# Procedural Generation in Unexplored

---

### *Course details*

**Name**: Interactive Agents and Procedural Generation
**Code**: ECS7016P

### *Student details*

**Name**: Pranav Narendra Gopalkrishna
**Number**: 231052045

---

# TABLE OF CONTENTS

# Introduction

## About the game

Unexplored is a video game — specifically a roguelite action-RPG dungeon-crawler — that applies procedural content generation (PCG) to create dungeon levels (20 overall), including puzzles and encounters. It features real-time action and one overarching objective, namely going down to the lowest (i.e. 20th) level of the dungeons to retrieve the "Staff of Yendor" to get it back to the surface; hence, it is a two-way journey that is — in essence — a cycle. As we shall see, cyclic generation is the keystone innovation that makes Unexplored stand apart in terms of both game-design and gameplay.

## Level-generation & cyclic generation

A key focus of the PCG for level-generation is *coherence with complexity*; the PCG must produce levels that are coherent (i.e. internally consistent) while also being complex (involving many interrelated, interacting elements). Cyclic generation is the abstract solution to the game-design's key objective of coherence with complexity. It takes inspiration from the principles of hand-drawn game maps that often involve cycles in paths (leading to a wider range of possible solutions to the level and more organic, non-linear traversal within the level). Technical details shall be discussed in the section "***System design***".

To concretise the abstract solution, level-generation for the dungeons involves three basic steps, corresponding to three basic AI systems, namely: (1) cyclic graph generator, (2) graph-to-tilemap translation, and (3) terrain-generation for the tilemap (involving another layer of PCG).

## Focus of the report

This report shall focus on the idea of cyclic dungeon generation, its implementation in Unexplored and how abstract level-design is concretised into playable levels.

# The game

## Role of the AI systems within the overall design

In Unexplored, the cyclic graph generator is used to generate the graph that represents the abstract structure of the levels. It involves generalised aspects that are further detailed in later steps. The graph-to-tilemap translation translates the graph to a tilemap that *directly* maps to the gameplay area. This system involves three key aspects: (1) grid-based graph generation, (2) grid-to-tilemap transformation (progressively increasing the grid's resolution) and (3) cellular automata to grow areas of the dungeon's naturalistic areas, e.g. caves and barriers (non-naturalistic parts such as rooms have fixed modification rules).

The terrain-generation is done separately from the tilemap and uses a kind of cellular automata with smoothing to generate terrains with respect to certain design patterns. The generated terrains are superimposed on the level tilemap to create the intended gameplay area in real-time. *Unexplored's terrain-generation is beyond the scope of this report.*

## Design requirements on the AI systems

**1. Two-directional playability**
The game's objective involves a two-direction traversal of every level. Hence, many of the lock-and-key puzzles need to be solvable in two directions (Dormans, 2017). While simple solutions exist (e.g. levers on both sides), a design objective is to use the greatest variety of lock-and-key puzzles possible (Dormans, 2017).

**2. Multi-stage generation**
The game-design must be integrated (i.e. not feel randomly put together) in many respects — i.e. in terms of navigation, progression and aesthetic. Each aspect has its own requirements, constraints and standards, and hence, must be handled by a separate AI system. However, each aspect is tied to the other; for example, navigation decides the layout, to which progression is tied, according to which aesthetic choices can be made. Hence, an integrated multi-stage approach to the dungeon level design is required.

## Other constraints & challenges

Although graph grammar follows a limited set of rules, the recursive application over many iterations makes its results hard to control for bugs. Some bugs that can happen are unreachable cells (i.e. parts of the map that a player cannot access) and non-backtracking paths (i.e. paths that do not let the player backtrack, thus preventing progression) (Dormans, 2016).

*Bug fixing…*
While some issues could be fixed (e.g. by adding a few conditions for barrier placements), the main solution is the "pray for help" feature which uses the level's graph representation to check (using pathfinding) whether the player is unable to traverse to neighbouring nodes (if yes, the game creates a path for the player). Running such a test for every player-position in-game is wasteful (especially due to the rarity of such a case), hence, since bug-proofing for transformational grammars is not yet foolproof, the "pray for help" feature helps solve for bugs more cost-effectively (Dormans, 2016).

# System design

## Controlling the graph generation process

Grammar-based generation can add complexity to the generated output with every step by modifying or expanding upon the previously generated output, which can make it hard to control, i.e. hard to keep it within abstract constraints such as coherence, relevance to the task and an upper bound to the level of complexity. To this end, the game applies a few methods.

### 1. Defining a general instruction sequence

To achieve an overall coherence in the output, “recipes” are used, i.e. a sequence of instructions about which rules to apply (Dormans, 2016). For example:

| **Grammar production rules** | **Recipe** |
|---|---|
| **start**: dungeon → rooms + goal<br>**goal**: goal → dragon + amulet<br>**grow**: rooms → rooms + room<br>**grow**: rooms → room + enemy<br>**entrance**: room → entrance<br>**enemy**: enemy → orc \| goblin \| rat … | SetStartSymbol(dungeon)<br>IterateRule(start)<br>IterateRule(goal)<br>ExecuteRule(grow)<br>IterateRule(entrance)<br>ExecuteRule(enemy) |

(Modified from Dormans, 2016)

Each name in bold is a non-terminal symbol associated with one or more production rules (if more than one, then any one applies, often picked at random). Each method used in the recipe specifies an instruction or a set of instructions associated with specific non-terminal symbols.

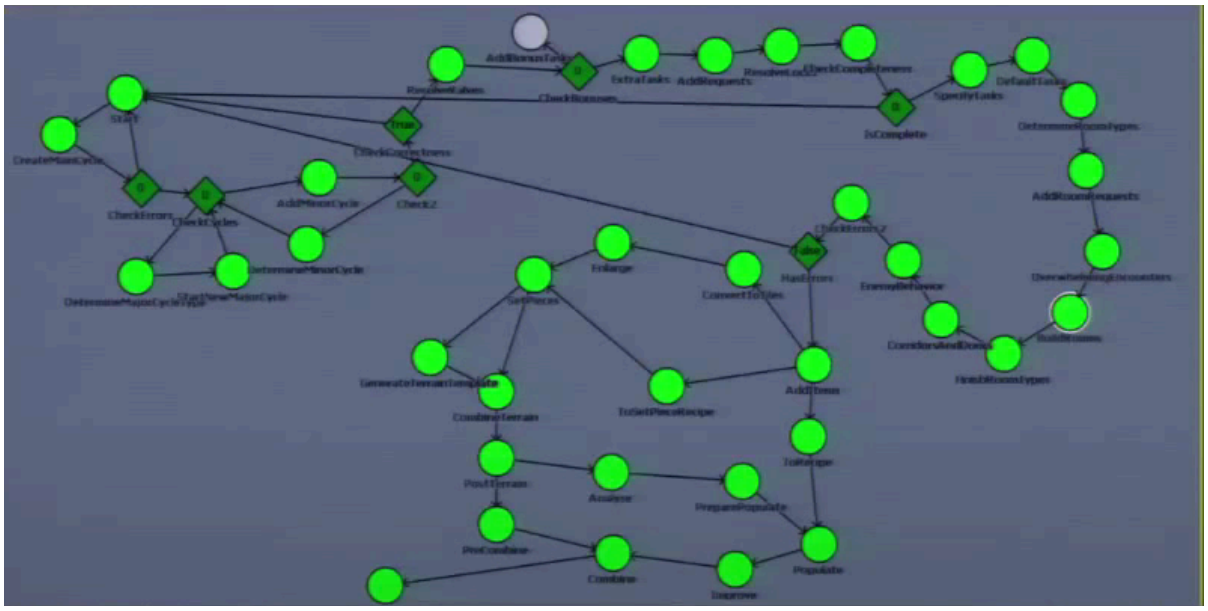
This contrasts with a classic generative grammar, wherein a start state is given from which rules are applied for a number of steps or until no rule applies. Note that if multiple rules can apply for a symbol, there are ways of choosing which to apply in the given iteration (the most common being random selection).

## 2. Specialisation of grammars per generative step

Instead of using one grammar for all tasks, specific grammars are used for specific tasks or task types. In Unexplored, each level's graph generation takes 44 steps (Dormans, 2016), each step representing a specific task, e.g. assigning node types (e.g. rooms, caves, shrines, etc.), creating pathways between nodes, adding abstract gameplay features (e.g. obstacles, valves, bonuses, etc.), creating lock-and-key relationships, etc.

## 3. In-generation constraint satisfaction checks

To ensure that key constraints are fulfilled *during* graph generation, checks are inserted between sets of tasks; if the respective grammars fail some constraints, then the tasks up to the check are redone by the respective grammars. A low-resolution screenshot of the graph generation process is shown, wherein the circles represent tasks (green for completed, grey for uncompleted) and the diamonds represent checks.



(Dormans, 2016)

## 4. Apply transformational grammar, not generative grammar

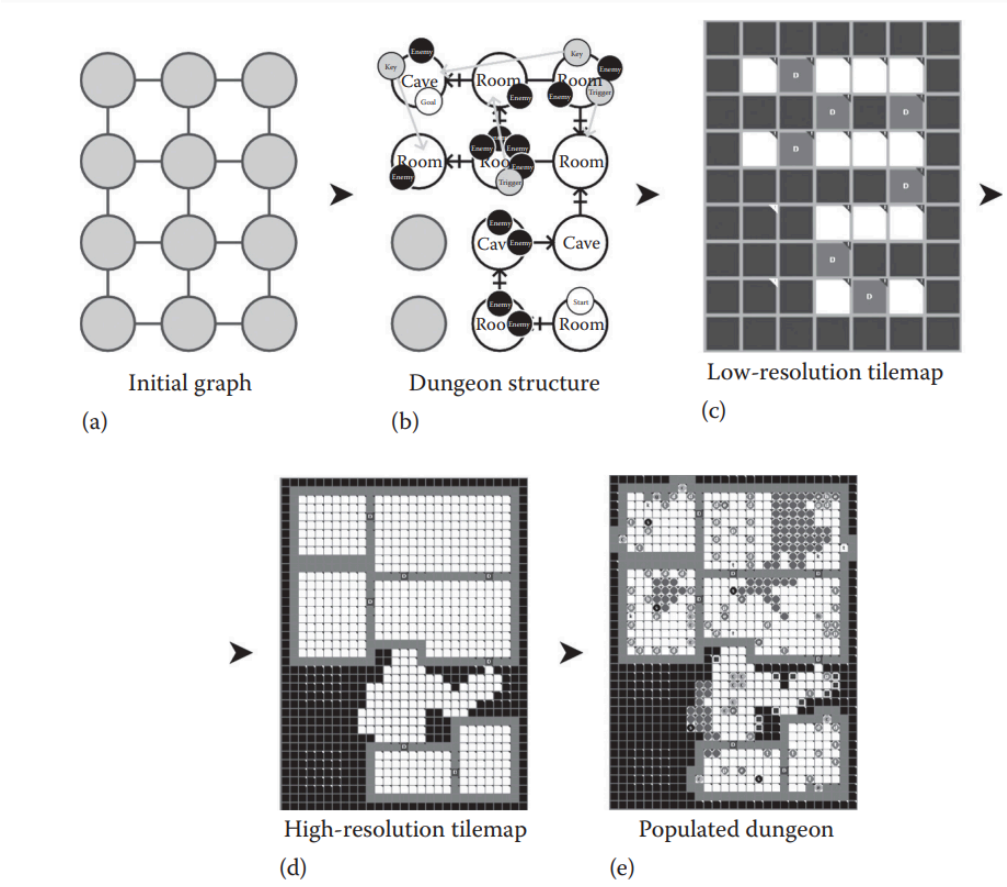
As a further way to control the graph generation process, rather than use generative grammars, which take an initial state and keep applying production rules up to

some point, Unexplored uses transformational grammars, which use recipes to transform the graph within a generalised but well-defined process. Each succeeding step of the graph generation takes the input from one or more of the previous steps and applies its own transformational grammar before passing its result to the next step (Dormans, 2016).

## Stages of dungeon generation

(*Stage ⇒ Degree of abstraction from the dungeon as experienced by players*)

To achieve an abstract structure while adding more fine-grained complexity, the generative process is multi-staged, starting with the abstract structure of the dungeon (represented using a graph) and ending with the populated dungeon. This helps retain key game-design objectives while also retaining the more concrete (i.e. experiential) variety, both aesthetic and functional. The diagram below shows the various stages of the generative process (from most abstract to most concrete).



(Dormans, 2017)

## Level graph generation

The level graph represents the logical structure (i.e. the connections between parts of the dungeon, the elements within each part and connection, etc.). The aim is cyclic generation, hence graphs are used as they represent cycles most efficiently (i.e. with the fewest irrelevant details). In particular, graph representation has the following advantages: (1) path distances and topography can be abstracted, and (2) simple, easy-to-design patterns can be used recursively to create complexity.

Note that in Unexplored, edges connect the various nodes (most importantly the start and endpoint) but the edges do not necessarily correspond to physical paths in the level. Instead, they represent the gameplay that is to occur between them irrespective of their distance physically (Thompson, 2021).

***Graph generation in Unexplored implements the following…***

**1. Limiting dungeon size**

Smaller but interconnected levels are more interesting to play in; hence, the graph is transformed by the grammar given a fixed number of nodes (e.g. 5×5) (Boris, 2021). The size of the graph is not expanded greatly (if at all), with most transformations either replacing existing interconnections with more complex ones between the same nodes or adding gameplay features to nodes and edges.

*Increasing level size until a limit…*

Extra nodes can be added to complicate the dungeon further, and short detours from the main cycle can also be added (often including more keys and obstacles). There are also rules for making cycles longer or adding dead ends. But such rules are run until the level has grown to the desired size (Boris, 2021).

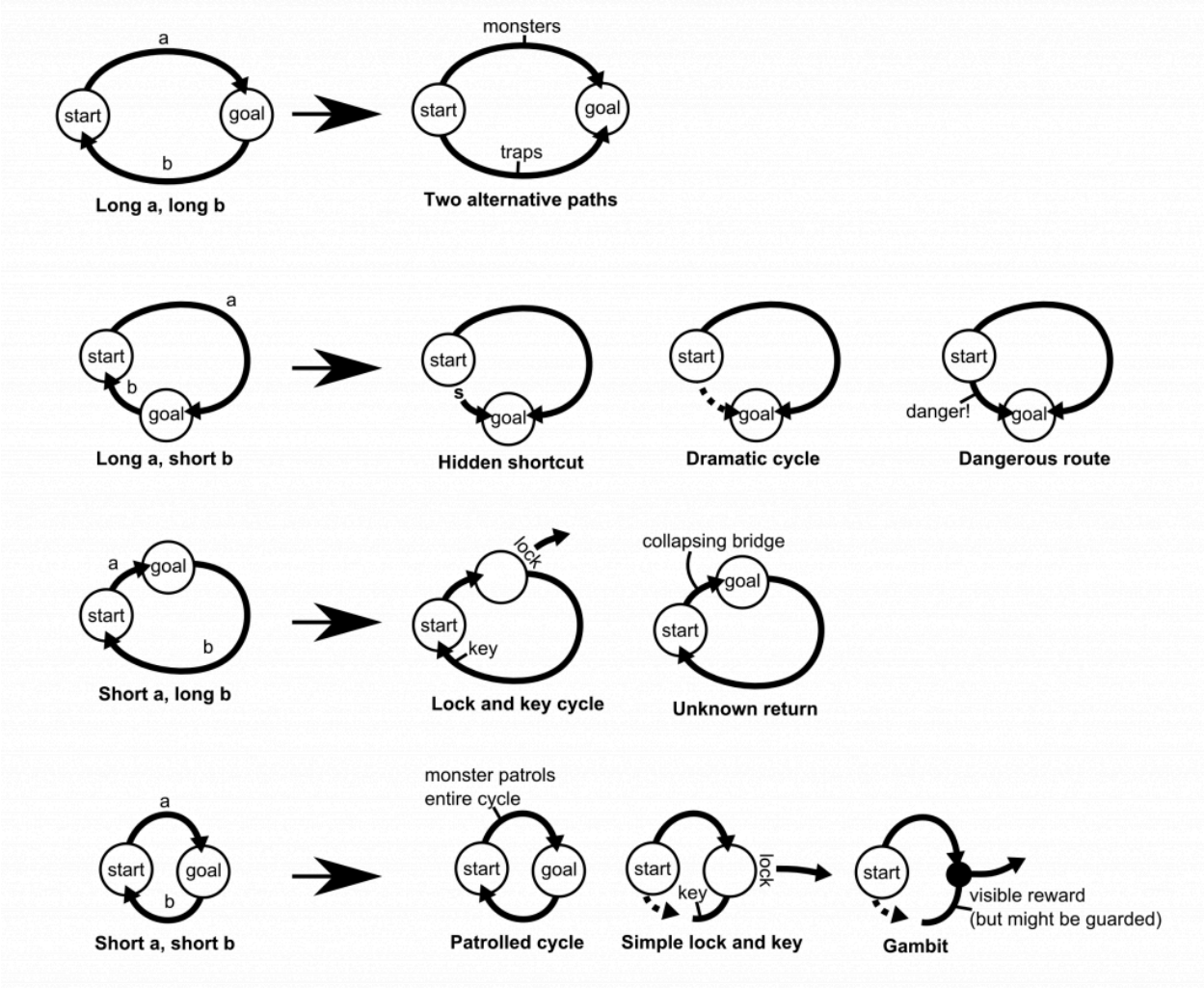
**2. Abstracting types of gameplay features**

There are a few broad game mechanics that may be realised in many forms. To implement each such mechanics while having and using a variety of particulars within each, the graph generation specifies abstract game mechanics that generalise

a variety of concrete game features. Some mechanics used in the dungeon are obstacles (which generalises progress-halting or dangerous encounters, e.g. enemy, trap or puzzle), lock-and-key (which generalises the features wherein a problem is to be solved by gaining and using items) and valves (which generalises a one-way passage between two areas, e.g. a one-way teleport or a one-way door).

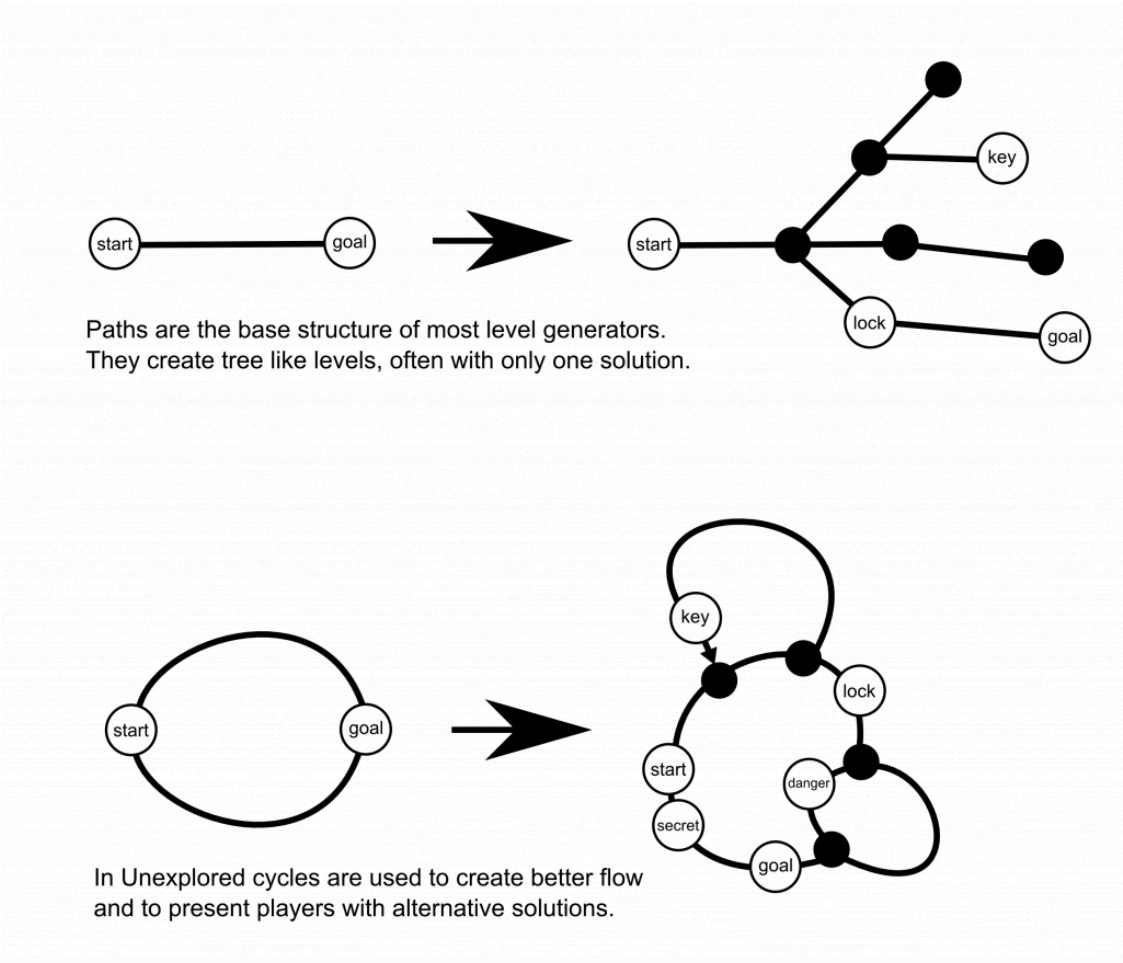
*Diversity in lock-and-key mechanisms…*
In-game, while a lock-and-key may be an actual lock-and-key, it can also be other kinds of problems, e.g. lever switches (key), time-sensitive pressure plates (key), a potion (key) with temporary resistance to lava (lock).

**3. Cyclic generation**
Cyclic generation (CG) is the generation of graphs with cycles as a key constraint of the process (i.e. each node exists as a part of a cycle). Each level has an overarching cycle that follows one of several level design patterns. Examples are given below.




(Dormans, 2024)

In Unexplored, CG is done using transformational grammars that add complexity (e.g. handcrafted patterns to replace more generic ones) or novelty (e.g. encounters) to existing graphs. While each level involves at least one cycle between the start and goal, using transformation rules, cycles can be nested within existing cycles, thus creating further the layout's complexity without adding new nodes. Such nested CG is a means to create complex, interconnected networks that translate into complex, interconnected level design (like handcrafted levels and unlike regular PCG levels). A comparison between branching and nested cycles is given below.




(Dormans, 2024)

**4. Attaching attributes to nodes and edges to condition transformation rules**
Attaching attributes to nodes and edges in the graph and allowing the graph's transformation rules to read and write the attributes is a way to activate different transformation rules according to the type of area (e.g. different design rules can be applied based on the room type). This adds variety to the map design while making sure the grammar does not implement incompatible design choices.

A key application of this method is using attributes to mark the functions of the locks-and-keys used, which is a way to avoid deadlocks and create variety in lock-and-keys at the same time (Dormans, 2017). For example, a "conditional" attribute for a lock means it opens to only one key, whereas a "non-conditional" attribute means it can be traversed in any direction but would need certain actions

to remove certain traversal hazards (e.g. torches to drive away ice-bats, poison gas to clear a room packed with spiders, etc.).

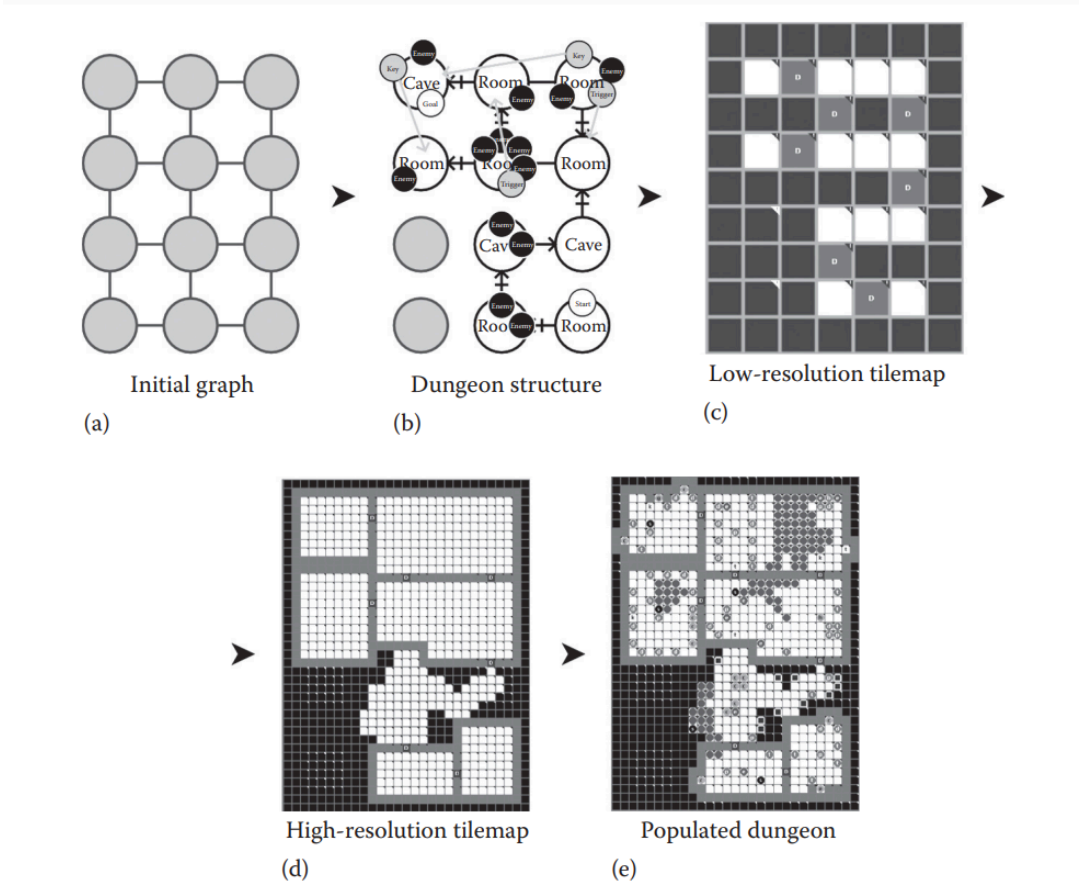
## Conversion of level graph to tilemap

While graphs represent the abstract structure of the dungeon efficiently, the end result as experienced by the player can be achieved most efficiently (i.e. with the least computational complexity) using a tilemap, i.e. a map divided into equally-sized tiles wherein features can be added. Furthermore, the graphs alone — being abstract — are interpretable in various ways (Thompson (YouTube), 2021) (e.g. translatable to various topologies, mechanics, appearance, etc.). To this end, tilemaps have the following uses:

- Decorations or extra details can be easily added (e.g. vegetation, pottery, etc.)
- Another tile graph can be superimposed to add wider features (e.g. terrain)
- Noise generator can handle randomised features (e.g. wind from chasms)[1]

(Dormans, 2016)

[1] This feature is not yet implemented, but shows the extensibility of the tilemap representation in terms of adding interesting gameplay features.




(Dormans, 2017)

*General process…*

Unexplored ensures the level graph (b) is generated in a grid shape given in (a) such that the graph maps to a two-dimensional space. The graph is then translated over multiple passes, starting from a low-resolution tilemap (c) to a high-resolution tilemap (d). The engine then applies the gameplay-specific information stored in the level graph (b) (e.g. corridor monsters, room-specific traps, locks-and-keys for the main puzzles, etc.) to reach (e) (Thompson (YouTube), 2021).

*More on increasing tilemap resolution…*

The low-resolution grid is expanded by a constant factor to give the actual grid of the map. Each block (associated to a node in the level graph) has special rules applied to shape it; doors are shrunk to one tile, rooms grow into larger rectangles with boundary walls, and most other areas (e.g. barriers, caves, tunnels etc.) have small cellular automata to give them rougher shapes. Some more elaborate patterns

are also applied, such as narrow bridges or tauntingly out-of-reach rewards (Boris, 2021).

# Industry context

## One-way progression with branches

Platformers using PCG for level-generation involve a one-way progression with branches sprouting from the main path (e.g. Rogue Singularity), as opposed to the two-directional cyclic paths generated in Unexplored.

*Level example in Rogue Singularity…*

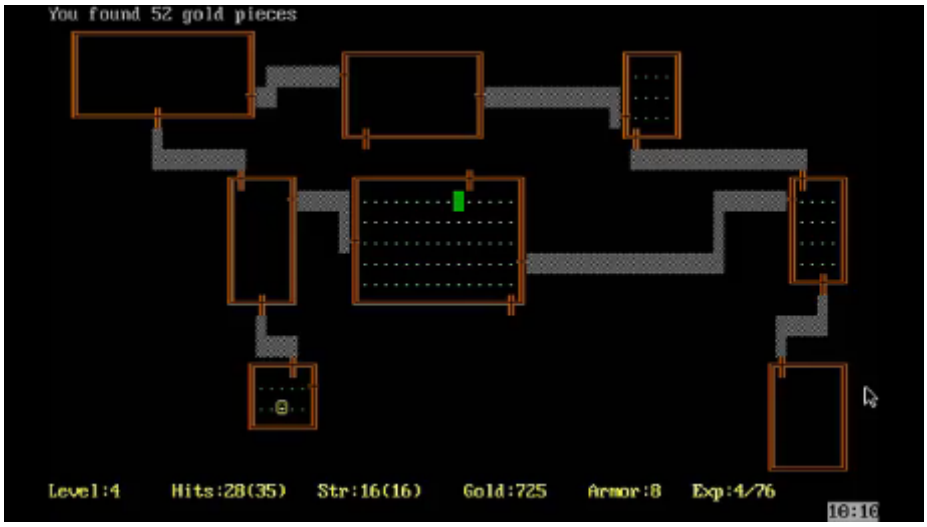
(Grubdog, 2019)

## Concrete-bound generative process

Most dungeon generation is done at the tilemap level, working on a representation closer to the playable level. However, such a concrete-bound approach makes larger-scale interconnectivity harder to implement. Instead, we see rules for placement of game-elements (e.g. rooms), for connecting the neighbourhood of each node or grid-cell and for constraint satisfaction checks (e.g. for playability and level size). Such an approach is seen in popular platformers like Spelunky or dungeon crawlers like Rogue (1980) and Binding of Isaac. In Rogue, for example, we see a dungeon crawler that uses a process of first placing rooms at random, then connecting neighbouring rooms with passages according to a set of rules selected at random. However, only neighbouring rooms are considered when creating

passages with no consideration given to aspects of the wider layout (Maizure's Projects, 2024). Such concrete-bound methods do not abstract the topography of the level when generating it, whereas such abstraction is indispensable for working with cycles in a map (as done in Unexplored).

*Screenshot from Rogue (1980)*




(IGDB, 2024)

## Ad hoc naturalistic dungeon layouts

Many games with caves or dungeons focus more on terrain-generation, which is often done using gradient noise (especially Perlin noise) — at least in part — as in popular games such as Minecraft (Minecraft Wiki, 2024) and Terraria. Having set the terrain, agent-based PCG methods are used to carve the terrain in a pseudorandom, naturalistic way, thereby leading to the cave-generation seen in these games. By contrast, Unexplored focuses first not on terrain-generation but level-design, only using cellular automata at the end to expand on the level's tilemap's shapes to make its dungeons more naturalistic.

*Terraria cave-generation example*

(CTG, 2024)

# Conclusion

Hence, we observe in Unexplored that a key to human-like design is abstract (high-level) design that applies principles grounded in human experience (e.g. cyclic layouts). Furthermore, abstract design enables the use of generalisations (e.g. lock-and-key, obstacle, etc.) that translate into a range of possible results when concretised.

With abstraction comes the need for a hierarchical generative process to help concretise the abstract design, as we see with the multi-stage level-generation. Each stage retraces the abstraction done from observations (e.g. observed cyclic environments in cities, parks, hand-drawn games, etc.) and brings the result closer to a unique playable level recreated from human design principles.

Grammar-based generation — which by itself can be hard to control — was guided (1) using recipes (instructions of how to apply grammar rules), (2) applying periodic constraint satisfaction checks across the graph generation process, (3) using transformational grammar rather than generative grammars and (4) by attaching of attributes to the level graph's nodes and edges (to take into account the nature of the associated gameplay feature in later generative stages). Such measures ensure overall coherence while encouraging diversity in outcomes.

The creator of Unexplored (Dr. Joris Dormans) wanted to use cycles to simulate (using PCG) the experience of playing on a handcrafted dungeon map, i.e. playing on a reasonably complex yet coherent layout with a more organic (i.e. intuitive or natural) exploration experience. To this end, Unexplored was successful both as a proof-of-concept and an experience. My experience in playing the game showed me a well-connected level-design whose cycles led me to progress within the dungeons more organically (i.e. without excessive backtracking, complex branches easy to get lost in (as in Minecraft caves) or contrived fixes like teleports).

*Example of a generated level in Unexplored*

(Dormans, 2024)

*Personal in-game examples (only maps)...*



Compared to other games using PCG level-design or cave-generation, Unexplored stands out by making PCG feel less like PCG and closer to human-made dungeon design while also fulfilling complex abstract goals (e.g. nested cyclical layouts, two-directional playability with complex lock-and-key puzzles solvable from either direction, etc.).

# Citations

## Content citations

**In-text: Dormans, 2017**
Dormans, J. (2017). ‘Cyclic Generation’. In: Short & Adams (eds). Procedural Generation in Game Design, CRC. pp. 83-95.

**In-text: Dormans, 2016**
Dormans, J. (2016). EPC2016 - Joris Dormans - Cyclic Dungeon Generation. [online] Available from: https://www.youtube.com/watch?v=mA6PacEZX9M [Accessed 17 March 2024].

**In-text: Boris, 2021**
Boris (2021). Dungeon Generation in Unexplored. [online] Available from: https://www.boristhebrave.com/2021/04/10/dungeon-generation-in-unexplored [Accessed 17 March 2024].

**In-text: Thompson, 2021**
Thompson, T. (2021). Unexplored’s Secret: ‘Cyclic Dungeon Generation’, [online] Available from: https://www.gamedeveloper.com/design/unexplored-s-secret-cyclic-dungeon-generation- [Accessed 18 March 2024].

**In-text: Maizure’s Projects, 2024**
MaiZure’s Projects, Decoded: Rogue, [online] Available from: https://www.maizure.org/projects/decoded-rogue/index.html [Accessed 18 March 2024].

**In-text: Minecraft Wiki, 2024**
Minecraft Wiki. Carvers. World Generation. [online] Available from: https://minecraft.wiki/w/World_generation [Accessed 19 March 2024].

**In-text: Boris, 2020**

Boris (2020). Dungeon Generation in Binding of Isaac. [online] Available from: https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/ [Accessed 19 March 2024].

**In-text: Thompson (YouTube), 2021**
Thompson, T. (2021). The Secret Behind Unexplored: Cyclic Dungeon Generation | AI and Games #57. [online] Available from: https://youtu.be/LRp9vLk7amg?si=isOXdduEuFI2TWlk [Accessed 20 March 2024].

## Image citations

**In-text: IGDB, 2024**
IGDB (2024). Rogue (1980). https://www.igdb.com/games/rogue [Accessed 20 March 2024].

**In-text: Dormans, 2024**
Dormans, J. A Handcrafted Feel: 'Unexplored' Explores Cyclic Dungeon Generation. [online] Available from: https://ctrl500.com/game-design/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/ [Accessed 20 March 2024].

**In-text: Grubdog, 2019**
Grubdog (2019). *Rogue Singularity – Infinite Possibilities*. [online] Available from: https://pietriots.com/2019/08/25/rogue-singularity-infinite-possibilities/ [Accessed 21 March 2024].

**In-text: CTG, 2024**
CARL'S TERRARIA GUIDE. Exploring and Mining the Underground. Terraria Walkthrough. [online] Available from: https://www.carlsguides.com/terraria/walkthrough/exploring-underground-caverns.php [Accessed 20 March 2024].