UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Adaptation of Cyclic Level Generation to 3D Environments

Ethan Martín Parra

Director: Salvador Bolarín

Grau: Videojocs (Barcelona)

Curs: 2024 - 2025

Universitat: UPC - CITM

Contents

Abstract

Procedural level generators have advanced immensely throughout the years, but few of the resulting levels make it to the point of seeming designed by hand, by a designer. One of these games is “*Unexplored*”, making use of what has been coined as “Cyclic Dungeon Generation".

In this project this kind of generation will be studied in order to adapt it to 3D environments and hopefully give visibility to more complex kinds of generation, and the algorithm will be integrated in an already existing ongoing multiplayer game which is under development.

On top of the algorithm, a new pipeline has been designed and implemented to produce more level patterns, possible combinations and new rules for the generator, to be able to expand the possibilities of the generator without the need of modifying the code.

Resumen

Los generadores procedurales de niveles han avanzado muchísimo con el paso del tiempo, pero pocos de los niveles resultantes de estos llegan a parecer diseñados a mano, por un diseñador. Uno de esos juegos es “*Unexplored*”, usando lo que ha sido acuñado como “Generación Cíclica de Mazmorras”.

En este proyecto este tipo de generación será estudiada por tal de adaptarla a entornos 3D y con suerte dar visibilidad a tipos de generación más complejos, además de integrar el algoritmo en el desarrollo en curso de un juego multijugador ya existente.

Un nuevo pipeline se ha diseñado e implementado sobre el algoritmo para producir más patrones de niveles, combinaciones posibles y nuevas reglas para el generador, por tal de ser capaz de expandir las posibilidades del generador sin necesidad de modificar el código.

Key Words

Procedural generation, Level Design, Cyclic generation, Generative grammar, Graphs

Links

Trailer:
https://www.youtube.com/watch?v=ptIoQd7Fh2g

GitHub Repository:
https://github.com/xGauss05/TFG_Project

GitHub Release:
https://github.com/xGauss05/TFG_Project/releases/tag/v.1.0

List of tables

List of figures

Glossary

Asset: Any digital resource that might be used to create the game. This could include 3D models, 2D images, pieces of code, music, animations and others.

Blockout: Overall and rough representation of the level shape, using only gray boxes or simple shapes, but not final assets and without adding any detail.

Cellular automaton: A grid-based system where each cell changes state based on the states of its neighboring cells. It is often used to simulate complex patterns and behaviors, or the evolution of a discrete system.

Grind/Farm: To spend large amounts of time repeating a task to achieve a certain goal or obtain in-game rewards such as game currency, resources or others.

Noise: Mathematical function used to generate pseudo-random numbers. In the field of procedural generation it is commonly used to create natural looking textures, patterns, smooth terrain amongst many others.

Perma-death: You lose all your progress upon death. Once you restart, you generate again all dungeons, loot, player character, etc.

Procedural: Procedural means that something follows a procedure. This means a specific set of instructions. For example: a recipe is procedural, a croissant is not. As long as you have ingredients, you can create as many croissants as you want following the recipe, but with a croissant you can not create more croissants. If you have different recipes, you can create different pastries, if you have different croissants, you simply have different croissants.

Rogue-like: Game similar to "Rogue", the 1980's game for Unix-based systems. This kind of games are characterized by randomly generated dungeons/levels and perma-death.

Run: A game or attempt you do in a game featuring perma-death. The period from the moment you begin an adventure until the moment when you die.

Souls-like: Game similar to "Dark Souls", the 2011 game by FromSoftware published by Bandai Namco. Nowadays the term has broadened a lot, but these games are usually characterized by their high difficulty, slow paced and intentional combat, rolling to avoid enemy attacks and stamina management. On top of that, they tend to include many bosses, which would simply be a challenge at the end of a level or zone.

Third person shooter: A videogame genre in which you control a character you can see on screen as opposed to you seeing what the character sees or being their eyes, thus the concept “third person”. Moreover, the camera is usually behind the character and framing it. The game is also built around the main action you will be doing in the game, which is shoot.

1. Introduction

1.1 Motivation

In recent years the term “Procedural Generation” has been increasingly used to market all kinds of games. However, the quality of the algorithms behind this generation has not followed the rising trend. Since the concept is being used simply as another selling point and not taken that much into account, levels generated procedurally might make the game they are implemented in more replayable, but they do not feel new or fresh anymore. The algorithms and approaches to generate them are always tailored to the game, but follow similar patterns and are either too simple or yield empty results.

With this project I wanted to study what some of the best procedurally generated levels bring to the table in order to come up with an innovative solution which can generate interesting levels which are not only more complex than current trends but also more fun to play and explore. On top of that, doing the initial research on some of these topics I stumbled upon a specific method that seemed to be the proper next step in terms of level generation but which was only implemented in one game and its sequel, so I thought of trying to implement it in a game of my own to try and make this approach a bit more popular.

1.2 Problem formulation

"Procedurally Generated Levels" is a term that is slowly being frowned upon with the pass of time, especially due to it being used as a marketing strategy more and more. This is because most procedurally generated levels or worlds rely on the same or similar techniques and algorithms, which end up giving the same results. These results are usually characterized by either being too dull or too linear, mostly in the case of procedural open worlds and procedural closed levels and dungeons respectively.

This creates a very clear problem when it comes to user experience, and it affects replayability. If your world is massive and empty but you have to spend a lot of time navigating it, it can become uninspiring. On the other hand, if your level is full of rooms packed with detail but the rooms repeat after every other door, it can become tedious. What saves a lot of games from falling under this categories is that the main focus of the game is not traversing the levels, but what happens in those levels and all other mechanics that come into play during gameplay, so the level becomes just a mere stage where the rest of the game happens, but this does not mean that their levels are not empty, repetitive or linear.

1.3 General objectives of the project

The objective of this project is to create an algorithm which will address the problems typical level generators present through the implementation and expansion of a fairly recent approach known as "Cyclic Dungeon Generation", developed by Dr. Joris Dormans and his team from "Ludomotion". This system has only been used commercially once in "Unexplored", and later expanded in "Unexplored 2", both games from "Ludomotion". The main goal is to give the system a twist expanding it by applying it onto a 3D level.

This algorithm will then be added to an ongoing project in order to make the first playable prototype of a videogame which could be categorized as a third person extraction shooter.

1.4 Specific objectives of the project

- Study different approaches from a range of videogames no matter their age to have resources to be able to implement them in a new algorithm if needed.
- Understand Cyclic Generation, a very unique method designed to generate organic levels to be able to expand upon it.
- Implement a Cyclic Generation system to replicate the original in the resulting simulations.
- Expand said system to adapt it to 3D scenarios and environments.
- Integrate the new system into a functioning prototype of a 3D shooter videogame.
- Test and validate whether the results have achieved the original expectations through user testing and their feedback.

1.5 Scope of the project

The aim of this project is to have a functioning and playable prototype of a game implementing the cyclic generation approach by the end of its execution. The target objective would be to have a complete algorithm using a certain amount of assets which could be final in order to get an idea of how generated levels might look. Given that this is a technically focused project, any kind of asset creation will not be part of the project. The minimum viable product would be to have a "blockout" type scene showcasing the algorithm, since open or free to use assets that fit the project might be difficult to find. The ideal case however, would be to have a full working demo of the game using asset store assets.

The contents of this document and the prototype that comes with it are mainly addressed to any developers who are willing to venture off the beaten path and try new ways of creating immersive procedural worlds to make them seem more believable and hand-crafted every time. With new implementations such as the one proposed players could feel more immersed in levels featuring procedural generation and get to experience more content without it being at the expense of its quality. Both developers and players would benefit from new and different approaches to level generation driving the industry forward, keeping the medium in a state of constant evolution and making better games off of innovative ideas.

2. State of the art and theoretical framework

2.1. Market analysis

2.1.1 History

One of the first procedurally generated games there is is "Conway's Game Of Life" (1970). This game only featured the option to place a few pixels on screen and begin a simulation, which was based on a cellular automaton. It was technically not a game itself, but more an environment for simulation.

Later on in 1978 "Beneath Apple Manor" was launched. In "Beneath Apple Manor" you play as a hero whose goal is to acquire the Golden Apple from the bottom floor of a dungeon. The different levels of the dungeon are generated procedurally, thus making this one of the first games to feature procedural generation. It also introduced the concept of permadeath.

In 1980 however, "Rogue" was launched. This is the game that popularized generative content and gave name to the "rogue-like" genre. In this game, much like in "Beneath Apple Manor", players control a character who must descend into a dungeon through various procedurally generated caves in search of the "Amulet of Yendor", located in the dungeon's lowest level. In these caves, the player must defeat monsters, improve their equipment and gain loot. The game was rendered in ASCII and it was turn-based.

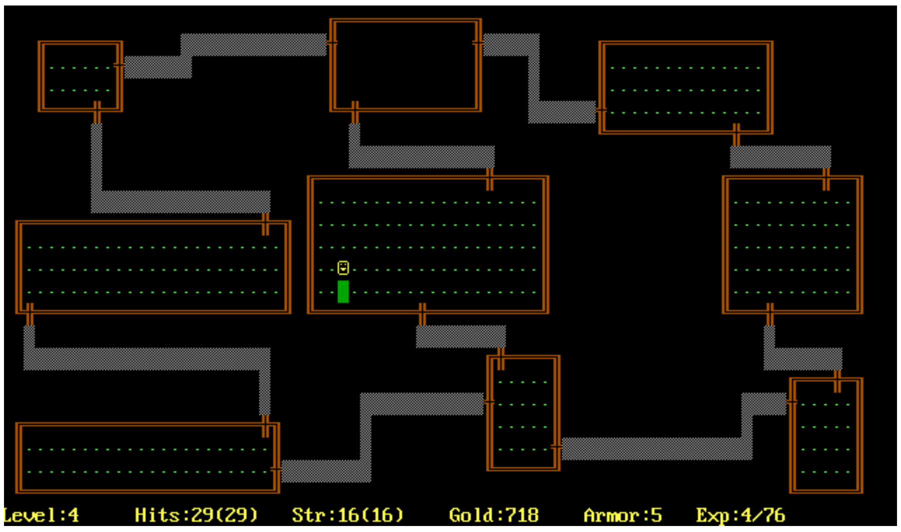


Figure 1: A screenshot of “Rogue”, showing its dungeon generation.

Even though some attempted to implement procedural generation after “Rogue”, most of them were not really successful. However, this changed when “Diablo” was launched in 1997. Diablo is an action RPG developed by Blizzard in which the player controls a hero battling through sixteen procedurally generated dungeons in order to rid the world from the evil "Diablo", the main antagonist of the game.

2.1.1 Examples of modern games

Over time, computers became more powerful and this improvement got reflected on games as an increase in performance, and the possibility to exploit this additional computing power. This meant that developers were freer to experiment with new ways of generating levels, or in this case full open worlds at the scale of hand-crafted ones or even more.
However, many procedural open worlds feature similar if not the same methods of generation. Most rely on different kinds of noise such as Perlin noise or fBM (Fractional Brownian Motion), and they deform it or shape it from there to fit their needed approach. One way it might be deformed is to apply a different number of octaves to it to create “smoother” resulting patterns to use in more or less crisp terrain deformations, or domain warping to produce the illusion of ridges and furrows.

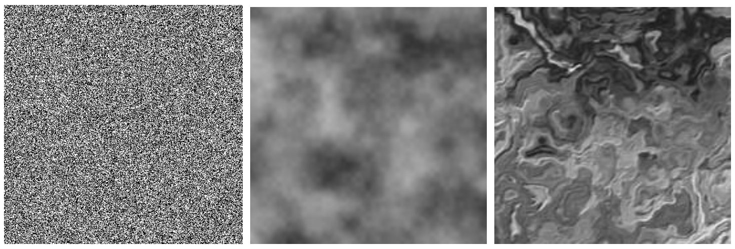


Figure 2: Different types of noise.
From left to right: Random Noise, Perlin Noise and an example of Domain Warping.

When talking about games with procedurally generated open worlds, one of the most popular ones is Minecraft. This is a sandbox survival game where you place blocks to create whatever your imagination can come up with, but the world is an immense playground for you to explore. Minecraft, as many others, combines multiple layers of noise in different passes to know where to place each block and what block to place, generating different biomes, structures and differences in terrain along the way.

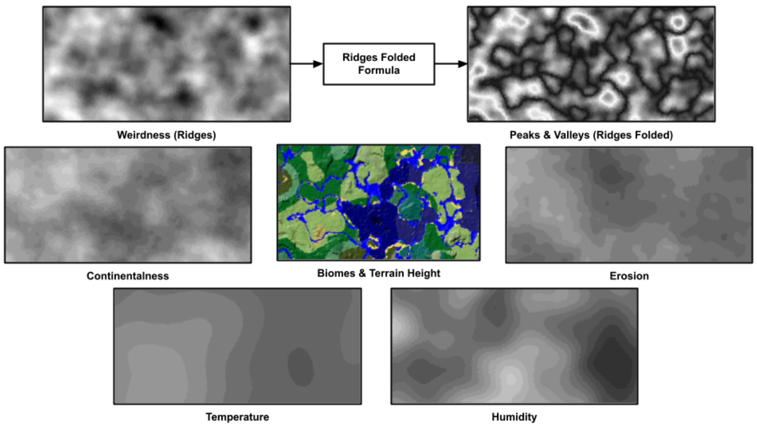


Figure 3: Different types of noise Minecraft combines to generate terrain.

However, there is one game that brought procedural generation to a whole new scale. No Man's Sky is a space exploration survival adventure game. In the game you set out to unravel the mysteries of the universe in your path to the center of the galaxy, a path in which you have the potential to find 18 quintillion planets, each with their own ecosystem, flora, fauna, geological composition and more. To create such diversity, No Man’s Sky also uses a combination of different techniques, such as Domain Warping, but also Slope Erosion, Altitude Erosion or the Analytical Derivative of the noise maps among others.

Figures 4 and 5: Examples of No Man's Sky Generation

This section's purpose is to give some context and show how far we have come in terms of generating our worlds and the scenarios we play in games, however, this project's objectives are closer to what might be considered "closed levels", where the generation process usually involves a combination of hand-made content and procedural content. In a lot of cases pre-made rooms will be arranged procedurally in order to create new layouts every time to have more control over how the levels are generated but still offer the player a fresh experience with each game session.
The purpose of the following sections is to analyze in-depth how some non-open world games handle procedural generation for their maps or levels in order to get a better understanding of what approaches are being taken, and to find similarities and differences to improve on them.

Spelunky

Spelunky is a 2D platformer videogame where you embody a spelunker who goes on an underground adventure in search of treasure through caves filled with enemies, traps and many other dangers. Although the core of the game is the platforming and the interactions with the environment, it also follows the structure of a rogue-like.

It was one of the first games to implement this kind of 2D side-scrolling platforming with procedurally generated environments. The game became very popular due to its clever level design, which makes it highly replayable, and its fast-paced platforming combined with fully destructible environments allowed for very satisfying emergent gameplay.

The game's levels are composed of a 4x4 room grid, where each room type will be determined as the algorithm advances. There are four basic types of rooms that can be placed, which are categorized by their shape. Type 1 is a horizontally arranged room with openings on the sides, type 2 is a T-shaped room with three openings, two on the sides and one on top, type 3 is an inverse T shape, with openings on the side and on the bottom, and the special type 0 are miscellaneous rooms which might have no openings, forcing the player to break through the walls in order to access it. Each of these types of rooms have eight to sixteen possible template layouts the algorithm can choose from at random whenever a new room is placed.

To create a level, the generator goes through the following steps:
First, a type 1 room is placed in a random spot in the first row by default. For every following room, a number from 1 through 5 will be chosen randomly. If the number is a 1 or 2, the next room will be placed to the left of the current one, and it will be of type 1. If it is a 3 or 4, it will go on the right of it, resulting in a type 1 too. If it is a 5 however, it will be placed directly below, and it will be either a type 2 or 3, making the previous type 1 room into a type 2 so it can have a bottom opening and carving one in the current one in case it is needed (type 2). On top of that, there are a couple other rules that must be followed. If the algorithm reaches the edge of the grid, it will automatically place a type 2 room to go down and will switch direction, and if we randomly get a number 5 to go downwards while being on the bottom line on the grid, an exit will be placed in the room. Once the main path from entrance to exit has been completed, the remaining empty rooms are filled with type 0 rooms. A third rule is applied in case there are three or four type 0 rooms arranged vertically, where the rooms will be replaced with a special type 7, 8 and 9 rooms forming a “snake pit”, a big hole with snakes and treasure at the bottom.

Every room template has what is known as “probabilistic tiles” and “obstacle blocks”. Probabilistic tiles are spaces where in this step of the process can be either filled with a block or left empty to increase the variability of the layouts. To further push this, “obstacle blocks” are 5x3 predefined groups of tiles which act exactly like room templates. They can either be placed or not in this step, and in case they are, additional static tiles will be placed in the room and the probabilistic tiles in the obstacle will be defined.

When the level layout process has finished, every free tile is evaluated to place monsters, objects and treasure randomly. Despite certain objects like gems or crates having a higher chance of spawning in tiles surrounded by walls and enemies usually spawning in more open spaces, the layout of the elements is not completely random though, since the position of these is taken into account to make a weighted distribution of the objects throughout the level.

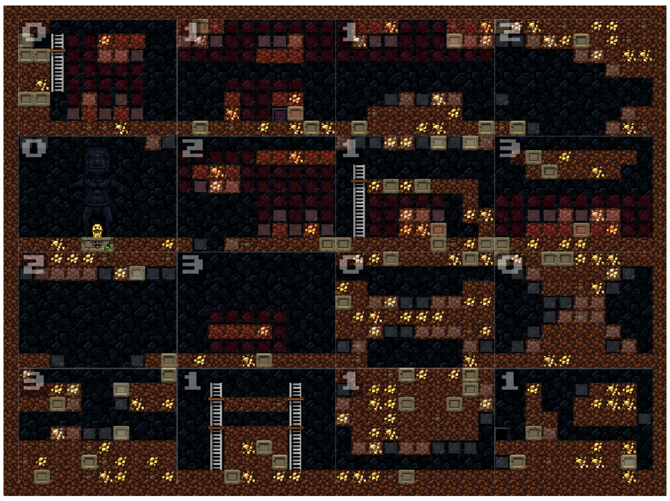


Figure 6: Main Spelunky layout. White tiles are probabilistic and red tiles are obstacle blocks

The Binding of Isaac

The Binding of Isaac is a rogue-like dungeon crawler with action-adventure elements. You play as Isaac, a toddler who must escape into his basement and fight to survive after his mom receives a message from God instructing her to kill him. In the game you'll find yourself going through levels of the basement, descending deeper every time into dungeons and darker places to finally confront your mother.

The game is known for the amount of different items and thus builds and combinations you can end up making throughout the different runs, not particularly for its level generation. However, levels feel fresh and understanding how the layouts work becomes intuitive easily.

Even though its level generation might be the simplest on the list given that the first version of the game was developed in under three months, the re-make "The Binding of Isaac: Rebirth" places second in the searches for rogue-likes on *Steam* at the time of writing, so it is not to be underestimated. The Binding of Isaac's maps are basically composed of squares in a grid. The algorithm generates the rooms layout and then picks the room interiors from a selected pool of different hand-crafted rooms. In order to generate the floor plan, first the starting room is placed in the middle of a 9x8 grid, and the number of rooms is determined by the formula *random(2) + 5 + level * 2.6*. Then, it will place the starting cell in a queue and iterate over the que until it is over. For each room, four sets of instructions will be executed (one for each cardinal direction) and they are the following:

1. Determine the neighbor cell by adding +10/-10/+1/-1 to the currency cell.
2. If the neighbour cell is already occupied, give up
3. If the neighbour cell itself has more than one filled neighbour, give up.
4. If we already have enough rooms, give up
5. Random 50% chance, give up
6. Otherwise, mark the neighbor cell as having a room in it, and add it to the queue.

If a room can't determine any of the surrounding cells as a new cell, the former will be marked as a dead end and those can and will be used later. Special rooms have unique parameters to be selected after all rooms have been placed, like the secret room trying to fit in a place where there are at least three adjacent neighbors or the boss room, which always tries to be placed as far away from the entrance as possible. Once the layout has finished generating and special rooms have been selected, the rest of "normal" rooms will be filled with interiors designed by hand. The algorithm draws a fitting interior from a big room layout pool and the room is a valid option it will place it, creating a simple yet efficient network of square rooms on a grid that serve exactly their purpose. "Rebirth" develops this algorithm a bit further, allowing for a little more variety on the room sizes, having the traditional 1x1, but also 2x1, 2x1 and some L shapes, but the main algorithm is the same.

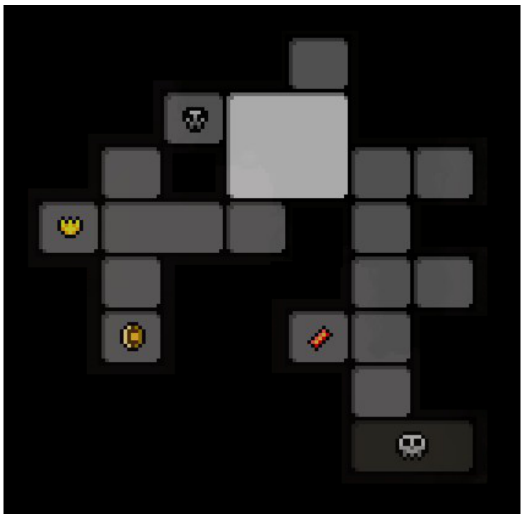


Figure 7: A Binding of Isaac example map

Warframe

Warframe is a third person shooter action-RPG set in the distant future in which you control a race named "Tenno" in biomechanical armors known as Warframes. In the game you will find yourself completing different types of missions across the "Origin System" (Solar system) while expanding your arsenal as well as upgrading your weapons and Warframes.

Even though procedural generation is not the main selling point of the game, every time you enter a mission the entire layout of the level is generated procedurally. This ensures that the amount of hours the players will spend grinding/farming do not become as repetitive and monotonous.

In Warframe the levels are generated using lots of different little pieces called "blocks" or "tiles" which are handcrafted. Each planet has a different "tileset" so all of the blocks of a certain mission will be selected from the same tileset to match the planet's team and faction. The levels are generated procedurally, arranging these tiles linearly first to ensure there is at least one viable path from start to exit and then branching off of some areas in the middle of the level. This generation follows a very specific set of rules. All tiles fall under the categories of *Start*, *Connector*, *Intermediate*, *Objective* and *Exit*, and levels adjust to a certain structure depending on the mission type. An example of structure using the initials of each type could be "SCICOCICE". To connect two rooms together, the first rule is that the "doors" connecting two rooms must be the same size. A 5x3 door from a room must connect to a 5x3 door on the next tile, but can not connect to a 9x3 room, for example.

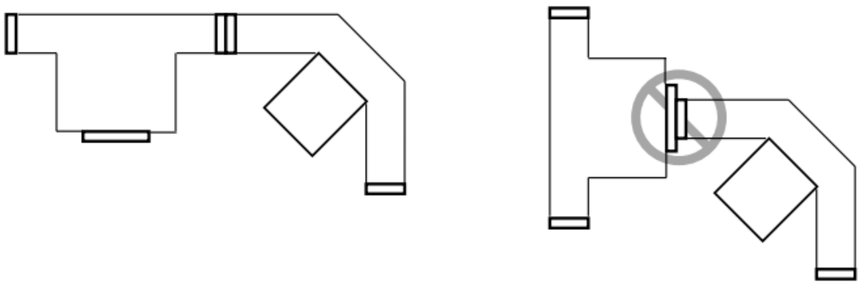


Figure 8: Warframe door placement

After checking if the doors match, the algorithm will check for overlapping blocks, to ensure no tiles get in the way of others. If two rooms collide, the process will backtrack and try to place a different room from the pool in the same spot to see if it is a valid match for the position.

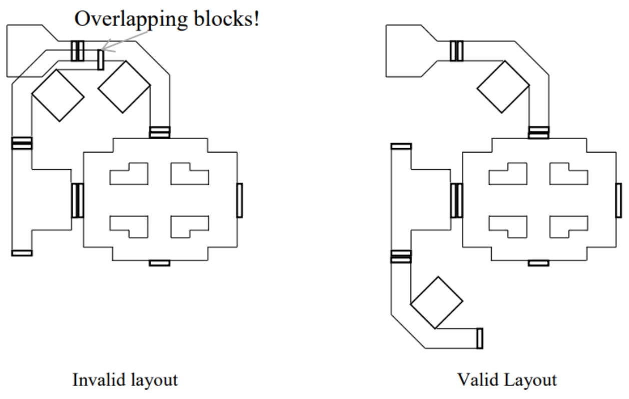


Figure 9: Warframe overlapping validation

When the main structure is complete, the algorithm will check for any remaining doors which have not had any other tiles attached to them. If that is the case and a "depth" parameter has not been maxed out, it will try to attach new blocks to that door and proceed from there to create branching in the level and in general make a more interesting layout. When the entire process is finished, remaining free doors will be capped to ensure there are no free unclosed spaces and the level is complete.

Remnant: From the Ashes

Remnant: From the Ashes is a lovecraftian action survival game which could be considered a Souls-like. Despite this, the main combat is ranged, with a style closer to a shooter. In the game you are a survivor of Earth's invasion by an entity known as "The Root", and embark on a journey following the steps of a previous survivor who was close to finding a way to defeat "The Root", but disappeared.

Even though the game mentions it slightly but does not market having procedural content, this is never mentioned in the game, and if you didn't pay attention to the game's public pages you would never know its worlds are procedural. It is amazing how well the different parts of the level are stitched together and how it all blends in with the main storyline.

Remnant mixes procedurally generated content with hand-crafted narrative and events in the most proficient manner, making procedural content and scripted one merge seamlessly. The game consists of four worlds with unique aesthetics, and these worlds will have a different tileset each, with the tiles usually being big chunks of the map. Even though not much information is available, it would seem that the layouts are built around specific checkpoints which determine what tiles with which shape can go where. Once the checkpoints are in place, random nodes are connected to the checkpoints and then the optional dungeon entrances are placed.

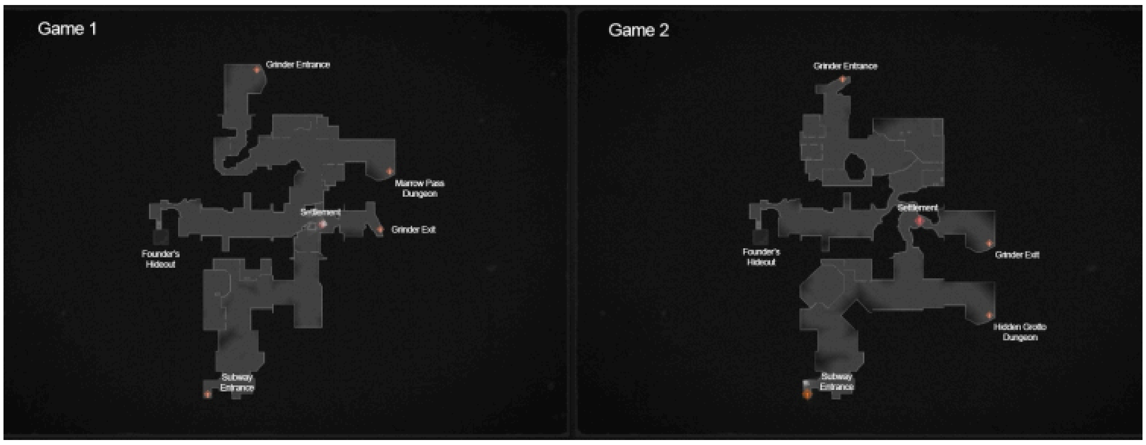


Figure 10: Example of Remnant's Generation

In this image we can clearly see how even though the main points of interest are the same and located similarly, the paths to get to them change completely, since the center checkpoint is not the same tile type. As stated before, the seamless blending of the tiles is remarkable, and the aim of this project would be to get close if not imitate the way in which tiles blend together to achieve the same sensation of unity and cohesiveness for the level.

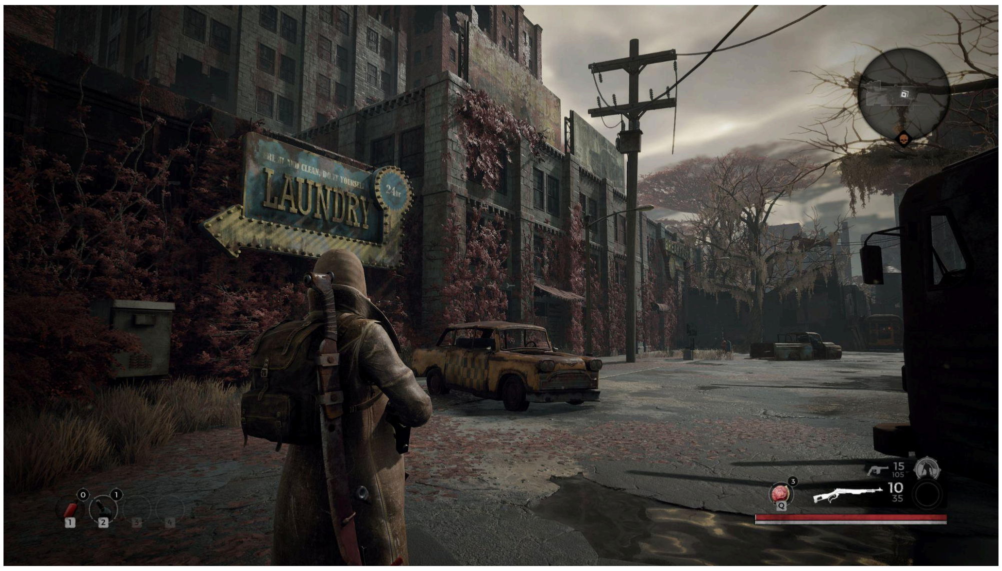


Figure 11: Tile separation is almost imperceptible

In conclusion, it becomes clear that there are as many approaches to world generation as games that feature it. However, the results that most common generators yield are often not that satisfactory, especially from the point of view of gameplay. As it has been described, open levels tend to feel empty or seem just like eye candy, while closed levels usually end up having tree structures with optional branching paths, in the end forcing the player to backtrack or retrace through certain parts of the level which don't feel as interesting after traversing them for the second or third time. To solve this, some games are opting for new and original ways of structuring their levels differently, and others are resorting to older hand-crafted levels considered pillars of game design to seek inspiration for their generators. These two solutions can be found in the case of *"Unexplored"*.

2.2. Unexplored and Cyclic Dungeon Generation

Unexplored is a rogue-lite action RPG which features real time dungeon crawling and melee based combat, although its creators also like to think of it as kind of an exploration game. In it you will embody an unnamed character going down into the Dungeons of Doom to retrieve the Amulet of Yendor and then come back out, just like in "Rogue".

If this game is separated from the others and has its own section, it is because of the revolutionary way in which it generates levels. Unexplored features what its developers called "Cyclic Dungeon Generation", a way of generating levels that is based on cycles, instead of the usual branching tree structure featured in most games.

The inception of this concept dates back to 2016, when Dr. Joris Dormans attended the Banff Workshop in Canada, and together with a group of people tried to come up with a

generator that could create parks that were interesting to walk in. Looking at urban planning and hypertext theories they found this idea of working with cycles which stuck with him. The next few days paying attention to his surroundings he also noticed his three year old child running in circles around a picnic table during a family field trip, and started looking for Dungeons and Dragons maps that had cycles in them as well.
He came to the conclusion that cycles are a very natural structure to humans and a very common way to design space. Even when looking at Zelda games, he bore out what Shigeru Miyamoto once said in an interview. *"Cycles foreground growth"*, meaning that a player coming back to an area stronger than it was when it first traversed it makes the player notice the difference in their own skills and abilities. To implement this in Unexplored, Dormans made use of *"Graph Grammars"*, which are based on a concept known as *"Transformational Grammars"*.

Transformational grammar is a system that defines a set of rules to create a series of sequences or structures from an original input. It usually consists of an *"alphabet"* and a set of rules. The grammar system will then get the given input from the alphabet and apply a series of rewriting operations according to the set rules to create a new output. This process can be executed in various iterations to expand a single input into a developed output.
For example, suppose that we have an alphabet with the symbols "A", "B", "a", "b" and "c". We can set some rules that will define the grammar and transform the symbols into new ones. If we set the rules "A -> Bc", and "B -> ab", the left hand symbols will be replaced with the ones on the right hand side. Therefore, if we apply the rules of the grammar to an input like "AB", we will get an output that looks like "abcab".

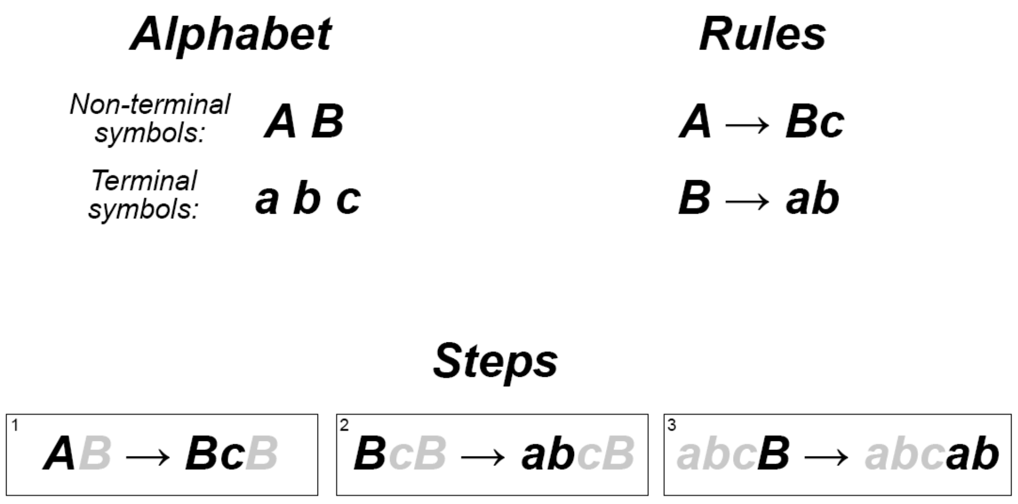


Figure 12: Transformational Grammar example.

This concept can also be applied to graphs, creating what is known as *"Graph Grammar"*. In this case a grammar would also consist of an alphabet and a set of rules in the same way, but instead of the alphabet containing letters or words, it would contain the different possible shapes a graph could adopt, and the set of rules would apply transformations on the graphs of a certain input to evolve the graph into a more complex structure.

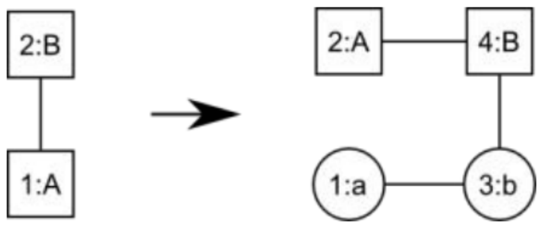


Figure 13: Example of a graph grammar rule. Dormans, Joris. (2010)

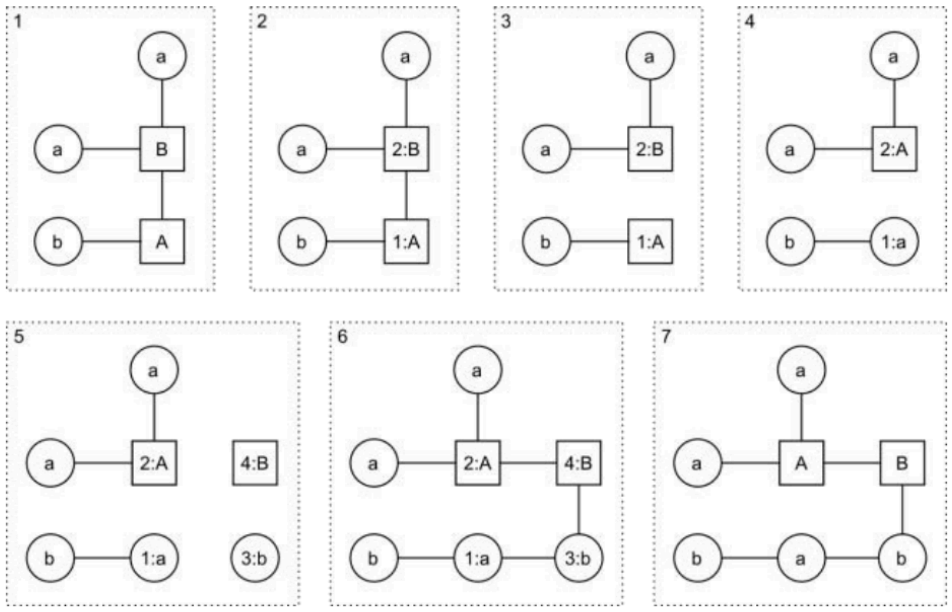
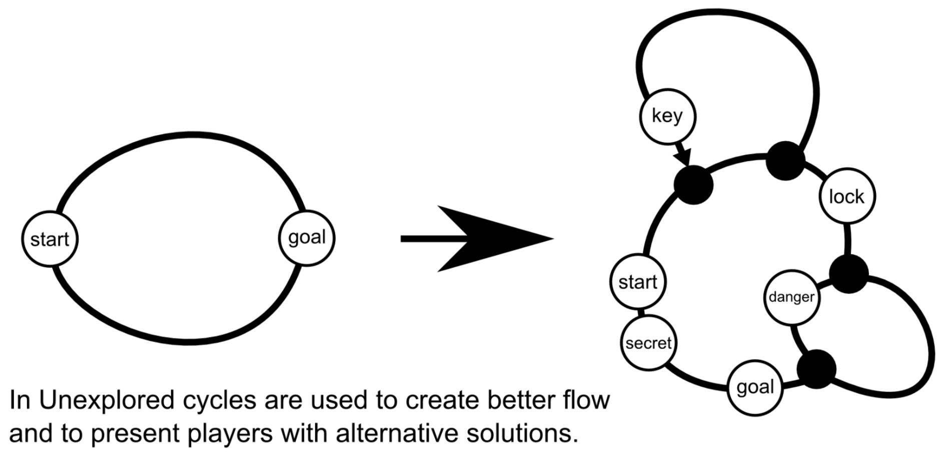


Figure 14: Transformation of a graph according to the previous rule. Dormans, Joris. (2010)

Developing the game, Dormans knew from the beginning that instead of making a single path from start to goal and then branching off it, he would create two paths, one going from start to goal and one going from goal to start, thus making a cycle. Then, making use of the previously mentioned graph grammars he would evolve this very simple initial graph into a more complex, more interesting one, which would represent the layout of the map at the conceptual level.



In Unexplored cycles are used to create better flow and to present players with alternative solutions.

Figure 15: Level layout graph evolution. Dormans, Joris. (2017)

When it comes to the actual implementation of the algorithm he uses a tool created by himself, *"Ludoscope"*, but the concept can be extrapolated to any other scenario without knowledge of the tool. The layout starts as a 4x4 graph of empty nodes that the first transformational step rewrites to add the main start - goal cycle. Then, two to three sub-cycles are added to the main one through more steps according to other grammar rules. There are many kinds of different cycle possibilities, so in order for it not to spiral out of control, what Dormans does is to apply a grammar with different sets of rules to every step of the process, and in this way it can make more accurate adjustments according to the current state of the graph.

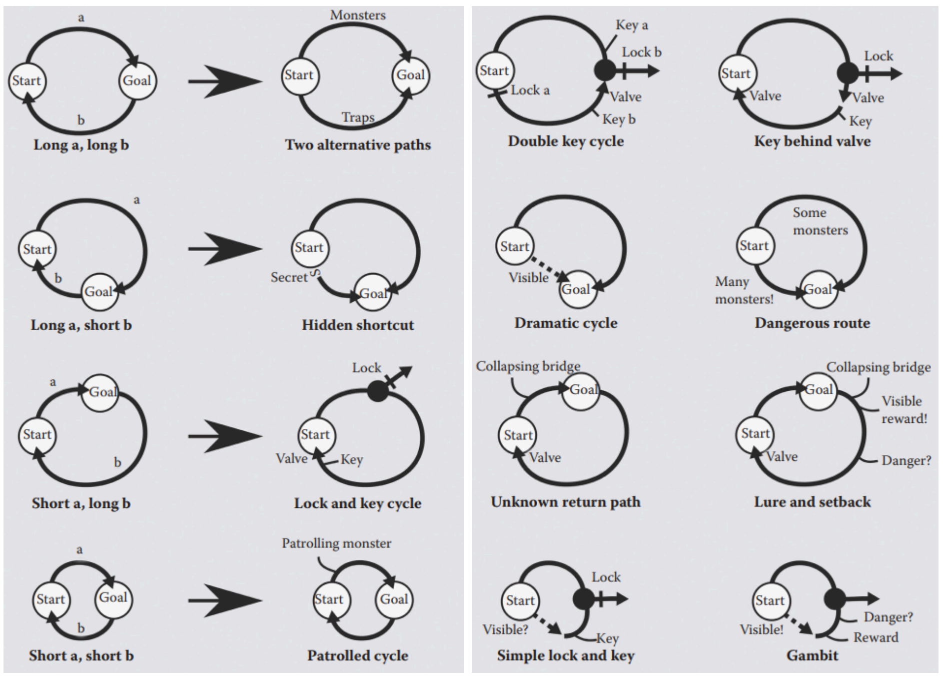


Figure 16: Examples of sub-cycle types. Dormans, Joris. (2017)

It is important to note that at this stage of the generation process grammars are still very generic, so lock and key combinations or enemies can be placed in the graph in their appropriate positions, but which type of combination or enemy might be determined later in the algorithm by another set of rules. However, just as generative grammars for languages are capable of capturing concepts beyond words, generative grammars for games should be able to capture higher level design principles that yield interesting results both at macro and micro scopes.
The next steps in the generation process add further items and relations between nodes according to more grammar rules in new steps, until the full node layout with all of the necessary items and puzzles is complete.

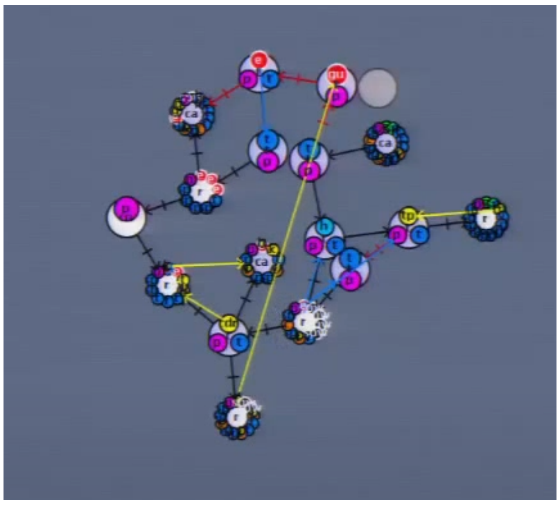


Figure 17: Final node layout. Dormans, Joris. (2016)

Once the layout of the level is completed, the generation of the actual physical level begins. The process starts with the algorithm producing a low resolution tile map based on the finished level graph, placing all of the rooms in the proper distribution and acknowledging their relations. After that, a higher resolution map is generated from the low resolution map shaping what would be the outline of the final level, and set pieces such as the entrance or exit zones are placed. Some noise is generated on the side to provide the map with biome-specific zones, and it is later superimposed onto the level. Lastly, decorations are added with trees, columns, wind zones and other populating the dungeon to make the game more interesting and certain zones more appealing or fun to play in.

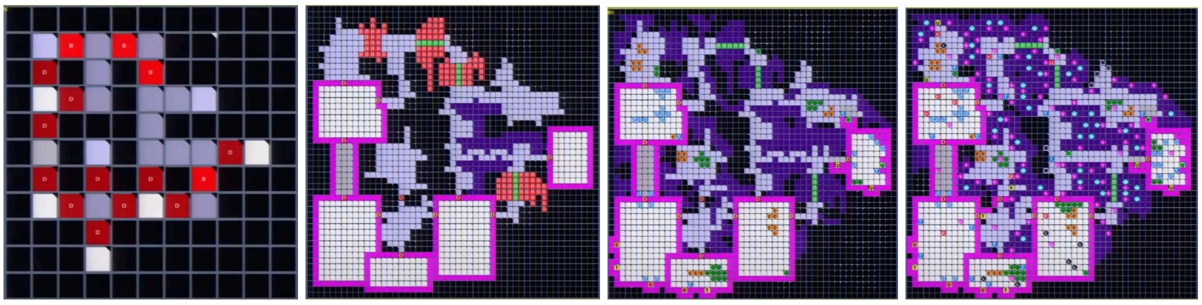
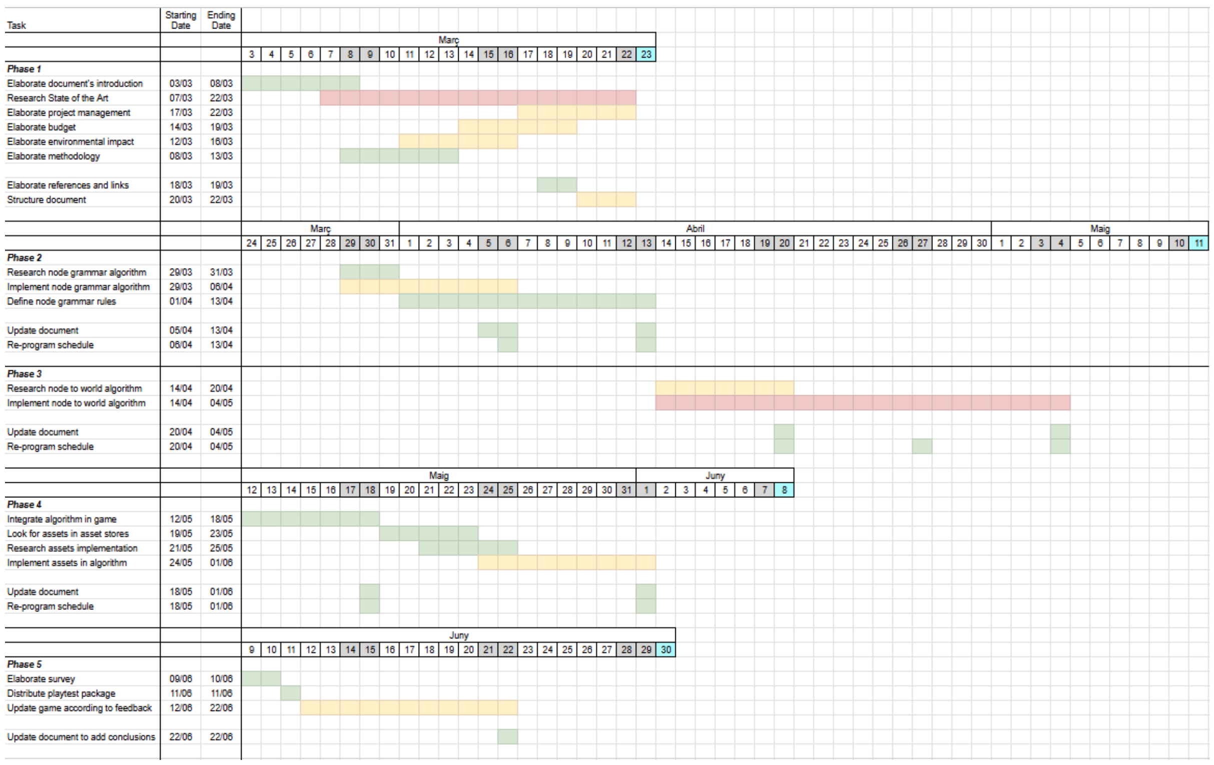


Figure 18: Different steps of the level generation process. Dormans, Joris. (2016)

In conclusion, the approach to dungeon generation implemented in Unexplored was quite revolutionary, even if games featuring procedural dungeon generation have not followed the trend after it. The game's generator produces very natural layouts that to many people feel even hand-crafted, making a positive impact on the players that explore them in depth and setting the gameplay aside stealing the spotlight when talking about Unexplored.
Due to the depth and complexity given to the generated levels, this project will be based on the concept of cyclic layouts implementing graph transformational grammars and mimicking with a custom implementation how the game does it up to the point of creating the physical level. After that, the elaborated graph will be used to build the levels where our game will be played in 3D.

3. Project management

In order to plan the development of the project certain previous evaluations have been conducted to determine its viability and structure the different phases of its production in time accordingly.
The first step towards structuring the project was to elaborate a Gantt chart with which to determine the different tasks involved in development and their duration.



Task	Starting Date	Ending Date
Phase 1		
Elaborate document's introduction	03/03	08/03
Research State of the Art	07/03	22/03
Elaborate project management	17/03	22/03
Elaborate budget	14/03	19/03
Elaborate environmental impact	12/03	16/03
Elaborate methodology	08/03	13/03
Elaborate references and links	18/03	19/03
Structure document	20/03	22/03
Phase 2		
Research node grammar algorithm	29/03	31/03
Implement node grammar algorithm	29/03	06/04
Define node grammar rules	01/04	13/04
Update document	05/04	13/04
Re-program schedule	06/04	13/04
Phase 3		
Research node to world algorithm	14/04	20/04
Implement node to world algorithm	14/04	04/05
Update document	20/04	04/05
Re-program schedule	20/04	04/05
Phase 4		
Integrate algorithm in game	12/05	18/05
Look for assets in asset stores	19/05	23/05
Research assets implementation	21/05	25/05
Implement assets in algorithm	24/05	01/06
Update document	18/05	01/06
Re-program schedule	18/05	01/06
Phase 5		
Elaborate survey	09/06	10/06
Distribute playtest package	11/06	11/06
Update game according to feedback	12/06	22/06
Update document to add conclusions	22/06	22/06

Table 1: Gantt chart. Link to the table found in the annex.

3.1. SWOT

Throughout this document the difference between current algorithms and the proposed solution have been discussed multiple times, so to address the market potential of the project a strength, weaknesses, opportunities and threats analysis has been conducted.

	Positive	Negative
Internal	Strengths Layouts generated with this approach tend to feel very organic if implemented properly, and usually feel hand-made. The algorithm will be implemented in a third person shooter game, which is a very popular genre.	Weaknesses The algorithm is a bit complex, so development can get complicated quickly if not handled correctly. Since only one game has implemented this approach, there is very little information about the subject.
External	Opportunities There is a lot of untapped potential in the approach, since only one game has implemented it and it is 2D. No games implement it in 3D except for the sequel, which has a top-down perspective nonetheless.	Threats Might not receive a positive reception due to the current lack of trust in procedural generation-heavy marketing campaigns. Due to the current trend of adding some sort of procedural generation to everything, there is a lot of competition in this segment of the market.

Table 2: SWOT analysis.

3.2. Risks and contingency plan

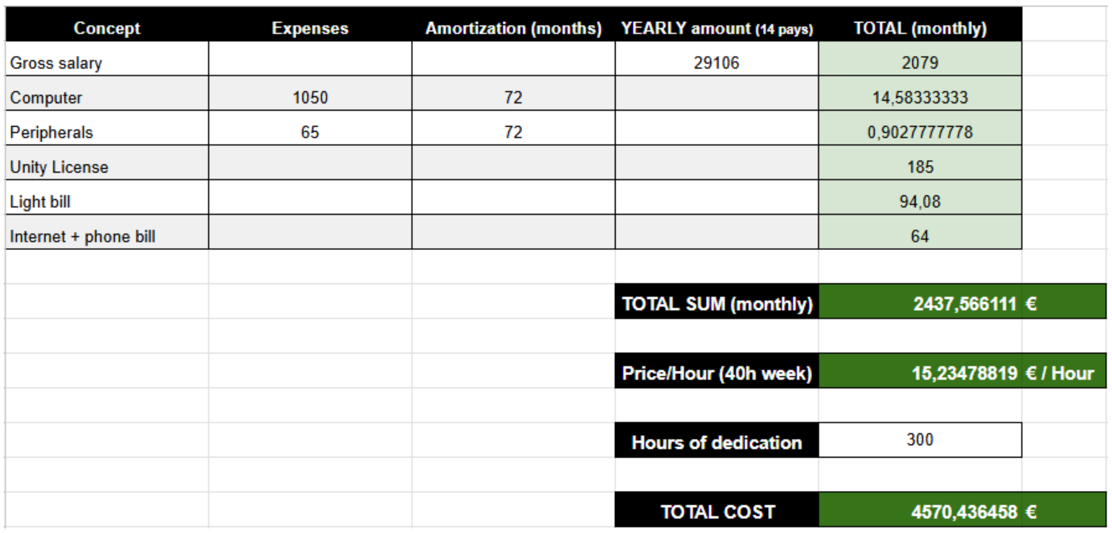
Given that even though complex in nature, this project is quite simple in terms of elaboration, the potential risks involved in the production process do not pose a major threat to its overall development. However, some minor setbacks may still arise, so a contingency plan to solve or prevent some of said setbacks has been elaborated.

Risk	Solution
When looking for assets in asset stores to visualize the results of the algorithm, there might not be free assets available that fit the project.	Blockout shapes with different colors or modes of representation could be implemented temporarily to showcase functionality.
Lack of experience and documentation resources about the specific implementation.	Allocate time for research during development, not just before it.
Main working station or computer malfunctions during development.	Use cloud based version control and refer to the latest version from a different machine to lose as little progress as possible.
Some bugs and errors might appear during development.	Have time allocated for bugs and problem solving to not exceed original planning.
During the development of the algorithm, process time can take longer in lower end machines.	Use profiling tools to help determine which parts of the algorithm are causing bottlenecks in order to optimize them.

Table 3: Risks and contingency plan.

3.3. Initial cost analysis

A budget has also been elaborated to estimate the cost of production. In the chart, gross salary is based on an average of the salaries of a junior gameplay programmer and a junior game AI programmer in Spain, since there is no specific job opening for a “Procedural Generation Programmer”. The electricity concept in the invoice is based on an average of last year's monthly electricity bill, and the internet cost is a flat rate.



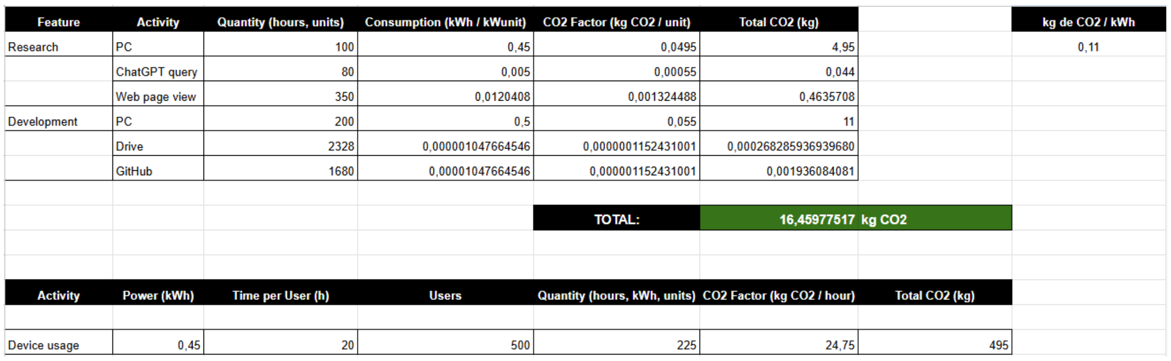
Concept	Expenses	Amortization (months)	YEARLY amount (14 pays)	TOTAL (monthly)
Gross salary			29106	2079
Computer	1050	72		14,58333333
Peripherals	65	72		0,9027777778
Unity License				185
Light bill				94,08
Internet + phone bill				64

TOTAL SUM (monthly)	2437,566111 €
Price/Hour (40h week)	15,23478819 € / Hour
Hours of dedication	300
TOTAL COST	4570,436458 €

Table 4: Project’s budget. Link to the table found in the annex.

3.4. Environmental impact

Similarly to the initial cost analysis, an environmental impact analysis has been conducted to determine how much of a carbon footprint the project will generate. In the breakdown, the “Consumption” value for a ChatGPT query is an approximate that has been provided as a reference, the “Web page view” value was referenced from *transitionnetwork.org*, and the value for the cloud storage platforms was referenced from *greenly.earth*. The CO2 kg per kWh conversion in Spain was referenced from *lowcarbonpower.org*.



Feature	Activity	Quantity (hours, units)	Consumption (kWh / kWunit)	CO2 Factor (kg CO2 / unit)	Total CO2 (kg)		kg de CO2 / kWh
Research	PC	100	0,45	0,0495	4,95		0,11
	ChatGPT query	80	0,005	0,00055	0,044		
	Web page view	350	0,0120408	0,001324488	0,4635708		
Development	PC	200	0,5	0,055	11		
	Drive	2328	0,000001047664546	0,0000001152431001	0,000268285936939680		
	GitHub	1680	0,00001047664546	0,000001152431001	0,001936084081		

TOTAL:	16,45977517 kg CO2

Activity	Power (kWh)	Time per User (h)	Users	Quantity (hours, kWh, units)	CO2 Factor (kg CO2 / hour)	Total CO2 (kg)
Device usage	0,45	20	500	225	24,75	495

Table 5: Environmental impact. Link to the table found in the annex.

4. Methodology

The project will be developed using an iterative incremental methodology. Development will happen in phases as in the traditional waterfall model, but it will consist of different iterations for the different parts of the algorithm. The project will be carried out using the Unity engine, given that it is the engine that us, the developers are most comfortable using. In order to keep track of the different phases and the current state of development, the previously mentioned Gantt chart will be used and updated frequently based on the needs of production.

The first phase will consist of research and setting the project's theoretical framework. In addition to it, the project's motivation, the problem it solves, the objectives and the scope of it will be determined. A market analysis will be carried out to determine what the state of the art is, and what kind of implementations could be carried out to make advances in the field and possibly innovate creating something new.
In order to make the project as viable as possible, different analyses will be developed to determine how to manage the project. This will include a SWOT diagram, a risks and contingency plan, a cost analysis with a budget for the project and what the environmental impact of development will be.
Lastly, a methodology will be set to be able to structure the different phases of the project, determine what will be done in each of the phases and be efficient while doing it. This methodology, even though set in the first phase, is open to modifications and subject to change.

The second phase will start the development process for the algorithm. The objective of this first development stage is to lay the foundations of the grammars and rules that will define the node structure of the level. To do so, various iterations of more research about the implementation of transformational grammars and development will be conducted according to the needs of development, and some testing will be done to ensure that the algorithm yields satisfying results.
Once this part of the implementation is considered complete, some time will be allocated to reflect the process of creating this part of the generator onto the document. This time allocation and reflection of the process will be also carried out in all of the following phases.

The third phase will pick up where the second left the algorithm and will continue to research and develop how to translate the generated graphs to a physical 3D level. More iterations will be conducted after testing for satisfactory results, just as in the second phase, and whenever said point is reached, the algorithm itself could be considered complete.

The fourth phase will be carried out in the same way as phases two and three, iterating on the development of the project but the focus will be different, since the main emphasis of this phase is to integrate the algorithm for the generator into the already existing base game mentioned in paragraphs 1.3 and 1.4, and final assets will be added to the project's library to also integrate them if found in asset stores.

The fifth phase will be the closing phase. Not only the algorithm ought to be finished, but the game prototype should be in a playable state too. The aim of this phase is to validate whether the implementation of the generator was successful or not. To do so, a build with the

prototype will be delivered to around twenty to twenty-five people (ideally more), alongside a Google Forms survey which all test candidates should answer.
Feedback will later be taken into account to perform some last tweaks to the generator if necessary to improve the overall satisfaction of potential players.

A possible future improvement to the validation system would be to implement callbacks in the code to upload certain KPIs to a database while the test users are playing with the objective to later analyze the data and be able to determine the weak points of the algorithm, such as places where the player might get stuck or imbalances on the player experience that were not foreseen. However, given the current scope of the project, the objective of validation is to determine whether the players enjoy the experience at a qualitative level, so the previously mentioned metrics will not be implemented nor taken into account in this document.

5. Project Development

As mentioned previously, the project will be carried out in the Unity engine and using C#. Even though the end goal of it is to be implemented in an already existing project, the algorithm will be developed as a standalone and it will be made from scratch, occasionally referring to external algorithms to speed up the development of the generation code.

5.1 Graph Generation

The core of the generator are graphs. In order to work with them, we first need a code representation we can work with. There are many ways to create this kind of representation when dealing with computers, and the best one will always be the one that better suits your needs. The selected approach will consist of two essential components: the Graph class and the Node class. Note that a traditional graph would also contain edges, but as will be explained later, these will be inferred from the connections between the nodes.

5.1.1 Class Structures

In this implementation, nodes will represent different parts of the level, so we need them to have a *type* variable that represents what will be in that part of the level. Since it is still a node in a graph, we need a list of the nodes it will be attached to as well. At the beginning, this was a list of nodes, given it is what makes the most sense, however, this was later changed to a list of unsigned integers which will represent the id of the node in the main graph, given it was easier to work with them in the graph isomorphism and replacement algorithms. This will be further explained in sections 5.2 and 5.5. An important remark is that with this structure the graph will be directed, since node A will be able to connect to node B even if node B does not connect to node A.

The resulting Node class looks like this:

```
CSharp
public class Node
{
    // The enum representiong the node type
    public NodeType type;

    // List of connected notes, with uints representing their IDs in the
    // main graph
    public List<uint> neighbors;

    public Node(NodeType nodeType)
    {
        type = nodeType;
        neighbors = new List<uint>();
    }
}
```

When it comes to the Graph class, the structure that will hold all of the information will be a Dictionary which will use uints as keys and the Node class as values. This uints will represent the ID of each node, this way each node can be accessed by ID without needing to iterate over all of the possible nodes. The Graph class will only contain this structure, as well as all of the member functions that will be used to operate on the graph.

The resulting Graph class looks like this:

```csharp
public class Graph
{
    // uint represents Node id
    public Dictionary<uint, Node> nodes { get; private set; }

    public Graph()
    {
        nodes = new Dictionary<uint, Node>();
    }

    // Function takes type, automatically assigns ID and adds node to graph.
    public uint AddNode(NodeType type)
    {
        Node node = new Node(type);
        uint assignedID = GetLowestFreeID();

        nodes.Add(assignedID, node);
        return assignedID;
    }

    // Passing in the ID of the original node and the ID of the destination,
    // access the original node and add the target to its neighbors list
    // only if it is not already present in the list.
    public void AddEdge(uint from, uint to)
    {
        if (!nodes[from].neighbors.Contains(to))
            nodes[from].neighbors.Add(to);
    }

    // Helper function to get the lowest unassigned ID starting from 0;
    private uint GetLowestFreeID()
    {
        uint currentId = 0;
        while (nodes.ContainsKey(currentId))
        {
            currentId++;
        }
        return currentId;
    }
}
```

5.1.2. Graph isomorphism

A graph is isomorphic to another, in short, when there exists a node and edge in a graph that matches exactly to a node and edge from another. In this case, we need to implement a “Contains” function in the Graph class to be able to replace subgraphs with target graphs if the condition graphs are found in the main graph. To ease the process of mapping nodes in the main graph to the ones in the condition, a simple helper struct has been created.

```
CSharp
public struct NodeRelation
{
    public NodeRelation(uint mainID, uint conditionID)
    {
        this.mainID = mainID;
        this.conditionID = conditionID;
    }

    public uint mainID;
    public uint conditionID;
}
```

Given that graph theory is not the main topic of this project, this “Contains” function has been based on the Ullman algorithm, and the code for it is originary from the following StackOverflow thread:

https://stackoverflow.com/questions/13537716/how-to-partially-compare-two-graphs/13537776#13537776

The mentioned code snippet is written in python, so it needed to be translated into C#. After manually translating it and adapting it to the current implementation, it looks something like this:

```
CSharp
public bool Contains(Graph subgraph, out List<NodeRelation>
mainGraphNodeRelations)
    {
        List<uint> findResult = FindIsomorphism(subgraph);
        mainGraphNodeRelations = new List<NodeRelation>();

        if (findResult == null || findResult.Count <= 0)
        {
            return false;
        }

        for (int i = 0; i < findResult.Count; i++)
        {
            //Value is main graph, index is subgraph
            mainGraphNodeRelations.Add(new NodeRelation(findResult[i],
(uint)i));
        }

        return true;
    }
```

A custom get field was also made for the auxiliary functions:

```
CSharp
public struct Edge
    {
        public uint from { get; private set; }
        public uint to { get; private set; }

        public Edge(uint from, uint to)
        {
            this.from = from;
            this.to = to;
        }
    }
    public List<Edge> edges
    {
        get
        {
            List<Edge> returnList = new List<Edge>();

            foreach (var node in nodes)
            {
                if (node.Value.neighbors.Count <= 0) { continue; }

                foreach (var neighbor in node.Value.neighbors)
                {
                    returnList.Add(new Edge(node.Key, neighbor));
                }
            }

            return returnList;
        }
    }
```

And the “Contains” auxiliary functions look like this:

```
CSharp
private List<uint> FindIsomorphism(Graph subgraph)
    {
        List<uint> assignments = new List<uint>();

        List<HashSet<uint>> possibleAssignments = new List<HashSet<uint>>();
        foreach (var subNode in subgraph.nodes)
        {
            HashSet<uint> setToAdd = new HashSet<uint>();

            foreach (var mainNode in nodes)
            {
                if (mainNode.Value.type == subNode.Value.type)
                {
                    setToAdd.Add(mainNode.Key);
                }
            }

            possibleAssignments.Add(setToAdd);
        }

        if (Search(subgraph, assignments, possibleAssignments))
        {
            return assignments;
        }

        return null;
    }
```

```
CSharp
private bool Search(Graph subgraph, List<uint> assignments,
List<HashSet<uint>> possibleAssignments)
    {
        UpdatePossibleAssignments(subgraph, possibleAssignments);
        int assignmentsCount = assignments.Count;

        //Make sure that every edge between assigned vertices in the
subgraph is also an edge in the graph.
        foreach (var edge in subgraph.edges)
        {
            if (edge.from < assignmentsCount && edge.to < assignmentsCount)
            {
                uint mappedFrom = assignments[(int)edge.from];
                uint mappedTo = assignments[(int)edge.to];

                if (!edges.Contains(new Edge(mappedFrom, mappedTo)))
                    return false;
            }
        }

        //If all the vertices in the subgraph are assigned, then we are
done.
        if (subgraph.nodes.Count == assignmentsCount)
        {
            return true;
        }

        //Fix for removing during iteration
        List<uint> iteratingAssignments = new
List<uint>(possibleAssignments[assignmentsCount]);

        foreach (var assignment in iteratingAssignments)
        {
            if (!assignments.Contains(assignment))
            {
                assignments.Add(assignment);

                //Create a new set of possible assignments, where graph node
j ("assignment")
                //is the only possibility for the assignment of subgraph
node i ("assignmentsCount").

                //Make possibleAssignments deep copy
                List<HashSet<uint>> newPossibleAssignments = new
List<HashSet<uint>>();
                foreach (var entry in possibleAssignments)
                {
                    newPossibleAssignments.Add(new HashSet<uint>(entry));
                }
                newPossibleAssignments[assignmentsCount] = new HashSet<uint>
{ assignment };

                if (Search(subgraph, assignments, newPossibleAssignments))
                {
                    return true;
                }

                assignments.RemoveAt(assignments.Count - 1);
            }
            possibleAssignments[assignmentsCount].Remove(assignment);
            UpdatePossibleAssignments(subgraph, possibleAssignments);
        }

        return false;
    }
```

```
CSharp
private void UpdatePossibleAssignments(Graph subgraph, List<HashSet<uint>>
possibleAssignments)
    {
        bool anyChanges = true;

        while (anyChanges)
        {
            anyChanges = false;
            foreach (var subNode in subgraph.nodes) //i
            {
                if (subNode.Key >= possibleAssignments.Count) { continue; }

                List<uint> assignmentsToRemove = new List<uint>();
                foreach (var possibleMaingraphNode in
possibleAssignments[(int)subNode.Key]) //j
                {
                    foreach (var subnodeNeighbor in subNode.Value.neighbors)
//x
                    {
                        bool match = false;
                        foreach (var mainNode in nodes) //y
                        {
                            if (subnodeNeighbor >= possibleAssignments.Count
|| possibleMaingraphNode >= nodes.Count) { continue; }

                            if
(possibleAssignments[(int)subnodeNeighbor].Contains(mainNode.Key) &&
nodes[possibleMaingraphNode].neighbors.Contains(mainNode.Key))
```

```
                                {
                                    match = true;
                                }
                            }
                            if (!match)
                            {
                                //Removing directly while iterating over it will
throw an exception
                                assignmentsToRemove.Add(possibleMaingraphNode);
                                anyChanges = true;
                            }
                        }
                    }
                    foreach (var assignment in assignmentsToRemove)
                    {

possibleAssignments[(int)subNode.Key].Remove(assignment);
                    }
                }
            }
        }
```

5.1.3. Json structures (init automatization)

After testing the implementation of the isomorphism algorithm on a few possible permutations of a condition graph, it became clear that a simpler way to declare and initialize graphs was needed. The tests were performed by constructing graphs manually using the graph's *AddNode()* and *AddEdge()* functions, so the next logical step was to automate this process. The easiest way to do so would be to add the possibility of loading them from a file, and given the clarity of the file type's structure and Unity having a built-in JSON utility, it was concluded that building a pipeline based on JSON files would be the most comfortable way to approach automation.
Using Unity's JSON utility, JSON files can be automatically loaded into classes as long as the field types are public primitive types and the classes are public and marked as serializable. The utility also supports lists and using classes only if they are public and they have been previously marked as serializable.

Taking into account what is needed for the class structures, the resulting wrapper classes would look like this:

```CSharp
[System.Serializable]
public class Node_Data
```

```
{
    //This id will be used to set the node key in the graph's dictionary
    public uint id;

    public NodeType type;
    public List<uint> neighbors;
}

[System.Serializable]
public class Graph_Data
{
    public List<Node_Data> nodes;
}
```

In order to be able to initialize the class structures using these wrapper classes, a new constructor was needed that was able to take the wrappers as a parameter and set all of the variables inside their respective classes according to what was parsed from the JSON files into the wrappers.

These are the new constructors that must be added to the respective classes.

```
CSharp
//For the node class:
public Node(Node_Data data)
{
    type = data.type;
    neighbors = new List<uint>(data.neighbors);
}

//And for the graph class:
public Graph(Graph_Data data)
{
    nodes = new Dictionary<uint, Node>();
    foreach (var nodeData in data.nodes)
    {
        nodes.Add(nodeData.id, new Node(nodeData));
    }
}
```

With this pipeline set up, it is only a matter of defining the graphs in a JSON structure.

```
None
{
    "nodes": [
        {
            "id": <the id you want to set to the node>,
            "type": <an int representing the entry in the type enum>,
            "neighbors": [
                <a list with the ids the node will be connected to>
            ]
        },
        {
             <this is a different node in the graph>
            "id": <all nodes must have unique IDs>,
            "type": <type can be the same or different>,
            "neighbors": [
                <the list of neighbors can be empty, so nothing here>
            ]
        }
    ]
}
```

It is important to remember that with this implementation the graphs are directed, so if a node has no neighbors it can still be attached to other nodes if it is set as the neighbor of another node.

Once different JSONs have been created, creating a new Graph in code is as easy as calling the constructor of the Graph class using the wrapper class and the JSON utility:

```CSharp
//--- Example usage ---

//TextAsset represents the JSON file
TextAsset graphToInitialize;
Graph graph;

//We read the contents of the JSON file and save them on a string
string parsedJSON = graphToInitialize.text;

//We load the Graph_Data wrapper passing the parsed info to the JsonUtility
Graph_Data graphWrapper = JsonUtility.FromJson<Graph_Data>(parsedJSON);

//Using the Graph's new constructor, we pass in the initialized wrapper
graph = new Graph(graphWrapper);
```

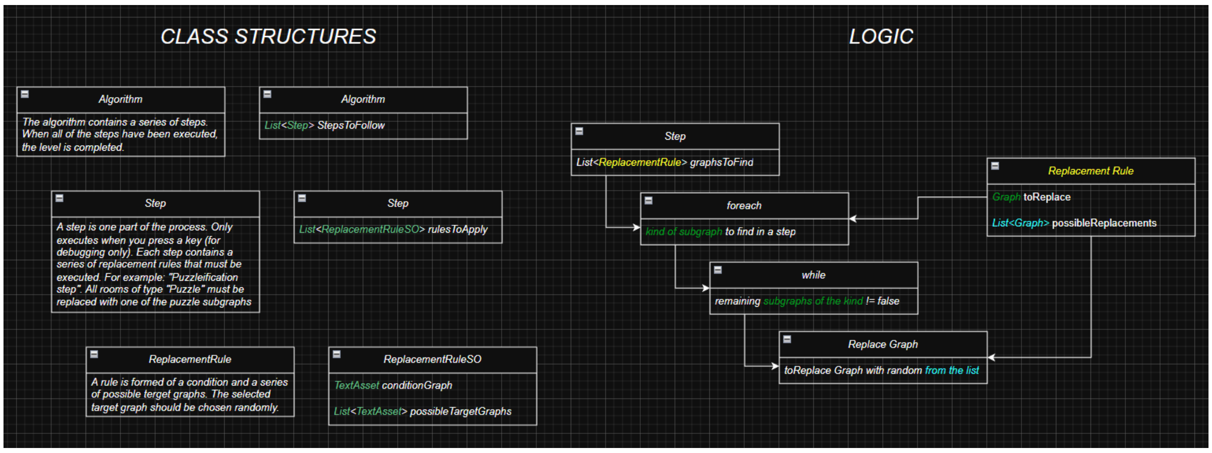
Or if we were to do it in an inline way, we could also just do:

```CSharp
//--- Example usage ---

TextAsset graphToInitialize;
Graph graph;

graph = new Graph(JsonUtility.FromJson<Graph_Data>(graphToInitialize.text));
```

5.1.4. Replacement rules

For implementing the replacement rules, the same approach that is mentioned in Dormans' *Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games* paper is taken. However, Dormans explains it on a high level without mentioning any code, so the implementation in this project was developed based on his descriptions.

Before starting to produce code related to the replacement, a preliminary conceptual diagram was developed to help design and visualize how the data would be structured. Mind that while the class structures seen in the diagram are final, the logic is only conceptual and does not reflect the final implementation.


CLASS STRUCTURES
Algorithm
Algorithm
List<Step> StepsToFollow
Step
Step
List<ReplacementRuleSO> rulesToApply
ReplacementRule
ReplacementRuleSO
TextAsset conditionGraph
List<TextAsset> possibleTargetGraphs
LOGIC
Step
List<ReplacementRule> graphsToFind
foreach
kind of subgraph to find in a step
while
remaining subgraphs of the kind != false
Replace Graph
toReplace Graph with random from the list
Replacement Rule
Graph toReplace
List<Graph> possibleReplacements

The algorithm contains a series of steps. When all of the steps have been executed, the level is completed.

A step is one part of the process. Only executes when you press a key (for debugging only). Each step contains a series of replacement rules that must be executed. For example: "Puzzleification step". All rooms of type "Puzzle" must be replaced with one of the puzzle subgraphs

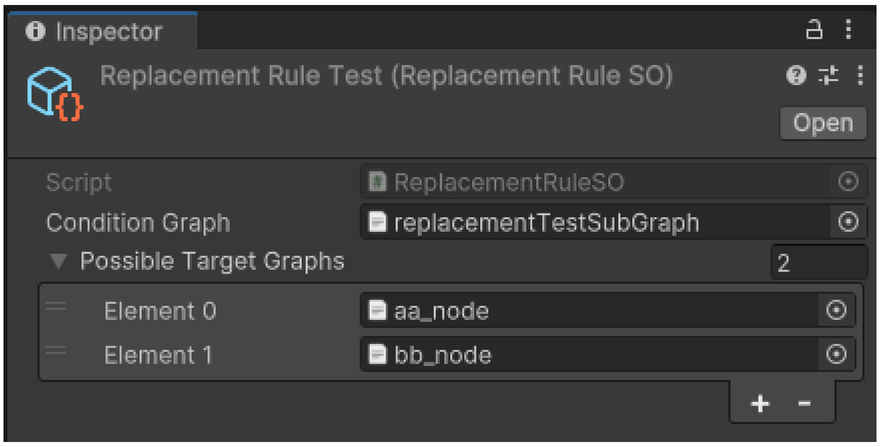
A rule is formed of a condition and a series of possible terget graphs. The selected target graph should be chosen randomly.

Figure 19: Preliminary diagram of replacement structures and logic.

The first step to build the replacing system is defining the rules that will form the core of the algorithm. For this, a Scriptable Object has been created to hold the condition subgraph and a list of possible target subgraphs. As it will be explained later, one of these subgraphs will be selected randomly to apply the rule onto the graph.

```CSharp
[CreateAssetMenu(fileName = "New Replacement Rule", menuName = "Rule")]
public class ReplacementRuleSO : ScriptableObject
{
    public TextAsset conditionGraph;
    public List<TextAsset> possibleTargetGraphs;
}
```

When taking a look at the Scriptable object in the Unity inspector we should see something like this, where all the fields can be filled with JSON files.



Script: ReplacementRuleSO
Condition Graph: replacementTestSubGraph
Possible Target Graphs: 2
Element 0: aa_node
Element 1: bb_node

Figure 20: Rule Scriptable Object Inspector view

Having the rules Scriptable Objects, the Step class can be made. This will take a List of rules and apply them onto the active graph when the step is called provided that the condition graph of the rule that is being executed is present in the active graph. The idea is that the algorithm will consist of a list of steps that will be executed one after the other, completing the graph once there are no more steps to be executed. More functions will be added when developing the algorithm, but for now the Step class looks like this:

```CSharp
[System.Serializable]
public class Step
{
    [SerializeField] List<ReplacementRuleSO> rulesToApply =
                     new List<ReplacementRuleSO>();

    //The seed parameter will be used to initialize the Random generator
    public void ApplyRules(Graph activeGraph, int seed)
}
```

If we create a serializable list of Steps and everything has been set up correctly, once we fill some fields with rules the Unity inspector should look like this, where every group of rules is a step to follow.

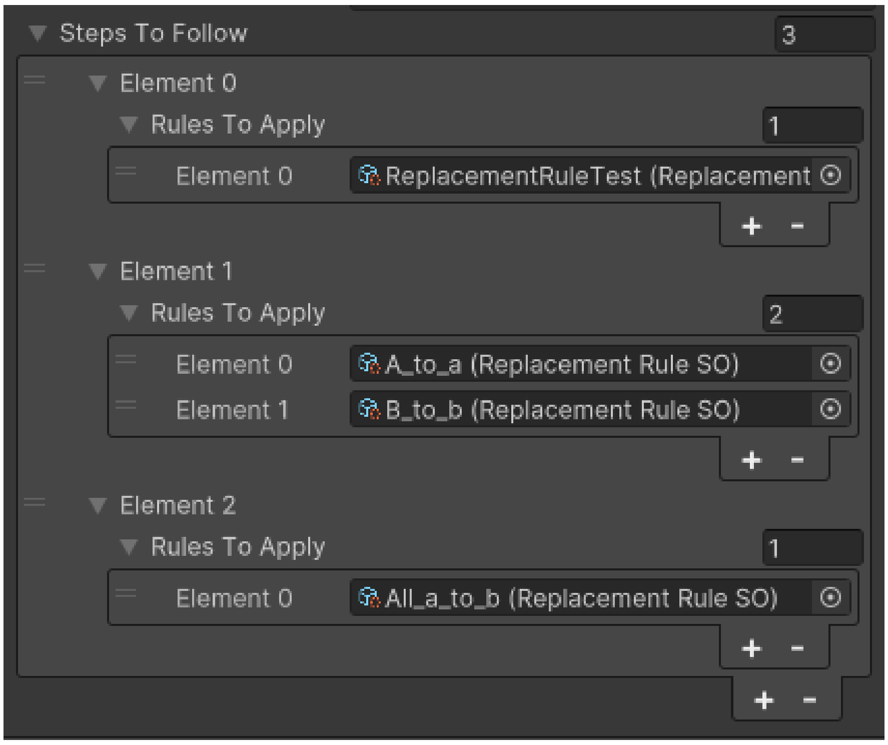

Steps To Follow 3
Element 0
Rules To Apply 1
Element 0 ReplacementRuleTest (Replacement
Element 1
Rules To Apply 2
Element 0 A_to_a (Replacement Rule SO)
Element 1 B_to_b (Replacement Rule SO)
Element 2
Rules To Apply 1
Element 0 All_a_to_b (Replacement Rule SO)
+ -

Figure 21: List of Steps in the Inspector view.

5.1.5. Replacement algorithm

With all of the necessary structures set up, the actual algorithm can be created. To implement it, a MonoBehaviour class has been created to be able to assign it to a Unity object so that its logic can be executed during playtime. This class will contain the initial graph the generator will start from and the previously mentioned list of steps to follow as well as the seed used to initialize the random generator. This seed can be set by hand from the Inspector if a specific combination is to be tested, but if set to 0 will be generated randomly.

```CSharp
public class Graph_Rewriting : MonoBehaviour
{
    [SerializeField] TextAsset initialGraphJson;
    [SerializeField] int seed = 0;

    [SerializeField] List<Step> stepsToFollow = new List<Step>();

    public Graph graph;

    private void Awake()
    {
        //We initialize the graph using the JSON constructor
        graph =
         new Graph(JsonUtility.FromJson<Graph_Data>(initialGraphJson.text));
    }
    private void Start()
    {
        //If the seed is set to 0 in the Inspector, initialize it to a
        //random value
        if (seed == 0) { seed = Random.Range(int.MinValue, int.MaxValue); }

        //For each listed step, apply its rules onto the graph using the
        //seed for the Random generator
        foreach (Step step in stepsToFollow)
        {
            step.ApplyRules(graph, seed);
        }
    }
}
```

When it comes to applying the rules, first we need to set the Random generator to the seed and then, for each rule in the list we will check if the rule's condition graph is present in the current active graph or not, using the Graph's "Contains" function, explained in section 5.2.. If it is not, we will just move on to the next rule, but if it is we will first randomly choose one of the possible target graphs in the rule and then replace the condition section of the main active graph with the selected target.

```
CSharp
public void ApplyRules(Graph activeGraph, int seed)
{
    //Set the Random generator seed to the seed set in the inspector
    Random.InitState(seed);

    this.activeGraph = activeGraph;

    //Traverse all of the rules in the rules list
    foreach (var rule in rulesToApply)
    {
        conditionGraph =
      new Graph(JsonUtility.FromJson<Graph_Data>(rule.conditionGraph.text));

        //Check if the rule's condition graph is present in the active graph
        if (activeGraph.Contains(conditionGraph,
                                     out List<NodeRelation> nodeIdsList))
            {
            //Generate random number and select from list
            int targetGraphIndex = Random.Range(0,
                                       rule.possibleTargetGraphs.Count);

            targetGraph = new Graph(JsonUtility.FromJson<Graph_Data>
            (rule.possibleTargetGraphs[targetGraphIndex].text));

            //Perform the replacement of the condition in the main graph
            ReplaceSubgraphInstance(nodeIdsList);
        }
    }
}
```

For replacing the section of the active graph that contains the condition graph with the selected target graph, the approach that is mentioned at the beginning of section 5.4. is taken. Following Dormans' paper, we would first need to assign IDs to the nodes, but thanks to how the Graph's "Contains" function works we already have that in the list of NodeRelations, so this step can be omitted. The rest follows as usual, except for a couple extra added steps to make the approach work in code.

```CSharp
void ReplaceSubgraphInstance(List<NodeRelation> nodesList)
    {
        //We will need to add the nodes form the target graph that are not
        //existing in the active graph, so we add them all to a list and we
        //will remove each node that is contained in the main graph later **
        List<uint> nodesToAdd = new List<uint>();
        foreach (var node in targetGraph.nodes)
        {
            nodesToAdd.Add(node.Key);
        }

        //For each main graph node existing in the condition subgraph
        foreach (var relation in nodesList)
        {
            //Process Step 1: We don't need to assign values since that is
            //already done in the subgraph check process, we store these
            //values in the "NodeRelations" list.

            //Process Step 2: Break all relations present in condition graph
            List<uint> neighborsToRemove = new List<uint>();
            foreach (var mainNeighbor
                     in activeGraph.nodes[relation.mainID].neighbors)
            {
                //If there is a neighbor (edge) in the main graph that is
                //the same as a neighbor in the condition graph, remove it
                if (conditionGraph.nodes[relation.conditionID].neighbors.
                    Contains(FindConditionValueFromMain(
                    nodesList, mainNeighbor)))
                {
                    neighborsToRemove.Add(mainNeighbor);
                }
            }
            foreach (var neighbor in neighborsToRemove)
            {
                activeGraph.nodes[relation.mainID].neighbors.
                 Remove(neighbor);
            }

            //Process Step 3: Convert to target types
            activeGraph.nodes[relation.mainID].type =
           targetGraph.nodes[relation.conditionID].type;

            nodesToAdd.Remove(relation.conditionID);//**Remove exsting nodes
        }

        //Step 4: Add missing nodes
        foreach (var missingNode in nodesToAdd)
        {
            uint newNodeID =
             activeGraph.AddNode(targetGraph.nodes[missingNode].type);

            nodesList.Add(new NodeRelation(newNodeID, missingNode));
        }

        //Step 5: Create resulting connections
        foreach (var relation in nodesList)
        {
            foreach (var targetNeighbor
                     in targetGraph.nodes[relation.conditionID].neighbors)
            {
                activeGraph.nodes[relation.mainID].neighbors.
                 Add(FindMainValueFromCondition(
                 nodesList, targetNeighbor));
            }
        }
    }
```

5.2 Level Generation

At this point we have generated the graph that will represent the layout of the level, but we don't have any physical representation of it yet. In order to generate the level, we will treat each node as a "Room" in the map, and each connection as a "Corridor". We will first define the positions of the rooms in the space, and then place the rooms in their defined positions. Lastly, we will connect them using said corridors.

```
CSharp
void GenerateLevel()
    {
        activeGraph = graphGenerator.activeGraph;

        Dictionary<uint, Vector3> positions = GenerateLayoutPositions();

        List<GeneratorRoom> placedRooms = PlaceRooms(positions);

        PlaceCorridors(placedRooms, positions);
    }
```

5.2.1. Defining Room Positions

The first step we need to take to give our conceptual layout a physical form is to define where each room will be in space. To do that, we will assign a Vector3 (position) to each node index, but since the nodes have specified connections and relations we can not set these positions at random. The taken approach will be to “untangle” the graph. This means that we will start with random positions for each node but then we will do somewhat of a physics simulation and apply forces on the nodes based on the relations and closeness to each other, so we end up with the least amount of relations or edges crossing each other. That way we end up with a clean topology for our graph so that the final level is as intuitive and easy to understand as possible.

```
CSharp
Dictionary<uint, Vector3> GenerateLayoutPositions()
    {
        //The NodePositioner class is a helperwhich will hold a position and
        //a velocity for the simulation
        Dictionary<uint, NodePositioner> nodes = new Dictionary<uint,
NodePositioner>();
        //For each Node we want a Positioner
        for (uint i = 0; i < activeGraph.nodes.Count; i++)
        {
            nodes.Add(i, new NodePositioner(GenerateRandomPosition(),
Vector2.zero));
        }

        SolvePositions(nodes);

        //Once the graph has been untangled, we want to return the final
        //positions in a Vector3 form
        Dictionary<uint, Vector3> positions = new Dictionary<uint,
Vector3>();
        for (uint i = 0; i < nodes.Count; i++)
        {
            Vector2 pos = nodes[i].position;
            positions.Add(i, new Vector3(pos.x, 0, pos.y));
        }

        return positions;
    }
```

This untangling process is not an easy one, so to speed up the process some research was conducted to find an already existing implementation that solved this specific problem. A web page hosting a little game simply called "*Untangle, from Simon Tatham's Portable Puzzle Collection*" was found where the objective was to untangle a graph of a set of randomly distributed nodes in order to make a planar graph:

https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/untangle.html

The interesting thing about this particular implementation as opposed to others such as "*Planarity*", is that this one features a "Solve" button, which does exactly what I was looking for. After taking a look at the script that was running the game in the browser, it was translated to C# as it was done with the Ullman algorithm in section 5.1.2, and after some adjustments it was ready to be inserted into the existing pipeline.

```CSharp
void SolvePositions(Dictionary<uint, NodePositioner> nodes)
    {
        //We must do many iterations of the simulation to produce an
        //accurate result. This final value was considered not too heavy
        //on the machine, but not too small to produce inaccurate results.
        int iterations = 50000;
        float repulsionStrength = 5000f;
        float springLength = 2f;
        float springStrength = 0.1f;
        float damping = 0.9f;

        for (int iter = 0; iter < iterations; iter++)
        {
            // Apply repulsion between all pairs of nodes
            for (uint i = 0; i < nodes.Count; i++)
            {
                Vector2 force = Vector2.zero;
                for (uint j = 0; j < nodes.Count; j++)
                {
                    if (i == j) continue;
                    Vector2 diff = nodes[i].position - nodes[j].position;
                    float dist = Mathf.Max(diff.magnitude, 0.01f);
                    force += diff.normalized * (repulsionStrength / (dist *
dist));
                }
                nodes[i].velocity += force;
            }

            // Apply spring force for each edge
            foreach (var edge in activeGraph.edges)
            {
                NodePositioner a = nodes[edge.from];
                NodePositioner b = nodes[edge.to];
                Vector2 delta = b.position - a.position;
                float dist = delta.magnitude;
                Vector2 springForce = delta.normalized * (dist -
springLength) * springStrength;
                a.velocity += springForce;
                b.velocity -= springForce;
            }

            // Update positions and apply damping
            foreach (var node in nodes)
            {
                node.Value.position += node.Value.velocity;
                node.Value.velocity *= damping;
            }
        }
    }
```

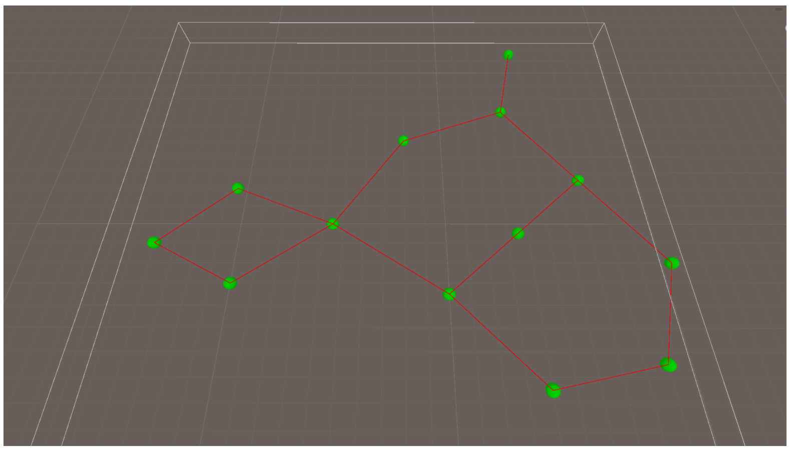


Figure 22: Positions represented in the 3D space.

5.2.2. Room Placement

Once the rooms' positions have been decided, we can go on to place them. However, there is an important consideration to take into account. Later on we will have to place corridors to connect the rooms, and we will do so by running a pathfinding algorithm (explained in section 5.2.3). To make this process easier, it has been decided to situate all of the physical parts of the level on a grid. The rooms don't need to be square necessarily, but they will occupy a space in this "invisible" grid. This grid will be composed of "*Cells*", which will hold different types of information.

```csharp
//This struct doesn't have much now, but we will add more fields in the
//future
public struct Cell
{
    public NodeType type;

    public Cell(NodeType type)
    {
        this.type = type;
    }
}
```

In order to be able to place the rooms in the grid, we will also need to read some information from the rooms we are going to place, such as the size they will occupy. To do so, we will create another script that will go in each room prefab we create, and we will set the needed information from the prefab's inspector.

```csharp
//Same as with the Cell struct, we will add more later down the line
public class GeneratorRoom : MonoBehaviour
{
    public Vector2Int size;
}
```

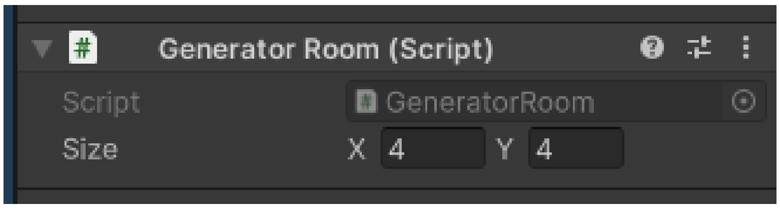

Generator Room (Script)
Script GeneratorRoom
Size X 4 Y 4

Figure 23: Room script from the inspector.

With the necessary structures in place, placing the rooms comes down to selecting the 3D model (prefab) we want to use according to the node's type, using the assigned position to the node’s index while snapping it to the grid, making sure the actual grid is filled and Instantiating the prefab.

```
CSharp
List<GeneratorRoom> PlaceRooms(Dictionary<uint, Vector3> positions)
    {
        List<GeneratorRoom> placedRooms = new List<GeneratorRoom>();

        for (uint i = 0; i < activeGraph.nodes.Count; i++)
        {
            GameObject selectedRoomPrefab;

            //Select the room we are going to place
            switch (activeGraph.nodes[i].type)
            {
                case Start: selectedRoomPrefab = startRoom; break;
                case Goal: selectedRoomPrefab = goalRoom; break;
                case Room: selectedRoomPrefab = normalRoom; break;
                case Lock: selectedRoomPrefab = lockRoom; break;
                case Key: selectedRoomPrefab = keyRoom; break;
                case Bridge: selectedRoomPrefab = bridgeRoom; break;
                case OneWayDrop: selectedRoomPrefab = oneWayDropRoom; break;
                default: selectedRoomPrefab = normalRoom; break;
            }

            GeneratorRoom toSet =
selectedRoomPrefab.GetComponent<GeneratorRoom>();

            Vector2Int posInGrid = WorldToGrid(positions[i]);

            //Check overlap
            GeneratorCollisionSolver.CheckOverlap(ref posInGrid, toSet.size,
grid);

            //Once generation is solved, fill the grid with the computed
            //information
            for (int j = 0; j < toSet.size.x; j++)
            {
                for (int k = 0; k < toSet.size.y; k++)
                {
                    grid[posInGrid.x - j, posInGrid.y + k] = new
Cell(activeGraph.nodes[i].type);
                }
            }

            //And then place the physical room in the level
            GameObject roomInstance = Instantiate(selectedRoomPrefab,
GridToWorld(posInGrid.x, posInGrid.y), Quaternion.identity);
```

```
            placedRooms.Add(roomInstance.GetComponent<GeneratorRoom>());
        }

        return placedRooms;
    }
```

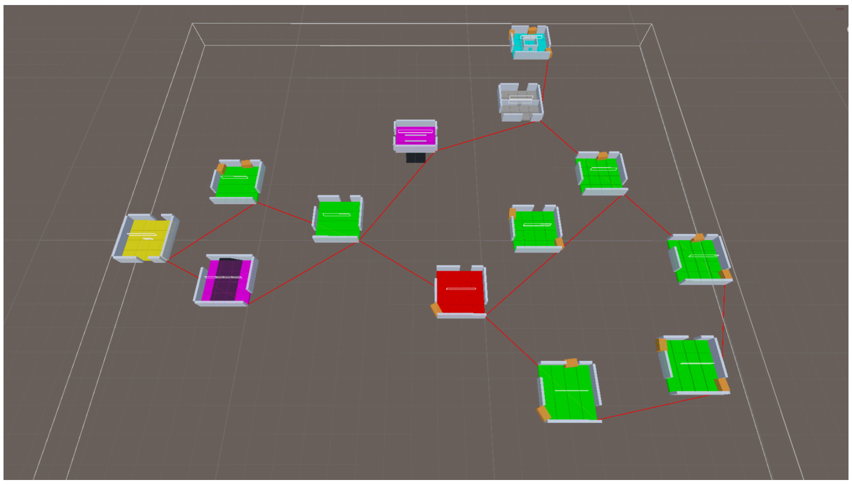


Figure 24: Rooms placed with their respective type representations.

Notice the function called "*CheckOverlap*". This function is very important to detect whether the room will be placed in a free space or not. In case the room trying to be placed is overlapping another one, its pivot (the position where it was originally going to be placed) will be moved in the most favorable direction to solve the collision, checking for overlap at each step until the collision is solved.

```CSharp
//In CheckOverlap we check if any position of the room Cells already has a
//value, and if so we detect a collision and then solve it
static void SolveCollision(ref Vector2Int origin, Vector2Int size,
Vector2Int conflictPoint, Cell?[,] grid)
    {
        Vector2Int targetOrigin;
        Vector2Int targetSize;

        //We know the limits of the current room, but we also need the
        //limits for the one we detected
        FindLimits(conflictPoint, out targetOrigin, out targetSize, grid);

        RectInt placingRect = new RectInt(origin, size);
        RectInt targetRect = new RectInt(targetOrigin, targetSize);

        //We also want to leave at least a gap of one between the rooms to
        //be able to place corridors later
        targetRect.position -= Vector2Int.one;
        targetRect.size += Vector2Int.one * 2;

        //With that we find the overlapping area and depending on its size
        //we will separate them vertically or horizontally
        RectInt overlappingRect = FindOverlappingArea(placingRect, grid);

        bool solveHorizontal = (overlappingRect.width <=
overlappingRect.height) ? true : false;

        Vector2 dir = overlappingRect.center - targetRect.center;

        //Finally, we just move in the direction that is furthest from the
        //center of the other room
        if (solveHorizontal)
        {
            if (dir.x > 0) origin.x++;
            else origin.x--;
        }
        else
        {
            if (dir.y > 0) origin.y++;
            else origin.y--;
        }
    }
```

At the end of this placing step, we want to return a list with all of the placed rooms so we can feed it to the next step of the algorithm, and get information about how to connect the corridors from said list.

5.2.3. Corridor Placement

Having all of the rooms placed in their positions, all that's left to do is to connect them with corridors following the graph's edges. We will need the previously mentioned positions list from 5.2.1 and the placed rooms list from section 5.2.2. However, we will also need to know where in the room the corridors must connect and to do so we will create an "Entrance" class that will store information useful to create the connections between rooms and corridors. We will also update the rooms' script to fit a list of possible entrances so that they can also be set from the prefab's inspector alongside the blockages that will limit the players' movement in case the entrances are not opened.

CSharp
```
[System.Serializable]
public class Entrance
    {
        //The direction where the entrance is "facing"
        public Direction direction;

        //The position with respect to the room's pivot or origin
        public Vector2Int localPosition;

        //The object in the way that we have to remove to allow passage
        public GameObject blockage;

        //A control variable to know if it has been already used
        [HideInInspector] public bool unlocked;
    }
```

CSharp
```
public class GeneratorRoom : MonoBehaviour
{
    public Vector2Int size;
    public List<Entrance> entrances;

    private void Awake()
    {
        foreach (var entrance in entrances)
        {
            entrance.unlocked = false;
        }
    }
}
```

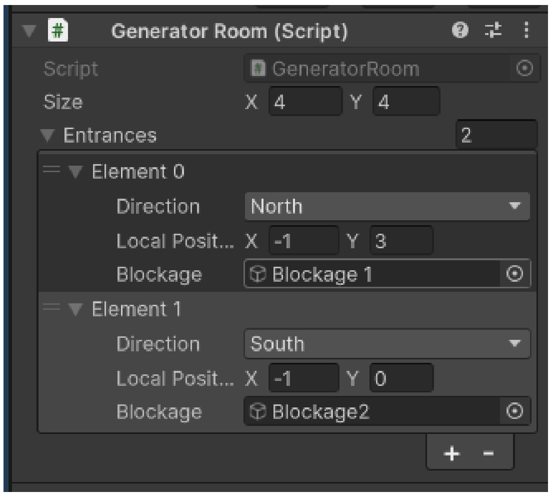

Generator Room (Script)
Script: GeneratorRoom
Size: X 4 Y 4
Entrances: 2
Element 0
Direction: North
Local Posit...: X -1 Y 3
Blockage: Blockage 1
Element 1
Direction: South
Local Posit...: X -1 Y 0
Blockage: Blockage2

Figure 25: Updated Room script inspector.

To place the corridors, we want to iterate through each neighbor of each node in the graph, which represents an edge. For each edge, we will iterate the free “entrances” in the room at each side of the corridor and will pick the closest entrance pair. Once selected, we will mark them as “unlocked” and remove the blockage to allow passage.

```
CSharp
void PlaceCorridors(List<GeneratorRoom> placedRooms, Dictionary<uint,
Vector3> positions)
    {
        for (uint i = 0; i < activeGraph.nodes.Count; i++)
        {
            for (int j = 0; j < activeGraph.nodes[i].neighbors.Count; j++)
            {
                //Find closest entrances
                GeneratorRoom startRoom = placedRooms[(int)i];
                GeneratorRoom endRoom =
placedRooms[(int)activeGraph.nodes[i].neighbors[j]];

                //We declare some variables to hold info of the closest
                //entrances
                float lowestDistance = float.MaxValue;
                Entrance selectedEntranceStart = startRoom.entrances[0];
                Entrance selectedEntranceEnd = endRoom.entrances[0];
                Vector2Int selectedStartPos = WorldToGrid(positions[i]);
                Vector2Int selectedEndPos =
WorldToGrid(positions[activeGraph.nodes[i].neighbors[j]]);

                //We will iterate all the free entrances in both rooms until
                //we find the closest pair
                foreach (var startEntrance in startRoom.entrances)
                {
                    if (startEntrance.unlocked) continue;

                    foreach (var endEntrance in endRoom.entrances)
                    {
                        if (endEntrance.unlocked) continue;

                        Vector2Int currentStartPos =
WorldToGrid(positions[i]) + startEntrance.localPosition;
                        Vector2Int currentEndPos =
WorldToGrid(positions[activeGraph.nodes[i].neighbors[j]]) +
endEntrance.localPosition;

                        float currentDistance =
Vector2.Distance(currentStartPos, currentEndPos);
                        if (currentDistance < lowestDistance)
                        {
                            lowestDistance = currentDistance;
                            selectedEntranceStart = startEntrance;
                            selectedEntranceEnd = endEntrance;

                            selectedStartPos = currentStartPos;
                            selectedEndPos = currentEndPos;
                        }
                    }
                }

                //Once the closest pair has been selected, we will mark it
                //as unlocked and remove its blockage
                selectedEntranceStart.unlocked = true;
                selectedEntranceEnd.unlocked = true;
                Destroy(selectedEntranceStart.blockage);
                Destroy(selectedEntranceEnd.blockage);

                // ...
            }
        }
    }
```

With the rooms prepared, we will execute a pathfinding algorithm to know the route the corridor has to take to reach from one entrance to another, avoiding rooms and other corridors. The algorithm used for it will be A*, given our previous experience with it, and being one of the most used for videogame development. Developing an algorithm which worked on top of the current implementation, debugging and optimizing it would be very time consuming, so some research was conducted to find an existing implementation. A C# repository by Val Antonini has been found with an extensive implementation with many options, so all that is needed is to integrate it into the project.

https://github.com/valantonini/AStar

The actual use of this implementation is very simple. You first define the space where the algorithm will be run using a matrix of values, then set the options for the pathfinder, initialize the pathfinder, and lastly call a function which will return an array of positions defining the path based on the start and the end positions you pass it as parameters.

```CSharp
        // ...
        Destroy(selectedEntranceStart.blockage);
        Destroy(selectedEntranceEnd.blockage);

        //Set up the environment for the Pathfinder
        short[,] tiles = MakeTilesFromGrid(grid);

        //Configure its options
        var pathfinderOptions = new AStar.Options.PathFinderOptions
        {
            PunishChangeDirection = true,
            UseDiagonals = false,
            Weighting = AStar.Options.Weighting.Negative,
        };

        //Initialize it
        var worldGrid = new WorldGrid(tiles);
        var pathfinder = new PathFinder(worldGrid, pathfinderOptions);

        //Calculate the starting and goal positions
        Vector2Int pathFindingStart = selectedStartPos +
GetDirectionCoords(selectedEntranceStart.direction);
        Vector2Int pathFindingEnd = selectedEndPos +
GetDirectionCoords(selectedEntranceEnd.direction);

        //Execute the algorithm
        Position[] path = pathfinder.FindPath(new
Position(pathFindingStart.x, pathFindingStart.y),
                                              new Position(pathFindingEnd.x,
pathFindingEnd.y));

        if (path.Length <= 0)
        {
            Debug.LogWarning("Could not find suitable path");
        };

        // ...
```

Since the implementation itself is pretty self-contained and straightforward, we only need to declare the tiles where it will be run properly, but the rest is quite self-explanatory. To do so, we will create a matrix of “shorts” as the program demands, using zeros to indicate that the cell is occupied by a room, ones to indicate that the cell is free, and fifties to indicate that the cell is occupied by an already placed corridor. This differentiation for corridors will be useful and explained later on when we talk about subways in section 5.2.4.

```CSharp
short[,] MakeTilesFromGrid(Cell?[,] grid)
    {
        short[,] returnTileMap = new short[gridSize * 2, gridSize * 2];
        for (int i = 0; i < gridSize * 2; i++)
        {
            for (int j = 0; j < gridSize * 2; j++)
            {
                if (grid[i, j] != null && grid[i, j].type == Corridor)
                {
                    returnTileMap[i, j] = 50; //Corridor
                }
                else if (grid[i, j] != null)
                {
                    returnTileMap[i, j] = 0; //Room (Subway entrance)
                }
                else
                {
                    returnTileMap[i, j] = 1; //Free cell
                }
            }
        }

        return returnTileMap;
    }
```

Once we know the path the corridor will take, we can work on filling the grid and placing the actual geometry, as we did with the rooms. We will iterate the pathfinder positions list to define each corridor cell. The corridor models can be either straight or a shoulder, but there is only one model for each, so we need to determine its orientation. To do so, we will modify the Cell struct to store a direction, using the same structure as in the entrances.

```
CSharp
public struct Cell
{
    public NodeType type;
    public Direction? direction;

    public Cell(NodeType type)
    {
        this.type = type;
        direction = null;
    }

    public Cell(NodeType type, Direction direction)
    {
        this.type = type;
        this.direction = direction;
        }
    }
```

We will fill the grid with a Cell of a “Corridor” type, and we will determine the direction of the current Cell by comparing the position of the next cell in the path with the position of our current Cell. Then the “GetPathCellDirection” function simply returns a Direction struct based on the direction vector we provide to ease readability.

```csharp
        //...

        //We traverse the positions list to set all of them
        for (int k = 0; k < path.Length; k++)
        {
            Vector2Int currentPos = new Vector2Int(path[k].Row,
path[k].Column);

            //We fill the grid with directions comparing with the next
            //position
            if (k == path.Length - 1)
            {
                Vector2Int dir = selectedEndPos - currentPos;
                grid[currentPos.x, currentPos.y] = new
Cell(NodeType.Corridor, GetPathCellDirection(dir));
            }
            else
            {
                Vector2Int dir = new Vector2Int(path[k + 1].Row -
currentPos.x, path[k + 1].Column - currentPos.y);
                grid[currentPos.x, currentPos.y] = new
Cell(NodeType.Corridor, GetPathCellDirection(dir));
            }

            // ...
        }
```

Knowing the position and the direction of the corridor cell we are evaluating, the next step is placing the corresponding corridor prefab. This process isn’t as simple as it might seem, because having only one model for shoulders and one for straight lines means that we will have to rotate them and re-adjust their position to fit the cell we are evaluating. This is done through the “GetCorridorShape” function.

To be able to decide which prefab we need and how we should place it, we will need to evaluate the current and previous directions, and return the needed prefab while also outputting the position adjustment and rotation adjustment they will need. If the current direction and the previous one are the same, it means that the current corridor will be a straight one. If it's different, the corridor will be a shoulder. However, as previously mentioned, depending on the directions of the two cells after checking whether the corridor is straight or a shoulder, the parameters for the adjustments will be defined.

```csharp
Object GetCorridorShape(Vector2Int currentPos, Vector2Int previousPos, out
Vector3 positionAdjustment, out Quaternion rotationAdjustment, Direction?
firstCellDirection = null)
    {
        //Get positions' directions
        Direction? currentDir = grid[currentPos.x, currentPos.y].direction;
        Direction? previusDir = grid[previusPos.x, previusPos.y].direction;

        if (firstCellDirection != null)
            previousDir = firstCellDirection;

        //Straight path
        if (currentDir == previousDir)
        {
            //Vertical
            if (currentDir == North || currentDir == South)
            {
                positionAdjustment = Vector3.left * 5;
                rotationAdjustment = Quaternion.Euler(0, 90, 0);
            }
            //Horizontal
            else
            {
                positionAdjustment = Vector3.zero;
                rotationAdjustment = Quaternion.identity;
            }

            return straightCorridor;
        }
        //Shoulder path
        else
        {
            //Up-right
            if (previousDir == North && currentDir == East ||
                previousDir == West && currentDir == South)
            {
                positionAdjustment = Vector3.left * 5;
                rotationAdjustment = Quaternion.Euler(0, 90, 0);
            }
            //Down-left
            else if (previousDir == South && currentDir == West ||
                     previousDir == East && currentDir == North)
            {
                positionAdjustment = Vector3.forward * 5;
                rotationAdjustment = Quaternion.Euler(0, -90, 0);
            }
            //Up-left
            else if (previousDir == North && currentDir == West ||
                     previousDir == East && currentDir == South)
            {
                positionAdjustment = Vector3.forward * 5 + Vector3.left * 5;
                rotationAdjustment = Quaternion.Euler(0, 180, 0);
            }
            //Down-right
            else
            {
                positionAdjustment = Vector3.zero;
                rotationAdjustment = Quaternion.identity;
            }

            return shoulderCorridor;
        }
    }
```

With the prefab selected, we just need to Instantiate it, but taking into account the position and rotation adjustments.

```CSharp
                    //Instantiate
                    Vector3 posToAdd;
                    Quaternion rotation;
                    Object corridorToInstantiate;

                    //Get the corridor shape from the return of the function
                    //and fill the adjustemnts passing them as parameters
                    if (k == 0)
                    {
                        corridorToInstantiate = GetCorridorShape(grid,
currentPos, currentPos, out posToAdd, out rotation,
selectedEntranceStart.direction);
                    }
                    else
                    {
                        corridorToInstantiate = GetCorridorShape(grid,
currentPos, new Vector2Int(path[k - 1].Row, path[k - 1].Column), out
posToAdd, out rotation);
                    }

                    //Instantiate the corridor accounting for the
                    //adjustments
                    Instantiate(corridorToInstantiate,
GridToWorld(currentPos.x, currentPos.y) + posToAdd, rotation);
```

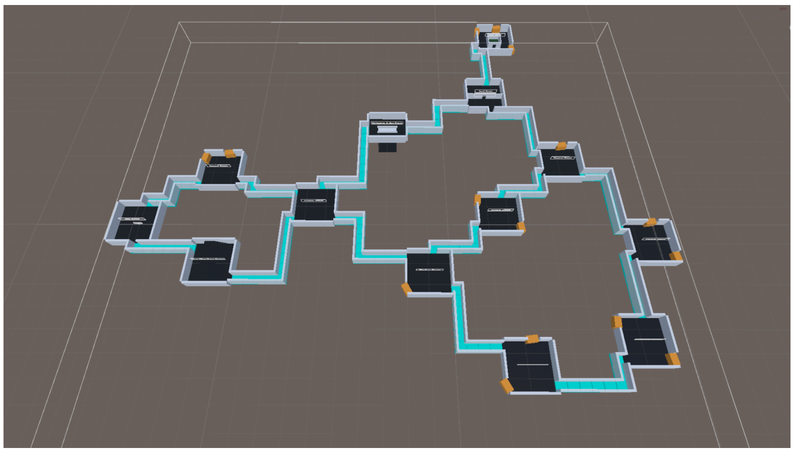


Figure 26: Corridors placed with their grid representation.

5.2.4. Subways implementation

With the current implementation of rooms and corridors, there is a problem. When we decided to set the value of the corridor cells to 50 in the Pathfinder, we were indicating that the cells with corridors are 50 times less likely to be used for the path, since we want to avoid crossings. This would break the flow of the level and the graphs we designed in all of section 5.1 would lose all their meaning. However, that leaves room for crossings if the algorithm is unable to find a path which does not cross any other (it is better to have a crossing path than to not have one at all). To solve this problem, the proposed solution was to use the third dimension, since we are adapting the whole generation approach to 3D after all. A system for generating subways in these cases was designed, and it works as follows.

Instead of using just one grid to execute the pathfinding algorithm we will use two, one for the ground level and one for the subways. When we try to find a path for the corridor (as shown in section 5.2.3 code snippet 3), we will check whether the path contains a 50 value in it, and if so, we will continue the path creation using the implementation for tunnels instead of the base one, and once finished we skip to the next iteration of the neighbors of a node, so the next corridor.

```
CSharp
        // ...
        //First part of PlaceCorridors function

        //We exacute pathfinding as usual
        Position[] path = pathfinder.FindPath(new
Position(pathFindingStart.x, pathFindingStart.y),
                                                      new
Position(pathFindingEnd.x, pathFindingEnd.y));

        //Before moving on to place the prefabs for the cells, we check if
        //there is a corridor cell in the path
        bool underground = false;
        foreach (var item in path)
        {
            if (tiles[item.Row, item.Column] == 50)
            {
                Debug.Log("Corridor must be underground");
                PlaceTunnel(selectedEntranceStart, selectedEntranceEnd,
selectedStartPos, selectedEndPos);
                underground = true;
            }
        }
        //When we finish, we skip the rest of this corridor's execution
        if (underground) continue;

        //Setting individual path cells
        for (int k = 0; k < path.Length; k++)
        {
            // ...
        }
```

The subway implementation works a little differently than the normal corridors. First we need to get some information about the room entrance to the subway, where the pathfinding will start with respect to the room entrance, the orientation of the subway model among others.

```
CSharp
struct SubwayInfo
    {
        //cell the pathfinding uses with respect to the entrance position
        public Vector2Int pathfindingCellDelta;
        //start of the space the subway occupies on the grid (world coords)
        public Vector2Int gridFillerPivot;
        //amount you must move from entrance to snap to grid after rotating
        public Vector2Int positionDelta;
        //the current rotation of the subway
        public float rotation;
        //Whether the current rotation positions it horizontally or not
        public bool horizontal;

        public SubwayInfo(Vector2Int pathfindingCellDelta, Vector2Int
gridFillerPivot, Vector2Int positionDelta, float rotation, bool horizontal)
        {
            this.pathfindingCellDelta = pathfindingCellDelta;
            this.gridFillerPivot = gridFillerPivot;
            this.positionDelta = positionDelta;
            this.rotation = rotation;
            this.horizontal = horizontal;
        }
    }
```

Having the information about the subway entrances, we can now Instantiate the model in the proper position and rotation, while filling the subway grid with the appropriate info to then run the pathfinding on it. We will be using the “Room” type to differentiate it from the corridor cells. We must take into account that this has to be done for both the start and end of the corridor, since otherwise it would result in an incomplete passage.

```CSharp
void PlaceTunnel(Entrance startingEntrance, Entrance endingEntrance,
Vector2Int startEntrancePos, Vector2Int endEntrancePos)
    {
        //We generate the needed info for the subway based on the position
        //and direction of the room's entrance
        SubwayInfo startInfo = GetSubwayInfo(startingEntrance.direction,
startEntrancePos);

        //Then we instantiate and fill the subway grid as with any other
        //room but taking into account the subway adjustments
        Vector2Int startGlobalPos = startEntrancePos +
startInfo.positionDelta;
        Instantiate(undergroundEntrance, GridToWorld(startGlobalPos.x,
startGlobalPos.y), Quaternion.Euler(0, startInfo.rotation, 0));

        for (int i = 0; i < 3 + (startInfo.horizontal ? 1 : 0); i++)
        {
            for (int j = 0; j < 3 + (startInfo.horizontal ? 0 : 1); j++)
            {
                subwayGrid[startInfo.gridFillerPivot.x + i,
                           startInfo.gridFillerPivot.y + j] =
                           new Cell(NodeType.Room);
            }
        }

        //We do the same process for the subway entrance at the end of the
        //corridor
        SubwayInfo endInfo = GetSubwayInfo(endingEntrance.direction,
endEntrancePos);

        Vector2Int endGlobalPos = endEntrancePos + endInfo.positionDelta;
        Instantiate(undergroundEntrance, GridToWorld(endGlobalPos.x,
endGlobalPos.y), Quaternion.Euler(0, endInfo.rotation, 0));

        for (int i = 0; i < 3 + (endInfo.horizontal ? 1 : 0); i++)
        {
            for (int j = 0; j < 3 + (endInfo.horizontal ? 0 : 1); j++)
            {
                subwayGrid[endInfo.gridFillerPivot.x + i,
                           endInfo.gridFillerPivot.y + j] =
                           new Cell(NodeType.Room);
            }
        }

        // ...
```

Then, we can run the pathfinding algorithm as normal, but we just need to make sure to run it on the subway grid and not the normal one. After that, all of the logic for setting the direction, making the calculations for which prefab we need and so on stay the same, but for one exception. The GetCorridorShape function would return a normal corridor if we kept it the same, but the subway corridors are different to the ones on the surface. To make sure that we will return the proper prefabs, we will make a modification to the function. We will add a parameter for which grid we will be checking, and we will read the directions from that. Additionally, when returning the prefab, we will return a corridor or subway prefab based on whether the grid we passed it is the surface or the subway one.

```CSharp
Object GetCorridorShape(Cell?[,] gridToCheck, Vector2Int currentPosition,
Vector2Int previousPosition, out Vector3 positionAdjustment, out Quaternion
rotationAdjustment, Direction? firstCellDirection = null)
    {
        //Read the directions from the grid parameter
        Direction? currentDirection =
            gridToCheck[currentPosition.x, currentPosition.y].direction;
        Direction? previousDirection =
            gridToCheck[previousPosition.x, previousPosition.y].direction;


        // ..

        //Straight path
        if (currentDirection == previousDirection)
        {
            // ...

            //Return prefabs depending on which grid was passed
            return (gridToCheck == grid)
            ? straightCorridor
            : straightUnderground;
        }
        //Shoulder path
        else
        {
            // ...

            //Return prefabs depending on which grid was passed
            return (gridToCheck == grid)
            ? shoulderCorridor
            : shoulderUnderground;
        }
    }
```

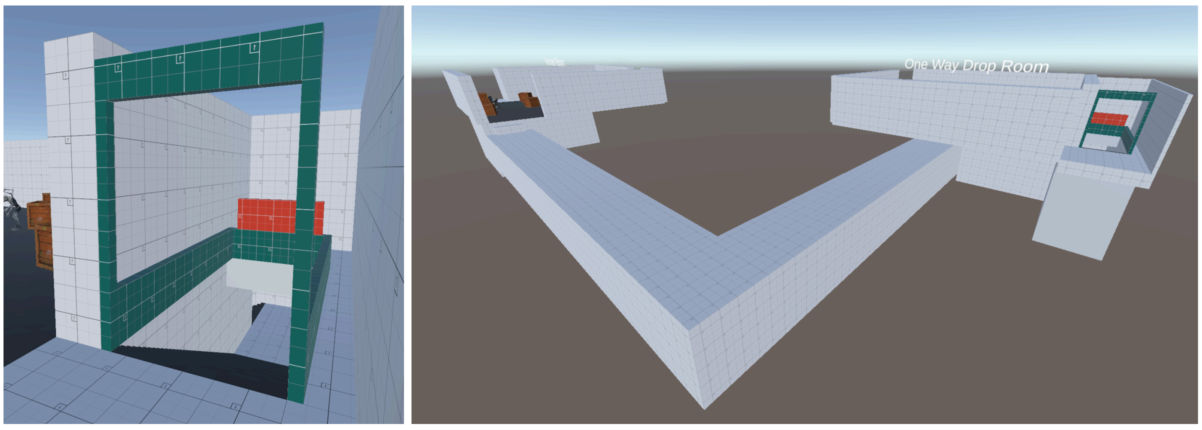


Figure 27: Subway system implementation.

5.3 Network implementation

As stated in section 1.3, this algorithm is to be implemented in an ongoing project for a multiplayer extraction shooter. This means there are some considerations to take into account and some modifications that must be done in the code.

One of the most important aspects is that in order for all the players to generate the same level, they must share a seed. For this, the “LevelManager” class inside the level, will generate a seed randomly only if the player executing it is the host of the session, and this seed variable will be synchronized among all the players. With the seed generated, all players will start the level generation using the synchronized seed.

```csharp
CSharp
public override void OnNetworkSpawn()
    {
        if (IsServer)
        {
            //Server generates seed which will be synced among all the
            //players
            seed.Value = Random.Range(int.MinValue, int.MaxValue);
        }

        //Subscribe to the "SceneLoaded" event to start the Level Generation
        NetworkManager.SceneManager.OnLoadEventCompleted += (sceneName, _,
_, _) =>
        {
            if (sceneName == "LevelGenerator")
                GraphGenerator.StartLevelGeneration(seed.Value);
        };
    }
```

With the current network implementation, all entities that must be synced between players must contain a “NetworkObject” script. These objects must be located on the root of the scene, so they can not be placed under the room prefabs even if they are boxes, destructible cars or other. What we will place instead will be spawners, which will be empty objects with a script indicating which object they should spawn in that position. This spawning process must be done once the level generation is complete, so we will declare an event to invoke it once the generation is complete, so any class that needs it can subscribe to it and execute these assignments or any others as needed. We will also need to generate a NavMesh on top of the level so that enemy AI can navigate the space.

```csharp
CSharp
void GenerateLevel()
    {
        activeGraph = graphGenerator.activeGraph;

        //Generate layout positions
        Dictionary<uint, Vector3> positions = GenerateLayoutPositions();

        List<GeneratorRoom> placedRooms = PlaceRooms(positions);

        PlaceCorridors(placedRooms, positions);

        //The server will generate the NavMesh when the level is finished
        if (LevelManager.Singleton.IsHost)
        {
            GetComponent<NavMeshSurface>().BuildNavMesh();
        }

        //We will also call all of the subscribers to the event
        OnLevelGenerationComplete?.Invoke();
    }
```

Some examples of the classes that will subscribe to the event and must be adapted are the PlayerSpawner class and the general EntitySpawner.

```CSharp
//PlayerSpawner
void SceneLoaded(string sceneName, LoadSceneMode loadSceneMode, List<ulong>
clientsCompleted, List<ulong> clientsTimedOut)
    {
        if (IsHost && sceneName == "LevelGenerator")
        {
            //When we enter the scene the host subscribes to
            //OnLevelGenerationComplete
            LevelGenerator.OnLevelGenerationComplete += () =>
            { OnLevelGenerated(clientsCompleted); };
        }
    }

    void OnLevelGenerated(List<ulong> clientsCompleted)
    {
        foreach (ulong id in clientsCompleted)
        {
            GameObject player = Instantiate(Player);
            player.GetComponent<NetworkObject>().SpawnAsPlayerObject(id,
true);
        }

        //Once we have created all the players, we can un-subscribe the
        //function
        LevelGenerator.OnLevelGenerationComplete -= () =>
        { OnLevelGenerated(clientsCompleted); };
    }
```

CSharp
```csharp
//EntitySpawner
void SceneLoaded(string sceneName, LoadSceneMode loadSceneMode, List<ulong>
clientsCompleted, List<ulong> clientsTimedOut)
    {
        if (IsHost && sceneName == "LevelGenerator")
        {
            //Same as with PlayerSpawner, we subscribe it
            LevelGenerator.OnLevelGenerationComplete += OnLevelGenerated;
        }
    }

    void OnLevelGenerated()
    {
        SpawnEntityWithTag("AmmoBoxSpawn", AmmoBox);
        SpawnEntityWithTag("AssaultRifleSpawn", AssaultRifle);
        SpawnEntityWithTag("BoxSpawn", Box);
        SpawnEntityWithTag("CarSpawn", Car);
        SpawnEntityWithTag("CarAlarmSpawn", CarAlarmed);
        SpawnEntityWithTag("DoorSpawn", Door);
        SpawnEntityWithTag("MedkitSpawn", Medkit);
        SpawnEntityWithTag("ShotgunSpawn", Shotgun);
        SpawnEntityWithTag("Horde_BasicZombieSpawn", Horde_BasicZombie);
        SpawnEntityWithTag("Horde_FastZombieSpawn", Horde_FastZombie);

        //And when all of the Entities have been spawned, we unsubscribe it
        LevelGenerator.OnLevelGenerationComplete -= OnLevelGenerated;
    }
```

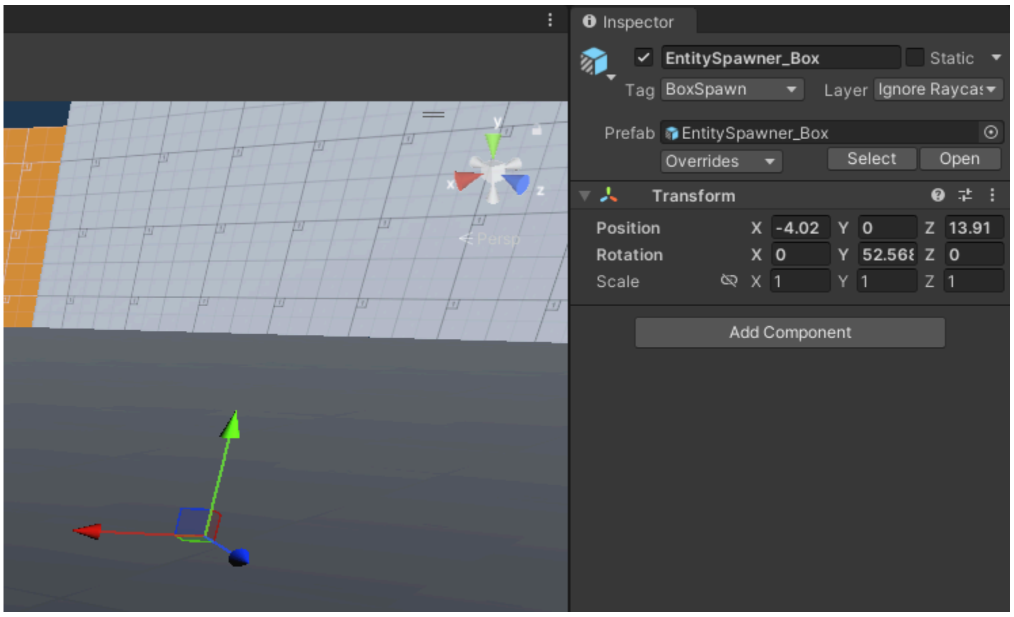

Inspector
EntitySpawner_Box
Static
Tag BoxSpawn
Layer Ignore Raycas
Prefab EntitySpawner_Box
Overrides
Select
Open
Transform
Position X -4.02 Y 0 Z 13.91
Rotation X 0 Y 52.568 Z 0
Scale X 1 Y 1 Z 1
Add Component

Figure 28: Placing an entity spawner in a room prefab.

6. Validation

As specified in section 4, to validate the project a survey was to be conducted. After the distribution of the survey, twenty-one people filled the Google Forms, gathering some feedback for the generator. Each section of the survey consisted of a question and a comment box so that the testers could provide written feedback. The questions surveyed and their answers will be discussed in this section.

Question 1: Do you think the layout of the level resembles more a branching structure or a cyclic structure?

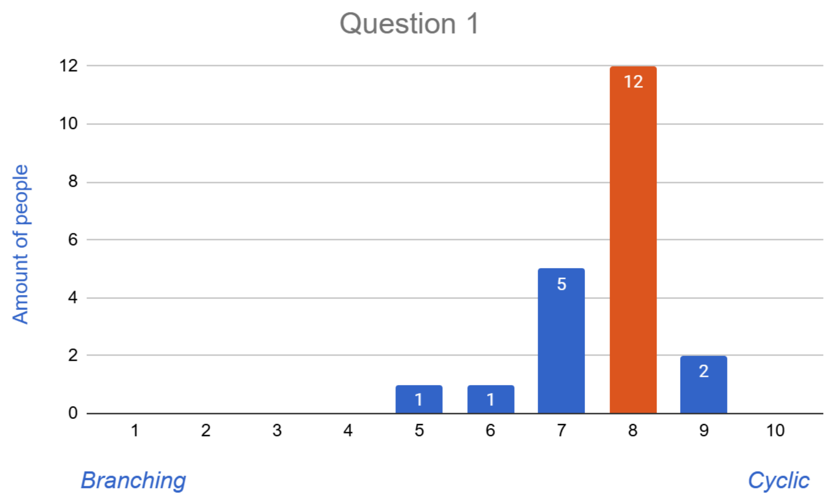


Figure 29: Question 1 answers.

From the first graph we can deduce that the overall objective has been achieved since, in general, most testers agree that the generator is more cyclic than branching. We can even discard the one in number 5, since one comment states they are unsure of the difference. However, one of the comments mentions the following:

"There weren't any ends (what you expect of a branching tree structure), but the cyclic aspect wasn't apparent either. It felt more like all the rooms connected to all the rooms nearby and that's it."

From this comment we can assume that the answer was given by one of the high voters, but the second part of the comment must be addressed too. Since the current implementation only features white squared prototype rooms, maybe there should be more effort put in conveying the intention of each room and their flow through the level design in the future.

Question 2: Have you found any Subways? If so, could you please give your opinion on it?

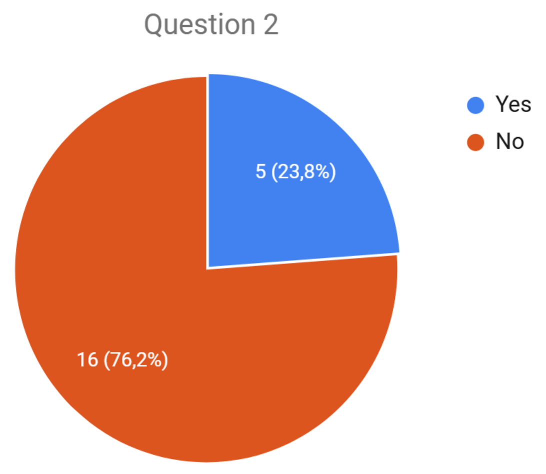


Figure 30: Question 2 answers.

This question confirms what was already suspected and intended. Subways are not a core part of the algorithm that will exist in every generation, but a safety measure added to prevent corridor crossings in case they happen. There were no relevant comments made in this section further than a general consensus that it is an interesting feature.

Question 3: Did you go through the same room more than once? If so, was it intentional?

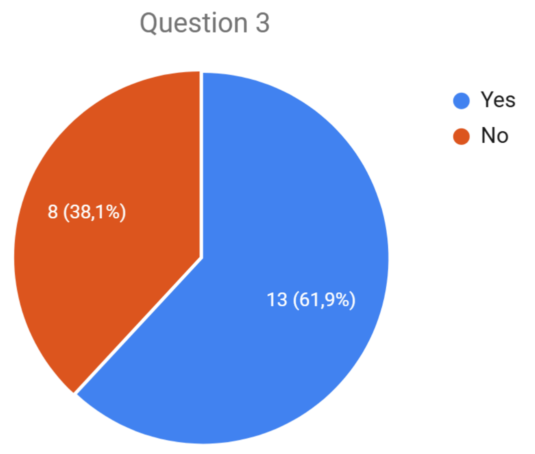


Figure 31: Question 3 answers.

Comments for this question mention that without using the free cam feature implemented for the demo build they would have probably done so and potentially got lost.

"*Without looking at the level beforehand I probably would have, but seeing the map made clear which path to take.*"

Seeing how this feature was useful to the playtesters, it could be interesting to discuss the possible addition of a minimap or map of sorts to help the player navigate the space. This was not intended to be a part of the game, so it would be important to evaluate how this affects the overall experience to see if it aligns with the original intended experience.

Question 4: Did you get lost?

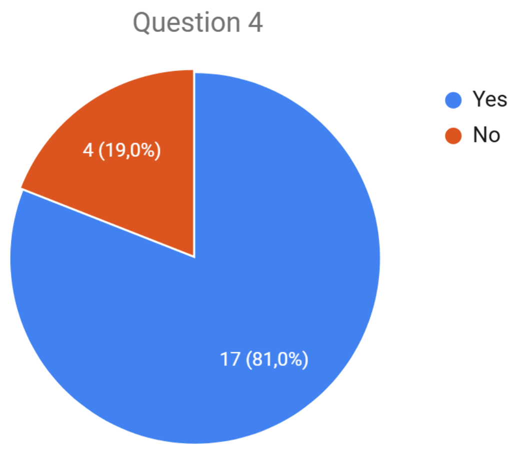


Figure 32: Question 4 answers.

This question is where it gets a bit worrying. Most playtesters got lost navigating the space. This reinforces the previous question's conclusions and is a problem that should definitely be addressed.

"Yes, but more because of the similar rooms and multiple paths rather than the level layout."

However, testers attribute this fact to the rooms and paths being similar in nature. This reaffirms the need for more variety of rooms or prefabs for the same type of node, since for the time being there is only one for each type. It would be of utmost relevance to do another round of playtesting with this question when more variety of rooms have been introduced and the different sections of the map are more recognisable.

Question 5: Is the scale of the map appropriate?

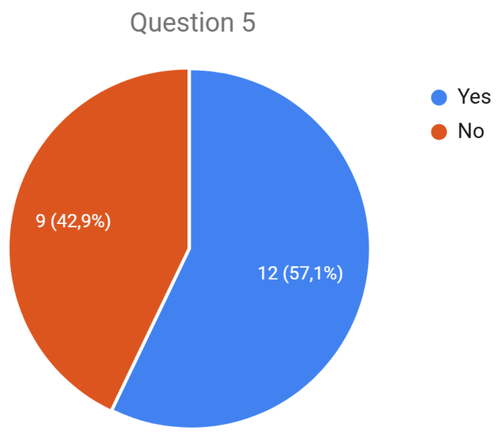


Figure 33: Question 5 answers.

More than half the testers agree that the map has an appropriate size, but those who don't say something along the lines of:

"The corridors are way too long sometimes, and the movement is a bit slow."

"The map is too big and there is nothing to do."

This demo was empty on purpose, but when it comes to the actual size of the map, the forces used to untangle the graph could be adjusted to try and generate layout positions which are closer to one another, while still maintaining some space between the rooms to generate the corridors which connect them.

Question 6: Did you get stuck at any point (as in not knowing how to advance)?

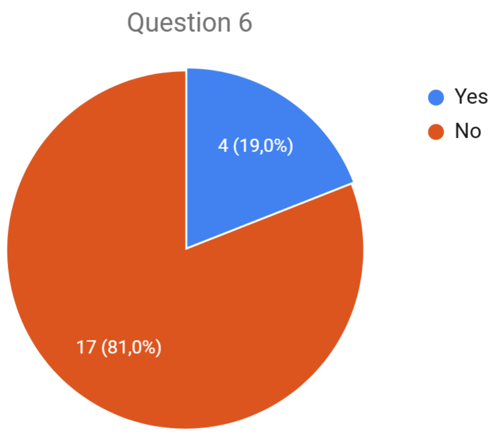


Figure 34: Question 6 answers.

Despite the answers to the previous questions, the testers in general knew what they had to do at all times. Even though they might have gotten lost at some point, they still knew that if they kept on walking they would eventually find the goal room.

"No, but I can see myself getting lost and genuinely thinking I got 100% stuck"

This comment also aligns with the previous questions. It may seem at first then, that the main problem with the generator in general is the lack of easily recognizable rooms and landmarks so that players could locate themselves in the level, together with not communicating clearly enough the intended flow of the level.

Question 7: Did you find yourself unable to advance due to an issue in the level (walls where there shouldn't be, gaps in the map)?

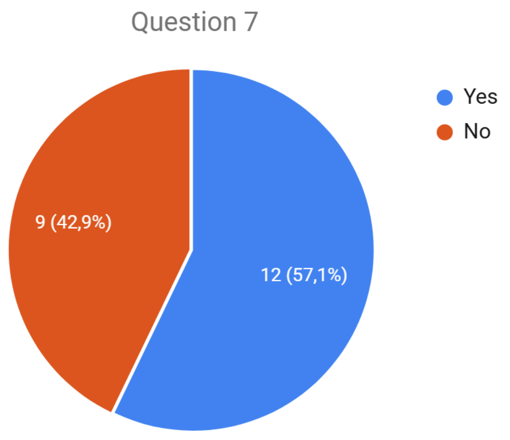


Figure 35: Question 7 answers.

The answers to this question might be the most worrying of all. More than half the people reported to have found walls, gaps or others in their way blocking their path.

"Fell off the map because of a gap in the wall in a normal room."

During development, some bugs have appeared that have been fixed before releasing the demo alongside the survey. The results come to indicate that testing has not been thorough enough, and even if the testers answered "Yes" after finding errors in one of the few generations they might have attempted, the percentage is rather high and is the problem which should be addressed with the highest priority.

General Comments

At the end of the survey, a section was added for testers to provide general feedback. These are some of the answers.

"I believe that I had bad luck the first two times that I opened the app, because the room next to the start room was the goal room. After answering your survey, I restarted it once more because I was curious about the subway and the other elements you mentioned. I got the seed 1164701588, and it was so much more interesting! I got a subway, a collapsing bridge, a key room, a locked door and a one way room. It was so intriguing and I got really engaged."

"I know that the project is not supposed to focus on level design in the "good gameplay experience" way, but I think the generator can generate levels that feel really unfair or boring if you choose the "wrong" path or extremely easy if you get lucky."

What these comments tell us, as previously suggested, is that there must be more care put towards the design of the levels, but in this case the graph layouts of them. Having only two possible final arrangements can make it so that even if a player plays the game more than once they always get the same layout, and not being careful enough when designing these possible layouts or combinations of configurations can completely break the intended experience.

7. Lines of future

Given that the time to develop this project is limited, there are things which could not be finished and many that could have been added or improved.

One of the things that would cause the most impact would be to add more rules, conditions and, in general, more possible configurations. This wouldn't be difficult nor require changes in the code, given that the pipeline for loading graphs from JSONs was built precisely for that. All it would require would be to design and test new ideas and see how the generator combines them, and the possible combinations would grow exponentially, due to the generation system being modular.

On the other hand, adding more room prefabs for each node type would give the generated levels more variety as well as help solve some of the problems encountered during playtesting. This would require a slight modification of the room selection code, but this wouldn't take long nor affect negatively the ease of use of the implementation.

As an addition to designing more rooms, the possibility of rotating rooms when placing them could be implemented. This would not only give the generated levels more variety, but also fix some current problems with softlocks or intended blocks or conditions to be missed altogether. This implementation would be a bit harder since rotations would have to be implemented not only for the Instantiated models but for the grid information as well.

Closing on the last days of development, a change in sizing was attempted. Corridors and subways were doubled in size, and that made the whole experience much less claustrophobic, more enjoyable and in general more fun to play. This implementation was not finished and thus temporarily discarded because the change required modifying the sizes of the grid and more importantly its cells, and even though the general change can be made rather quickly, testing it and checking it didn't introduce new bugs or produce unexpected results would be more time consuming. Trying to fully implement it however, has been proved to greatly improve the overall experience of combat and exploration. Pairing it with the survey's suggestion of closing the rooms together might further increase its potential.

To fully finish the generator, the gameplay features related to the different rooms such as the collapsing one way bridge or the key-lock combinations should be implemented. These implementations are a bit more tricky, since the multiple implementations should be tailored to the different types of rooms, and they would depend on many possible factors which must be taken into account. It is something that has to be thought and planned thoroughly in order to make it work, but it would get the prototype much closer to being a proper prototype with complete and enjoyable gameplay.

As a last observation, the visual aspect of the used rooms and corridors can be improved. Whether it is through the use of store assets or custom-made ones, any thing that we add to the rooms will significantly enhance this aspect, especially if we take into account that the current implementation only features prototype colored boxes. On top of that, some thorough testing and bugfixing must be conducted to fix the current errors of the generator and any others that might be found.

8. Conclusion

Overall, the initial objectives for this project have been accomplished. Despite having reached most of the original milestones, what has been achieved would better fit the description of a very solid starting point with lots of potential.

At the beginning of the project, an in-depth study of the generation process of “Unexplored” was carried out, laying the foundation for what would later be the achieved procedural graph generator together with the graph loading and rule creation pipeline. This pipeline allows designers to easily create new graphs, combinations and rules on how and when the graphs should evolve without the need of modifying a single line of code.

On top of the achieved graph generator, a whole level generator was built from scratch to bring the generated graphs to life. Nodes would be placed in the level as rooms, and edges would take the shape of corridors. A subway system was also created as a safety measure to ensure there was always a path from one room to another without disrupting the intention of the designer, even if there were no possible routes on the surface.

The algorithm was successfully integrated in an already existing on-going multiplayer project, and slight additions were made to handle server calls, event callbacks, synchronized maps and entities, amongst others. The prototype is fully functional and has reached a satisfactory state. It can be completed from beginning to end, however there is much room for improvement, as stated at the beginning of this section.

On the personal side of things, I am satisfied with how much we have achieved with this project, but I would like to continue developing it seeing the potential it has come to have. Before starting to research the topic, the project was going to be about creating a procedural generation algorithm that maximized the complexity of the generated levels, but following existing approaches and taking as a reference games with some of the best procedural levels. However, while conducting my investigation I found Unexplored, and I immediately saw a gap in the market with tons of untapped potential.

Certain parts of the process have been really hard for me such as all of the graph theory related concepts or generative grammars at the beginning of all this, especially since I am not a maths nor a linguistics major. However, reading papers and trying to bring unknown concepts to my terrain, computation, has made me find a new way of absorbing knowledge I would have probably not found had I not exposed myself to this challenge.

9. References

- Inigo Quilez. (2022, April 28). Domain Warping [Blog Post]. *Domain Warping*. https://iquilezles.org/articles/warp/
- Murray, S. (2017, April). *Building Worlds in No Man's Sky Using Math(s)* [GDC Talk]. GDC, San Francisco, California. https://www.youtube.com/watch?v=C9RyEiEzMiU
- Kazemi, D. (n.d.). Spelunky Generator Lessons [Interactive Website]. *Spelunky Generator Lessons*. https://tinysubversions.com/spelunkyGen/
- 'Boris'. (2020, December 9). Dungeon Generation in Binding of Isaac [Blog Post]. *Dungeon Generation in Binding of Isaac*. https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/
- Brewer, D. (2013). *Managing Pacing in Procedural Levels in Warframe*. 13. https://www.gameaipro.com/GameAIProOnlineEdition2021/GameAIProOnlineEdition2021_Chapter07_Managing_Pacing_in_Procedural_Levels_in_Warframe.pdf
- skillet_gamer. (2021, April 1). Randomly Generated Worlds [Blog Post]. *Remnant: From the Ashes - Guide and Walkthrough (PC)*. https://gamefaqs.gamespot.com/pc/243491-remnant-from-the-ashes/faqs/78376/randomly-generated-worlds
- Joris Dormans. (2010, June). *Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games*. Hogeschool van Amsterdam. https://pcgworkshop.com/archive/dormans2010adventures.pdf
- Cyclic Generation. (2017). In Dormans, Joris, *Procedural Generation in Game Design*. CRC Press.
- Electricity in Spain in 2024. (2024). [Blog Post]. *Electricity in Spain in 2024*. https://lowcarbonpower.org/region/Spain
- Ed Mitchell. (2010, April 20). How much energy do we use on the web? [Blog Post]. *How Much Energy Do We Use on the Web?* https://transitionnetwork.org/news/how-much-energy-do-we-use-on-the-web/

- Stephanie Safdie. (2024, December 20). What is the Carbon Footprint of Data Storage? [Blog Post]. *What Is the Carbon Footprint of Data Storage?* https://greenly.earth/en-us/blog/industries/what-is-the-carbon-footprint-of-data-storage
- Vaughn Cato. (2020). *Ullman Algorithm Implementation* [Python]. https://stackoverflow.com/questions/13537716/how-to-partially-compare-two-graphs/13537776#13537776
- Simon Tatham. (2025). *Untangle* [JavaScript]. https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/untangle.html
- Val Antonini. (2024). *AStar* (Version v1.3.0-beta.1) [C#]. https://github.com/valantonini/AStar

10. Annex

- Table 1. Gantt chart:
  https://docs.google.com/spreadsheets/d/1Z9rSohppWlxwr_u0iJE2U4LaPiTI0Tum_58moOd6rfk/edit?gid=1685557521#gid=1685557521

- Tables 4 and 5. Budget and environmental impact (different sheets):
  https://docs.google.com/spreadsheets/d/1Xzdzu9yDNPU_Myh7k51sXG02TkTwKJqPk3Qt89qqRNA/edit?gid=1306094701#gid=1306094701

- Survey:
  https://forms.gle/RR7T6hLJvtLEh7Ez6