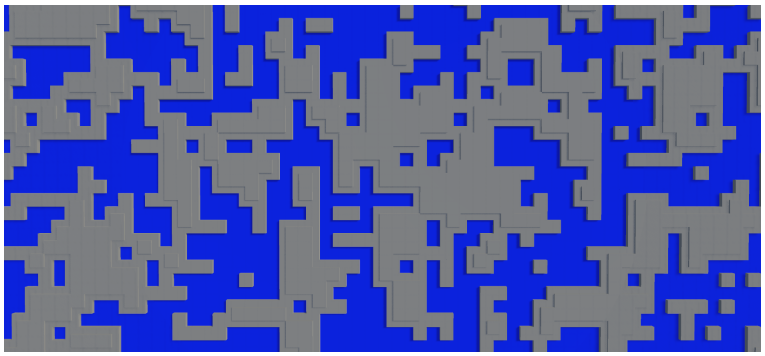

Figure 2.1: A combination of tiles with equal weights.

[9]Most notably in arena-based games, a location of disproportional importance compared to its size.

[10]Edge of a game's playing area, typically obstructing movement past it.

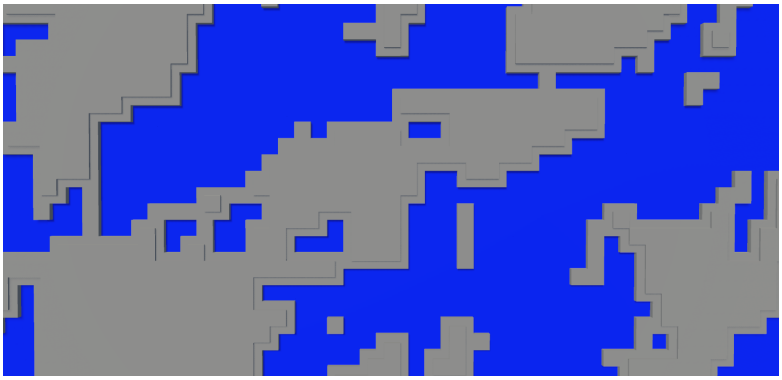

Figure 2.2: A combination of tiles; border ("shore") tiles have a weight of $\frac{1}{20}$.

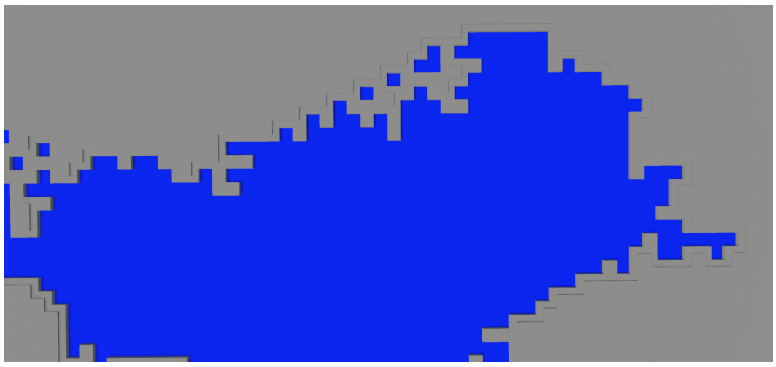

Figure 2.3: A combination of tiles with weights as a function of position.

2.2.2.4 Complete Tile Sets

To completely prevent conflicts, a tile must exist for each combination of edge types. For t types of tile borders with n sides, this amounts to t^n (!) total tiles. This can be reduced to $t^{\frac{n}{2}}$, given optimal generation order.[11] As perhaps the most limiting constraint for a developer using such systems, several methods of reducing this amount will be discussed throughout the rest of this work.

2.2.2.5 Tile Equivalence

Two tiles with identical border types can be considered identical. This is a simplification aimed mainly at implementation. Conversely, rotating a tile effectively constitutes a new tile unless rotational or axial symmetry is present. Defining only one rotation for each tile and automatically filling in the rest allows to reduce the amount of needed input tiles; For square tiles up to 4 times (less than 4 times for tiles with symmetry).

[11] E.g. each tile is generated when at most half the neighboring tiles are known. Load order optimization is further discussed below.

2.2.2.6 Optimising Load Order

Load order optimization is primarily a method of decreasing the amount of needed tiles. It is possible to tile the plane while generating tiles that only have at most half their neighbors already generated by starting at any coordinate and then proceeding in a spiral.[12]

This may seem like a powerful tool, but it also leads to the computational complexity of generating two tiles far apart being proportional to the square of their distance, at minimum. There are "compromise" approaches leading to, for example, (almost) never generating a tile with all surrounding tiles already generated. These will not be further explored.

2.2.2.7 Tile Layers

Splitting tiles into several only partially dependent layers can aid in reducing the amount of different tiles needed. An example could be a tile set generating terrain (3 types of tile borders - "grass", "soil", "rock") and houses on top (again 3 types - "inside house", "wall", "outside house") for a total of $n = 9$ combinations. Separating the tiles into two independent tile sets saves on tiles needed significantly:

layer	n	n^2	n^4
single layer	9	81	6561
"terrain" layer	3	9	81
"building" layer	3	9	81
both layers	$3 + 3 = 6$	$9 + 9 = 18$	$81 + 81 = 162$

Table 2.1: Tile count for a square tile system with two layers.

While this concept for layers could be potentially what "makes or breaks" the system's usability, it is also barely possible in systems of current game engines. Instead, a more realistic approach is to use layers in a less powerful form.

As two tiles with same borders are considered equal by the generator, it is possible to encode many tile variations into layers of a single tile. For example, terrain tiles with multiple different types of foliage and terrain irregularities. While this approach doesn't directly decrease the total amount of tile border combinations, it decreases the amount of border combinations needed to achieve a sufficient variety.

2.3 Dykeman, 2017

Isaac Dykeman: Procedural Worlds from Simple Tiles (9)

[12]For squares, OEIS sequences A174344 and A268038 form the x and y coordinates of the n-th generated tile.(33)

2.3.1 Description

A PCG for tile connectivity-based, single-layer worlds, allowing both static and real-time generation and square or cubical tiles in a 2D or 3D lattice, respectively.

2.3.2 Relevance

A critical concept introduced in this publication is conflict *reduction*. Simply put, conflict situations can be mostly avoided, even when given a tile set, with which random tile generation would generate conflicts, e.g. an incomplete tile set.

Several algorithms allow fixing or preventing conflict states and will be presented in this chapter.[13]

2.3.2.1 Backtracking

The most basic method of conflict reduction is to simply delete one or more generated tiles and try again. Backtracking is often undesirable as a significant portion of already generated content may get removed. Additionally, given a non-plane-filling tile set, backtracking will get "stuck" in a loop.

2.3.2.2 State Space Navigation

Each tile affects which tiles can be placed around it. This can be expressed as an array of potential tile weights (probabilities of placement), assigned to each coordinate. In addition, this narrowing of possible states also affects not-yet-placed tiles neighboring the not-yet-placed tile, and so on.

This effect may continue without bounds[14]. Generally, if a potential tile placement would remove the last option for any not-yet-generated tile anywhere on the lattice, it should not be placed.

2.3.2.3 Two Pass Generation

In Dykeman's toolset, the above concept is realized by first calculating the possible tile states for near empty spaces and eliminating what "would lead to" conflicts, and then choosing which tiles to place in a second iteration. The calculation does not eliminate all conflicts, as it has a limited radius, which the author calls *sphere of influence*.

There are several effectively similar algorithms that can be used for such calculations; the most straightforward approach is to focus on calculating which states are *not* possible. This approach constitutes a constraint satisfaction problem (CSP)(48), solvable by algorithms like arc consistency 3 (AC-3)(22).

[13] A passing mention should be made to tile load order optimization (see 2.2.2.6) as a primitive method of conflict reduction.

[14] Consider a tile set with only two black and white tiles. Placing one tile determines the future state of *all* tiles.

2.3.2.4 Undecidability

There is no algorithm that can, in general, decide if a tile set is plane-filling or not(38), even for many simplifications of this problem(26). This problem is called *undecidablility*.

Even given a relatively small area to tile, it can be hard to determine if the given tile set tiles it as the problem is still NP-complete(14). [15]

2.3.2.5 Incomplete Tile Sets

Possibly the most important notion the work expands upon, is that with sufficient conflict reduction, it is *beneficial* to not have a tile for each permutation of edge types. From a game design perspective, there is little use for homogeneous[16] randomization of tile placements. Collision reduction can prevent unfavorable situations and thus force some degree of heterogenization. An extremely simplified example is as follows: there are two types of borders. The second type represents a contour line passing though the border.

[15] A great example is the Eternity II puzzle from 2007(30), which set *a $2 million bounty* for finding a valid ordering of a mere 256 square tiles in a 16-long square grid. The puzzle is still unsolved.

[16] In this case equivalent to *lacking any larger size structures*

Figure 2.4: A full tile set, with a total of 16 automatically generated rotations

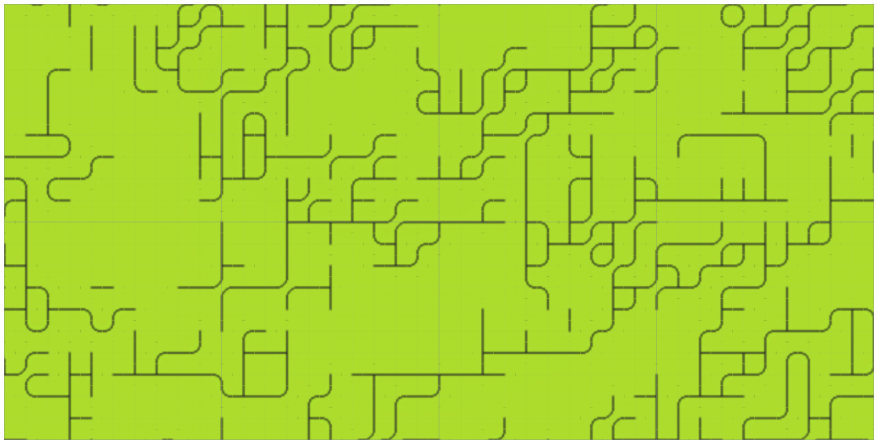

Figure 2.5: The result does not produce the expected contour lines, as contour lines do not end, split, or join up.

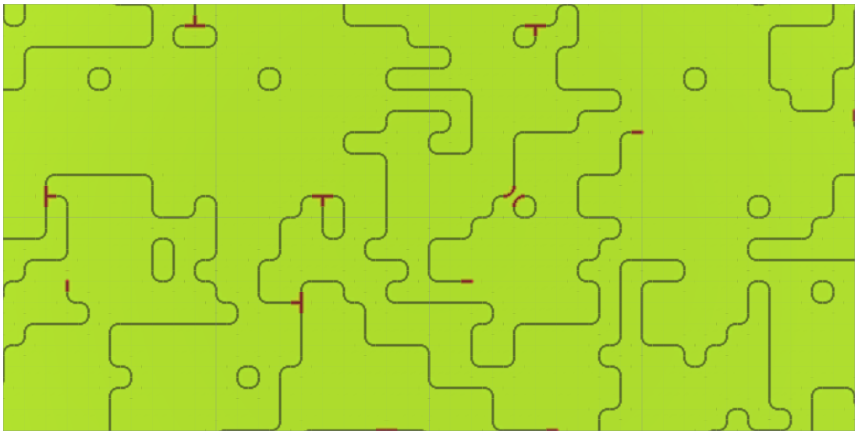

Figure 2.6: Modifying how often the unwanted tiles appear to an absolute minimum will still result in their placement (highlighted in red) being forced where there are no other options. The only solution is thus to remove some tiles from the "palette" entirely.

Figure 2.7: A reduced tile set, with only a subset of all permutations. A total of 7 rotations are automatically generated.

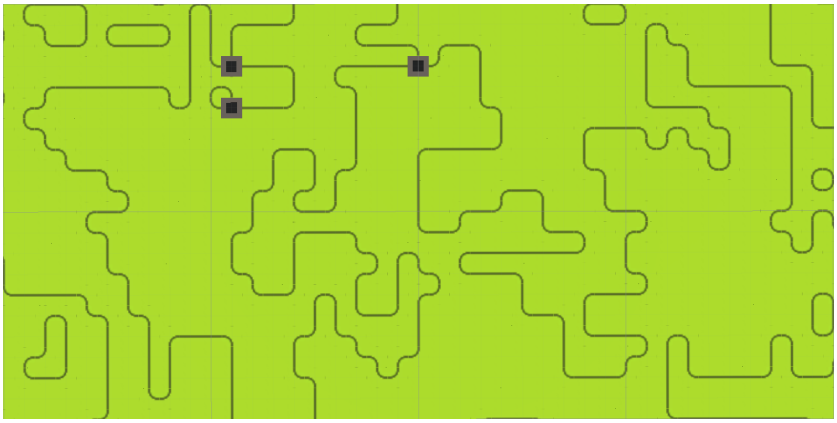

Figure 2.8: Without conflict reduction. Black tiles indicate positions where no tile can be placed.

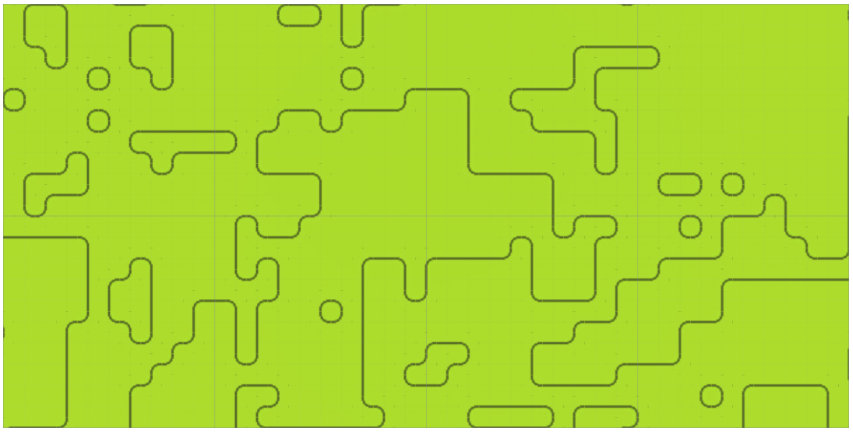

Figure 2.9: With sufficient conflict reduction.

The expected result could only be created with a combination of *incomplete* tile sets and conflict reduction. A system with the given meaning could not be created with a complete tile set. In other words, using incomplete tile sets gives the system a *meaning*.

2.4 Further Concept Definitions

This section details additional concepts and terms that are not directly related to any of the relevant works.

2.4.1 Tile Elevation

In 3D systems, each tile can have not only a position, but also an elevation, a height compared to the baseline plane of the system. This expands the range of

sensible tile sets with features such as terrain, while keeping the generation logic two-dimensional.[17]

To facilitate faster development, the elevation range can be discretized, e.g. integer values only. Leading from this, tiles at different height can thus be considered as different tiles, similarly to tiles with different rotations.

2.4.2 Vertical Borders

With the tile layer definition from 2.2.2.7, a further simplification can be made by declaring tile layers as full-fledged tiles, with their own border connectivities and the such.

A *vertical border* is a parameter of a tile that defines which layers' tiles can be stacked upon it. Essentially, it works in the same way: both the default (main) tile set's tiles and layer tiles define a vertical border; if they are compatible, they can be on the same coordinate together.

This works even for multiple distinct layers, but unfortunately constitutes a 3D state space. A computational simplification can be made in which tile layers do not need to be as strictly generated in terms of connectivity; E.g. if a conflict arises, it is ignored.[18]

2.4.3 Soft Zero

Several concepts related to connectivity-based tile systems can be improved with the concept of a *soft zero*. Soft zero represents a weight that is so low, it will never be selected, unless there is no other option (or the only other options are also soft zeroes). Conversely, a *hard zero* represents a weight that should never be selected, usually due to a conflict arising. This concept can be used in border compatibility weights too, or for example to aid non-uniform distribution (see 2.2.2.2).

2.4.4 Pathfinding

In the scope of this work, pathfinding will refer to the process of finding the shortest path between two tiles in a tile system. The path is typically convoluted by impassable areas. The meaning of *impassable areas* will be explored in the implementation of such a process, as it does vary between applications (e.g. height differences for a mountainous terrain, walls in a maze, etc.).

[17] Of course, this means expanding any given tile set and the relevant borders to accommodate all desired changes in elevation, e.g. various sloped tiles. An example implementation can be found in OpenTTD(34).

[18] Perchance this seems too limited, like in the case of the above mentioned "house on terrain"; but this still allows generating with tile variations efficiently, esp. visual tile variations.

2.5 Summary

This concludes a non-exhaustive list of relevant theoretical concepts. Further concepts related to various implementation details will be explicated in chapters related to the implementation itself.

Chapter 3

Methodology

3.1 Technology used

The following is an overview of the most important technology and tools chosen to implement the PCG.

3.1.1 Unity With C#

No feature described previously is engine-dependent. Unity(45) was chosen as a game engine, because it offers a good combination of usability and flexibility. Unity also allows for simple exporting and importing of toolsets as *Asset Packages*(46).

The resultant toolset should be compatible with both 2D and 3D Unity scenes, although testing will be done primarily with 3D scenes. Unity supports multiple scripting languages, but C# will be used for all written code. The final package will be compatible with any platform Unity is able to build projects for; any Unity version newer than the one used (2021.3.11f1) should be fully compatible.

3.1.2 Git

Git as a versioning system will be used both as a method of backing up the work in progress, as well as a means of distribution of the resultant package though the university Gitlab(24). Unity asset packages are especially appropriate for this as they can be installed with a single click, simply by pasting the git repository URL.

3.1.3 Json.NET

Json.NET(31) is a C# JSON serializer and deserializer. This will be used to save the state of the generated tile system between program sessions. JSON will also be utilized for storing documentation tooltips.

3.1.4 Reference System Specifications

As one of the goals of this work is to optimise for performance, it is necessary to list the most relevant used system specifications. Specifications primarily related to this goal are:

- OS — Windows 11 Version 22H2
- CPU — 12x AMD Ryzen 5 5600H
- memory — 16GB RAM DDR4

3.2 Generator Specifications

As lattice-based PCGs are inherently limited, focus will be on allowing a broad range of usability, at least within the confines of the tile connectivity niche.

The following is a list of basic feature requirements for the generator.

1. Separate data definitions for tiles, tile borders, and tile grouping. The definitions should optimally be usable for any conceivable scenario, and should allow for different definitions of connectivity.
2. Support for multiple terrain layers.
3. Tile selection and elevation randomization using weights and smooth noise.
4. Sufficient conflict reduction.
5. A pathfinding algorithm allowing for any arbitrary definition of traversibility.
6. An editor for manual specification of tiles in the system and setting up data definitions, pathfinding and other PCG settings.
7. Object pooling, e.g. loading/unloading of tiles' in-game objects separately from generation logic.
8. Serialization of the tile system state to a file and subsequent state recovery.

3.3 Research Questions

It is important to question and give focus to both finding and expanding the computational and other limits of PCGs. This section is dedicated to defining several metrics of success for various goals, with the intent of exploring how much is actually possible.

3.3.1 Run Time Performance

What are the limits of real-time generation?

One of the bigger research goals of this thesis is to find out just how much can be done in real time[19]. In games, this refers to updating the tile system on-demand, and fast enough to not fall behind or delay other parts of the game logic. On the other hand, the "overhead" upon startup is not limited. At least one calculation cycle is performed between each graphics update. Anything above cca. 20 milliseconds on average (and 100 at most) between each graphics update will not be considered real time.

This area of PCGs is not well explored for connectivity-based tile systems. Dykeman's work (see 2.3) shows that it is possible to generate connectivity-based worlds in real time, including a good amount of conflict reduction; While Grelsson(13) comes to the conclusion that for any more complex PCG it is very hard to fit all calculations into a 20 ms window.

The PCG implemented will be optimised for a single thread. While this does not utilize the full performance of the game engine, not using parallelization allows for simpler and more straightforward code that is easier to understand, maintain, and expand. Additionally, the Unity API is majorly not thread-safe.

3.3.2 Conflict Reduction

How to effectively reduce conflicts?

From a game design performance, a single conflict can have far-reaching consequences, from breaking game logic, obstructing gameplay, to breaking immersion. Conflicts should be avoided at almost any costs.

There are two main scenarios: reducing conflicts with an entirely optimal load order, and reducing conflicts with an entirely nonoptimal load order.

Although this is not possible to achieve completely, an effort will be made in exploring ways of reducing conflicts, and, as a secondary goal, exploring ways of mitigating the consequences of conflicts.

[19]Although different, the terms *real time* and *run time* will be used interchangeably, as whenever the program calculates *at* run time, it is also required to calculate *in* real time.

3.3.3 Flexibility

How many different problems does this tool help with?

Although not soundly defined, a focus will be on implementing features in such a way as to be usable simultaneously in as many ways as possible.[20] Just allowing usage in different manners is not enough; the primary metric of evaluation is total time saved while developing a specific PCG.

In order to achieve this goal, emphasis is placed not only on the generator itself, but also on UI usability, documentation, default settings, examples, and other accessibility.

[20] Without becoming one of those tools that *requires only minimal configuration and tweaking.*

Chapter 4

Generator Structure

A tile connectivity-based, run-time-based PCG has been developed to accomplish the goals stated. This chapter details the implementation odds and ends, not in their ideal state, but in a practical state. This is influenced by many factors, such as object design, optimization, and Unity asset pack structure.

Several limitations of Unity, C# and other used tools are inevitably reflected in how the resultant tool performs. These will also be detailed in this chapter.

4.1 Architecture

In order to accomplish a high degree of flexibility, the generator structure is decentralised. The central logic is split into 8 base components, each derived from Unity's MonoBehaviour(47). Each also provides a given API-like set of public methods to the other components, as well as overridable methods for simple expansion. For the sake of decentralization and expandability, components commonly pool their own data separately, leading to some redundancy.

Several other scripts are also present, and constitute either Unity Editor scripts (e.g. UI), data definitions for tiles, or other data structures and related logic.

4.1.1 TileSystemBase System

Each of the 8 main components is derived from the abstract TileSystemBase class, which is derived from Unity's MonoBehaviour. This class provides easier access to the other components and helps ensure all components are correctly loaded.

Most MonoBehaviour components, have an abstract base class defining basic (non-overridable) behavior and all necessary abstract methods. TileManager is still overridable but in a limited manner.

4.1.2 Communication Between Components

This section shows more details from the generator structure. Each component will be described in detail in the following sections.

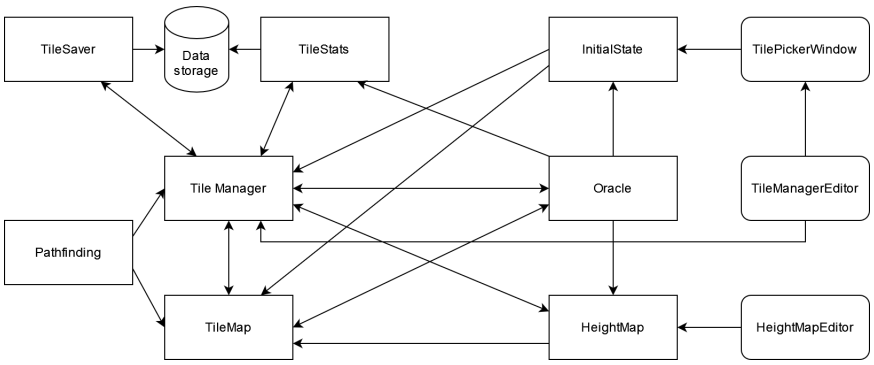

Figure 4.1: Graph of interactions. Modules with sharp and round corners represent classes derived from MonoBehaviour and editor scripts, respectively. Arrows lead from interactors to interactees. Physical storage is displayed as a cylinder.

4.1.3 Unique Identifiers

Each part of the generating logic has to be able to operate with multiple instances of content. This ranges from having multiple tile types to having multiple PCG instances running simultaneously.

Where indexation is needed, a centrally and deterministically distributed 0-indexed integer will be used as an identifier (e.g. tile type ID); Otherwise a user-defined string will be used (e.g. generator name). In all cases, having multiple instances of the same name will result in undefined behavior.

4.2 Interface Design

As a toolset, a user interface (UI) is needed to facilitate the project feature usability. A program is only as good as its UI is. The usual UI design principles apply: maximizing the learning rate, intuitiveness, responsiveness, accuracy, and minimizing error rate.

4.2.1 Main UI Utilities

There are 4 distinct types of UI used in this implementation, each with different design principles and use cases:

1. Unity inspector — Unity's inspector allows for setting persistent default values of object fields for each instance of a MonoBehaviour and ScriptableObject, including setting references to other object instances, list and array elements, etc. This is primarily used to set up initial component settings.

2. Custom UI classes — what isn't possible or isn't effective in the inspector, will be implemented as a custom Editor- or EditorWindow-derived class.

   These use the same GUI framework as the inspector, but are displayed in a separate window or as a separate part of the inspector. They are used mainly to display data unconventionally or to allow for more complex data visualization.

3. In-scene UI — two components, InitialState and TerrainMap, can display their state directly in the scene using Unity's Gizmo system or by instantiating objects in edit time.

4. Context menu items — various context menu (i.e. right-click) lists can be expanded. Creating new data structure instances and adding a PCG variant to a scene is implemented in this way. For example, adding a tile system is done by selecting one of several component combinations from the context menu.

4.2.2 Other UI Utilities

A number of other improvements have been implemented, mainly aimed at overcoming specific limitations in the inspector UI. The following are the most prominent inspector-improving utilities implemented:

- ReadOnlyAttribute — makes a field non-editable.(17)

- SerializableDictionary — displays a Dictionary's contents in the inspector as a ReadOnly list.(6)

Other minor details that could also be considered UI, like error messages, will not be explained further in this work.

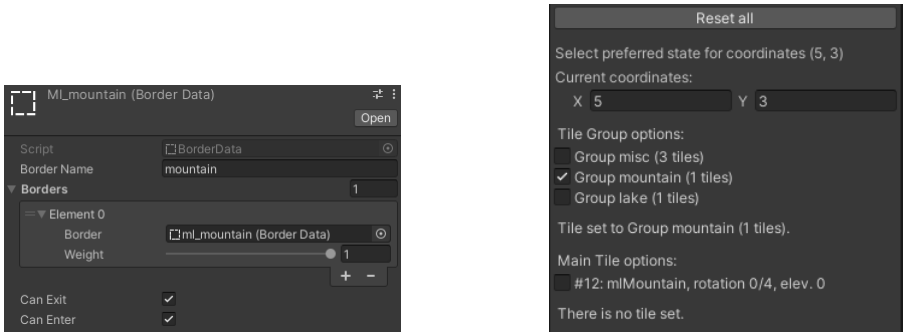

Figure 4.2: Example of standard and custom inspector UI

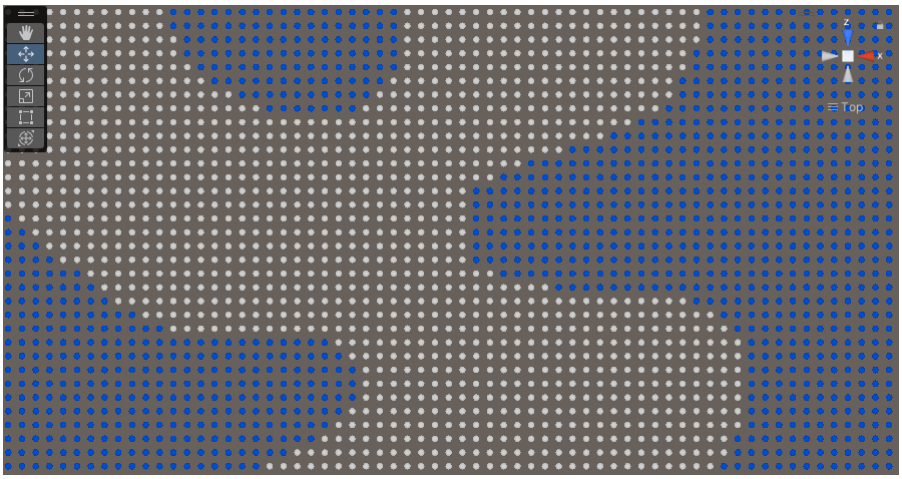

Figure 4.3: Example of Gizmo-based GUI

Chapter 5

Tile Management

This chapter details the components and data structures related to tile management.

5.1 Tile Data Structures

A number of classes facilitating various data structures have been implemented. The core of tile structure consists of TileData and BorderData.

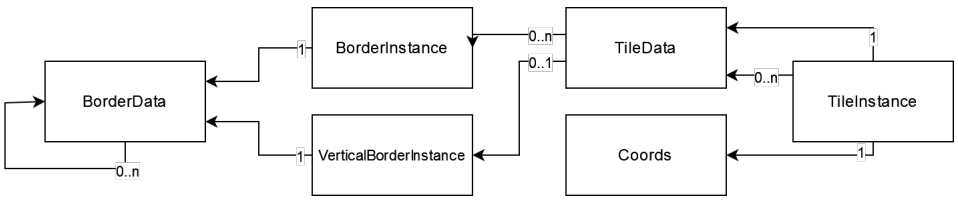

Figure 5.1: Tile data structures. Arrows represent has-a relations.

5.1.1 Coords

To allow for serialization of coordinates, a new Coords class is used instead of Unity's Vector2Int. This also allows for more options with operator and constructor overloading, and custom JSON operations. For convenience, the two fields of this class are called x, z instead of Unity's x, y.

5.1.2 BorderData

BorderData is a ScriptableObject(47) with definitions of reusable tile border information, e.g. every piece of information, that is common to each border of the same type. This means that each border type should correspond to one instance of this class. Instances can be created and configured in the editor.

The data contained consists of a unique string identifier, pathfinding data, and a connectivity definition.

Connectivity is defined as a list of compatible BorderData entries, with weights from 0 to 1. The border is compatible with all listed borders. A weight of zero represents a *soft zero*, while a non-existent value represents a *hard zero* (as described in 2.4.3). A serialization override can modify these lists to keep the connectivity graph they represent symmetric.

While the list is initialized with a single reference to the object it is in (e.g. standard Wang-like connectivity), it can be changed to anything. A notable use case is having a pair of borders that are mutually connective, but not self-connective.

Figure 5.2: A tile set with 5 tiles and 3 apparent border types: blue, brown, and transitions. This is a simplification of a lake-ground terrain system.

Figure 5.3: An example of a valid configuration of tiles given 3 border types; all borders shown are compatible.

To fix the obvious error, a separate type of border can be defined for (in clockwise order) blue-to-brown and brown-to-blue transitions. These should be compatible with one another, but shouldn't allow for compatibility with a border of the same type.

5.1.3 BorderInstance

A BorderInstance constitutes of a BorderData reference, as well as data, that is not reusable between all borders of the same type. Notably relativeElevation, which is reserved for terrain generation. Elevation denotes how high the edge of a tile is

above its center. This adds a layer of connectivity restrictions, as a tile must fit with both border types and border elevations.

5.1.4 VerticalBorderInstance

Essentially equivalent to BorderInstance, but denotes a Vector3 offset to the main tile instead of an elevation.

5.1.5 TileData

TileData is a ScriptableObject and can represent either a standalone tile type, or a tile layer type. It defines all data reusable between all tiles of the same type, notably:

- id — a unique[21], deterministic identifying value assigned at start time. This is crucial for precalculation in the Oracle and tile serialization.
- gameObject — a prefab to be placed in the game world upon generating the given tile.
- weight – the tile weight expressed as a value between 0 and 1.
- borders, verticalBorder — a list of BorderDataInstance, with one border for each side of the tile, and one VerticalBorderInstance for defining compatibility with layer tiles. All tiles should have the same amount of borders.[22] Borders should not be null, instead a border compatible with nothing should be defined.
- currentElevation — the elevation the tile should be placed at. One tile variation is generated for each value between minElevation and maxElevation. For example, a tile representing the sea should only have one elevation.
- currentRotation — a rotation denominating how many times the tile has been rotated by one unit clockwise. This is proportional to the final rotation of the gameObject when instantiated. Ranges from 0 to uniqueRotations. One TileData variant is generated for each of these rotations.
- uniqueRotations — the number of unique rotations for a given tile. This is assigned manually to allow specification of single-rotation tiles with any borders. If a tile has k unique rotations, then rotations 0 to $k-1$ are unique (see proof in section 5.1.5.1).

[21] Unique within each layer only.

[22] Although this is a not-always-useful limitation, it is also rooted in technical limitations. Circumventing this could lead to some interesting options, like square-octagon tilings.

Crucially, two TileData instances with different rotations or different elevations are considered different. Therefore, from one specified TileData, a large amount of different tiles may be pre-generated.

5.1.5.1 Obtaining Every Unique Rotation

If a tile has k unique rotations, then rotations 0 to $k-1$ are unique.

Proof. Let T be a tile with n sides and k unique rotations. This means that the sequence of border types repeats every $\frac{n}{k}$ sides, i.e. rotating the tile k times yields the same sequence of borders.

Therefore there is a repeating pattern with $\frac{n}{(\frac{n}{k})} = k$ elements. This means that elements i and $i + lk, l \in \mathbb{Z}$ are not mutually unique.

Thus any set of more than k neighboring rotations has just as many unique rotations as a set of k neighboring rotations.

The set of n (e.g. all) neighboring rotations has k unique rotations, thus a set of neighboring k rotations also has k unique rotations. Rotations 0 to $k-1$ are thus a set of k unique rotations. □

5.1.6 TileInstance

TileInstance instances are exclusively auto-generated and represent a single generated tile, including:

- coords — a Coords instance denoting where the tile was placed.
- mainTileData — a reference to the so-called "main layer" tile; that is, the basic layer as described in section 2.2.2.7.
- layerTileData — an array of TileData, one for each additional layer.

5.1.7 Layer

In the context of this PCG tool, a Layer will refer to a set of all tiles (and variations thereof) that are part of the same terrain layer — whether the main layer or one of the additional layers. A Layer effectively behaves like a list.

5.1.8 Group

A tile Group contains primarily a list of tiles. This is another data structure used to help organize TileData instances in the Unity inspector. Its primary purpose is to facilitate data exchange between TileManager and TerrainMap components, explained in the subsequent sections.

5.2 TileManager

TileManager encapsulates a dictionary of previously generated TileInstance instances, in effect constituting an object pool[23]. This is unavoidably the centerpiece, and thus a limitation, to almost all other components' functioning.

It also enforces load order in synchronization-critical components. The TileManager, while not disallowing inheritance, doesn't have an abstract defining class; this is one of the core limitations of this system.

Most inspector-based configuration will take place in the confines of this component. This includes tile Group instances and terrain layers.

Additionally, the TileManager creates and handles tile loading requests, both by retrieving tiles from storage and by interacting with the Oracle component. Tile load requests are enqueued for each new position as defined by TileMap. Tile unload requests for tiles that are too far just deactivate the corresponding GameObject instances in the scene.

This makes the PCG inherently generate on-demand with all the drawbacks associated (tiles are never "generated in advance", and tile load order is harder to optimise.

In order to prevent "lag spikes", tile generation requests are added to a queue and there is a configurable cap on how many can be processed per frame.

5.2.1 OptimalOrderTileManager

As discussed in section 2.2.2.6, the load order can be optimised in a number of different ways. An example implementation of an optimal load order system is presented in this override of TileManager functionality. It is only compatible with 4-sided tilings.

Tiles are loaded in a spiral order(23); this is only effective for some tile shapes, most notably square. A request for loading any tile is processed by first checking for the previous tiles in the spiral and generating them if not present.

Unlike many conflict prevention methods, this system can prevent conflicts completely in square tile sets with more than t^2 tiles, where t is the number of border types, as no matter the situation, a compatible tile will always exist.

[23]Object pooling is simply the fact that no generated information is being forgotten. A generated tile can never be un-generated.

5.3 TileMap

The TileMap is an abstract class defining spatial and logical tile ordering with 6 abstract methods:

- GetPositionOfCoords(), which converts Coords into a Vector3 of the tile position in the local space within the given scene.
- GetNearestTileCoords() is, in effect, inverse to GetPositionOfCoords().
- GetElevationOfTile(), calculating a tile's actual y position based on its elevation.
- GetLocalTileRotation(), by default zero, but can return any degree of horizontal rotation. For example, a triangle tile system will have tiles at different rotations.
- GetNeighboringPositions() returns a dictionary with keys being integer directions[24] from the tile and values being Coords of each neighboring tile.
- OppositeSide() finds, for a neighboring tile in some direction, the direction of the tile from the neighboring tile.

These 6 methods are enough to define any single-shape tile system. The remaining logic is common to any tile system, and includes linear transformations and Unity interactions.

The component also keeps track of currently loaded tiles and provides methods for tile loading and unloading.

Two basic tile orderings have been implemented, although more are possible.

5.3.1 RectangleTileMap

RectangleTileMap has a configurable tile size using a 2D vector in the inspector. Calculations in rectangular logic are particularly simple, as demonstrated in the below code snippet.

```
1 protected override Vector3 GetPositionOfCoords(Coords coords)
2 {
3     Vector2 pos = coords * tileSize;
4     return new Vector3(pos.x, 0, pos.y);
5 }
```

[24]Direction 0 is toward -z; directions increment clockwise. A tile's n-th border is toward the n-th direction and rotating a tile n times will make former border 0 face direction n.

5.3.2 HexagonalTileMap

Unlike RectangleTileMap, this component only supports regular hexagons. The positioning for a hexagon tile is a bit more complicated as axial coordinates need to be converted to a Cartesian coordinate system(16):

```
protected override Vector3 GetPositionOfCoords(Coords coords)
{
    float x = Mathf.Sqrt(3) * tileSize * (coords.z / 2f + coords.x);
    float y = 3f / 2f * tileSize * coords.z;
    return new Vector3(x, 0, y);
}
```

5.4 TileSaver and TileStats

These two components have a lot of common logic in that they both use Json.NET(31) to save and load data to a permanent memory by serializing a list of TileInstance instances.

JSON is not ideal for what should be stored as binary data. Its main advantage lies in its readability and flexibility towards changes in stored data.

5.4.1 TileSaver

TileSaver is invoked once upon start time, where it loads all tiles to the memory and "feeds" them one-by-one to the TileManager. It also updates the Oracle through the OnTileGenerated() method, to ensure the state of the Oracle is consistent with how it was in the previous run time.

An example save file for a world consisting of a single tile with an ID of 3 could look like this:

```
[
    {"pos": {"x": 0,"z": 0 }, "id": 3, "elev": 0},
]
```

5.4.2 TileStats

This component keeps track of generated tile statistics, including tile type frequencies, conflict count, and elapsed time. These can be and are used in tile generation.

5.5 InitialState

The goal of the InitialState is to pool data about a preferred initial state for some tiles in the generated world. This is useful in a variety of game design situations, such as having borders to a game map.

Most of this component's functionality is based off having an Editor UI script to service it. This component encapsulates several dictionaries of relevant data and provides an API-like access for UI scripts.

Like TileSaver, tiles defined in this component are guaranteed to appear in the scene as if they were placed before any other tiles.

In the scene, a Gizmo is shown for each coordinate. The gizmo changes color based on whether it is selected, and if there is an initial state set for it. If an initial tile is set, the entire tile is instantiated and displayed at the position.

5.5.1 TilePickerWindow

This UI script services an entire new editor window. It is the only singleton in this project, which means that for multiple tile systems, it switches contexts each time a different tile system is interacted with.

This is done through another script, TileManagerEditor in turn servicing this script by pre-processing Unity event data for the TilePickerWindow.

The window allows to select a coordinate either by clicking on a Gizmo in the scene view, or by manually inputting the coordinates. A number of selecting toggles are available to set a preferred initial tile group or tile.

Chapter 6

Tile Generation

This chapter details the components and data structures related to tile generation. Novel concepts are explained side-by-side with implementation descriptions.

6.1 Oracle

The simplest possible definition for an Oracle is just a function that outputs a tile type given a set of tile types and a coordinate request, and outputs which tile type should be placed at those coordinates.[25]

Oracle is perhaps the single most complicated component. For this reason, its internal logic was split into several independent abstract methods, which are then overridden in multiple inherited components, each of which can be used as a base class with different combinations of overriding possibilities.

Two methods are central to the Oracle's operation:

- GenerateTileAtCoords() — selects a tile to be generated based on a variety of factors. Returns a new TileInstance instance.
- OnTileGenerated() — called after a tile is generated, which happens in exactly two cases: the previous method was called, or a tile was loaded from storage. This method typically contains conflict reduction calculations.

6.1.1 Oracle Base Class

The base class, Oracle operates both at edit time and run time.

[25] In more poetic words, *the Oracle knows the future*

In edit time, it precalculates mutual border compatibility of tiles within one layer and tiles from different layers.[26]. This behavior can be overriden.

In run time, its operation is based upon a dictionary of TileData and their calculated weights. In each step of the calculation, the weights of each tile represent how likely it is to be placed at the given coordinate. Weights are typically from 0 (*soft zero*) to 1. *Hard zero* values can be achieved by removing the tile from the dictionary entirely. Each tile present is called a *candidate*.

The default calculations include modifying the tiles' initial weight based on several simple factors, such as tile frequency compensation, InitialState values, and TerrainMap values. All of these are described in the next sections.

Upon the conclusion of these calculations, the following outcomes are possible:

- A tile has infinite weight — then the tile is automatically selected.[27]
- Multiple tiles with nonzero weights are left — a random one is chosen with a probability proportional to its weight.
- Multiple tiles are left, but with zero weights — a random one is chosen with equal probabilities.
- No tile is left — this signalises a conflict. Conflict solving policy is dedicated to an abstract method, which is also overriden in StandardOracle.

The Oracle also defines a few helper methods aimed mainly at separating the precalculated data structures from direct access.

6.1.2 BasicConnectivityOracle

The simplest possible Oracle simply iterates over the list of candidates and decreases weights of values based on their compatibility with all present neighbors. The following is a reference implementation of this functionality.

```
1 protected override Dictionary<TileData, float> SelectRelevantTiles(
      Dictionary<TileData, float> candidates, Coords coords)
2 {
3     return candidates.MultiplyWeights(GetCompatibilitiesForLayer(
      tileManager.mainLayerTileList, coords));
4 }
```

[26] Also referred to as *intra-layer compatibility* and *inter-layer compatibility*.

[27] Having multiple infinite weight tiles results in undefined behavior.

GetCompatibilitiesForLayer() automatically removes all hard-zero values, and multiplication using the MultiplyWeights() extension method only returns values for common keys.

In fact, the above described is enough to avoid all conflicts for complete tile sets. Unfortunately, the only advantage of such approach is its low computational cost.

This approach is used to generate non-main tile layers regardless of the main tile generation.

6.1.3 AC3ConnectivityOracle

The AC-3 based Oracle represents the metaphorical pinnacle of this work.

The algorithm it is based on, AC-3[28], is based off keeping track of the possible state space for each coordinate.

6.1.3.1 State space navigation

Conceptually, as an CSP algorithm, AC-3 operates by navigating the state space of the system. In this case, it is 3D (2D lattice, 1D list of tile choices)[29].

As it is impossible to represent the entire plane in data, the state space is represented as a vector of values for each coordinate — and coordinates' vectors are loaded into a dictionary lazily, e.g. on-demand. This vector of values is called a Possibility.[30].

Primarily, operations with possibilities use Boolean logic. States are either allowed (don't lead to a conflict) or disallowed (lead to a conflict), although fuzzy logic calculations are also possible.[31]

The goal of any derived conflict reduction algorithm is to remove as many values as possible from this vector. Placing a tile, for example, will lead to a "collapse" in the corresponding coordinate's vector, with only one value remaining positive: the one of the tile placed.

The more values culled, the better; except of course those, that don't lead to conflicts.

[28]Introduced in 2.3.2.3.

[29]Here lies the real reason tile layers do not have advanced conflict reduction; the state space would be 4D and thus as much slower in calculations.

[30]A Possibility, well, shows what is possible.

[31]Allowing for fuzzy logic calculations leads to several interesting changes in the system's behavior. This is further discussed in the evaluation.

6.1.3.2 AC-3

After any value is removed from any coordinate's Possibility, the tile is enqueued in a queue representing all changed state space vectors.

Each neighbor's every value is checked, if at least one value from the changed Possibility supports it. "Support" is defined for any combination of two tiles simply as their mutual compatibility.

If an option is not supported by the neighboring Possibility, it is removed from the vector, and also enqueued.

AC-3 creates new Possibility instances during its operation. Some of these instances can be directly initialized with some values marked as unsupported even before a tile is placed on their coordinates. This happens when the InitialState has a preferred state for the coordinates. As some values are newly unsupported, the just added Possibility is also enqueued.

6.1.3.3 AC-3 iteration limit

As AC-3 iterations can cause a "chain reaction" without limits, arbitrary limits are enforced — the distance from the first processed element, and the number of iterations per tile placed.

Worst-case scenario, the number of iterations is proportional to the distance cutoff squared times the number of tile types (e.g. each iteration removes only one option).

For most tile sets though, the minimum distance two tiles can be apart is quite low. In the ocean-mountain example, this is at most 2 for any two tiles. The largest minimum distance for any two tiles corresponds to the distance a typical AC-3 iteration will reach. Tiles further away will almost always have a Possibility that consists of only positive values.

In other words, tiles further away will almost always have all options available. In his work, Dykeman calls this effect *sphere of influence* of a tile.(9)

Thus for many tile sets, the limits are not relevant and serve more of a "failsafe" purpose.[32] The distance and iteration cutoff have default values of 10 and 100, respectively. Reaching the cutoff limits during a computation will result in less effective conflict reduction.

[32]Although if the limit is set too low, it is possible that the preferred initial state will not be propagated correctly.

A special case is terrain with many different height levels, for which the sphere of influence is directly proportional to the number of height levels. Due to terrain volatility, setting the radius too low can and will result in conflicts; these conflicts will often spread in a chain reaction-like manner.[33]

6.1.4 StandardOracle

The standard oracle provides standard logic for connectivity precalculation and conflict processing.

Conflicts are resolved by choosing a tile that is compatible with as many neighbors as possible.

6.1.5 Further Oracle Possibilities

Bar certain structural limitations, the Oracle system is designed to allow for further expansions in a simple manner. This includes:

- Connecting with Pathfinding to make terrain that is always traversable.
- Heuristic conflict reduction, further culling some options in an approach based on utilising the remaining inter-frame downtime.[34]
- Different logic for connectivity, tile weight compensation, and conflict solving.

6.2 TerrainMap

A component closely related to the InitialState and Oracle, TerrainMap adds an additional level of structure to the PCG. Being yet another abstract component, it provides the base logic for its task.

It has two main tasks: choosing an optimal tile group for each coordinates, and choosing an optimal tile elevation.[35]

For example, the already mentioned ocean-mountain terrain world could be split into areas belonging to one of two tile Group instances:

- Ocean — big, interconnected areas with zero elevation differences.
- Mountains — smaller, disjoint areas with a rugged height profile.

[33] Interestingly, this problem can be almost entirely avoided by setting the tile load distance to less than the radius.

[34] A MCTS-like approach could be used by pretending to place random tiles and seeing how often different options lead to conflicts.

[35] The name is a bit unfortunate, but had to stay for a lack of a better one.

In other words, each tile group should have configurable sizes and other parameters for procedural generation. For single group tile sets, only the elevation profile is relevant.

6.2.1 GroupSettings

Each tile Group has an automatically generated GroupSettings instance, editable in the inspector. This means, that a different set of parameters can be passed to the TerrainMap for each Group. The class is derived from ScriptableObject and can be overriden to enable custom settings.[36]

6.2.2 GUI

An array of spherical Gizmos is displayed in the scene view for each coordinate. Each Gizmo has a color unique to the group that should be placed there, and an elevation corresponding to the calculated optimal tile elevation placed. A screenshot of this behavior can be seen in figure 4.3.

Colors are by default distributed using an 1D variation of Fibonacci hashing across a state space of HSV colors. Fibonacci hashing selects such a value from a range, that is furthest from all already selected values.(37) This is employed for selecting the hue; for any amount of groups, an almost-optimal set of hues will always be preselected.[37]

The color for each group can always be selected manually.

6.2.3 BasicTerrainMap

For 1-3 layer systems, the BasicTerrainMap alongside the BasicGroupSettings may be used. The settings class doesn't allow setting any parameters outside the group's Gizmo color. The height map itself has two parameters, one for area size and one for "hill" size.

The functionality is based off sampling scaled Perlin noise(36) and discretizing it to obtain a preferred tile group. For elevation, an offset Perlin noise is amplified and sampled.

[36]The initialization method can be overriden if specific initialization is needed.

[37]A small variation in color darkness (value) is also added to hopefully aid with colorblind usage.

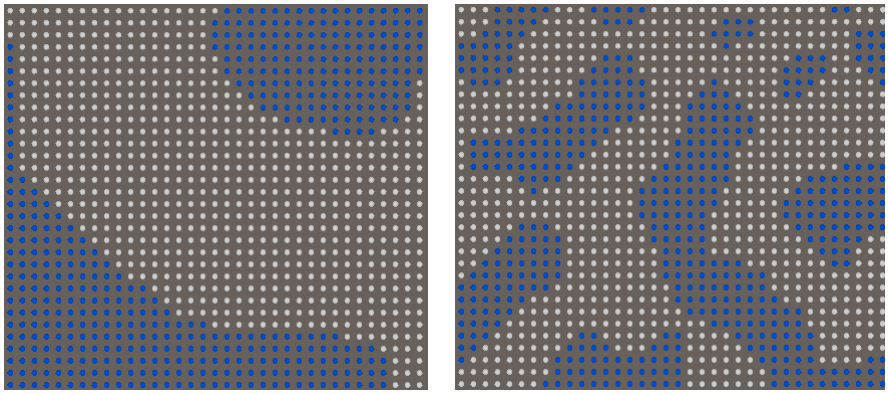

Figure 6.1: A group size scale of 9 (left) and 2 (right).

6.2.4 BiomeTerrainMap

This component offers a more usable amount of abstraction for terrain generation. Perlin noise sampling is wrapped in several algorithms, allowing more "humanly understandable" values to be used as input. While this may, and will, still lead to unexpected behavior, it is much simpler to "get a feel" for what the variables correspond to.

In this system, the concept of *biomes* is used. The first part of the calculation consists of splitting the world into various areas, assigning each one its own customised height map. In this case, each biome has its own Perlin map variation, and for each coordinate, the one which has the largest value wins.

The second part of the calculation aims to smooth out the transitions between biomes' height maps. This is done by sampling the height of near coordinates and averaging the results. This process is called *blending*.

6.2.4.1 Biome Border Blending

Blending has been implemented as a weighted average of the height values of several coordinates in a limited configurable radius. A value contributes to the average if it is of a different biome than the central coordinate. The weight decreases linearly with distance from the center.

Blending through all tiles in a radius is computationally expensive, and scales square-proportionally with the given radius. This led to two optimizations: a cache for already computed intermediate values, and a "fast blending" option, which only selects a linear amount of tiles in any given radius, on behalf of blending accuracy.

Simple stress testing showed the maximum radius with a reasonable calculation time to be around 10 with fast blending. The default is set to 2.

6.2.4.2 BiomeGroupSettings

The GroupSettings has been overriden to provide 5 different input variables, two for biomes and three for height maps:

- size — roughly corresponds to each biome's size. Works well when paired with biomes of similar size.
- frequency — how often the biome appears.
- middle elevation — the average height of the biome.
- height spread — maximum tile height in a given biome.
- ruggedness — regulates steepness of terrain which in turn dictates hill shape.

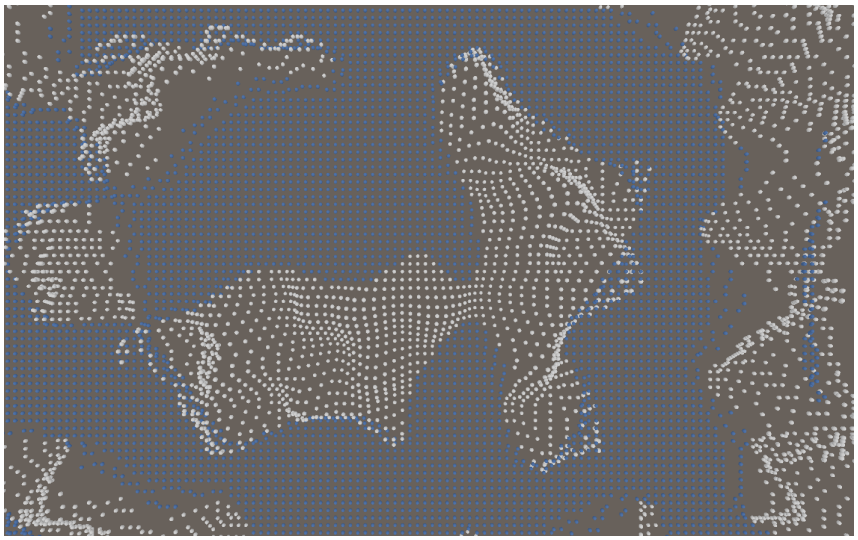

Figure 6.2: Ocean-mountain demonstration of BiomeTerrainMap

6.2.5 Further TerrainMap Possibilities

It is important to note that the above two examples are just rudimentary proof-of-concepts. There is no system that encapsulates all, or even many, options that are available for terrain generation. These can be arbitrarily complex. Examples include Minecraft's(29) 5D state space navigation (Three dimensions, plus two for humidity and temperature of biomes). Even then they can be unsuitable for use outside of niche applications.

This component was designed to be used in many different ways, but requires prior customization by the developer. A versatile UI, some helper methods, and an API-like access system does precisely that.

Even in the case of $\mathrm{BiomeTerrainMap}$, there are a number of things that could be improved[38]; but still, these aren't improvements for every use case. The proof-of-concept shown here is a starting point. This is just the start of the figurative path, leading to vast, unexplored, unmappable areas.

6.3 Random Generation

This section offers a complete walkthough though the process of generating a single tile. The main purpose of this section is to show the mechanics that were omitted in previous chapters and to put them into perspective. Most of this logic is part of the Oracle.

6.3.1 Start Time

Some calculations are conducted in advance. This is mainly done for tile type variant generation and compatibility precalculation, including tile elevation profiles. All saved tile data is loaded into memory.

6.3.2 Candidate Initialization

When the reference point moves to "uncharted" territory, a new tile generation request is created and enqueued.

The process upon dequeuing starts with a dictionary of candidate ($\mathrm{TileData}$) and weight pairs. The weight is equal to the given tile's initial weight, divided by the number of unique rotations it has.

6.3.3 Weight Compensations

There is one other major weight compensation, notably tile frequency compensation. If a tile is placed disproportionately often compared to its weight, its weight is modified to negate this effect.

For a tile $t \in C$, the set of all candidates, $p_{ideal} = \frac{weight(t)}{\sum_{c \in C} weight(c)}$ is the ideal, or expected frequency of the tile, and $p_{real} = \frac{count(t)}{\sum_{c \in C} count(c)}$ is the actual frequency. The compensated weight is thus $w = weight(t) \cdot \frac{p_{ideal}}{p_{real}}$

While this is effective at offsetting typical statistical outliers, it won't be able to compensate for massively unbalanced tile sets. In fact, compensation pushing for an

[38] As in: multi-tiered Perlin noise, river biomes, 3D biomes, fuzzy biome selection, custom transitions, ...

unrepresented tile type to have a higher frequency may result in other tiles having less opportunities to be placed.[39] This behaviour is further explained in chapter 9.

Figure 6.3: An unbalanced tile set. All tiles have equal weight.

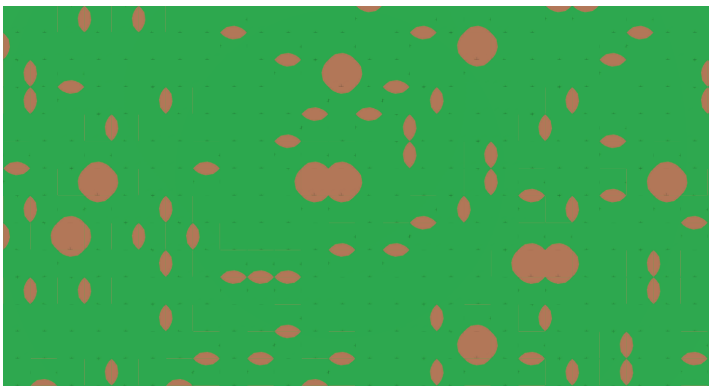

Figure 6.4: Without compensation, some tiles appear much more often than others.

Compensation	meadow tiles	border tiles	city tiles
ideal	33.3%	33.3%	33.3%
none	19.0%	74.6%	6.4%
rotation	49.3%	48.4%	2.3%
rotation and frequency	17.2%	74.5%	8.3%

Table 6.1: Compensation mechanic results for a sample size of 100489 tiles

6.3.4 InitialState and TerrainMap

Tiles that have a preferred tile state in the InitialState component are automatically selected, and the generation skips everything until layer selection.

Tiles without a preferred initial state will behave according to the TerrainMap. Tiles of different groups than the selected one have their weights set to a soft zero.

[39] In the extreme example of fig. 6.3, it is almost impossible to reach a balance, as more city tiles result in more border tiles, offsetting the balance. The tipping point (achievable with weights around $0.05, 0.05, 1$) is very unstable and leads to a combination of normal meadows and gigantic city formations.

Tiles of different heights have their weights decreased exponentially for each unity of height difference they have towards the height map. This does not mean, however, that the generated tiles will closely match the preferred state.[40]

6.3.5 Connectivity

Tile candidates, which are deemed incompatible, typically due to leading to a conflict, are discarded (hard zero). From the remaining candidates, a single tile is chosen by weighted probability. If no tiles remain, the state is conflicted and the most suitable, albeit incompatible tile is chosen.

6.3.6 Layers

After a tile is generated, a layer tile is generated for each layer. The process is simpler, consisting only of weight compensations and simple connectivity-based culling. In case of a conflict, the most suitable tile is chosen as discussed in the previous section

6.3.7 Finalization

The tile choice is then sent to the TileInstance constructor. Typically the tile is then activated on the tile map by instantiating its GameObject. TileStats statistics are updated, and

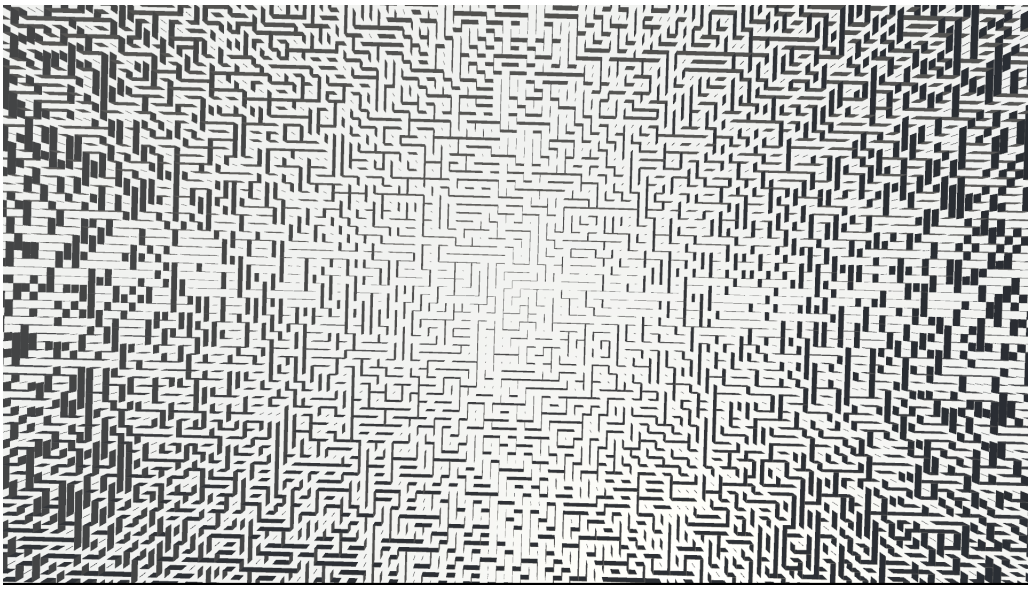

Figure 6.5: Example maze world generated in this system.

[40]For example, setting any height map for a system of flat tiles will still result in a flat tiling.

Chapter 7

Pathfinding

Pathfinding has been implemented as a separate component; it provides several premade settings for terrain navigation calculations, as well as an overridable method for defining any other settings.

The centerpiece of this component is the GetPath() method, which returns a list of traversed coordinates on the shortest path from one coordinate point to another.

7.1 A* Algorithm

To calculate the shortest path, a standard A* "greedy best-first" algorithm(35) is used.

The main calculation loop of A* pathfinding iterates over a priority queue of potential shortest path candidates. Every iteration, the highest priority candidate is taken: if it ends at the goal, it is selected as the shortest path; else, all valid moves are added to the candidate's path and enqueued.

The candidate's priority[41] is calculated as a sum of the candidate path length and a heuristic function.(49)

Since the priority is organized so that paths with low sums of lengths and heuristic values (e.g. estimated remaining lengths) are investigated first, the algorithm will usually quickly locate the shortest path.

If no path was found, an empty list will be returned as the result.

[41] Priority is "more important" the *lower* its value is.

7.2 Parameters

- maximumSteps defines the maximum number of iterations of the A* algorithm before the search is given up.
- maxHeightUp, maxHeightDown — specifies the maximum height offset of two neighboring tiles that is still considered traversable. Defaults to +1 and -1.
- nonGeneratedIsTraversable — if and only if true, tiles that haven't yet been generated are considered passable. This is useful for ensuring generated terrain is traversable.

Additionally, each tile border defines if it is traversable while moving in and out from the tile.

7.2.1 A* Iteration Limit

The maximumSteps default was chosen to be 1000 based on a simple stress test. The test was conducted within the Unity editor, meaning actual use cases will run faster. Both memory time complexity appear to be quadratic; this aligns with theory, as in 2D space, each iteration consists of a linearly increasing number of operations.[42]

iterations	average time (s)
1000	0.20
2000	1.0
3000	2.2
4000	4.1
7000	11
10000	24

Table 7.1: Calculation time for number of iterations.

7.3 Heuristics

In simple terms, a heuristic function estimates the length of the remaining path from the goal[43]. Different tile sets will have different useful heuristics; consequently, the heuristic function in the Pathfinding module can be overridden.

The default heuristic function simply computes the distance between current coordinates and those of the target:

[42]For one iteration, a linear amount of queue elements is processed, each in constant time.

[43]This estimate should always underestimate the real distance. Overestimating functions are not guaranteed to find the shortest path, but can run faster.(49)

```
1     protected virtual float Heuristic(Coords current_pos, Coords
      goal_pos)
2     {
3         return (current_pos - goal_pos).manhattanDistance;
4     }
```

7.4 VisualPathfinder

VisualPathfinder is a module that allows setting new tasks and viewing pathfinding results in the inspector, as well as visualising them in the game world.

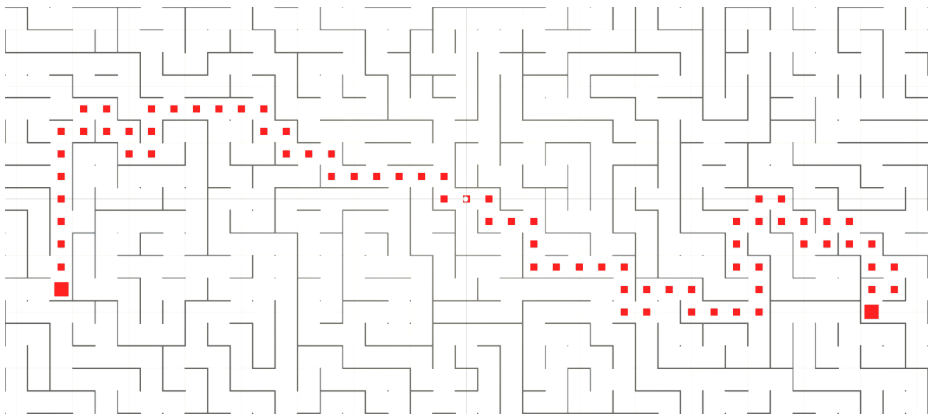

Figure 7.1: A shortest path with a length 75, navigating through a maze. This was calculated in 950 iterations.

Chapter 8

Usage

The reference implementation of this work has been published in a publicly accessible Git repository.(24) Installation is best done with Unity's Package Manager tool.

The package structure complies with unity guidelines(44) for better orientation. It is described in appendix B.

8.1 Basic Workflow

One of 12 basic setups can be placed directly within a scene through the hierarchy context menu. TileData and BorderData can be created using the project context menu.

Most of the work needed to transform a set of tile 3D objects into a functioning PCG then consists of inspector-based editing.

Default values should be usable for most cases, as described in several approximate performance trials in the previous chapters. Non-default values can always be set in the inspector. Every displayed input setting has a tooltip explaining it. Data validation is performed for complex inputs, warning messages are utilized if a discrepancy occurs.

For use cases requiring more customization, code is set up to allow for simple derived classes. For example, any standard biome/height map generator can be adapted for use by the TerrainMap, using a derived class as a wrapper.

All classes, overridable methods, public methods, and some private methods are documented using XML documentation.(28) Less intuitive code has comments attached.

8.2 Example Usages

The package is bundled together with several samples. These serve both as demonstrations of the project and as reference use cases for better familiarization. Additionally, a quick guide can be found in the README file.

These include:

- Carcassonne — an example containing complete tile sets based on the popular Carcassone board game is implemented for both square and hexagonal tiles.
- ContourLines — example usage of conflict reduction (see figure 2.9) and an initial state.
- TerrainHeight — a sample showcasing terrain elevation.
- Maze — a scene showcasing pathfinding and layers.
- MountainLakes — a sample showcasing biomes in a 2D setting.

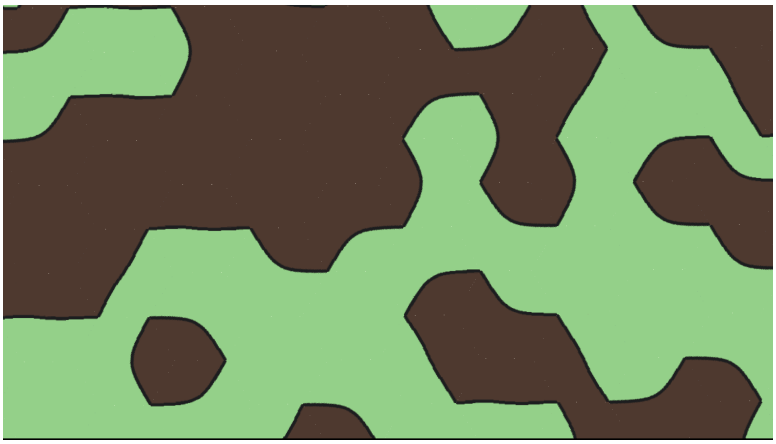

Figure 8.1: A hexagonal variant of the Carcassonne scene

8.2.1 Example Game Prototype

ParcourGame is a first-person 3D platformer game prototype designed to showcase features of the PCG. The game has minimal backstory and graphical design, but is playable, meaning it has a complete enough set of game mechanics.

The playing field is a strip of procedurally generated obstacles, with new tiles generating in one direction, and old ones unloading in the other; the player has to navigate across the obstacles without falling off or staying too far behind, as falling off yields death. The generation rate increases linearly with time, as does the player's maneuverability.

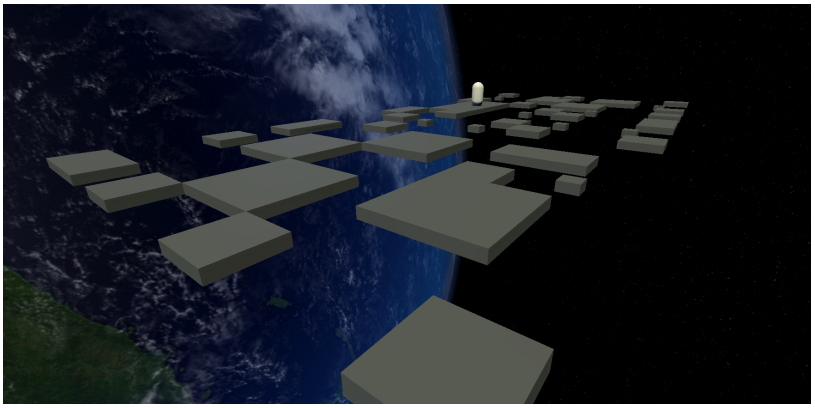

Figure 8.2: A third-person view of the world generated in the game prototype.

Some of the showcased features include:

- connectivity — tile's edges are of three different types, ensuring no gap is too large.
- tile elevation — there is added difficulty in some tiles being generated progressively higher and thus being harder to reach. This uses a slightly modified tile height system.
- reference point — the reference point for generation is not set to the player's location, but rather moves on its own.

8.3 Generic Use Cases

For a toolset as generic as this one, the ideal state is to allow *any* tile connectivity-based generator to be built upon it, with or without the need to create derived classes. In other words, the goal is to *minimise* development time for custom generators on this basis.

In practice, any tool is limited in its scope. Current limitations are described in chapter 9. The following are various possible use cases:

- 2D biome generation
- hilly terrain generation
- room-based worlds (mazes, dungeons)
- procedural texture generation
- board generation for desktop game adaptations

Chapter 9

Evaluation

This chapter is both a retrospection and a look ahead.

All stated generator specifications (See 3.2) have been implemented. Some additional features are also present.

Specifications 2 (layers) and 6 (initial state) still leave a lot of room for improvement, but their current implementations are sufficient for basic use.

9.1 Research Question Evaluation

9.1.1 Run Time Performance

What are the limits of real time generation?

This was one of the most tricky limitations to this work, requiring constant thought throughout each phase of the implementation process.

Two most notable optimizations have each led to about an order of magnitude better performance: rewriting Possibility and some other data structures to operate on lower-level code, and adding queues (buffers) to various components to even out uneven loads.

Although several other optimizations were added, the generation is still at its limits in terms of real-time usage, both in the editor and in run time.

Some features, and especially feature combinations, are barely usable in real time. Often multiple different components approach the real-time limitations in different manners:

- It is not possible to load more than $\sim$ 100 tiles per second, even for the simplest tile sets.

- For tile sets of above $\sim$ 150 tiles its is barely possible to fit *one* generation request into a whole frame. AC-3 will use up more calculation time than allowed for a single frame for spheres of influence with a radius of about 20.

Further optimizations face diminishing returns. Even if optimizations were applied to address one aspect, other critical areas would remain constraining factors. To conclude, this type of generation is largely not suitable for real-time applications.

9.1.2 Conflict Reduction

How to effectively reduce conflicts?

Implementing AC-3 was a success in this regard; although its limitations still prevail: there is no option for fuzzy logic, including soft zeroes. Layer conflicts are not significantly avoided, leading to a narrower range of usability. But the main constraint remains: real-time performance.

Expanding the already extensive calculations could lead to further hindering of the defined performance goal, but is possible. Not being able to avoid conflicts for a concave tile generation order is the main downside of the current Oracle.

Overall, the conflict reduction provided is suitable for most use cases.

9.1.3 Flexibility

How many different problems does this tool help with?

A wide range of features is present in the implementation. All features are usable and can be used in combination with each other. Components offer a ready-to-use base implementation that can be further expanded by the developer depending on the use case.

A lot of the flexibility is based off the versatile data definitions, especially the border connectivity relation, and the decentralization of logic into semi-independent calculations.

In terms of usability, the least developed part of this implementation is the InitialState UI. Setting up an initial state is tedious and the visualisation is rudimentary.

9.2 Limitations

Other limitations include:

- Pathfinding cannot generate tiles when navigating through unknown terrain.

- Tiles cannot be unset (removed). This leads to a more static world than generally desirable. The decision to omit this feature lies in the Oracle's state; Reversing a tile placement operation can be very hard to implement properly as to not lead to an undefined state.
- The initial state editor UI is limited in its usability as it cannot modify multiple tiles simultaneously.
- Tile generation requests are not entirely optimised, and can lead to inconsistent performance.

9.2.1 Tile Predicting

Perhaps the largest limitation can be seen in figure 2.3: The Oracle isn't capable of predicting which tiles will lead to the preferred tiles being compatible at their positions. E.g. on group area transitions, this leads to border tiles being placed randomly until the correct tile group happens to be compatible.

This is a limitation that is entirely unmitigated. There is almost no relevant theory to base any mitigations on. Dykeman's generator(9), operating on a finite area, uses random initial tile placement and multiple iterations to smooth out a result. Setting and resetting tiles multiple times is common. In this PCG, tile unsetting is neither possible, nor favourable for an on-demand generation focus.

9.3 Future Additions

Expansions possible to the reference implementation include:

- Tile layer conflict reduction.
- Tile load order optimization.
- Dynamic worlds — tile setting, unsetting, modification.
- Advanced edit time state setting.
- Integration of further constraints into generation (e.g. semantic)

Conclusion

The purpose of this thesis was to explore the options in real-time procedural generation of tile-based content, and, more importantly, implement these in an exemplary toolset.

Several existing solutions were examined and discussed, and a set of relevant concepts was put together and expanded up to the limits of contemporary research. Current research in connectivity-based procedural generation is not very expansive — each work exploring what is possible in this area can still be called a pioneering one.

Concrete implementation goals, such as terrain layers, usage of smooth noise, pathfinding, state serialization, and initial state setting, were all accomplished. Their implementation is often basic, but is also robust, flexible and expandable. The implementation showcases what is possible in tile connectivity-based generation even with real-time limitations.

The overarching goal for the implementation part of this project was (as with any PCG in fact) to simplify the process of creating various formal systems. Although significant, satisfactory and novel progress has taken place in the development of such a PCG, there are still limitations in place, mostly of a structural kind.

Quite obvious from the evaluation in the previous chapter is a final, largest caveat of this work: the scope was *too wide*. Many features are too different to be reasonably coexisting, especially with a restrictive limitation of real-time performance.[44] This leads to the PCG system losing *meaning*, the above explained tile predicting being a prime example.

[44] In other words: flexibility at the expense of functionality is at the expense of flexibility.

Of course, this could be fixed with additional logic. Perhaps an extension to Oracle that optimizes load order, and can simulate future situations and evaluate which tiles are best to place. This would necessarily require several further optimizations.

But perhaps it is a better idea to stop this string of research here. Perhaps instead of constant feature additions, it is time to think of feature *subtractions*.

Not trying to force together terrain and connectivity, on-demand loading and predictivity, giant tile sets and real-time restrictions — but rather focusing on one or a few of the features mentioned in this thesis and expanding upon them. There are so many just barely explored possibilities.

Like concluded in 2.3.2.5, *limiting variety gives the system meaning*.

Sources

1. ACAEUM.COM. *D&D Basic Set* [online]. [visited on 2023-01-17]. Available from: https://www.acaeum.com/ddindexes/setpages/basic.html.
2. AMANDINE ENTERTAINMENT. *Ultimate Terrains* [Unity asset pack]. [N.d.]. [visited on 2023-01-15]. Available from: https://assetstore.unity.com/packages/tools/terrain/ultimate-terrains-voxel-terrain-engine-31100.
3. BANDAI NAMCO ENTERTAINMENT. *Pac-Man* [online]. [visited on 2023-01-17]. Available from: https://pacman.com.
4. BLOODRIZER. *Kittens Game* [online]. [visited on 2023-01-17]. Available from: https://kittensgame.com.
5. BOLLER, Sharon. *Learning Game Design: Game Mechanics* [online]. [visited on 2023-01-17]. Available from: http://www.theknowledgeguru.com/learning-game-design-mechanics/.
6. CHRISTOPHFRANKE123. *How to serialize Dictionary with Unity Serialization System* [online]. 2014. [visited on 2023-05-21]. Available from: http://answers.unity.com/answers/809221/view.html.
7. COLLINS, Karen. An Introduction to Procedural Music in Video Games. *Contemporary Music Review*. 2009, vol. 28, no. 1, pp. 5–15. Available from DOI: 10.1080/07494460802663983.
8. CR31.CO.UK. *Wang tile definition* [online]. [visited on 2023-01-15]. Available from: http://www.cr31.co.uk/stagecast/wang/intro.html.
9. DYKEMAN, Isaac. *Procedural Worlds from Simple Tiles* [online]. 2017. [visited on 2023-01-15]. Available from: https://ijdykeman.github.io/ml/2017/10/12/wang-tile-procedural-generation.html.

10. ELECTRONIC ARTS. *SimCity* [online]. [visited on 2023-01-15]. Available from: https://www.ea.com/games/simcity/simcity.

11. FIDE. *FIDE LAWS of CHESS* [online]. [N.d.]. [visited on 2023-01-15]. Available from: https://www.fide.com/FIDE/handbook/LawsOfChess.pdf.

12. GIANT SPARROW. *What Remains of Edith Finch* [online]. 2017. [visited on 2023-01-17]. Available from: https://store.steampowered.com/app/501300/What_Remains_of_Edith_Finch/.

13. GRELSSON, David. Tile Based Procedural Terrain Generation in Real-Time. In: 2014.

14. HAMKINS, Joel David. *Conjecture on NP-completeness of tessellation of Wang Tile up to finite size* [MathOverflow]. [N.d.]. Available from eprint: https://mathoverflow.net/q/157714. (version: 2014-02-16).

15. HELLO GAMES. *No Man's Sky* [online]. [visited on 2023-01-17]. Available from: https://www.nomanssky.com/.

16. ISAAC. *Hexagonal Grid Coordinates To Pixel Coordinates* [Stack Overflow]. 2010. [visited on 2023-01-20]. Available from: https://stackoverflow.com/a/2459541/6123979.

17. IT3RATION. *How to make a readonly property in the inspector?* [online]. 2014. [visited on 2023-05-21]. Available from: http://answers.unity.com/answers/801283/view.html.

18. JOHNSON, Curt. *Microsoft Minesweeper* [online]. [visited on 2023-01-15]. Available from: https://www.microsoft.com/en-us/p/microsoft-minesweeper/9wzdncrfhwcn.

19. JUNGMING SUH Han Zhang, Zian Wang. Procedural generating of plants models using L-system [online]. [N.d.] [visited on 2023-01-20]. Available from: https://hanzh015.github.io/Procedural_generating_of_plants_models_using_L_system.pdf.

20. KOSTER, Ralph. *A theory of fun for game design* [online]. Scottsdale, 2004 [visited on 2023-01-15]. ISBN 978-1449363215. Available from: https://www.theoryoffun.com/.

21. LAGAE, Ares; KARI, Jarkko; DUTRÉ, Philip. Aperiodic Sets of Square Tiles with Colored Corners. 2006.

22. LEYTON-BROWN, Kevin. *CSPs: Arc Consistency* [online]. 2010. [visited on 2023-01-15]. Available from: https://www.cs.ubc.ca/~kevinlb/teaching/cs322%5C%20-%5C%202008-9/Lectures/CSP3.pdf.

23. LHF. *On a two dimensional grid is there a formula I can use to spiral coordinates in an outward pattern?* [Mathematics Stack Exchange]. [N.d.]. Available from eprint: https://math.stackexchange.com/q/163101. (version: 2012-06-26).

24. LÍBAL, Rudolf. *Tile-based procedural generation* [online]. [visited on 2023-01-17]. Available from: https://gitlab.fel.cvut.cz/libalrud/proj-pcg.

25. LINDEN, Roland; LOPES, R.; BIDARRA, Rafael. Designing procedurally generated levels. In: [online]. 2013, pp. 41–47 [visited on 2023-01-15]. Available from: https://www.researchgate.net/publication/288320122_Designing_procedurally_generated_levels.

26. LUKKARILA, Ville. The 4-way deterministic tiling problem is undecidable. *Theoretical Computer Science*. 2009, vol. 410, no. 16, pp. 1516–1533. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2008.12.006. Theory and Applications of Tiling.

27. MAUNG, David. *Tile-based Method for Procedural Content Generation* [online]. 2016. [visited on 2023-01-15]. Available from: https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=osu1461077485&disposition=inline. PhD thesis. The Ohio State University.

28. MICROSOFT CORPORATION. *Documentation comments* [online]. Microsoft Corporation, 2023. [visited on 2023-05-23]. Available from: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/.

29. MOJANG. *Minecraft* [online]. [visited on 2023-01-17]. Available from: https://minecraft.net.

30. MONCKTON, Christopher. *Eternity II Instruction booklet* [Internet Archive Wayback Machine]. 2007. Archived on: 2007-10-06, URL: https://web.archive.org/web/20071006033127/http://uk.eternityii.com/Download.ashx?id=19934.

31. NEWTONSOFT. *Json.NET* [online]. [visited on 2023-01-17]. Available from: https://www.newtonsoft.com/json.

32. NYKAMP, Duane. *State space definition* [online]. Math insights. [visited on 2023-01-15]. Available from: https://mathinsight.org/definition/state_space.

33. OEIS FOUNDATION INC. *The On-Line Encyclopedia of Integer Sequences*. 2023. Published electronically at http://oeis.org.

34. OPENTTD DEVELOPMENT TEAM. *OpenTTD: An open-source transportation simulation game* [https://openttd.org/]. [N.d.]. [visited on 2023-01-20].

35. PATEL, Amit. *Introduction to A** [online]. [visited on 2023-01-19]. Available from: https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

36. PERLIN, Ken. *Making Noise* [Archived on Web Archive]. 1999. [visited on 2023-01-15]. Available from: https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/.

37. PREISS, Bruno R. *Fibonacci Hashing* [Archived on Web Archive]. 2004. [visited on 2023-01-15]. Available from: https://web.archive.org/web/20130515024557/http://brpreiss.com/books/opus4/html/page214.html.

38. ROBINSON, Raphael M. Undecidability and nonperiodicity for tilings of the plane. *Inventiones mathematicae*. 1971, vol. 12, pp. 177–209.

39. SMITH, Gillian. *An Analog History of Procedural Content Generation* [online]. [visited on 2023-01-17]. Available from: http://www.fdg2015.org/papers/fdg2015_paper_19.pdf.

40. TANYA SHORT, Tarn Adams (ed.). *Procedural generation in Game Design*. Taylor & Francis, 2017. Available also from: https://www.taylorfrancis.com/books/edit/10.1201/9781315156378/procedural-generation-game-design-tanya-short-tarn-adams.

41. TEUBER, Klaus. *Catan* [Board game]. 1995. Available also from: https://www.catan.com/understand-catan/game-rules.

42. THE TETRIS COMPANY. *Tetris* [online]. [visited on 2023-01-17]. Available from: https://tetris.com/.

43. TUPPER, Jeff. Reliable Two-Dimensional Graphing Methods for Mathematical Formulae with Two Free Variables. 2001. Available also from: http://www.dgp.toronto.edu/~mooncake/papers/SIGGRAPH2001_Tupper.pdf.

44. UNITY TECHNOLOGIES. *Package layout* [online]. [visited on 2023-01-20]. Available from: https://docs.unity3d.com/Manual/cus-layout.html.

45. UNITY TECHNOLOGIES. *Unity* [online]. [visited on 2023-01-17]. Available from: https://unity.com/.

46. UNITY TECHNOLOGIES. *Unity Asset Packages* [online]. [visited on 2023-01-17]. Available from: https://docs.unity3d.com/560/Documentation/Manual/AssetPackages.html.

47. UNITY TECHNOLOGIES. *Unity Documentation* [online]. [visited on 2023-05-05]. Available from: https://docs.unity3d.com/2021.3/Documentation/Manual/index.html.

48. VILIAM LISÝ, Branislav Bošanský. *Constraint Satisfaction Programming and Scheduling* [online]. [N.d.]. [visited on 2023-05-14]. Available from: https://cw.fel.cvut.cz/wiki/_media/courses/zui/slides-l9-2023.pdf.

49. VILIAM LISÝ, Branislav Bošanský. *Informed (Heuristic) Search* [online]. [N.d.]. [visited on 2023-01-19]. Available from: https://cw.fel.cvut.cz/b212/_media/courses/zui/slides-l3-2022.pdf.

50. WILD CARD. *Dwarf Quest* [online]. [visited on 2023-01-15]. Available from: https://www.gamersgate.com/product/dwarf-quest/.

51. WREDE, Klaus-Jürgen. *Carcassonne* [Board game]. 2000. [visited on 2023-01-15]. Available from: https://images.zmangames.com/filer_public/d5/20/d5208d61-8583-478b-a06d-b49fc9cd7aaa/zm7810_carcassonne_rules.pdf.

Appendix A

List of abbreviations used

(n)D n-dimensional (e.g. 2-dimensional)

AC-3 Arc Consistency 3

API application programming interface

BFS breadth first search

CPU central processing unit

CSP constraint satisfaction problem

GPU graphics processing unit

(G)UI (graphical) user interface

ID identifier

JSON JavaScript object notation

NP-complete nondeterministic polynomial-time complete

PCG procedural content generator (noun), or procedural content generation (verb)

URL uniform resource locator

Appendix B

Unity asset package structure

This is an overview of the created Unity asset package structure. Some files have been omitted for brevity.

```
proj-pcg
├── package.json
├── README.md
├── CHANGELOG.md
├── LICENSE.md
├── Third Party Notices.md
├── Editor
│   ├── Resources
│   └── Scripts
├── Runtime
│   └── Scripts
└── Samples
    ├── Carcassonne
    │   ├── Square
    │   └── Hexagonal
    ├── ContourLines
    ├── Maze
    ├── Biomes
    └── TerrainHeight
```

Appendix C

Contents of included CD

The implementation included is identical in structure to the one available at Gitlab (https://gitlab.fel.cvut.cz/libalrud/proj-pcg).

root
- thesis.pdf . . . . . This thesis as a PDF
- proj-pcg . . . . . The implementation as a Unity package
- screenshots . . . . . A collection of screenshots
- game prototype.zip . . . . . A playable game prototype

Appendix D

Installation

D.1 Installation with Unity Package Manager

In Unity, open the ”Package Manager” tab (Window -> Package Manager). Select + -> Add package from Git URL.... Paste the Gitlab repository URL (https://gitlab.fel.cvut.cz/libalrud/proj-pcg.git). Unity will do the rest automatically.

D.2 Installation from CD

In Unity, open the ”Package Manager” tab (Window -> Package Manager). Select + -> Add package from disk....

A file explorer will open. Navigate to the CD's proj-pcg directory, and select the package.json file inside. Unity will do the rest.

D.3 Sample usage

One of Unity's limitations is the inability to load read-only scenes. Package contents are hard-coded to be read-only. To view, edit, or use a sample scene, copy it into your project.