Procedural generation:

Known methods and applications for video game development

Version 1.2

Demonstration of LaserCave: https://lucinotion.itch.io/lasercave

By: *Lucinotion*

Originally published on 2020-07-13, last update: 2022-03-25

Table of Contents

Table of Illustrations

1. SUMMARY

"LaserCave" is a singleplayer roguelike game that combines puzzles with combat mechanics. In LaserCave you control VIC, a robot trapped in the depths of a procedurally generated cave who tries to escape to the surface. To achieve this, she will have to fight a lot of different enemies and solve the uncountable puzzles that she finds along her way to the exit.

The game is directed towards gamers that enjoy roguelike games like "Enter the Gungeon" and puzzle games like "Portal" in platforms like PC and console. LaserCave is a fast paced game that offers an endless game mode like most games of the genre without renouncing a more narrative experience with NPCs and a main storyline.

Keywords:

Procedural generation, mazes, dungeons, roguelike, puzzles.

2. INTRODUCTION

The LaserCave project began during my third year at university, when I became interested in the area of procedural generation. I decided that it would be a good idea to continue it during fourth year with the development of a game based on the procedural generation of environments as the subject for my Final Degree Project, so that I could study more about procedural generation methods and how to apply them to a game.

Procedural generation is a field of computer science which I want to master; in the future I do not seek to work with large development teams, but prefer to create video games independently. This is why procedural generation is key when it comes to saving effort and time.

LaserCave as a game, on the other hand, seeks to be an entertaining game that mixes logical elements with the fun of shooting up enemies. Originally the problem that I wanted to fix with LaserCave was the need to spend time creating new levels. But in the end what I really wanted to do with this game is add logical elements and narrative within the roguelike genre. Most roguelikes focus on giving you a lot of different weapons and different abilities, I wanted to do the opposite; in LaserCave you have 1 weapon which can be used to kill enemies in different ways. I believe that giving the player creativity in the use of a single weapon is much more fun than having the same mechanics for all weapons; the solution to monotony is not variety, it is the ability to do more with less, choose a simple concept, apply it, and let the player evolve the concept as much as they can.

Therefore, the expectations are:

- **Procedurally generated** content: The structure of the dungeons must be procedural so that the final game can be played indefinitely and thus can be re-playable many times.

- **Graphical editor of scenarios:** As part of the procedural generation, a visual level editor (or something similar), that facilitates the creation of complex rooms that require extra "human" logic to work.

- **Simplicity**: The game has 1 main mechanic and a series of sub-mechanics that revolve around it. The laser projectiles you fire bounce off walls, more bounces = more damage. There are different modifiers around the world that slightly alter the behavior of these projectiles, but the basis is that the rest of features only seek to expand the main gameplay mechanic.

- **Voxel style**: Pixel art is somewhat losing the strength it originally had within the indie game market. There are more and more developers who are jumping to make lowpoly games as an alternative. I think of voxels as the 3D equivalent of pixel art. The problem is that, they are difficult to use for modeling. This is because of the lack of tools and editors for making 3D voxel and very few artists have experience with them. There's also the problem of getting artists to create voxel models that triangulate well and don't generate an excessive number of triangles. Be that as it may, LaserCave is made with voxels in mind, I want to explore this medium to see what can be done with it, what looks good, what works and what does not**.**

- **Narrative**: A roguelike with narrative capabilities is somewhat difficult to create when you have to take into account the fact that the game has to allow unlimited gameplay, but still the story is finite. Most roguelikes end up leaving the story behind in exchange for a gaming experience based around the mechanics and game-play. It's hard to organize and code a roguelike so that it can have room for some narrative without affecting the gameplay, but I have some ideas on how to do it.

Structure of development

The structure to be followed during development consists of carrying out an initial investigation into the current state of the procedural generation, the most used methods and their possible uses in video games. Once the research is finished, some concepts will be applied to the development of the game in Unity. Once the base of the game (generation of the stage and movement) is done, the extra time will be used in the graphic polishing of the game, the creation of 3D assets and the expansion of gameplay with new game elements (enemies, items etc.).

Who is this game for?

At first the target audience of the game was not clear, the roguelike genre is very old and its popularity has varied a lot over time. However, if we take into account the target audience of puzzle games and also include people interested in narrative games, we can define the type of people who may be interested in the game.

For example, the main target is gamers who enjoy roguelike games such as "The binding of Isaac", "Enter the Gungeon", "Rogue Legacy", "Pokemon Mystery Dungeon" + those who also like puzzle games such as those in the "Portal" series. Although it can also be appealing to those who like non-intrusive narrative games like "Hotline Miami" or "Half-Life".

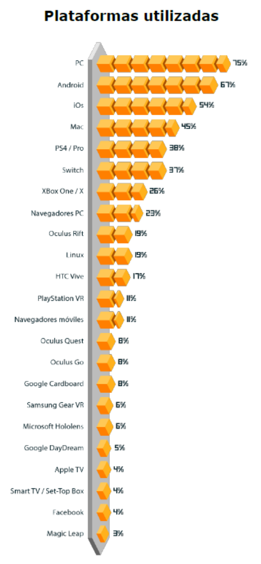
3. CONCEPTUAL AND CONTEXTUAL FRAMEWORK

3.1. The current market

According to the report presented in the 2019 edition of *the White Paper on the Spanish development of video games* promoted by DEV[1]: "The platform most used by Spanish companies and studios is still the desktop computer (PC and Mac). Followed by mobile devices (Android over iOS), although its use is decreasing significantly. In third place are consoles, where development for PS4/Pro dominates, although Nintendo's console could come to dominate this segment next year."

The study also shows that Unity continues to be the preferred engine for Spanish companies for the development of cross-platform video games.

Figure 1 - Popular platforms



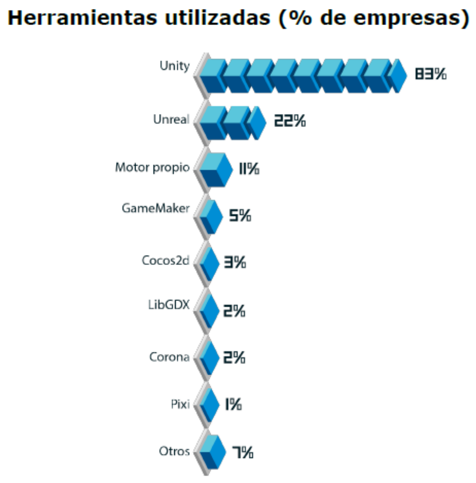


Figure 2 - Most widely used tools in the sector

The sudden decline of the mobile market sugges that markets such as the Nintendo Switch can be a good alternative, without neglecting the PC market, which remains along with the console market (PS4 and Xbox One) the most popular today, especially within the roguelike genre.

Speaking about the roguelike genre, it is difficult to obtain general statistics about this specific genre, however there are statistics of the popularity of some games of the genre within platforms such as Twitch. In the case of "*Enter the Gungeon*" there is a considerable

[1] (DEV, 2019)

increase in the popularity of the game every time it is updated with new content. Which shows the existence of an active community interested in the roguelike genre.

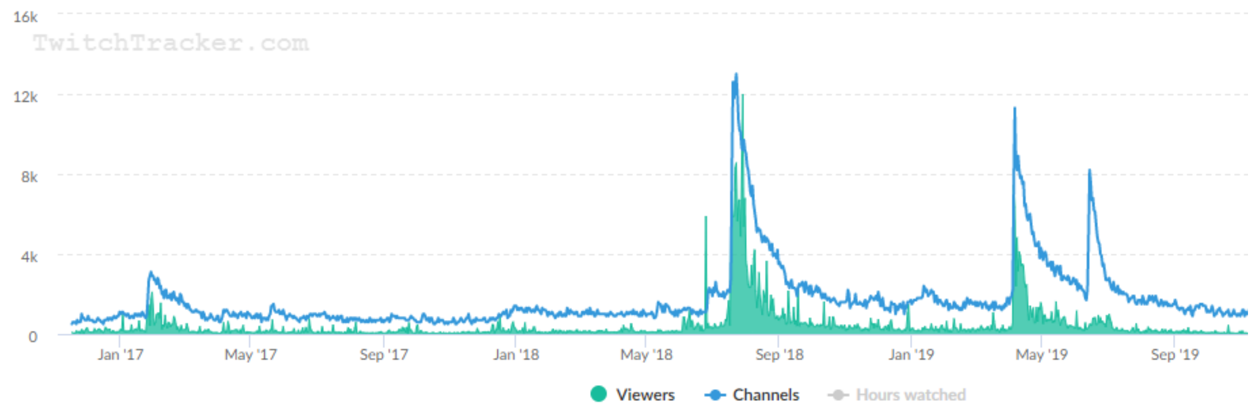


Figure 3 - Popularity of "Enter the Gungeon" on Twitch

3.2. Research

1. GENERAL INFORMATION

Abstract:

This research offers an overview of the different techniques of procedural content generation[2] currently on use by the video game industry, comparing the effectiveness and results of different generation systems. It also describes its most common use cases, the advantages and disadvantages of each system along with the conclusions obtained. The objective of this investigation is to find adequate and adapted procedural generation systems in order to combine them to give rise to a generator of labyrinth-like environments typical of the roguelike[3] video game genre. This generator must be able to use both predesigned structures by made by designers and predefined values to provide diversity and replayability to the gameplay of the final video game. It is also intended to apply research in the creation of enemies with different attacks and the positioning of decorative elements in the world.

Keywords:

Procedural generation, environments, perlin noise, grammars, l-systems, mazes, labyrinths, search-based, rule-based, cellular automata, dungeons.

[2] Procedural generation is a method used in software to generate new content from an algorithm.
[3] Roguelike is a genre of video games in which the player goes through a series of infinite dungeons full of enemies until he runs out of life points.

2. INTRODUCTION

The project in development consists of a dungeon game in which the players must make their way through different rooms until they reach the end while defeating enemies and solving puzzles. One of the biggest problems that arise when analyzing the nature of the game is the constant need to generate new content for the players as they progress through the game.

This content can be: rooms, corridors, successions of events, objects, enemies, puzzles etc. To meet the need for these structures, it is required the development of a procedural generation system, since it saves a lot of work for both designers and artists when it comes to generating and placing content in the game world.

This tendency to generate content automatically by computer has become very popular trend lately in a wide number of sectors, because of the large amounts of money that can be saved in human resources. Among these sectors stands out the music sector, which has made great progress in the automatic synthesis of melodies[4] in the last decade, and the 3D modeling sector, in relation to the automatic creation of textures[5] or the "smart materials" that we find in software such as Adobe's Substance Painter.

2.1. Research interest in the project

This research can help the project in relation to the generation of the structures of the game and, in this way, give rise to a game with an almost unlimited number of possibilities in terms of the structure of its scenarios, placement of elements, decoration, attacks of enemies etc.

It seeks to ensure that the game provides an almost unlimited number of hours of entertainment and replayability in a seamless manner, in the shape of scenarios, mechanics and challenges presented to the player.

In addition, in order to make the game expandable, the procedural generation that is sought to achieve will take into account a series of structures created in advance by a designer to generate its content and will use a set of easily interchangeable assets. This way the game can be easily adapted and expanded to add new content and avoid monotony.

[4] An example of this advance is present in services such as https://www.ampermusic.com/ of procedural generation of music.
[5] An example of this is https://www.pixplant.com/ .

2.2. Current research interest

Knowledge about procedural generation and artificial intelligence[6] is in increasing demand by technology companies. In recent years many tasks an artist used to perform have been automated, whether it be animation, texturing, rigging etc. The generation of stories, gameplay and ideas, tasks of a screenwriter or a designer, are also beginning to be optimized and supported by software to make the creative process simpler and cheaper, by accelerating the development process.

This new paradigm shift relies on the use of algorithms to provide support to the artistic process. Although these techniques are still a long way from being able to "replace" a good artist or designer on a development team, it is still a very interesting field of study, especially considering its rise in popularity in recent years[7], and its use as a marketing strategy in video games such as No Mans Sky.[8]

2.3. Research plan

The structure chosen to investigate each of these different procedural generation systems is based on an exhaustive search of the most used methods today in the video game sector, to find those that are most related to the generation of labyrinth-like environments. Apart from this, more generic algorithms related to the generation of pseudorandom numbers with samples adjusted to specific values and noise map generation algorithms will be taken into account. Research will also be carried out on quest sequencing[9] algorithms, among others capable of bringing extra diversity to the project.

Once a listing of generation systems has been created, a general analysis of each of these generation methods will be made, along with the share of first impressions about their possibilities of use in the project, their advantages and disadvantages.

As sources of information, books and works on the subject will be taken, along with general information obtained from the internet, supported by images, videos, articles and other formats of media.

Once all possible methods have been reviewed, a final conclusion will be formulated on which methods are the most suitable to be able to reach an effective implementation of a generation algorithm in the project.

[6] Artificial intelligence, or AI, is a field of computational research that seeks to generate intelligent behaviors using algorithms.
[7] https://trends.google.com/trends/explore?date=2008-12-14%202020-01-14&q=procedural%20generation
[8] https://youtu.be/ueBCC1PCf84?t=9
[9] Quest sequencing is based on generating missions based on an 'A' objective that provides a 'B' reward when completed with 'C' requirements.

2.4. Theoretical framework

As a theoretical basis, bibliographic works from different sources are used, included in the bibliography of this document, based on the current state of the procedural generation in the software industry, focusing especially on its uses in the video game sector. A personal opinion on this information and conclusions on the functionality and usability of each of the fields investigated will also be provided.

As a theoretical framework structure, the following will be used:

1) *What is procedural generation?* : A brief presentation on the main topic of the research, procedural generation, among other secondary topics necessary to understand the research, such as the needs of a procedural algorithm and its limits.

2) *Noise and randomness*: Concepts are presented to understand the inner workings and bases of procedural generation, such as pseudorandom numbers and procedural noise generation.

3) *Most common types of generation algorithms*: The different types of procedural generation algorithms most commonly used in video games are listed and described in depth here. Genetic algorithms, rule-based algorithms, and direct structure computation algorithms are taken into account. For each one there is an in-depth explanation of their sub types, operations, possible uses and applications in the final project.

2.5. Objective of the research

The general objective of this research is to clarify and demystify various concepts about procedural generation in order to obtain an understanding of the most common methods of procedural generation. To reach a conclusion about which methods are better suited depending on the needs of a video game.

I hope that the information gathered throughout this research will serve as a basis for the procedural needs of this project and can be used to discover and clarify the limits of this field of computer science.

It is also my goal to provide through this document a place of easy access to a good part of the set of algorithms that make up the spectrum of procedural generation, in order to help anyone who is not able to find this information on their own because of how scattered it is among different books and academic articles that are sometimes difficult to obtain.[10]

[10] Most of the information on procedural generation, as with most topics related to programming, are scattered across different publications written in different languages.

2.6. Expected results of the investigation

From this research it is expected to obtain a method, or set of methods, resulting from the combination of several of the resarched generation systems to be able to be implement in the generation of environments in the project. These methods seek to improve the variety of the game in terms of level design, structure and behavior of enemies.

In the case of not finding a definitive solution, a procedural generation method of its own will be developed; using what has been learned throughout this research as basis, in order to solve the shortcomings of the project.

3. BODY OF RESEARCH

Below is the total set of all the research done for this project.

3.1. What is procedural content generation?

Procedural content generation, commonly abbreviated as "PCG", is a method used to generate new content automatically from a mathematical algorithm. These algorithms are implemented in a programming language and executed with a series of initial parameters. The result is determined directly by the algorithm used in the generation.

The content to be generated depends on the objective and scope in which the algorithm is being used. In a video game, for example, the content is usually maps, levels, objects, enemies, quests, etc.

This content depends largely on the genre of the video game that is proposed to develop. For games of the Sandbox genre such as Minecraft[11], Spore[12] or Civilization[13], the procedural generation is responsible for creating the game world. In others such as the RPG genre, of which we can use The Elder Scrolls V: Skyrim[14] as an example, the generation focuses on creating missions with a series of requirements for the player.

The procedural generation is directly connected to AI algorithms, which in video games are generally used (although not exclusively) to define the behavior of characters and enemies. The generation of these behaviors, therefore, is not considered procedural generation, since the field of AI is specifically responsible for it. This is why we must understand that procedural generation and AI are closely related, but they are not the same and are responsible of different functions.

[11] Minecraft is a game developed by Mojang, which has an almost infinite open world. So far it has sold around 176 million copies.
[12] Spore is a game developed by Maxis and released in 2008. The game world and its inhabitants are procedurally generated.
[13] Civilization is a series of turn-based strategy games developed by Firaxis. The world is procedurally generated.
[14] The Elder Scrolls V : Skyrim es un juego RPG desarrollado por Bethesda game studios.

Now that it has been defined what procedural generation consists of. Here are the bases and characteristics of a procedural content generator:

a) *Input*

   All content generators need input parameters that alter the result of the generation algorithm. If there is no input, the result will always be the same. Examples of common input arguments are usually: seeds[15], pseudorandom numbers, and generator-specific parameters.

b) *Output*

   It is the resulting content generated by the algorithm. It can be of any format depending on what you want to create. E.g: a maze generation algorithm could return an array of values that specify where there are walls and where there are not.

c) *Complexity*

   The complexity of the algorithm determines its execution speed. This is crucial when implementing it within a video game, especially if the generation has to run dynamically while the player is playing and long loading screens need to be avoided.

d) *Customization*

   It directly depends on the complexity of the generator's input. Some generators only require one parameter to operate, while others allow several parameters to adjust the result to better suit specific needs. E.g: a tree generator can generate a tree with only one input parameter, or it can use several parameters to determine extra characteristics, such as the number of branches or their length.

e) *Randomness*

   There are procedural generators that use algorithms to generate specific values, while others tend to use more random values in their calculations. This determines the diversity of the result obtained and divides the types of generators between more deterministic generators and more random generators.

[15] Seed is the name given to a value used as a basis by a pseudorandom number generator to generate pseudorandom numbers.

With this last characteristic of procedural content generators it can be observed that there is a very close relationship between these and the use and creation of pseudorandom numbers. Pseudorandom[16] numbers are used by most procedural algorithms as the basis for producing a different result in each execution. A section is devoted to this topic below.

3.2. Pseudorandom numbers and procedural noise generation

To understand the basis of procedural algorithms, we must take into account a series of programming concepts about what is considered something "random" or stochastic.[17]

First, in computing, to generate a "random" number an initialization value is required, commonly called a "seed", which is used by the algorithm to generate different values. The origin of the generator seed and the complexity of the algorithm define how random the number generated will be. In many cases the seed is taken from the internal clock of the system, in other more complex cases it is taken from images of changing environments, such as a camera that takes photographs of clouds, the static of a radio or a television, among others...

The benefit of using a seed with a static algorithm is that the same sequence of values can be obtained again by introducing the same seed previously used.(Watkins, 2016)[18]

The disadvantages are that eventually and depending on the complexity of the algorithm, the sequence of numbers, or part of it, will end up repeating. A very easy way to test the effectiveness of a pseudorandom number generator is by visually representing these numbers in a two-dimensional image (Figure 4).

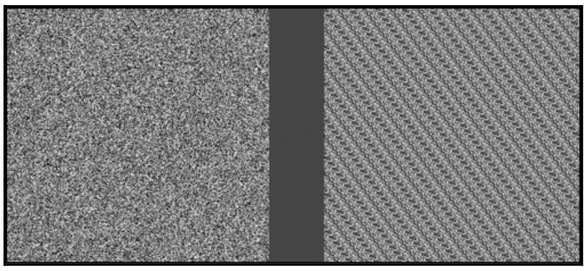


Figure 4 - Representation of pseudorandom numbers

[16] In computing there are no purely random numbers, so every time a reference is made to random numbers, they are actually considered pseudorandoms.
[17] A stochastic process is a mathematical process that generates random magnitudes over a given period of time.
[18] Watkins, R. (2016). Procedural Content Generation for Unity Game Development. Birmingham B3 2PB, UK: PACKTLiB

In the image above you can see one of the main uses of pseudorandom numbers, the generation of noise maps. These maps are a very useful resource in the generation of 3D terrains and textures, since they allow to generate organic terrains and textures automatically (Figure 5).

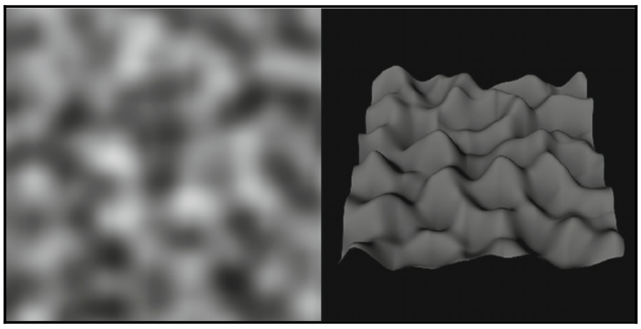


Figure 5 - Map of noise generated with pseudorandom numbers

Noise maps can be black and white, grayscale, color, two-dimensional or one-dimensional. In the field of 3D modeling and texturing, the use of a grayscale map, commonly called Heightmap, is usually preferred. "A heightmap should be a grayscale image, with white areas representing high areas and black representing low areas." (Unity3D, 2018)

There are many different ways to generate noise maps, although they all share 2 types of generation: 2D and 3D. 3D maps have a particularity, they are designed to look good when projected on a sphere, meanwhile 2D maps will show a seam [19]. In the next section the most widely used noise generation algorithms will be discussed, along with their most common uses, advantages and disadvantages.

3.2.1. Interpolated

There is a very simple way to generate procedural noise that consists of assigning each pixel of the noise map a random value, this is what is considered "Random Noise". The main problem with this noise is that there is no direct relationship between one pixel and the next, as can be seen in (Figure 4).

Because of this, the terrain generated from these maps is excessively steep.

One solution to this problem is to interpolate[20] the different values on the map to generate a gradient. These values are separated by a distance called "lattice" that defines the

[19] Seamless textures can be tiled without having a noticeable “cut/seam/transition” where the texture begins and ends.
[20] Interpolate is the action of finding the intermediate values between two different values using a linear function.

resolution[21] of the map. Since the goal is to generate a two-dimensional map we have two possible options:

(a) Bilinear interpolation

Bilinear interpolation consists of interpolating two values in a two-dimensional space, interpolating linearly on the x-axis and then on the y-axis.

• First, a lattice size is chosen and the points to be used are selected.

E.g . *P[0,5] and P[0,10] with lattice 5.*

• Each intermediate point is linearly interpolated on its *x and* y *axis*.

E.g . *P[0,7] = 0.6 * P[0,5] + 0.4 * P[0,10]*

This process is simple, but it has a number of disadvantages. The most notable can be seen with the naked eye, and is that by the nature of performing interpolation on two axes, the points located on these axes (that is, directly above, below or on the sides of the reference points) have a steeper gradient that causes the final map to have the appearance of a grid (Figure 6).

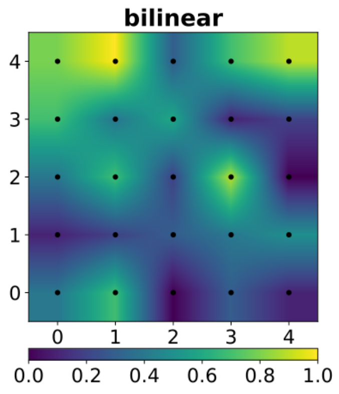


Figure 6 - Result of a bilinear blur on a 5 * 5 image

This grid effect is only visible when placing datum points outside the edges of the image. To avoid this there is another method of interpolation:

(b) Bicubic interpolation

For a smoother result, instead of interpolating linearly using the mean of the values of the control points, a non-linear function can be employed that calculates the vertical slope of each point based on its proximity to the control points (Figure 7).

[21] The resolution of a noise map refers to the size of the gradient between its points, or lattice.

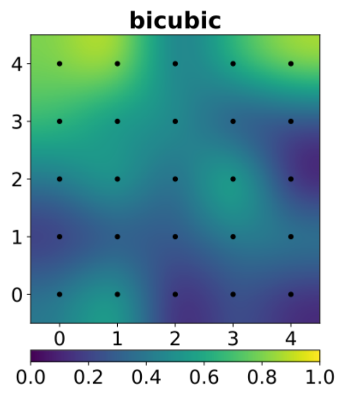


Figure 7 - result of a bicubic blur on a 5 * 5 image

The function for calculating the slope can be any, as long as it suits our needs. If our goal is to generate gentle slopes, a very common and easy to calculate function is:

$f(x) = -2x^3 + 3x^2$

It generates a softer S shape than that of a sine or cosine with values between 0 and 1. It is also very easy to compute (Figure 8).

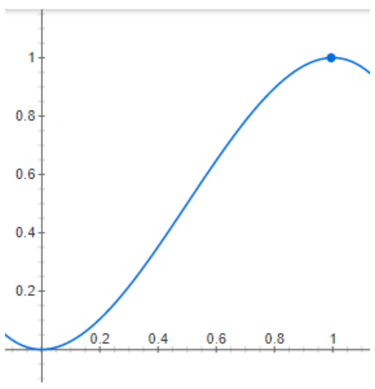


Figure 8 - Bicubic interpolation formula p(x) = −2x3 + 3x2.

The use of our slope function then replaces linear interpolation, so that the previous example would be:

$f(0.4) = 0.352$

$P[0,7] = 0.648 * P[0,5] + 0.352 * P[0,10]$

3.2.2. Gradient Noise

Instead of generating the height values of each slope by interpolation, each slope can be generated directly by using gradients. These gradients smooth out the change in height of each point on the slope rather than just the highest and lowest point, so the result is much smoother when the height of the slope changes between steeper values and flatter values.

In this method the lattice, mentioned above, is not used as a starting point to generate the gradient, but emerge naturally from the gradients used as slopes, so the result is much more organic.

The basis of these methods consists of generating a 2D vector in each lattice that serves as a gradient, and determines how high and inclined the slope is.

There are several implementations of gradient-based noise generation algorithms, the most popular being:

a) Perlin Noise

The first of these types of noise was created by Kenneth H. Perlin (Wikipedia, 2020)[22], a professor in the department of computer science at New York University, specializing in graphics, animation, multimedia... among others. He is best known for creating the first gradient-based procedural noise generation algorithm, which is now known as Perlin Noise. This was first used in the original film "Tron" (1982) to generate more realistic textures.

In this algorithm the four corners of the square of a plane are interpolated, but instead of interpolating between values, it is interpolated between vectors placed in each of the corners of the plane. In this way the result is smooth even when using several quacks with vectors, or gradients, shared between them in the same context (Figure 9).

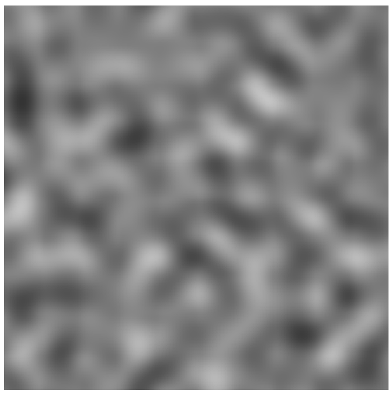


Figure 9 - Noise generated by the Perlin Noise algorithm

Perlin Noise is one of the most widespread noise algorithms today because of its easy implementation and medium-low computational cost. In procedural generation it is used, along with other noise algorithms, when generating terrain (using the values as determinants

[22] Wikipedia. (January 14, 2020). Ken Perlin https://en.wikipedia.org/wiki/Ken_Perlin

of the height of the terrain). It is also used to place elements in clusters (clustering) such as lighting, or the enemies of a map, so that they are distributed in groups evenly.

If instead of using gradients, as Perlin Noise does, we used integer or floating-point values, the result would be that of the so-called Value Noise, a noise in which the difference between each square of points is noticeable with the naked eye because of the interpolation carried out (Figure 10).

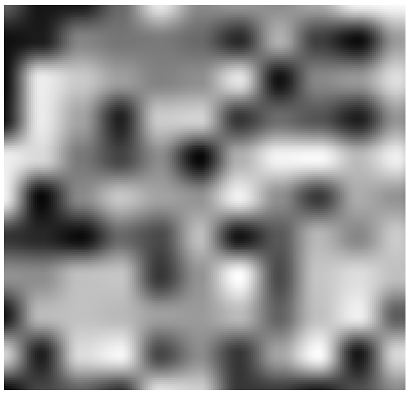


Figure 10 - Example of Value Noise.

This difference could be smoothed out by applying a convolution array[23], but it would consume much more in terms of runtime than using Perlin Noise directly instead.

b) Simplex Noise

The Simplex Noise is an improvement made by Perlin himself to his algorithm. It consists of using triangles instead of squares within what is called a Simplex Grid, so that there are fewer corners to calculate. The rest of the implementation consists of the same as the Perlin Noise, interpolating between the vectors (in this case 3 instead of 4 because they are triangles) to achieve a smooth transition of slopes.

Because of the triangular nature of the Simplex Grid, the shape of the slopes tends to be triangular. Despite being a much faster method, this triangular result may not be the desired one in many cases, which is why Simplex Noise is less used today than Perlin Noise. Another reason for its low use is because Perlin Noise has a much simpler implementation than Simplex, making it much easier to modify.

[23] A convolution array is a matrix used in graphical programming to generate image effects such as blur.

3.2.3. Fractal Noise

Fractal noise, as the name suggests, is based on fractals[24]. They can be obtained from any of the algorithms I have described by applying them recursively and increasing their wave frequency on a regular basis while reducing the amplitude. By superimposing noises with different frequencies (also called scale or resolution) a much more organic result is obtained.

This type of noise is used to make the structure of clouds, mountains and rocks. But it’s also used in texturing for organic elements.

Although any noise can be applied recursively on itself to obtain a fractal result, there is a type of noise used only to obtain fractal results because it is very easy to implement and efficient, even despite its recursive nature.

a) Square Diamond

It starts with an initial square, of which the central point must be found from its corners, each with a random noise value. The value of the center point is calculated by interpolating the values of the four corners with it and then adding an extra random value.

The next step is to find the central points of the segments of the square and assign a value resulting from interpolating the values of the two ends and the center. To this value is added a random value.

This way squares get generated recursively. The recursion is usually it stopped when the distance from a corner to the center of the square is lower than a unit (Figure 11).

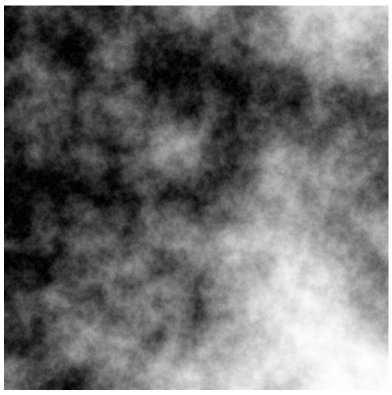


Figure 11 - Example of fractal noise

The random value that is added to the center of the square and the center of the segments must be in a specific range that determines the smoothness of the result. Higher values generate very contrasting values while lower values result in less clear results.

[24] A fractal is a resulting image of the recursive repetition of a pattern.

3.3. Procedural generation algorithms

Noise and randomness are not enough to generate scenarios with the logic expected by a player. Reaching that level of complexity requires the combination of a series of algorithms designed in order to provide logic to the structure and gameplay.

There are a lot of procedural algorithms, many of which are based on AI to apply logical rules to the generation of levels, objects, enemy missions etc. In this case, the following categories of procedural algorithms will be taken into account:

- Procedural generation algorithms based on genetic search algorithms: These are algorithms that use a population of individuals that evolves to give rise to an individual who meets a series of requirements.

- Rule-based generation algorithms: These are algorithms that are based on a series of coded symbols that translate into a specific generative behavior.

- Algorithms of direct generation of structures: They are simple methods of little complexity that allow solving problems such as the placement of general elements in the world, such as structures, objects, enemies ...

Currently, these types of algorithms are among the most popular when it comes to programming procedural generators for games. Its use is especially widespread in games of the sandbox[25] and roguelike genre, where it seeks to provide the exploration of scenarios of almost unlimited possibilities.

3.3.1. Search-based generation algorithms / evolutionary algorithms

This generation system employs evolutionary[26] search algorithms[27] in order to achieve functional levels. It requires a representation of the elements to work with, each with its own "genes" that define its behavior or components. To calculate the viability of the result, an evaluation function commonly referred to as the "fitness" function is used.

A simple yet widely used evolutionary algorithm is the *(μ + λ * Evolutionary strategy)*. In which *μ* are the individuals who remain in the generation, *λ* is the size of the population that is generated in each iteration and the *Evolutionary strategy* is an optimization required to choose the best specimens.

[25] Sandbox is a video game genre that offers the freedom to explore an open world or mechanic without constant barriers.
[26] "Evolutionary algorithm" https://en.wikipedia.org/wiki/Evolutionary_algorithm
[27] "Search algorithm" https://en.wikipedia.org/wiki/Search_algorithm

The generation process in this case follows the following process:

- Creation of an initial population of size $\lambda+\mu$ with elements or "individuals" with random genes (or that might come from a previous population).

- Selection of the best individuals (μ) through the *fitness function* and combine their genes to give rise to a new population of individuals.

- The number of best individuals to choose from is obtained by eliminating λ worst individuals. The λ eliminated individuals are replaced with copies of μ best individuals.

- A mutation is randomly applied to some elements to ensure that an optimal solution is finally reached. In the case of using one-dimensional vectors to represent the genes of an individual, the *Gaussian mutation*[28] is commonly used for the pseudorandom mutation of the genes of individuals.

- If an individual with an optimal score has been reached, the generation is stopped, otherwise this population is continued back to step 2.

The implementation of this type of evolutionary algorithms depends directly on the way of representing the genes of each individual and the complexity of the *fitness function*. For example, if the representation is made with real numbers, a particularly effective evolutionary strategy is the CMA (*Covariance Matrix Adaptation*[29]), used for the global optimization of functions without derivatives.

The representation of the different elements that make up the world depends on the objective of the generation. The size of the representation, called the dimension, exponentially affects the computational time needed to find a solution and validate it.

Each representable element to be placed at the level has a *genotype* that contains the data that identifies its behavior and characteristics. The result of the generation is called *phenotype* and is the determinant of the viability/quality of the level.

(a) Representation

How to represent each element of a level is a complicated task that requires a thorough analysis of the desired objective. The type of representation used can be direct, in which each individual element is generated with its position and attributes of the level directly, or it

[28] The Gaussian mutation / Gaussian distribution / normal distribution is a function that allows random numbers to be generated in a range within which the central values are more likely to appear.
[29] "The CMA Evolution Strategy" 4 Abril 2016, https://arxiv.org/pdf/1604.00772. Accessed 8 Dec 2019.

can be more indirect, in the case that predefined structures are placed or the algorithm is only used to decide specific values of some elements, such as the number of enemies, treasures etc.

As an example of an indirect representation, in the document "*Evolving dungeon crawler levels with relative placement*" (Valtchanov & Brown, 2012) they use evolutionary algorithms to generate the structure of the tree that makes up the rooms of a dungeon and its connections between them. In its representation each room is a Node with a series of possible connections.

In the generation of 2D levels, this kind of algorithms does not limit us to just connect trees as in the previous case, but the complete structure of the level can be generated element by element individually. In this case, it is common to choose to represent the map as an array of bytes/characters that is easy to alter by the algorithm. To represent floors and walls it is common to use 0s for floors and 1s for walls in such a way that the result of the generation of a level can be represented as a binary string, or even an integer. This binary matrix can be initialized with zeros or with ones to give rise to different results.

For example, a room of 6 units height and 8 units width could be represented as an array of Booleans in which *true* is a wall and *false* is a floor.

This is considered a direct representation, genes can be stored within a 64-bit unsigned integer. For larger room sizes, this bit-to-integer conversion may not be possible because of integer storage limits (usually limited to 128-bit integers) and is usually left as a one-dimensional vector to work with.

The placement of walls and floors is carried out by the evolutionary algorithm based on a series of preference parameters applied in the *fitness function*. As a basis, you can specify main rules, such as favoring the generation of contiguous floors to create rooms, or that these floors are only placed in one direction to create corridors.

A simplified example of this generation is demonstrated by Danniel Ashlock, Colin Lee and Cameron McGuinness in the paper "*Search-based procedural generation of maze-like levels*" (Ashlock, Lee, & McGuinness, 2011), which explains how the direct representation of rooms and corridors for a dungeon works by representing the genes of each room and corridor as a set of numbers that specify their independent attributes, such as length, size, position and direction. (Figure 12)

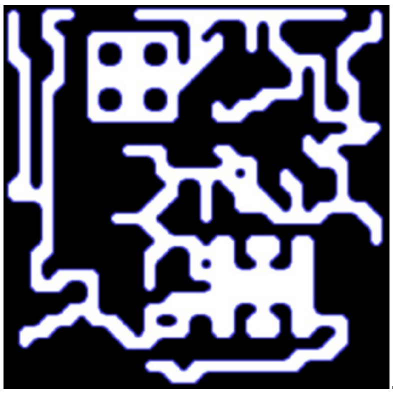

.

Figure 12 - Generation using evolutionary algorithms and pre-designed structures. Image extracted from (Shaker, Togelius, & Nelson, 2016)

To provide more variety and detail to the generation, the individual generation of walls and floors can be combined with the placement of pre-designed structures, such as rooms or special corridors that would be very difficult to generate through search algorithms.

(b) Evaluation functions

The evaluation functions or fitness functions are the basis for determining what result of the generation is desired to obtain. Depending on the method used to calculate the viability of the level, the evaluation functions can be: *direct, simulated or interactive*.

- Direct functions check that a series of rules are met in the resulting phenotype, such as that there is a specific number of rooms, corridors or that the distribution of these elements is adequate.

- Simulated functions require an agent to act instead of the player to validate the gameplay of the level. There are different ways to achieve this, the most common being to try to get an AI to complete the level by taking the control of the player.

- Interactive features use an external player to manually assess the quality of the level.

The biggest problem that arises when trying to evaluate a level is to define evaluation concepts. E.g. If the function has to rate each level based on how fun it is, you have to determine what makes a level fun to define the evaluation function.

3.3.2. Rule-based generation algorithms

The first thing that comes to mind when creating a content generator is the large number of rules and requirements that the content to be generated must follow in order to be considered viable. That is why there is a branch of procedural generation algorithms based on the direct substitution of elements in the object to be generated, by others, based on a series of parameters, or rules. Among these algorithms, the following stand out:

a) Grammars

In the book "Handbook of Formal Language Volume 1" (Rozenberg & Salomaa, 1997)[30] grammars are presented in the first chapter, as a language of free context, defined by an axiom *S,* and a series of literal replacement rules that are applied recursively on themselves, from left to right.

For example the grammar:

S → SAB

Indicates that S is represented as the replacement by the literals SAB, and since this replacement is recursive, S = SAB = SABAB = SABABAB.

Of course, the usual is to have multiple rules:

A → BB

B → a

In this case, if we apply all these rules and iterate, the following string would result:

S → SAB → SABBBa → ...

In this example a total of 3 iterations have been made, but the nature of the grammars allows them *to* continue iterating indefinitely, as long as there are "*non-terminator symbols*", that is, for example the symbol "a" is lowercase to determine that it is a terminator symbol, that is, there is no rule that determines a replacement of "a" so that part of the chain is not going to be altered. Therefore, if a string contains only terminator symbols, it cannot continue to be iterated.

This way of representing languages by a series of replacement rules was first introduced in 1956 by Noam Chomsky in his paper "*Three models for the description language*" presented to MIT's Department of Modern Languages and Electronics Research Laboratory. (Chomsky, 1956)[31]

[30] Rozenberg, G., & Salomaa, A. (1997). *Handbook of Formal Language Vol 1.* Berlin: Springer
[31] Chomsky, N. (1956). Three models for the description of language. Massachusetts: IEEE Transactions on Information Theory

This method of generating literals by the use of rules allows to obtain deterministic results that are easy to recreate and with an unlimited number of possibilities. However, what can they be used for in procedural generation? And even more importantly, what rules and how many should be used?

As already explained above, abstract data must be separated from what it represents. A number in a generation system can be anything; it can represent the number of enemies in a level, their difficulty, the number of walls in a room, or the number of doors. In this same way a string of characters and the rules used to write them can represent anything from the succession of events in a story to the order and type of objects you can find in a room.

Grammars can be combined with the aforementioned evolutionary algorithms as a solution to the representation problem. This method of representation offers a solution not only when it comes to shaping the genome of each individual but also in the process of selection and evolution of the same, put into practice in "*Procedural Generation using Grammar based Modelling and Genetic Algorithms*" (Haubenwallner, 2016)[32]

b) Graphs

An interesting use of grammars is their application in different data structures, such as strings, lists, trees, and graphs[33]. Since this research is aimed towards achieving the generation of the different components of a roguelike game, it is interesting to review its application in these last two structures (trees and graphs) since they can be applied to elements such as the structure of the world map, the succession of events during the game adventure, among others...

To achieve this we can imagine the map of a game as a graph, with different types of nodes[34], each with replacement rules that allow the iteration of the graph. Suppose we have a graph base structure, which defines the gameplay of a level.

[32] Haubenwallner, K. (2016). *Procedural Generation using Grammar based Modelling and Genetic Algorithms.* Austria: Institute of Computer Graphics / Graz University of Technology.
[33] A graph consists of a series of interconnected nodes, these nodes can be connected to one or more nodes. Some graphs are closed, while others have an initial and an end node.
[34] A node is an element of data structures found on linked lists, trees, or graphs. A node usually contains a value of its own, along with one or more connections to other nodes.

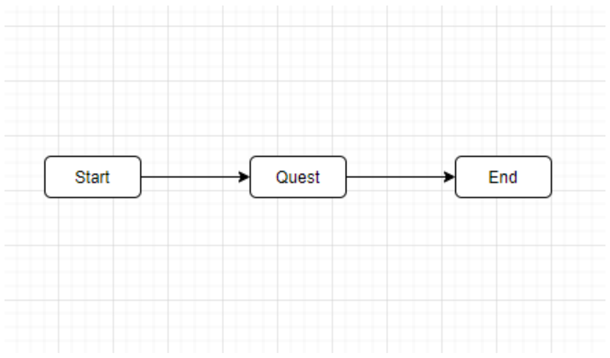


Figure 13 - Initial state of a graph representing a video game quest.

In this figure you can see 3 nodes, A **Start** node, a **Quest** node and an **End** node. From here, we can apply a series of replacement rules that allow us to specify more complex Quests.

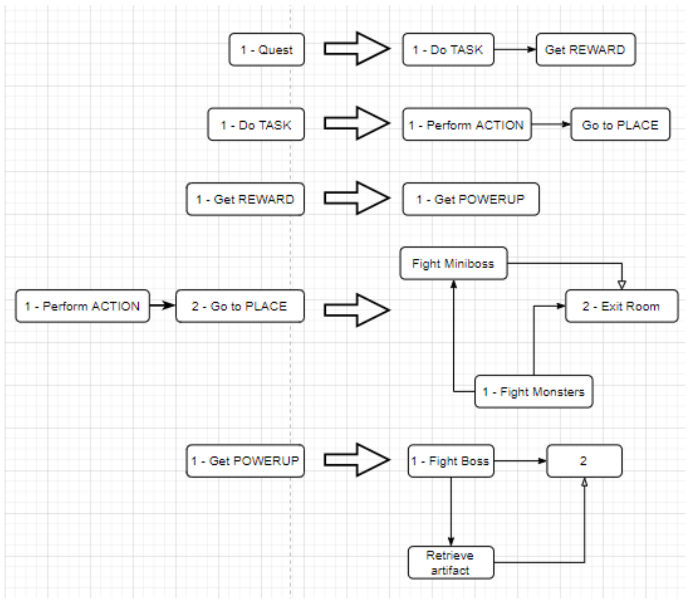


Figure 14 - Rules of replacing a graph, the value(s) on the left is/are replaced by the subgraph on the right, using the numbers of the nodes as reference anchors when doing the replacement

This example begins using the previous rules. As with the grammars explained above, replacement rules consist of a part that defines the section to replace (on the left) and a part to replace it with (on the right). The process to follow to replace each subset of the graph with the subset specified in the replacement rules depends on the complexity of the graph rules. In the document "*A Graph Grammar Approach to Graphical Parsing*" (Rekers & A., 1995)[35] an extensive explanation of the replacement standards used with graphs to parse

[35] Rekers, J., & A.S. (1995). A Graph Grammar Approach to Graphical Parsing. (Leiden, The Netherlands) (Aachen, Germany): Proceedings of the 11th International IEEE Symposium on Visual Languages

visual programming languages is made, but in this case, we will only take into account 2 basic rules:

- The black arrows indicate the accessible nodes. If the player is on a node, they can only access those to which their current node is connected to by a black arrow.

- The white arrows indicate the dependence of one node on another in order to access it. E.g. in order to access a node that has a white arrow connected (dependent node), you must first access the node on which it depends (dependency). In this way you can establish requirements of the likes of: To be able to move to the next room, you must first visit the rest of the rooms.

Now that the basic rules have been defined, the replacement method is quite simple:

In one rule, the nodes on the left have an identifier, (in this case a number). This number identifies the nodes by which nodes on the left will be replaced by nodes on the left. In such a way that if we have the following rule:

1:A ► 2:B → 1:C ► 3:A ► 2:E

Node 1:A corresponds to 1:C and node 2:B corresponds to 2:E because they share the same number. If we apply this rule to the following graph:

S ► A ► B ► S

The result would be:

S ► C ► A ► E ► S

Back to the example at (Figure 14), if the replacement rules are applied following what is explained here, with the first iteration we would get something like this:

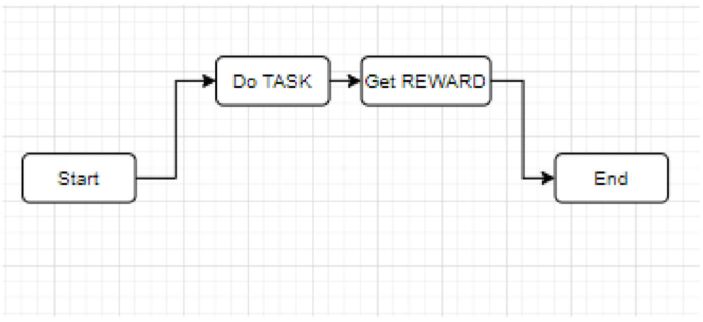


Figure 15 - First iteration of the graph

This initial replacement is quite simple because it is a simplified example, in a real case we would have several replacement possibilities for each rule, instead of just one.

If we perform another iteration:

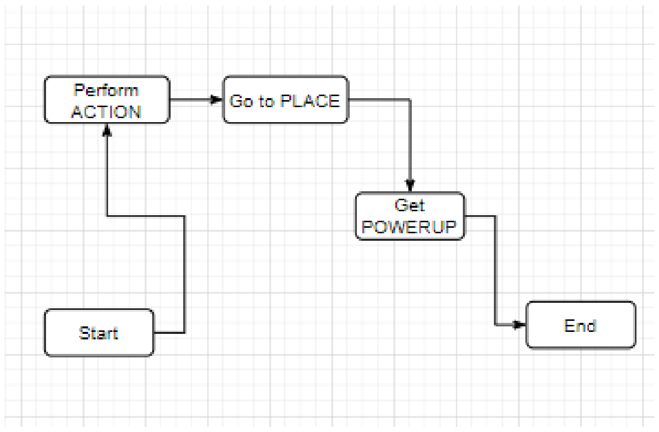


Figure 16 - Second iteration of the graph

We could also make a "Do TASK" node not only replace with (Perform Action > Go to Place), but we could put alternatives, such as (Perform Action > Talk to NPC). Which of the two would be chosen depends on what we seek to obtain in our generation, or it could be chosen randomly.

Replacing grammars consists of going from general concepts like "Go to Place" to more precise concepts, as you can see in the following iteration:

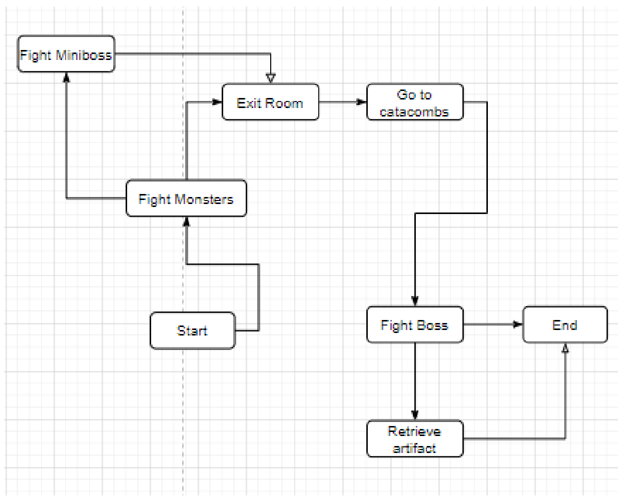


Figure 17 - Another iteration of the graph

In this case, "Go to Place" has been defined as the succession "Fight Monsters > Fight Miniboss > Exit Room". Even though "Fight Monsters" connects with "Exit Room" it is necessary to run "Fight Miniboss" first, since "Exit Room" depends on this node (white arrow).

There exists another dependency in this case. In order to continue to the node “End” the node “Retrieve Artifact” must be visited, but this node can only be accessed after visiting “Fight Boss”. In the definition of this rule an empty space has been left with a “2” (Figure 14), this indicates what is the node to connect to the next node of the graph, since in this case it is not clear if the next node is "Retrieve artifact" or the empty node (The empty node with a “2” in this case represents "End" since it is the next node to "Get Powerup"). In cases such as the first 2 replacement rules, the “2” has been omitted since only one free node remains, it is assumed that this is the one that connects to the next one in the graph. But in cases like rule 4 and 5, they are necessary to solve connection problems.

From these simple rule-based systems quite complex systems can be obtained with just a few iterations. However, there is a problem when trying to bring this into a game:

What if you want to use this system to represent the rooms of a game?

Graphs have a problem, no matter how many limits are stablished, if you try to "fit" the nodes of a graph into a grid problems will arise, many of which have no easy solution. For example, it is very difficult to determine a place in the grid, in which by positioning one node you can be sure that it will not prevent the placement of others (assuming that there can only be one node in each space of the grid), an example can be seen here:

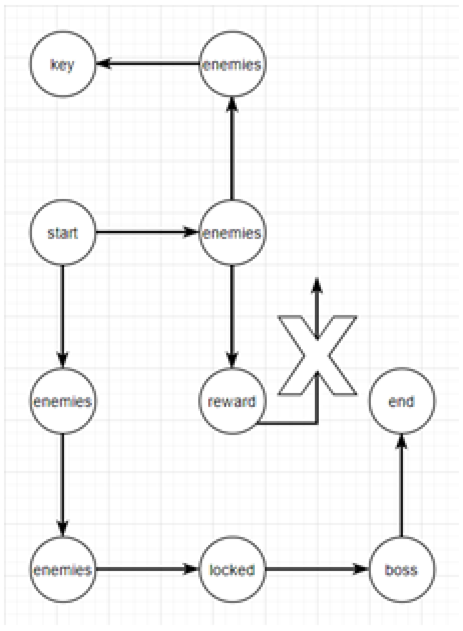


Figure 18 - Example of an error when trying to plot the generated nodes of a graph within a grid.

In this case reward cannot be iterated, since there is no space left in the grid to place nodes that are children of the same.

This problem could be solved by using evolutionary algorithms or brute force, to try to achieve an optimal placement of the nodes of the graph, but still many cases could occur in which there is no solution, either by the size of the graph, the number of interconnections or the limits of the grid.

c) L-Systems

In the grammars explained, the application of rules is done from left to right and the modifications of the chain are taken into account when evaluating the rest of the chain, as it is traversed.

This means that the result of the grammar "X" with rules:

X → XY

Y → X

YY → X

Results in the following iterations:

X → XY → XYX → XXX

In the last iteration ("XYX → XXX") the first "X" of "XYX" is replaced with "XY", giving rise to "XYYX" and because this is the working string, the next expression to evaluate is "YY", which has resulted from the previous operation applied within the same iteration, and is replaced with “X” giving the final result of “XXX”.

There is an alternative to this method, which is to generate the new string in parallel to the one being evaluated. The results may vary completely, or be exactly the same depending on the method used.

In the example above, by rewriting in parallel we would get the string "XYXXY" instead of "XXX".

This method is used by L-Systems (Lindermayer Systems) created by biologist Aristid Lindenmayer to describe the structure of plant organisms such as trees and algae.

L-Systems have a number of characteristic qualities:

- They have symmetry.
- Its nature is fractal.[36]
- Its growth is exponential.

They can be used to represent virtually anything, from the movement of a character, the arrangement of elements in the world, the musical accompaniment of a song, etc.

A very common use of L-Systems is the representation of fractals using some graphic representation language (such as Logo[37] or another based on turtle graphics[38]). Because of this there are a number of extra commands that are usually attributed to L-Systems, for example, 90-degree rotations are usually represented with the command "+" and "–" depending on the direction of rotation. There is also the possibility to save the current position of the "pen" (see Logo) using "[", and go back again to this position with "]".

An L-System can be used to define the structure of a level, and this representation can be used within an evolutionary algorithm to evolve it according to a specific criterion, which allows to obtain a structure much more adjusted to a series of requirements.

[36] A fractal is a structure recursive formed by a number *n* of versions of itself.
[37] Logo was a language of rVector presentation created in the late 67s, but which became especially popular when it was ported to 8-bit microcomputers in the late 70s.
[38] Turtle graphics are based in the idea of operating a digital pen making it follow a series of commands to draw lines and thus create graphics. It is a common method for drawing fractals.

3.3.3. Algorithms for the direct generation of structures

The algorithms mentioned so far can be applied to almost any paradigm of procedural generation. But there are a number of algorithms designed to solve a particular problem. Below are the most common within this aspect of procedural generation:

a) Maze generation algorithms

There are a number of maze generation algorithms[39] that can be responsible for interconnecting the different nodes/squares of a grid, so that the resulting connections are similar to those of a labyrinthine environment.

I. Linear algorithms: Eller's algorithm

These algorithms work at the same speed with each row of a grid (or array) to generate connections, so they are especially useful if you're looking to create mazes as quickly as possible.

Eller's algorithm[40] is one of the fastest linear algorithms by far and, unfortunately, one of the most difficult to understand. Its result tend to be a labyrinth with many short corridors and intersections (Figure 19).

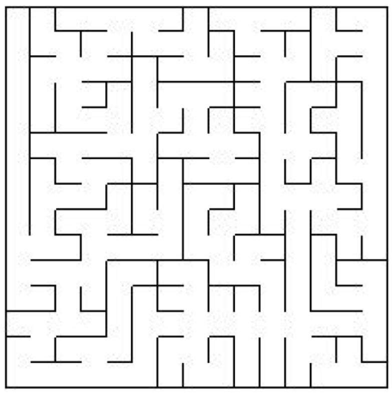


Figure 19 - Labyrinth generated using Eller's algorithm

[39] Maze generation algorithms are a branch of procedural content generation that seeks to achieve an algorithm capable of generating complex mazes, performing as few calculations as possible.
[40] Buck, J. (January 15, 2020). Maze Generation: Eller's Algorithm. Retrieved from Jamisbuck's Buckblog: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorith

To generate a maze using Eller's algorithm, you have to do the following for each row in the array:

1. Each cell in the row is assigned a set, for example, the first row of a 5 * 5 matrix will start with sets such that:

```
0|1|2|3|4|5
```

2. Next, the cells of different sets that are randomly adjacent are put together:

```
0 0 0|3 3 3 // In this case 0|1|2 are merged into 0 0 0 and 3|4|5 into 3 3 3.
```

3. For each set, in this case 0 and 3, at least one vertical connection with the following row is randomly created:

```
0 0 0|3 3 3 // The first, original row.
0|.|0|. .|3 // New row, with at least 1 random connection to each group in the upper row.
```

4. From here, go back to step 1 with the new row, but only a new set is assigned to the empty cells:

```
0|1|0|2|4|3 // 0|.|0|. .|3 gets filled with new groups in the empty spaces
```

Other types of linear algorithms of labyrinth generation are those based on binary trees, the problem with these algorithms is that despite being easy to implement and high speed, their result is very predictable, even having applied some improvements. Instead, later on, it is presented how to use binary trees to subdivide spaces, a much more useful application at the level of procedural generation of them.

II. Exponential algorithms: Backtracking algorithm

Unlike linear algorithms, exponential algorithms are capable of generating better labyrinths (less symmetrical and more complex), at the cost of a longer computation time .

Many of these algorithms are based around the idea of having a cell that places corridors as it progresses. This concept is inherited directly from the "cellular automata" algorithms[41], which are also explained later in this paper.

The backtracking algorithm stands out in this category for its popularity and extension when creating labyrinths, due to its easy implementation, good results and little computational expense . One benefit of this algorithm over the rest is that its creation revolves around going through each corridor of the maze from the beginning to the end as it is created. This

[41] Cellular automata is a method of procedural generation that employs "cells" contained in a grid, which follow a series of rules, usually related to the position of other cells in the environment around them.

is why it is especially useful when it comes to saving the location of each corridor, its length and its forks separately, this data might prove useful later in development.

To generate a maze using the backtracking method, do the following procedure until there are no free cells left:

1. A random cell is selected to start with.

2. The current cell is marked as visited.

3. A random adjacent cell that is not visited is selected and connected to. Now this is the current cell.

4. Repeat step 2 and 3 until there are no longer any adjacent non-visited cells available to connect. In this case, go back (backtrack) through the connected cells until you find one that meets those conditions, then go back to step 3.

5. If when backtracking you return to the initial cell, the maze is then complete.

The problem with this algorithm is that most generations consist of a main convoluted corridor with some divergences that end in dead ends (Figure 20).

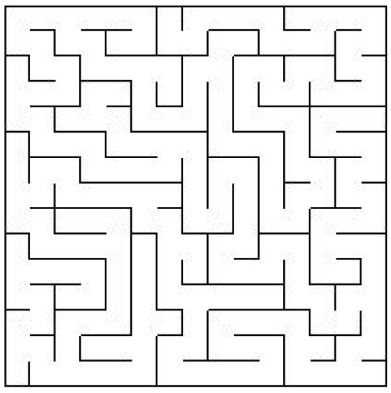


Figure 20 - Labyrinth created using the backtracking algorithm.

A possible improvement which I implemented to the backtracking algorithm is to start the new corridor in a random visited cell (with access to a non-visited cell), instead of using the first visited cell that is found when going back. This way you can get labyrinths with more divergent corridors of more or less the same length, and thus the navigation through the labyrinth feels more open, rather than linear. The problem is that this requires a longer computation time, since you have to be constantly storing the visited cells that still have a free connection with a non-visited cell.

The difference in generation between the two algorithms can be appreciated at first sight (Figure 19 and Figure 20).

Eller's algorithm tends to generate many intersections, while backtracking (without my modification) generates long corridors. There are many more corridor generation algorithms , but these two particularly stand out from among the most used.

b) Algorithms of subdivision of spaces with binary trees

The use of maze generation algorithms provides a possible solution to a very common problem in the field of procedural generation: Generate a series of randomly interconnected nodes, which do not overlap.

This problem arises very often from the need for a base for the map of a level on which to work, in such a way that different independent sectors can be defined, connected irregularly, representing different areas of the game world.

An alternative to maze generation algorithms is the use of space subdivision algorithms[42], commonly used in graphics acceleration to save calculations when rendering a scene. These algorithms work on different data structures, such as the Quadtree (2D), the Octree (3D) and the binary tree. Its main function is to subdivide a large space into smaller subspaces .

The improvements provided by this method compared to the use of maze generators are the following:

- Less computational expense than any exponential algorithm of labyrinth generation. This makes it especially useful in large scenarios.

- Generation of a tree with the complete structure of the level, with its rooms and the connections that exist between them. Useful when determining zones and establishing unlokeable areas.

- It is not limited to a grid of "x" columns per "y" rows. So the results are more organic.

[42] An algorithm of subdivision of spaces is responsible for dividing a scene (usually already created) into a series of subspaces that contain its different elements following a certain criterion.

In this case, I will explain the application of a BSP tree[43] in a scene to subdivide it into rooms, adapting the method explained in "*Basic BSP Dungeon generation*" (Roguebasin, 2017)[44] so that it best suits certain needs.

1. The base space (A) is divided into 2 subspaces, (B and C) either horizontally or vertically. In the tree is therefore placed (B and C) as child nodes of (A). For convenience, it is a good idea to leave the smallest subspaces to the left of the tree the largest to the right, in case we need to sort them by size or use this property for something.

2. Step 1 is repeated recursively with the created subspaces until the subspaces reach a certain minimum size.

3. The rooms are placed within the subspaces, I recommend not to place the rooms in a random place within each subspace, but you should use a normal distribution function that takes into account the size of the child subspaces when placing the room. In this way you could make rooms of the same size tend to be closer to each other, and rooms with more disparate sizes tend to be more separated.

4. Once the rooms are placed, you have to go from the child nodes of the tree to the parent node connecting the rooms by corridors. Parent nodes have no room to connect to, so one solution may be to connect the parent's child nodes to their sibling node.

In this way, an optimal result can be achieved both in terms of processing speed and the amount of data obtained with which to work. Using this data to determine certain things is very simple, for example, to lock an entire branch of the map with a padlock, all you have to do is place the key in one of the rooms of one of the branches of the tree, and then place the padlock on one of the sister branches of the previous branch. In this way a dependency is determined, that is, access to one of the branches depends on access to another in order to advance.

With this binary tree you can also determine which is the longest branch, and which is the room at the end of these. In such a way that you can do things very easily, such as placing rewards for the player at the end of these branches with a value equivalent to the length of them.

[43] BSP stands for Binary Space Partitioning, and consists in storing the subspaces of a scene within a binary tree, following a particular requirement.
[44] Roguebasin. (19 de Diciembre de 2017). Basic BSP Dungeon generation. Obtenido de Roguebasin - Basic BSP Dungeon generation: http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation#Building_the_BSP

c) Cellular automata

The cellular automaton is a generation algorithm based on the interaction of different cells between them and the environment that surrounds them. To do this, these cells follow a series of parameters that define their behavior.

A very famous example of cellular automata, and one of the first of its kind, in the world of AI and procedural generation is the one presented by John Horton Conway (who died on April 11 , 2020 due to COVID-19), in his algorithm "*The game of life*".[45]

To use this algorithm you have to prepare, first of all, an environment on which to work on. This environment is usually represented as a grid of n*n. And within this grid, there can be different types of boxes with different attributes.

It is also necessary to define the type of "neighborhood" to be used, this refers to the boxes near the cell used to check the "surroundings" of it. The most common is to use moore's neighborhood, in which the 9 squares surrounding the cell are checked, or the Neumann neighborhood, in which only the squares directly above, below to the left and right of the cell are checked.

The existing checkbox types in the environment are also determined. In this case, since it is a cave, there are squares that define "rocks" (not trespassable) and soil (trespassable).

Once the environment is established, the different parameters of the behavior of each automaton cell that travels through it must be defined. These parameters depend on the objective of the algorithm, in this case it is interesting to show its application in the generation of organic caves. In the document "*Cellular automata for real-time generation of infinite cave levels*" (Johnson, Yannakakis, & Togelius, 2018)[46] the main attributes used to obtain the desired result are:

- *r*: Amount of rock type squares.
- *s*: Number of iterations.
- *T*: Amount of "rocks" to detect in the neighborhood.
- *M*: Size of Moore's neighborhood or Neumann's neighborhood, in this case Moore is used.

[45] “Conway’s game of life” es un algoritmo de Cell phone automaton based on only 3 main rules, but with just that it is able to generate structures that show a minimum of intelligence.
[46] Johnson, L., Yannakakis, G. N., & Togelius, J. (2018). Cellular automata for real-time generation. Copenhagen: PCGames

To proceed with the generation, a grid is generated with a random normal distribution of rock-type squares and soil-type squares. The number of rocks is specified in *r*, the remaining squares are soils (Figure 21).

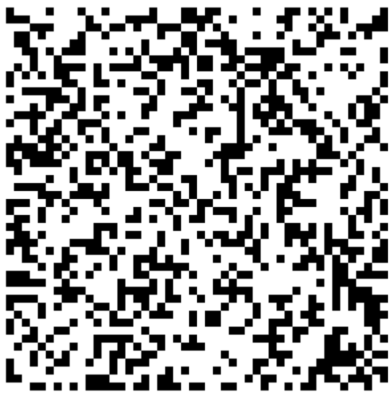


Figure 21 - Pseudorandom arrangement of *r* rocks in a grid.

Then, for each iteration *s,* go through each of the squares individually and check how many rock-type boxes are within their neighborhoods. If the number is greater than or equal to T, that box is stored on a new map apart as a rock type box, otherwise it is saved as a soil type box.

The result of this algorithm gives rise to complex soil packages, similar to those of the structure of a cave (Figure 22).

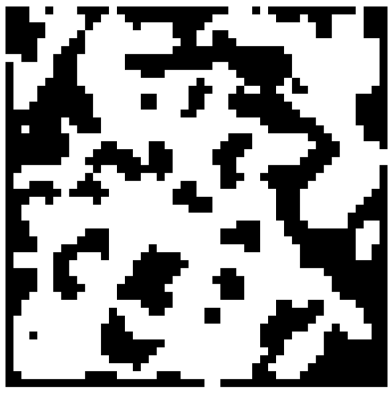


Figure 22 - Result of a generation using cellular automaton using the parameters r(0.4), T(5), s(2) and M(1)

4. CONCLUSIONS

Below are the results of the research, and its possible applications to a roguelike style game, both in terms of level structure and gameplay.

4.1. Noise

Randomness is a basic and indispensable concept in procedural generation. Of all the noise algorithms that I have researched (there are a lot of noises such as Voronoi, Pink, Blue… among others, that I have not covered here) I have come to the conclusion that noises such as Perlin Noise can be used in many different ways in the generation of both the structure of levels, the placement of objects, and division of the level into different zones.

The grayscale of the Perlin noise could be used to determine, for example, the probability of appearance of enemies or the difficulty of them. It could also be used to generate, or support other generators (such as the cellular automaton), to create the structure of an organic cave.

What problems does it present?

The amount of control over its outcome is very limited, so you wouldn't rely on it completely for a major generative task. E.g. you can’t generate a cave in which all your rooms are connected using Perlin, you would have to use an extra algorithm that is responsible for joining these rooms.

However, for aesthetic uses on which the gameplay of the game doesn't rely on, using Perlin might be a good idea. An example could be its use for the placement of light sources on the level, or to place elements that have different levels of wear such as cracks or broken walls, which require a gradient that feels continuity (cracked wall, followed by a more cracked wall, followed by a broken wall).

Fractal noises on the other hand only come in handy for generating procedural textures or in the creation of VFX such as particles, fire, clouds, etc.

4.2. Evolutionary algorithms

Evolutionary algorithms allow procedural generation to be applied to virtually any feature of a video game, whether it is the positioning of elements, the control of the difficulty of a level or the creation of the level structure. Especially in the latter case interesting results can be obtained by adjusting the fitness function.

However, these algorithms present a series of problems when implementing them for video games:

- They are computationally expensive, especially when their dimension is large. It is necessary to take into account the number of iterations necessary to obtain a desirable result and the requirement of having to check the viability of the level in each generation.

- A very common problem with this generation is the possibility of generating inaccessible rooms that make the level unplayable. This requires very well refined algorithms that prevent these situations from happening or global rules that do not allow these cases.

- The elements created have to be interpreted once generated in order to work with them. E.g. If a listing of the rooms on the level is required, but the algorithm is responsible for placing the walls, some way of defining which areas of the map are considered rooms is required.

It would be much more effective to generate a viable level from the beginning, which does not need to be evaluated to save computing time. To achieve this result it would be more optimal to use a construction system based on rules that would prevent the generation of unconnected rooms and other unwanted structures.

This evolutionary method could be used at a simpler level to generate the structure of the dungeons of a roguelike if it were necessary to determine characteristics such as: length of corridors, number of branching paths, number of rooms... Speed-wise, any rule-based system will be much faster and easier to program.

A solution to the speed problem of this algorithm would be the inclusion of *trained neural networks*, instead of the use of brute force to evolve the level. Apart from that if you seek to generate a complex level from 0 that respects a series of rules, this is one of the few "simple" solutions that does not require the definition and verification of thousands of different generation standards manually, but allows a much more generic approach.

In my case, I will go for the placement and subsequent alteration of predefined generic structures, in order to save the time of generation of the level, and thus allow to generate complex and interesting structures in terms of gameplay, without requiring an algorithm excessively complex and difficult to maintain / expand.

4.3. Rule-based generation

Rule-based generation algorithms have sparked my curiosity when it comes to quest generation. As shown in my example, you could start from a base structure for the dungeon to generate and iterate until you achieve a predetermined requirement. The only problem, already explained, is that graph structures are very complicated to capture in a grid, which is the base used by most roguelikes in their structure. That is why to make this possible, it would be necessary to generate the structure contained within the grid first using another algorithm, and then apply rules on it. Considering that some of these rules may not be applicable as explained above (Figure 18).

As for using L-Systems to create structures, they are especially useful as a form of representation for evolutionary algorithms, as they are easier to interpret and modify than something like a pixel array representation.

Suppose we want to generate a level structure as branched as possible, but with a large number of straight sections. This can be achieved by mutating an original axiom with a fitness function that favors these criteria, until an optimal result is achieved.

Another reason why this evolutionary approach is useful is to avoid symmetry. In certain cases, especially if it is an organic structure, a level of symmetry that is easily perceptible by the player is not something sought (since it makes the structures predictable). This can be achieved by mutating the genome in each iteration, so that there are subtle changes as it is represented. This makes it difficult to replicate the result (since several evolutions of the L-System are actually drawn in an overlapping way) but visually, it is more attractive.

4.4. Direct generation

When working with grids, the use of maze generation algorithms is one of the most effective solutions. They are simple algorithms that provide very good results to interconnect cells. They have certain limitations, such as the inability to decide the length of the corridors, or the number of intersections, but with some modifications they can generate a tree of nodes that allows working with them much more easily.

In the same way, binary partition algorithms are especially useful in terms of creation speed, which already generates a tree with the entire structure of the level, very useful to further develop the level with other algorithms, such as the use of grammars with rules on the tree itself to extend it and thus be able to determine locks (place keys and locks), place rewards and enemies, apart from being able to adjust the difficulty with the depth of each branch.

The dungeons generated with BSP provide an extra that can not be obtained using maze generation algorithms, and that is that they are not limited by a grid, so the rooms can be placed more irregularly by the level, creating a much more organic structure while preventing overlapping.

I will leave aside cellular automaton, more than anything because although it can be used on grids to define structures similar to rooms, generally an extra algorithm is always required that is able to connect those rooms that are not connected. And if this were not enough, another algorithm is also needed to determine which spaces of the generated structure are rooms and which corridors. This entails extra runtime and the result does not bring as many benefits as the binary tree that the other methods can provide .

4. METHODOLOGY AND ORGANIZATION

4.1. General objective

The objective is the development of a roguelike video game in Unity that has procedural generation of environments and voxel aesthetics. As I explained, Unity is the engine I have chosen it for the project. The roguelike genre is popular nowadays thanks to its popularity among streamers, which makes for a viable game.

4.2. Description of the project

The game will have the following features:

- Generation of procedural levels
- Puzzles
- Laser projectile rebound mechanics
- Destructible objects
- Controller and keyboard support
- 3D models with voxel aesthetic
- Postprocessing
- Shaders and other lighting effects

I seek to finish most of these points to get a playable prototype as a result, I doubt that a finished game is possible considering that I am doing this alone and there are time limitations.

4.3. Working methodology

I am doing the project alone so I will personally take care of the artistic section, as well as the design of the game and its programming. For this, a main research on procedural generation will be carried out that will then be combined with different mechanics and possible level designs. The artistic line in this case will seek to follow a retro style using the voxel as a modeling primitive, the aesthetics will also be influenced by the level design and the theme that is decided in the design process.

For the development of this project, due to the absence of other members and the consequent inability to hold meetings and follow any sort of “Agile” methodologies, I chose to follow a purely exploratory development based on holding biweekly meetings with the supervisor of the project in which to maintain follow-ups about the development.

In these meetings, weekly milestones to be met within development were specified and the different requirements of the game were specified as the development progressed.

I decided to use Unity as the engine for the development of the project because of its popularity in the sector, because I was familiar with it after been using it for 3 years in multiple projects and because of its plugins related to the import and optimization of models made from voxels. To host the project I decided to use GitLab for offering a free plan of 50GB of storage on the cloud that gave much more flexibility than the free plans of Bitbucket or GitHub.

First of all, I chose to create a roguelike due to its procedural nature and its flexibility in terms of level of gameplay elements. From there I was specifying the mechanics of the game and exploring different artistic styles:

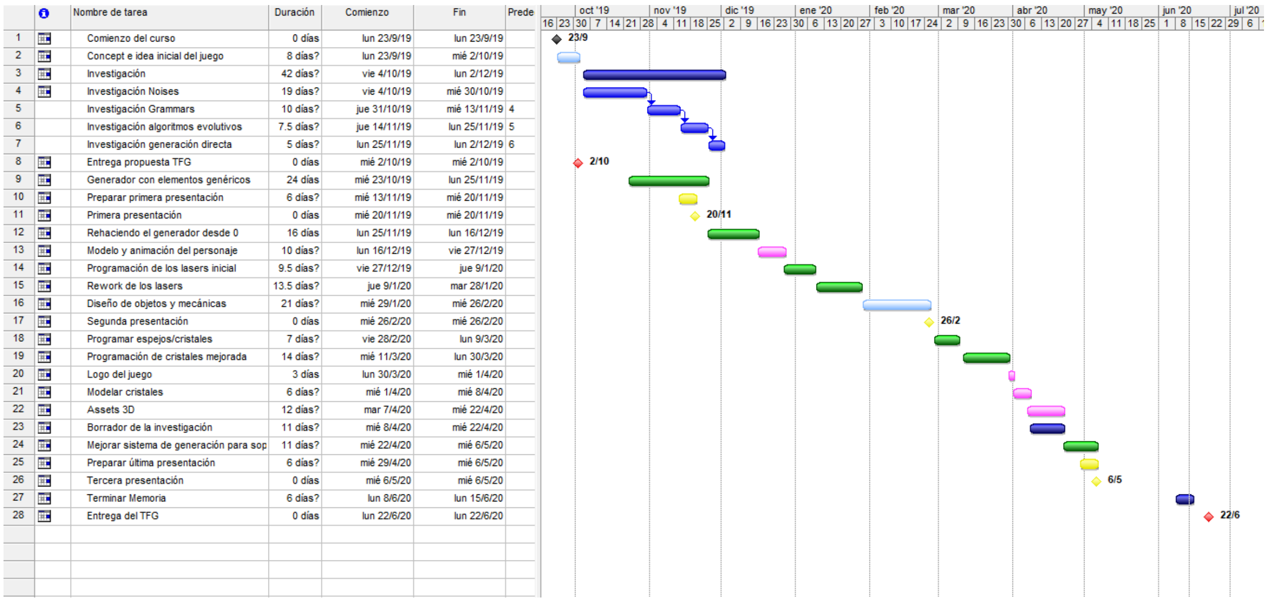
- Definition of the main mechanics, the bounce of laser projectiles against walls.
- Expansion of the main mechanics by means of 4 modifiers:
    - Spreader: A surface that multiplies the number of projectiles that hit it.
    - Directioner: A surface that reflects projectiles in a specific direction regardless of their original direction.
    - Move: A surface that alters the movement of the projectiles it collides with.
    - Teleporter: A surface that teleports projectiles to another Teleporter.
    - FX: A surface that alters the characteristics of a projectile, providing different powers to it, such as the ability to paralyze or burn enemies.
- Definition of 56 base elements for the game represented with ASCII characters, these elements are generic but can be used to specify further characteristics.
- With these elements, create the structure of the **.dun** file, used to contain the representation of pre-created rooms and that would allow you to define extra characteristics to each tile, such as the type of enemy in a tile of a room M (monster).

Once the research was finished, I began the development of the dungeon generator with the following elements:

- Room connector following a modification of the backtracking algorithm to interconnect the rooms.
- Level structure generator based on grammars.
- System of loading, selection, rotation and turning of rooms.
- Algorithm for generating corridors between rooms.
- Level structure based on rooms of different types:
    - Enemies: Room that contains enemies or puzzles.
    - Start: Room where the player starts
    - Finish: Room where the player ends.

- Boss: Room where there is a battle against a boss.
- Locked: Closed room that requires a key.
- Key: Room with a key that allows you to unlock a Locked room.
- Reward: Room with rewards (items, powerups, health, money, ammo...)
- Special: Room that diverges into 2 or 3 different paths, can contain shops and NPCs.

- Setter of the 3D elements in the Unity Scene.
- Algorithm of selection of elements with "context awareness" using the Moore neighborhood to select walls that follow a certain continuity.
- Character design, rigging, animation and programming.
- Programming of the logic of laser projectiles.
- Programming of the mirrors (spreader, directioner, mover...).
- Design of the game logo and title.
- Improvement of the "context awareness" algorithm to support the selection of different soils.
- Creation of models of walls, floors, obstacles (rocks), boxes, enemies, money, healing items, mirrors / glass, doors ...
- Optimizing voxel models for Unity.
- Room graphic design tool.



	ℹ	Nombre de tarea	Duración	Comienzo	Fin	Prede
1	▦	Comienzo del curso	0 días	lun 23/9/19	lun 23/9/19	
2	▦	Concept e idea inicial del juego	8 días?	lun 23/9/19	mié 2/10/19	
3	▦	Investigación	42 días?	vie 4/10/19	lun 2/12/19	
4	▦	Investigación Noises	19 días?	vie 4/10/19	mié 30/10/19	
5		Investigación Grammars	10 días?	jue 31/10/19	mié 13/11/19	4
6		Investigación algoritmos evolutivos	7.5 días?	jue 14/11/19	lun 25/11/19	5
7		Investigación generación directa	5 días?	lun 25/11/19	lun 2/12/19	6
8	▦	Entrega propuesta TFG	0 días	mié 2/10/19	mié 2/10/19	
9	▦	Generador con elementos genéricos	24 días	mié 23/10/19	lun 25/11/19	
10	▦	Preparar primera presentación	6 días?	mié 13/11/19	mié 20/11/19	
11	▦	Primera presentación	0 días	mié 20/11/19	mié 20/11/19	
12	▦	Rehaciendo el generador desde 0	16 días	lun 25/11/19	lun 16/12/19	
13	▦	Modelo y animación del personaje	10 días?	lun 16/12/19	vie 27/12/19	
14	▦	Programación de los lasers inicial	9.5 días?	vie 27/12/19	jue 9/1/20	
15	▦	Rework de los lasers	13.5 días?	jue 9/1/20	mar 28/1/20	
16	▦	Diseño de objetos y mecánicas	21 días?	mié 29/1/20	mié 26/2/20	
17	▦	Segunda presentación	0 días	mié 26/2/20	mié 26/2/20	
18	▦	Programar espejos/cristales	7 días?	vie 28/2/20	lun 9/3/20	
19	▦	Programación de cristales mejorada	14 días?	mié 11/3/20	lun 30/3/20	
20	▦	Logo del juego	3 días	lun 30/3/20	mié 1/4/20	
21	▦	Modelar cristales	6 días?	mié 1/4/20	mié 8/4/20	
22	▦	Assets 3D	12 días?	mar 7/4/20	mié 22/4/20	
23	▦	Borrador de la investigación	11 días?	mié 8/4/20	mié 22/4/20	
24	▦	Mejorar sistema de generación para sop	11 días?	mié 22/4/20	mié 6/5/20	
25	▦	Preparar última presentación	6 días?	mié 29/4/20	mié 6/5/20	
26	▦	Tercera presentación	0 días	mié 6/5/20	mié 6/5/20	
27	▦	Terminar Memoria	6 días?	lun 8/6/20	lun 15/6/20	
28	▦	Entrega del TFG	0 días	lun 22/6/20	lun 22/6/20	

Figure 23 - Gantt Chart

The "What if?" methodology has been used as a risk analysis technique due to my lack of knowledge regarding risk analysis and the experimental nature of this project.

5. DEVELOPMENT

5.1. Profile

When I started high school I wanted to learn how to write computer programs. Sadly I was born in a small town with ~20mb of broadband and at computer science class we were taught how to use Office. Even so, I had been thinking about the idea of making a video game for some time and although I did not know how to make it a reality, every day I dedicated a few hours to develop it, even if I could only do it on paper.

When I finished high school I changed my mind about studying computer engineering when I saw all the new video game development degrees emerge, so I went to [redacted]

There I learned how to program, and although I don't have published games I feel very capable of developing games by myself. In [redacted] we were not separated into groups of programmers, artists and designers like in [redacted], so I was able to increase my knowledge in the three branches all the time I was there and this allows me to perform much better as an independent developer.

This year at [redacted] [redacted] I have worked as an internship in [redacted], exercising the role of programmer.

In this project, on the other hand, I've had to exercise the roles of programmer, artist and designer to push it forward.

5.2. Planning of tasks and efforts according to work profile

5.2.1. Assignment of tasks

As already explained in the methodology chapter, I personally took care of the total development of all the game development tasks.

5.2.2. Tasks, subtasks and relationships between them

In this section I will only detail in depth the tasks related to programming because it is the branch to which I belong within the career. I will however cover some parts about art and design tasks that I consider important to contextualize my work.

A) Generation of dungeons

Design: The generation had to be able to create easy-to-parameterize dungeons that were well balanced and allowed manual editing of rooms in a simple way. Based on tests I decided that the minimum size of a room should be 4 * 4 and the maximum size 16 * 16 without counting walls, to avoid tiny or giant rooms. Apart from this, the structure of the level must make sense, so that if the player goes through 5 very difficult rooms and arrives at a room with no exit in which they have to turn around, at least in that room there should be some kind of reward for the player. One of those rewards does not necessarily have to be an item, but can be a key with which to open the door of a room through which to continue.

Programming: First, I started by dividing the "floor" into different interconnectable nodes, each node having access to its neighboring nodes within their corresponding Neumann neighborhoods. The size of all nodes is presented with a two-dimensional array of "tiles", and the nodes are arranged in a two-dimensional array of *H* nodes height and *W* nodes width.

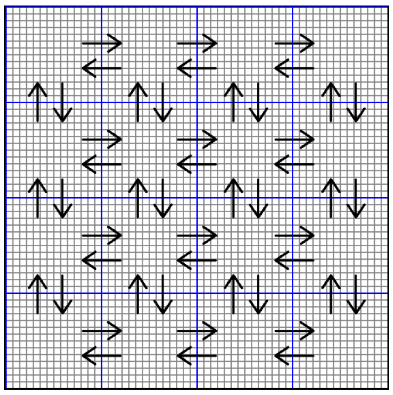


Figure 24 - Interconnected nodes

When interconnecting the nodes, I used a modification of the backtraking algorithm that when reaching a dead end, continues from another available random node. As I've already explained this modification balances the shape of the labyrinth and avoids generating too many intersections. This work is done by the *Floor* class, which contains an array of *Node* class objects, which once inicilized with *InitNodeLinks()* interconnect with each other using the *InterlinkNodes()* function, producing the following result:

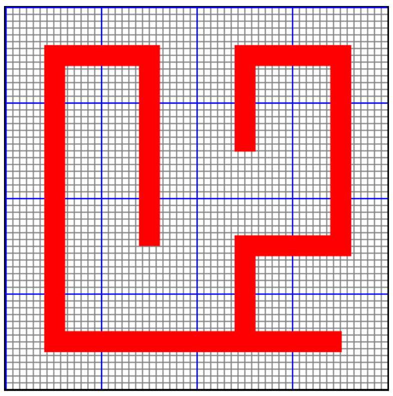


Figure 25 - Connection generated by modified backtracking

With this generation a tree of node connections is created which represents the different branches that exist on the floor, the parent node is the first used in the generation of the connections, in this case [2, 1]. At the end of this parent's branch is a fork into two branches whose father is [2, 3].

In the next step, a category is assigned to each of the nodes depending on the room they will contain, which can be:

- (E) Enemies: Room containing enemies or puzzles.
- (S) Start: Room where the player spawns.
- (F) Finish: Room where the level ends.
- (B) Boss: Room where there is a battle against a boss.
- (L) Locked: Closed room that requires a key.
- (K) Key: Room with a key that allows you to unlock a Locked room.
- (R) Reward: Room with rewards (items, powerups, health, money, ammo...)
- (S) Special: Room that diverges into 2 or 3 different paths, may contain shops and NPCs.

To do this initially all the children of a branch are marked as nodes of type *Enemies* and from there iterates using grammar rules as follows:

→E→E→E→E can be replaced (depending on length) by:

- →E→E→E→E→R
- →E→E→B→R
- →E→E→F
- →E→E→B→F

R←E←E←E→E→E→R can be replaced by:

- B←E←L←E→E→E→K
- R←E←L←E→E→E→K

←E→ is replaced by:

- ←S→

The first node used in the creation of the tree is assigned as the *Start* room:

- S→

Applying these rules in the previous case we obtain this result:

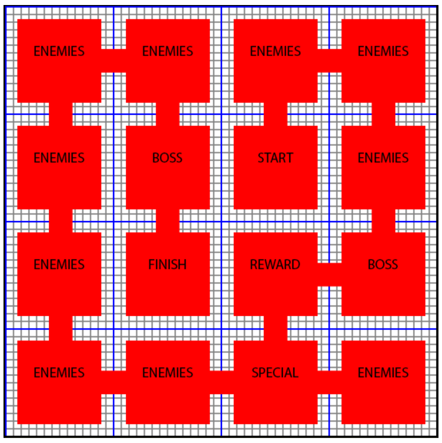


Figure 26 - Node Structure

Once the general structure of the level is ready, it is necessary to load the rooms that will be placed inside each node. This is done by using the *DngRoomLoader()* which uses the structure of the **.dun** file to load pre-designed rooms.

However, it is not enough to just load them and place them directly in a location within each node, but the type of room, its size and the orientation of its inputs and outputs must be the same as those of the nodes, for this **5 types of room direction shape were defined**:

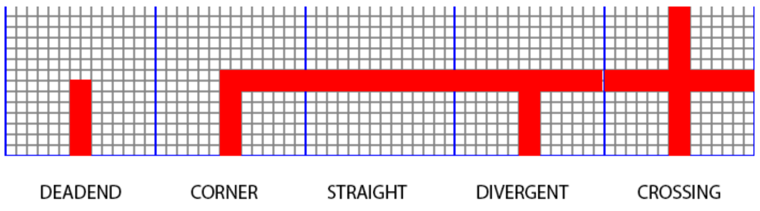


Figure 27 - Possible shapes of a room direction

To avoid having to make the same room with different entry and exit positions and thus be able to recycle rooms, the possibility of rotating and flipping rooms was implemented so that the algorithm is responsible for making the room fit inside the target node, E.g.

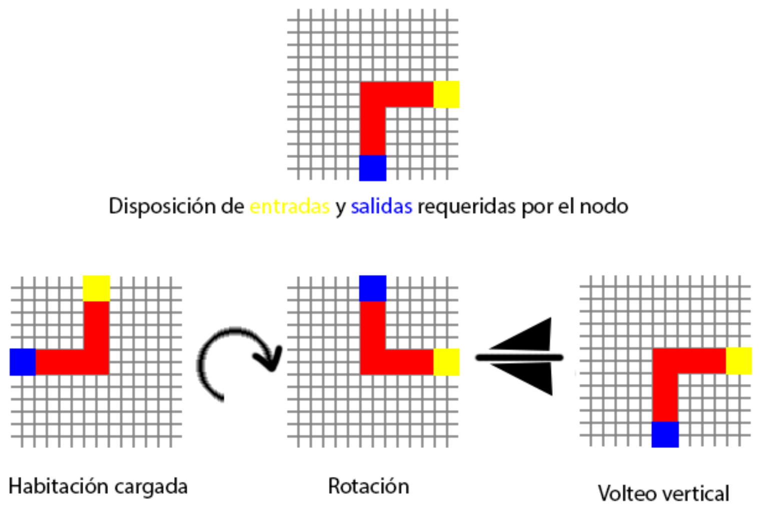


Figure 28 - Direction matching

After these transformations the room is ready to be placed inside the node, for safety a margin of 1 block is left when placing them to avoid collisions during the next step: placing the corridors.

For the connection of rooms using corridors I deduced an algorithm of connection of point A (the exit of a room) with point B (the exit of a room) following the steps that are used in cellular automata to travel in a specific way an array. An approximation of the formulas used in most roguelikes to connect corridors is as follows:

I. Opposite parallel walls | |

- The difference w in x and y is calculated between point S (start of the corridor) and E (end of the corridor).
- Next, Sx (if the corridor is horizontal) or Sy (if the corridor is vertical) is incremented. Each time Sy or Sx is incremented, w is decreased and the process is repeated until w is half of its initial value or a random value between initial w and 1.
- Each time Sx or Sy is incremented, that S position is marked as a corridor.
- Once this point is reached, Sy (if the corridor is horizontal) or Sx (if the corridor is vertical) is incremented until Sx is equal to Ex (in the case of vertical) or Sy is equal to Ey (horizontal case).
- For the remaining w, Sx (in horizontal case) or Sy (en vertical case) is increased until Sy or Sx are equal to Ey or Ex respectively.

II. Parallel walls _ _

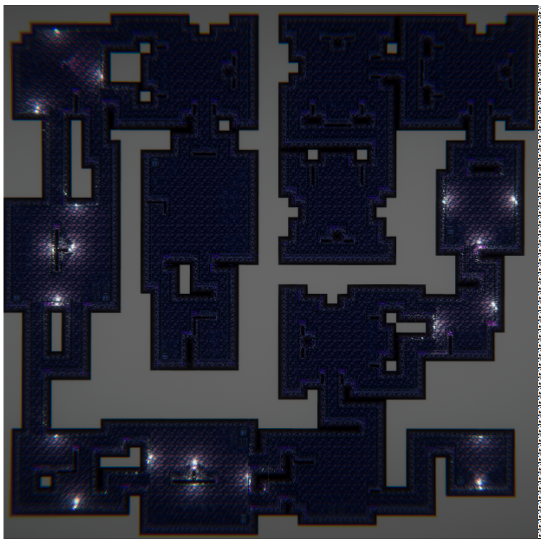
- The difference w in x and y is calculated between point S (start of the corridor) and E (end of the corridor).
- Sx or Sy is incremented (depending on whether it is a vertical or horizontal corridor) until it is equal to (Ex - Sx * 2) or (Ey - Sy * 2) respectively.
- Each time Sx or Sy is incremented, that S position is marked as a corridor.
- The rest of the operation is similar to that used to connect opposite parallel pairs, but the first step is repeated until w is 0.

III. Perpendicular walls | _

- If both Sx and Sy are smaller than Ex or Ey respectively, steps are performed to connect opposite parallel walls, making an offset at the objective point E. Once this offset is reached, it is continuous by increasing Sx or Sy until reaching the original E.
- Otherwise, Sx or Sy is first incremented until an offset is reached on that axis. Once here, we continue to use the method to connect opposite parallel walls using the offset point as S until reaching point E.
- As in the rest of the methods, each time Sx or Sy is incremented, that S position is marked as a corridor.

As an improvement to this method, corridors of different widths, or of irregular width, can be included, along with the possibility of adding enemies, objects and decorations within the corridors. But at first with the normal generation it is enough to move between rooms.With all this, the final result of the floor looks like this:

Figure 29 - Final Representation in Unity



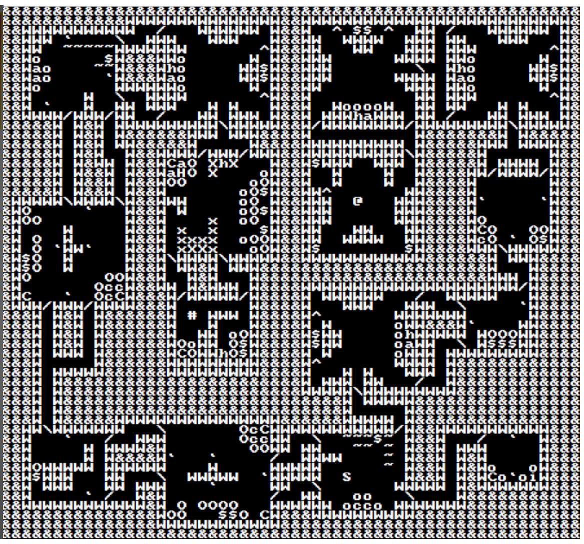
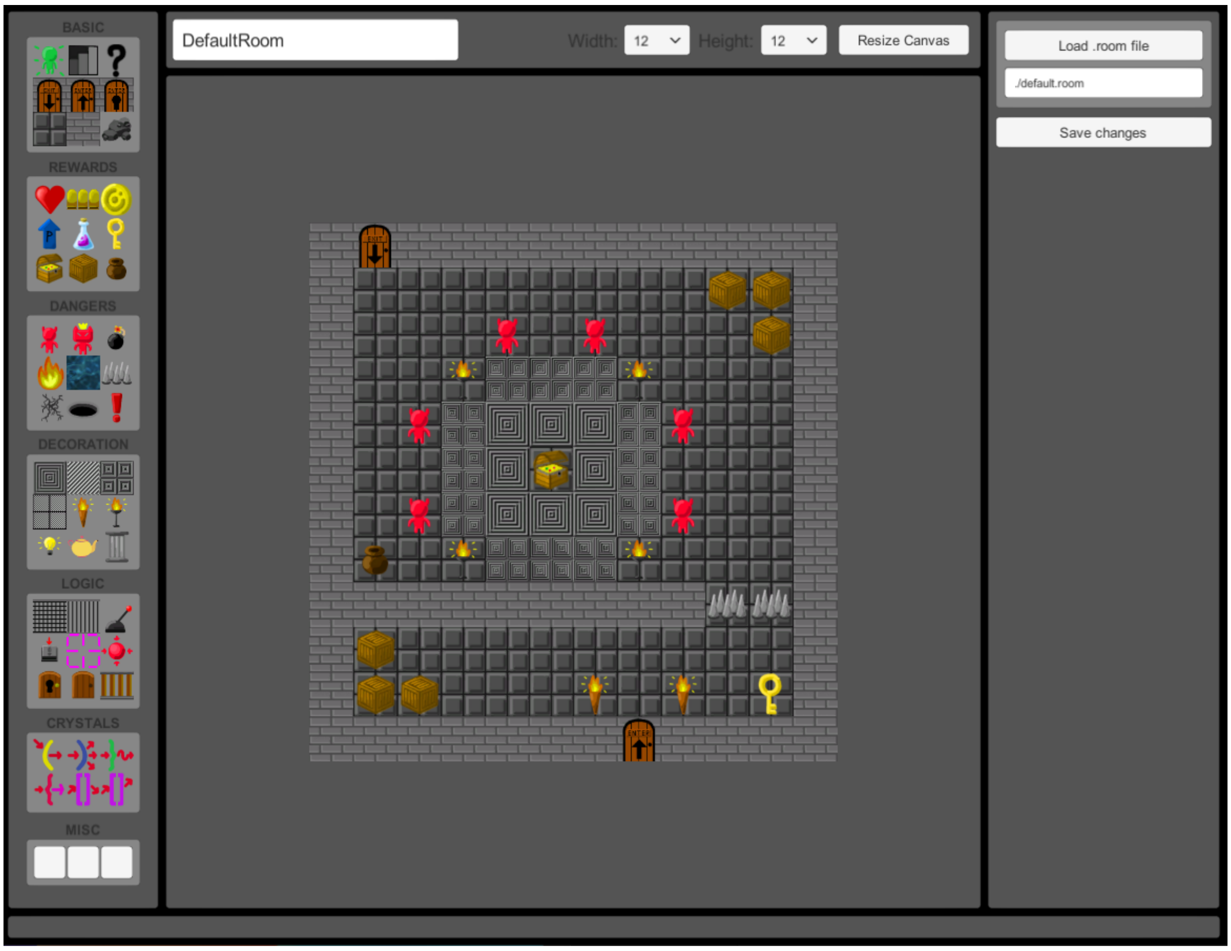


Figure 30 - ASCII Final Representation

A.A) Level Editor

Design: The goal of developing a level editor was to make the process of creating rooms much easier. Instead of a level editor, the program allows you to create individual rooms that can then be saved as a **.dun** file that defines all the rooms from which the algorithm can choose to generate the level. The objects that can be placed in the editor are the same ones that are defined further down in the "Object Design" task, which are generic objects that represent basic aspects such as Monsters, Items, Powerups without reaching specific data such as which specific Monster, Item or Powerup is going to appear in that area. These types of decisions should be left to the generation algorithm, but as an extra the possibility of manually specifying this type of details was contemplated in order to make some more specific rooms.

Art: In this case the art is not especially detailed. They were simply looking for a simple fix since this tool is not part of the game itself but is an added extra .



Programming: This program was part of an a project for middleware class, so we were forced to use DLLs written in C++ to perform most of the functions of the program. In the future if I continue with the development I plan to remake this tool from in javascript to avoid DLLs dependencies.

B) Main mechanics of firing laser projectiles

Design: For this mechanic I was inspired by the game "Deflektor" (Deflektor - C64-Wiki, 2020)[47] for the commodore 64 along with the lasers used in "Portal2". (Portal 2)[48] The concept was to create bullet hell-style projectiles, but with an added bonus of logic and complexity. Instead of using a continuous laser, I opted to use short, intermittent lasers that would make it more difficult to hit enemies.

The principle is simple, you have lasers that you can shoot and these bounce against some surfaces. To help you have a trail that tells you the trajectory that laser would follow if you shoot it with your current position and orientation. To encourage the use of the mechanics of reflecting projectiles against surfaces it thought it would be interesting to make the lasers more powerful the more they bounce, to encourage the player to look for a much more complicated shot instead of a direct shot at enemies.

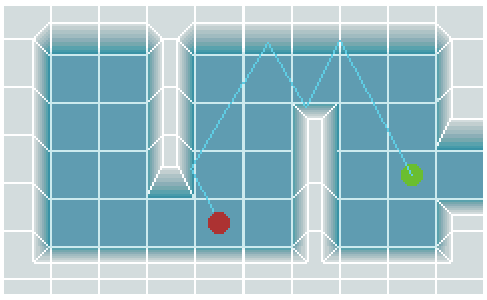


Figure 31 - Laser mechanics animation (click to play)

Art: For the visual style of the lasers I opted to use the Unity trails mixed with the bloom included in the post-porcessing package to give neon effect to the lasers. I also made different models in voxel in different sizes, but in the end these are barely visible when mixed with the emmisive of the material, the bloom and the trail.

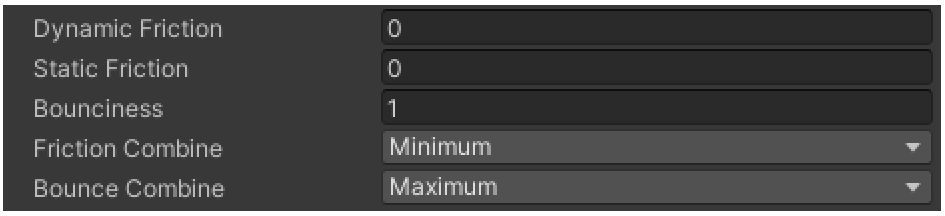
Programming: I originally started by programming the bounce of the lasers using raycasts. So that the laser launched a raycast in front when it appeared, and that raycast was reflected recursively on the different surfaces with the "reflectable" tag. This worked, but in order to detect collisions against enemies I ended up including a rigidbody component with a spherical collider. This made me realize that with the right settings I could use the physics system to make the bounces, and thus save me from having to throw all those raycasts each frame (throwing them all at once when spawnearing the laser is not enough

[47] https://www.c64-wiki.com/wiki/Deflektor
[48] https://en.wikipedia.org/wiki/Portal_2

since the scenario can change from the moment it appears until it reaches its destination. , an enemy can be placed in the middle of his trajectory or one of the obstacles it had to hit could break before reaching it). In addition, calculating the frame in which it is considered to have reached the point of collision was quite complicated.

With Unity's new physics-based system there were a number of settings that had to be changed to make the bounces accurate, first of all:

- To create a perfectly elastic bounce: Create a physical Unity material with the values shown here:



Dynamic Friction	0
Static Friction	0
Bounciness	1
Friction Combine	Minimum
Bounce Combine	Maximum

Figure 32 - Perfectly elastic material

  In this way the bounce is perfect and the resulting motion vector after the collision is equal to the motion vector before the collision reflected with the normal of the surface on which it is collides.

- These adjustments however were not enough to get "the perfect bounce". It was necessary to change a couple of things in the physics settings of the Unity project to achieve this:
  - Physics->Bounce threshold : By default is 2, but this required the projectiles to go at high speeds in order to bounce, so I lowered it to 0.05.
  - Time->FixedTimestep: By default is 0.02, but this caused the bounces to be inaccurate and the trajectory plotter not to square with the final trajectory of the projectile. This is why I ended up lowering it to 0.006, this makes the game perform more physics calculations per second, but it solves the problem.

The result of these changes allows the projectiles to reach a velocity of 32 units per second without losing accuracy with each bounce. Unlike the default values, in which the precision is lost if going faster than 16 units per second.

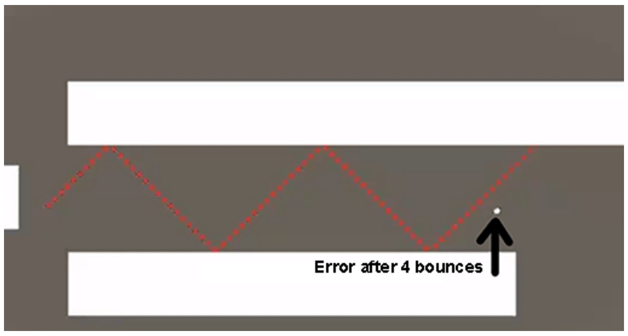


Figure 33 - Physics Accuracy Error in Unity

B.A) Projectile reflection crystals

Design: To give more interest to the main mechanics of the game it thought that it would be a good idea to add a series of elements that would modify the behavior of the lasers when colliding with them. From here I came up with several possibilities:

- Spreader: A surface that multiplies the number of projectiles that hit it.
- Directioner: A surface that directs projectiles in a specific direction regardless of their direction.
- Mover: A surface that alters the movement of the projectiles it collides with.
- Teleporter: A surface that teleports projectiles to another Teleporter.
- FX: A surface that alters the characteristics of a projectile, providing different powers to it, such as the ability to paralyze or burn enemies.

These elements are key when it comes to building interesting environments and puzzles because they allow different styles of gameplay and bring variety to the game.

Art: For the design of the crystals I used the voxel editor “MagicaVoxel” created by @Ephtracy. The shape of the crystal attempts to represent the effect of the crystal:

- The spreader is a prism, which causes the decomposition of colors.
- The directioner is shaped like an arrow, which points in the direction in which the laser redirects.
- Moving is a circular prism, which distorts the light that passes through it.
- The fx is a corundum crystal, which exists in nature in many different colors, the idea is that depending on its effect it has a different color.
- The teleporter shares the same glass as the fx until I find a type of glass that fits with it (for precision reasons, this glass has to have a flat shape that does not affect the puzzles in which it is used).

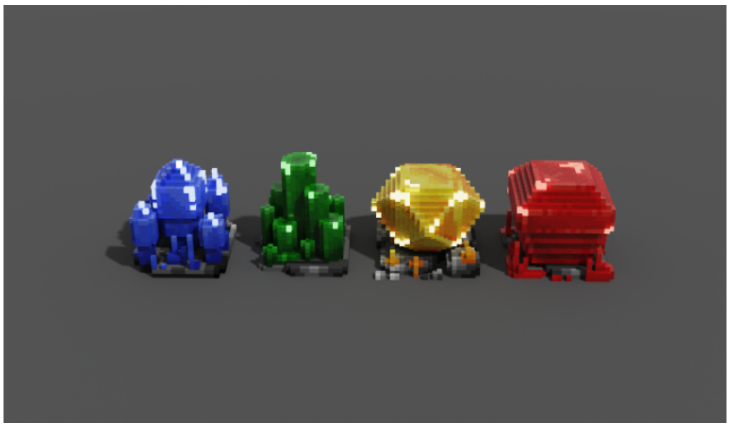


Figure 34 - Models of the crystals

Programming: I had to program the crystals 2 times, so I will explain what approaches I used on both occasions:

- The Gargollo method: It was taught by a professor at [redacted] and is especially useful when making flexible and easily extensible components using other components. The system workes like this:
    - Create a *CristalBase* component with *OnEnter(laser)* and *OnExit(laser)* events.
    - Have another secondary component for each specific crystal (CristalDirectional, CristalMover, CristalSpreader...) with a series of specific methods depending on what each crystal does.
    - In the *Awake()* method, the secondary components subscribe their secondary functions to the *OnEnter(laser)* or *OnExit(laser)* method of the *CristalBase* component, depending on what each crystal needs to do.
    - The *CristalBase* component invokes its *OnEnter(laser)* and *OnExit(laser)* events from its *OnCollisionEnter(collision)* and *OnCollisionExit(collision)* methods when the colliding object has a Laser component, which is passed as an argument to the event invocation.

  This method would have been useful if I was really interested in combining the effects of several crystals into just one (something I originally had in mind), but the overuse of events made the code quite convoluted, so I ended up changing it.

- The interface method: This method arose from the following problem: Imagine that we have the base class *Enemy*, and then from that class inherit *FliyingEnemy*, *GroundEnemy* and *WaterEnemy*. If we follow the cause-and-effect law, if Mario jumps on an Enemy, it is Mario who kills the Enemy, so the Enemy's death function should be invoked by Mario. And so a call is made to *enemy.JumpOn()*.

For simple cases this works well, but *it may be the case that we are not interested in WaterEnemies having a JumpOn() method because jumping on top of them should not be possible, since they are in the water*. In that case, the *JumpOn()* method can no longer be placed in the parent class and the *JumpOn()* of *GroundEnemy* and *FliyingEnemy* would be independent of each other, so within Mario we would have to do this awful thing:

```
// Worst code ever:

Enemy enemy = other.GetComponent<Enemy>();

if(enemy){
      if(enemy is FliyingEnemy){
            ((FliyingEnemy)enemy).JumpOn();
      }
      else if(enemy is GroundEnemy){
            ((GroundEnemy)enemy).JumpOn();
      }
}
```

This code will get worse as new classes who inherit *Enemy* are added that need to have a *JumpOn()* method. The solution? Create a Jumplable interface with the *JumpOn()* method implemented by *FliyingEnemy* and *GroundEnemy*, so you just have to do:

```
other.GetComponen<Jumpable>()?.JumpOn();
```

The? allows JumpOn() to be called only if the object is not null

Going back to the crystals, I made a Reflectable interface, so that when the laser hits the mirror, I can do:

```
other.GetComponent<Reflectable>()?.Reflect(this);
```

This breaks the laws of cause-effect a bit because the crystal actually reflects the laser, so it should be the crystal that calls the method *Reflect()*, rather than being the laser within its *OnCollisionEnter()*.

Regarding the implementation of the *Reflect()* method in each crystal, here is a small summary:

- Spreader: The vector of motion prior to the collision with the crystal is obtained, it moves to the opposite end of the crystal, it is normalized, a lerp of the rotation of the vector is made in such a way that it results in x vectors that form an arc. A new laser clone of the original is created in each vector x changing its direction of motion to that of its corresponding vector. You have to be careful not to enter an infinite loop of laser creation preventing these new lasers from being in contact with the crystal that created them.
- Directioner: The direction of the colliding laser is turned to the normal vector of the collision.
- Mover: The motion function defined within the laser is changed when it collides with it.
- Teleporter: The absolute transformation matrix of the laser is obtained when colliding and multiplied by the reverse transformation matrix of the teleporter, the result is multiplied by the transformation matrix of the target teleporter and the direction of the laser is reversed. In the case of using an inverted teleporter, the result is also rotated 180 degrees on the Y axis.

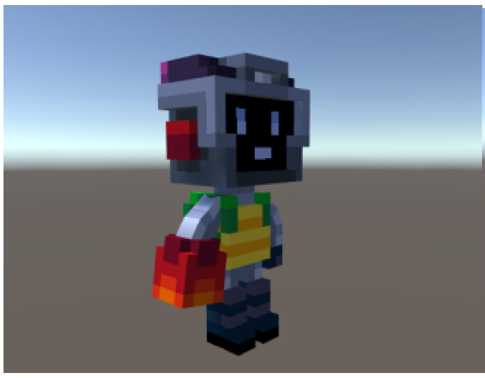
B.B) Character

Design: The character was necessary to be able to shoot the lasers, it had to have a large weapon that occupied its entire hand, Megaman style, from which it could shoot them. The game takes place in a cave, so the original model that was in mind was that of a miner, but after a series of failed attempts it was decided to change the aesthetics and the character became a robot, something that matches the neon retro aesthetic that ended up taking the game.

Art: A former [redacted] teacher noticed a tweet I put with the character's model and gave me a number of ideas to fix it. The final design did not convince me so I decided to discard it and start from scratch. [49] At one point the red handkerchief of the previous model reminded me of the jaw of a robot, so it occurred to me that, since robots are quite rectangular, the model would surely look quite well made in voxels. In the end I managed to model VIC, a robot still without history, but that served me to have a base on which to animate and move around the stage[50].

[49] https://twitter.com/i/status/1211029827144310786
[50] https://twitter.com/Lucinotion/status/1212504262531522560

Figure 35 - Final model of VIC protagonist



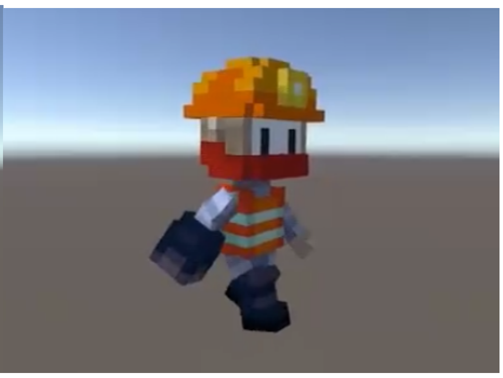


Figure 36 - Original model of mining protagonist

Programming: The main problem with character programming is the control and management of the multiple layers of animation needed to allow the character to do things like shoot while walking, aiming the gun, breathing while aiming... To achieve the blending of animations, different layers were used for the different states (breathing, walking, aiming) so that several animations could be played at the same time.

Apart from this, things like the angle of the camera or its position were not completely defined. I took into account the possibility of the camera rotating during the gameplay and adjusted the motion input so that it was relative to the camera.

C) Objects and elements of the world

Design: For the inventive process of designing new objects to place in the game, the classic objects of most roguelike games were taken as a base, such as healing objects, money, items, powerups... along with objects that support the logic of the game such as switches, pressure plates, event triggers, reflective walls, glass, locked doors...

On some occasions the limits imposed by ASCII characters tested my ability to create objects without losing a logical connection to the symbol they represent. In total, the following available tiles/objects were defined:

- @start: The protagonist's spawn place on the dungeon floor.

- # end: Exit from the current floor.
- / entranceDoor: Entrance door in a room.
- % lockedEntranceDoor: Closed entrance door that requires a key.
- \ exitDoor: Exit door of a room;
- floor: Floor of the room or hallway.
- . altFloor1: Alternative flooring generally used to create carpets.
- : altFloor2: Alternative floor used to place carpets on top of carpets.
- , floorTransitionToAlt1: Transition from normal soil to alternative soil 1. Useful at the programming level so as not to have to have different layers of floor, and useful at the art level to be able to design the edges of the carpets independently.
- ; floorTransitionToAlt2: Transition from alternative soil 1 to alternative soil 2.
- ' wallLight: Wall light used for torches and lamps.
- " floorLight: Floor light used for floor lamps.
- ' specialLight: Special ambient omnidirectional light located on the ceiling.
- * Switch: Switch used to activate different mechanisms. It is activated when colliding with a laser.
- _ pressurePlate: Switch used to activate mechanisms that is activated with a weight or when the player steps on it.
- = stairs: Stairs for descending into a lower ground elevation level.
- + untrasspasableCellWall: Grid wall that cannot be pierced by lasers.
- | passableCellWall: Grid wall that can pass the lasers.
- ~ liquid: A pool of liquid that damages the player and can burn or poison him.
- _ pit: Bottomless pit that damages the player if it falls into it.
- ^ spikes: Spikes that rise and fall off the ground at regular intervals.
- ¡ trap: Random trap.
- ? random: Random item.
- (directioner: Explained above.
-) spreader: Explained above.
- { move: Explained above.
- } fx: Explained above.
- [tp: Explained above.
-] inversedTp: Like the tp but the exit angle is the same as the input angle.
- $ money: In-game money that can be spent in stores.
- A ammo: Special ammunition from the game.
- B boss: Boss of the game,

- C crate: Destructible box that can contain items.
- D decoration: Decorative element such as statues, tables, altars...
- E explosive: An obstacle that can explode in certain circumstances.
- F fire: Fire that does damage and can burn enemies or the player.
- G gate: Decorative door that can be opened and closed, some enemies can not open it.
- H health: Life healing item.
- I item: Item usable by the player temporarily.
- J jar: Vase container of objects of lower level to the crate.
- K key: Key that opens closed doors.
- L lock: Closed door, different from the closed entrance, which is opened by a switch or a pressure plate.
- M monster: An enemy.
- N npc: An npc with which the player can talk and interact.
- O obstacle: A destructible obstacle.
- P powerup: An improvement that raises the player's statistics permanently.
- Q quest: A type of shop that gives you money in exchange for completing certain objectives and missions.
- R rails: Slippery floor on which the player can slide but not change his direction.
- S shop: A store.
- T treasure: Chest containing items better than those of the crate.
- U usable: Object with physics that can be moved around the room and that allows you to activate pressure plates with it.
- V lowerWall: Wall that is at a lower level than the rest.
- W wall: Wall that is at the normal level of the room.
- X crack: Cracked ground with different levels of breakage that breaks if the player steps on it too much, causing it to fall into a pit.
- Y pillar: A non-destructible decorative pillar
- Z zone: An invisible trigger that executes a script when the player activates it.

Programming: Since all objects are placed on a single layer, to determine what type of soil is under an object or enemy, an algorithm that deduces this is necessary. To help simplify the algorithm, transition tiles were created between alternative floor 1 and alternative floor 2 so that it could be more easily determined whether an object is within an alternative floor 1, an alternative floor 2 or a normal floor, something that is especially difficult when placing objects on the edges of a type of floor.

The solution provided by this system is based on Moore's neighborhoods and follows the following 3 simple rules:

- If in the Moore neighborhood of the object there are alternative floor tiles, in that case an alternative floor is placed.
- If in moore's neighborhood there are normal floor tiles and alternative floor tiles, in that case a transition is placed.
- If in the Moore neighborhood there are no alternative floor tiles, a normal floor is placed.

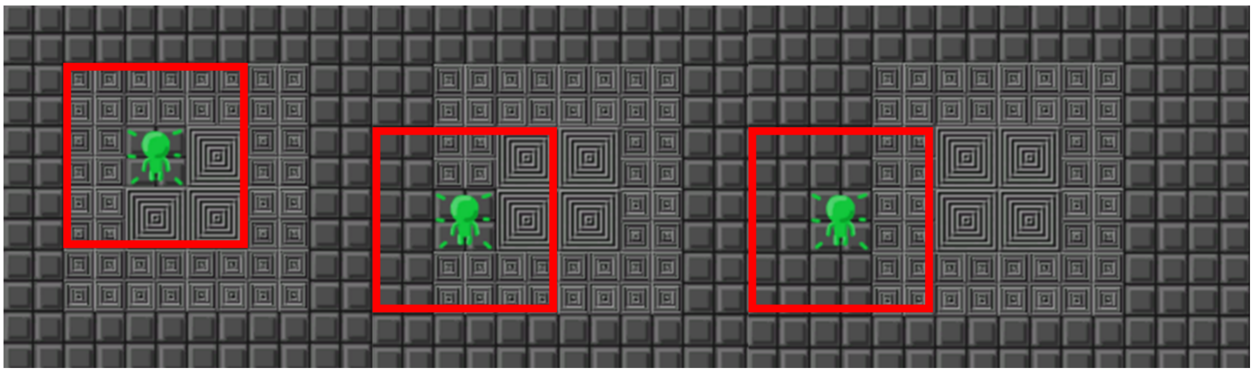
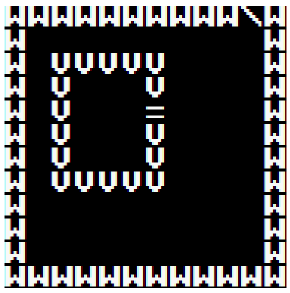


Figure 37 - Self-soil selector rule 1, 2 and 3

This system is not perfect, since placing many objects together loses the context and stops working, but if you are careful you can create levels in a simple way.

Another similar system is used to have different ground levels at the level. This system uses the Neumann neighborhood to detect if the first ones you find with reference to the soil/object to be placed are W(wall) or V(lowerWall) to determine its height. If it is first located at a lowerWall it means that the ground is at a lower height, if a wall is found it means that the ground is at the normal height:


```
WWWWWWWWWW\W
W          W
W VVVVV    W
W V   V    W
W V   =    W
W V   V    W
W V   V    W
W VVVVV    W
W          W
W          W
W          W
WWWWWWWWWWWW
```
Figure 38 - Example of different terrain levels

In this case all the tiles inside the rectangle of lowerWalls will be placed one unit below the rest. Apart from this, many objects need to use the Neumann neighborhood to calculate their orientation, for example the stairs (=) shown in the previous example, the bars (+) and (|), the doors (G), entrances (/), closed entrances (%), exits (\), closed doors (L) and the lights that are placed on walls (').

Apart from all this, there are some objects especially complicated to place since they use "context-awareness" to select a different 3D model depending on the elements that are within their Moore neighborhood. The two most important examples are walls and floors.

- Floors: The floors are connected logically to maintain continuity, this is achieved by something I call "Moore Patterns" (because I have not found any information about them on the internet), these are 47 patterns formed each by an array of zeros and ones of length 8 that represents the shape of the piece to be placed. For convenience I treat this as a byte to be able to perform operations with bitwise operators.

Figure 39 - Moore Patterns 1

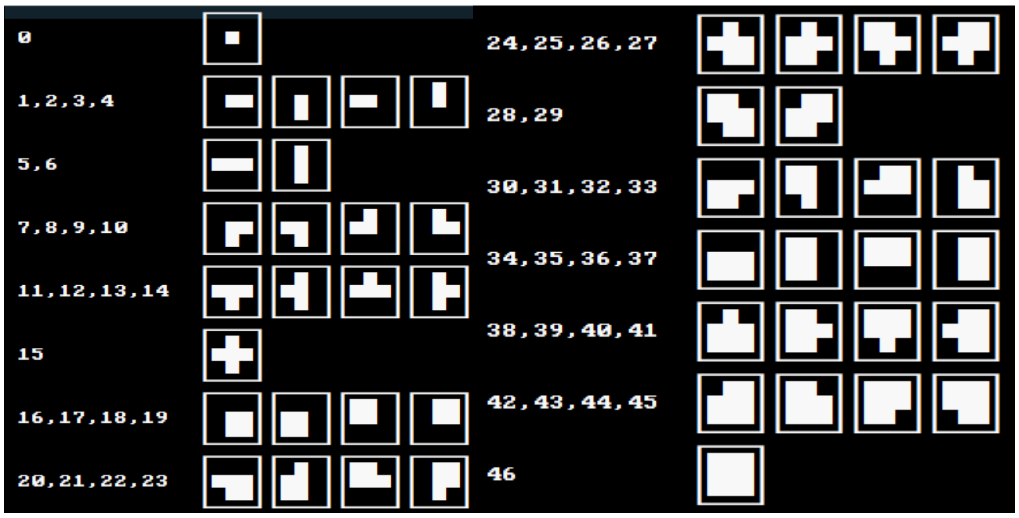


Figure 40 - Moore 2 Patterns (old numbering)

Let's say I need to know what kind of soil tile would go in this green zone:

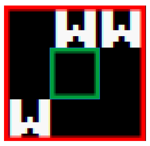


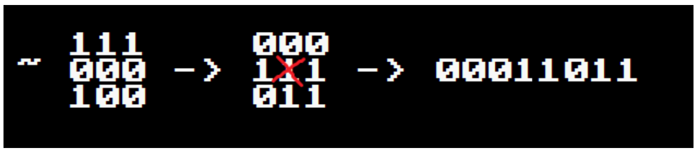
Figure 41 - Example of context-awareness algorithm input

The first step is to pass this to a byte, in which 0 represents the squares with floor and 1 represents the squares with walls, however this is not enough, since in some cases like this, the result does not correspond to any of the Moore patterns, this is why you have to take into account an extra rule when making this transformation:

- If in the central box of a column or row is not the object we want to place (in this case we want to place floors, so the opposite are looks) that row / column is filled with 1s. This means that in the example the top row would be all 1s, because there is a W in the middle.

This is useful because to insert the 1s and 0s within the chain the OR(|) operator is used, and as we use 1s to represent the objects of the opposite type to the one we are going to place and 0s for those that are of the type we want to place, if during this operation we try to place a 0 again where there is now a 1, the OR operator is not going to let us, because 0 | 1 = 1, however nothing prevents us from placing 1s where there are 0s, because in the same way 1 | 0 = 1.

The conversion gives the following result, note that the result is reversed with a NOT(~) and the center is ignored:



```
  111     000
~ 000 ->  1X1 -> 00011011
  100     011
```

Figure 42 - Bit Operations on Moore Patterns

This 11011 (27 in decimal) is the key of a dictionary that contains all the patterns and returns a number that is directly associated with the identifier number of the piece to be placed. It turns out to be the following sprite and leaves the floors of the rooms with this stoned look on the edges on which it meets the walls:

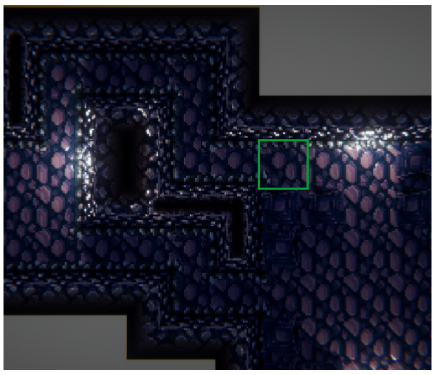


Figure 43 - Result of soil content-awareness

- Walls: The walls luckily do not require the 47 patterns to work, but 15 are enough, since the models of the walls can be rotated without problems, not as in the case of the floor, which when rotated breaks the visual pattern.

To support the two types of patterns I organized the pattern dictionary as follows:

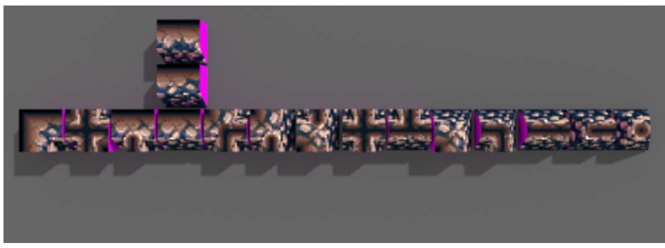
```
// KEY       MOOREx47    MOOREx15  ROT
{0b01010000, (9 << 8) |  (3 << 2) |  2}
```

In this case the contained value is a 16-bit variable divided into 2 groups of 8.

- The lowest 8-bit group contains the part identification number using Moorex15 in its highest 6 bits and reserves its 2 lower bits to save the number of 90-degree rotations on the Y-axis needed to match the part to its shape in the pattern.
- The highest 8-bit group keeps the part identification code in Moorex47, used in the case of floors, where rotating the part is not possible and a total of 47 different pieces are required.

Art: Art in this case was especially difficult in the case of the floor and walls because of the number of different models and sprites needed to make context-awareness work:

Figure 44 - The 14 + 2 primitive wall variants needed



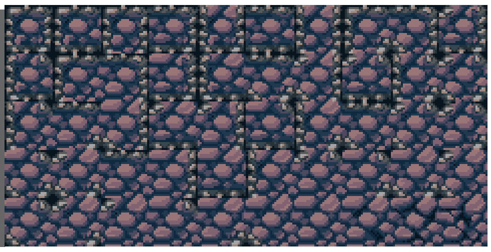


Figure 45 - The 47 + 3 primitive soil variants needed

In the case of the soil, I applied a bump map generated from the greyscale of the soil's own texture to give it more depth, this looks particularly good when bilinear filtering is disabled, to generate pixelated specular brightness.

For the models, the rule of creating tiles of 24x24, which are equivalent to 1 Unity Unit, was followed. The theme of the game is centered around retro technology, so some of the modeled enemies are computers, microprocessors and floppy disks. For health items I thought it would be interesting to use batteries of different sizes, since the protagonist is a robot.

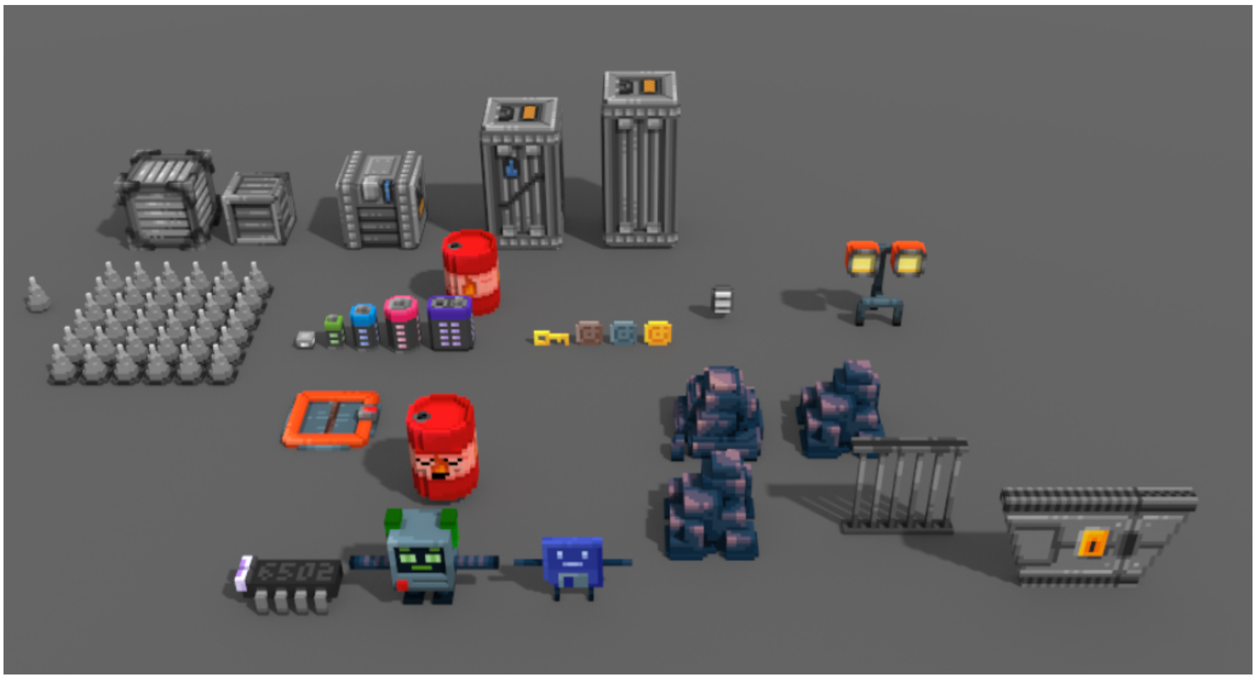


Figure 46 - Models of enemies and in-game objects (some unused)

The boxes and doors were given a "tech" style bwith lights and other ornaments.

6. CONCLUSIONS, RESULTS AND FUTURE LINES

6.1. Conclusions

I can conclude that the development of the project has managed to extend my knowledge of the procedural generation and has managed to obtain an optimal final result.

I've missed having teammates and following an organizational system in which to depend and work with other people. Because of this in some situations having a development plan *too flexible* has drived me away from priority work in favor of other optional but more interesting work in terms of research, such as the graphic section of the game or the development of objects and mechanics that were not implemented due to lack of time.

During development I also had to redo part of the code due to my lack of experience in large projects, this has shown me the importance at the beginning of a project of defining the operation of each class in a clear and concise way to prevent errors and have a much more fluid development process.

6.2. Results

Most expectations regarding the final result were met. Originally I did not expect to go that far in the development since I wanted to focus on the technical section of the game related to the procedural generation. However, the game was well received in the presentations and when it is finished it seems that it has a lot of potential to be able to be put on sale on the target platforms once I have finished incorporating the objects and enemies that are still missing from the game.

The game has a good visual section, interesting and extensible mechanics, a room editor for the community and an endless game mode that makes it highly replayable. There are some shortcomings in the sound section and in the variety of objects and enemies, but it can always be expanded in the future.

6.3. Future of the project

I originally decided to do this TFG on my own because all the projects were already complete. My goal was to learn a little more about procedural generation so that I could use this knowledge in a personal project that I have been working on for years (www.notestruck.com WIP).

For now, the development of this game will stop until the development of my other project is finished, I will compile a version with WebGL to be able to post the game online as part of my portfolio until I decide what to do with it.

When I resume the project, I'll start by making a real trailer for a crowdfunding campaign, investing something in advertising and trying to get some publisher interested. Then I'll add all the missing content and focus on polishing the game's aspect a lot more and adding music.

7. BIBLIOGRAPHIC REFERENCES

A. Lagae, S. L. (2010). *A Survey of Procedural Noise Functions.* The Eurographics Association and Blackwell Publishing Ltd.

Ashlock, D., Lee, C., & McGuinness, C. (2011). *Search-based procedural generation of maze-like levels.* Guelph, Ontario, Canada: University of Guelph.

Buck, J. (January 15, 2020). *Maze Generation: Eller's Algorithm*. Obtained from Jamisbuck's Buckblog: https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorith

Chomsky, N. (1956). *Three models for the description of language.* Massachusetts: IEEE Transactions on Information Theory.

Deflektor - C64-Wiki. (February 12, 2020). Obtained from C64-wiki.com: https://www.c64-wiki.com/wiki/Deflektor

DEV. (2019). *White Paper on Spanish Video Game Development.* Madrid: Games from spain.

Haubenwallner, K. (2016). *Procedural Generation using Grammar based Modelling and Genetic Algorithms.* Austria: Institute of Computer Graphics / Graz University of Technology.

Johnson, L., Yannakakis, G. N., & Togelius, J. (2018). *Cellular automata for real-time generation.* Copenhagen: PCGames.

Portal 2. (n.d.). Obtained from En.wikipedia.org: https://en.wikipedia.org/wiki/Portal_2

Rekers, J., & A. S. (1995). *A Graph Grammar Approach to Graphical Parsing.* (Leiden, The Netherlands) (Aachen, Germany): Proceedings of the 11th International IEEE Symposium on Visual Languages.

Roguebasin. (19 de Diciembre de 2017). *Basic BSP Dungeon generation*. Obtenido de Roguebasin - Basic BSP Dungeon generation: http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation#Building_the_BSP

Rozenberg, G., & Salomaa, A. (1997). *Handbook of Formal Language Vol 1.* Berlin: Springer.

Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural Content Generation in Games.* Springer.

Unity3D. (2018). *Unity Technologies. Publication 2018.3.* Obtenido de Unity documentation: https://docs.unity3d.com/es/current/Manual/StandardShaderMaterialParameterHeightMap.html

Valtchanov, V., & Brown, J. A. (2012). *Evolving dungeon crawler levels with relative placement.* Guelph, Canada: School of Computer Science.

Watkins, R. (2016). Procedural Content Generation for Unity Game Development. Birmingham B3 2PB, UK: PACKTLiB.

Wikipedia. (14 of 1 of 2020). *Ken Perlin - Wikipedia.* Retrieved from Wikipedia, the free encyclopedia: https://en.wikipedia.org/wiki/Ken_Perlin