Master Thesis - Procedural Content Generation in *Celeste* Overview of automated level generation in video games with a focus on playability in a 2D platformer

Louis ROBINET

August 30, 2024

Abstract

This thesis presents the development of a procedural content generation (PCG) pipeline tailored for the 2D platformer *Celeste*. The project leverages a Markov Chain-based model to generate individual rooms within levels, focusing on maintaining structural coherence and enhancing playability. By employing existing *Celeste* levels as training data, the model captures essential gameplay patterns to generate new, challenging, and engaging content. Key post-processing steps, including exit refinement and the strategic placement of respawn points, are applied to ensure that generated rooms are not only playable but also align with the gameplay experience expected in *Celeste*. The generated levels are evaluated based on metrics such as playability, difficulty, and interestingness, focusing on the quality and challenge of the generated content. The project aims to contribute to the field by offering a customizable and robust tool for level generation applied to *Celeste*, and potentially be used as a foundation to create a *Celeste* AI framework.

Contents

1 Introduction

1.1 *Celeste* and context of the project

Celeste is a 2D platformer released in 2018. The concept of this game is quite simple: you have to progress through various well-designed levels to climb successfully Mount Celeste, relying on 4 mechanics only: move, jump, dash and grab. You can combine these mechanics to perform more advanced gameplay (if you *move* vertically while you *grab* a wall, you will end up *climbing* the wall; more advanced combinations can be done but listing them would be time consuming and not really interesting within the scope of the project). Just like other platformers, you might interact with a lot of different entities (fatal, like spikes or monsters, or friendly, like bouncers or dash refills) that are really part of the identity of a level. Even more, some levels can just become unbeatable if you remove the entities. So level conception in *Celeste* is not only about the structure/the foreground of a level, but the harmony between structure and entities.

For the rest of this report, I need to make a vocabulary precision: each *level* is subdivided into *rooms*. Procedural content generation concerns the latter: I first generate rooms procedurally, then I assemble them to make a whole level. I am not generating whole levels at once as I am willing to make use of all the available information and potentially extract low-level structures. From a structural point of view, rooms contain **exits** connecting them together, while a level is a set of rooms assembled together and has a **starting point** and an **ending point**.

1.2 Video Games PCG - Level Generation

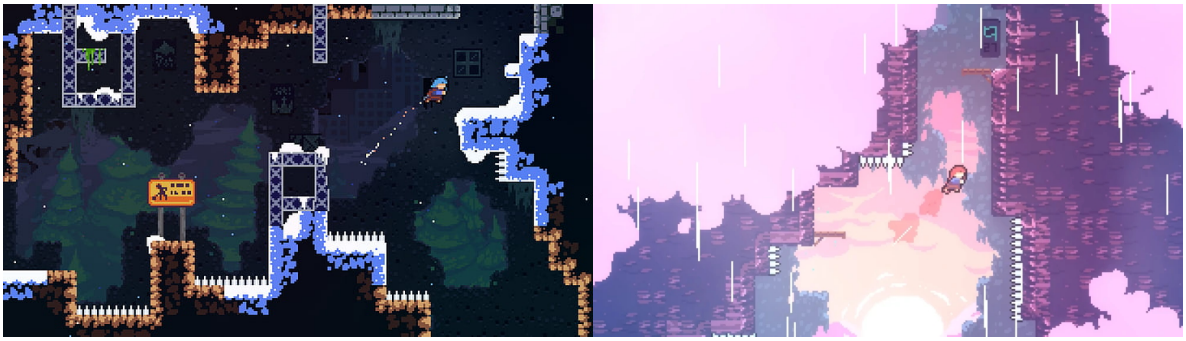
Over the past years, people got interested in level generation in video games, and a few approaches have been considered to that end. A lot of this work concerns the video game *Super Mario Bros.* since a framework has been specially developed for using AI methods within a version of this game. This **Mario AI framework** has been used in the context of Reinforcement Learning applied to video games, by training an agent to play Mario. But this framework also has been used to develop another part of video games AI, that is level generation. For platform games (such as *Celeste* or *Super Mario Bros*), levels are usually hand-crafted by developers whose task is to make sure that they are as interesting as challenging (beyond the obvious requirement that a level should be clearable). Level generation-oriented PCG is a field of research where AI is used to try to generate levels that meet both requirements stated before.

1.3 PCG in *Celeste*

When it comes to level generation applied to *Celeste*, a natural question that comes to mind is whether I should use a supervised or unsupervised method. While the latter requires to define somehow scores to evaluate the interestingness and difficulty of a room to quantify how good a level is, the former allows generating levels from data (in this case, the multiple levels of the game that already exist) and then the main concern is no longer to define obscure evaluation functions but to develop a model that manages to seize as best as possible what characterizes a good level - id est, the ones that already exist. There are two major difficulties in *Celeste* level generation:

- Unlike Mario, *Celeste* levels do not always consist of an entrance on the left and an exit on the right! Some rooms do even have exits on the ground and the ceiling and are meant for more vertical gameplay (cf. figure 1). One of my major concerns is that I have to make sure that the rooms that will be generated are clearable; that is, that there is always a way to reach the exit.
- There is no "*Celeste* AI framework", meaning that I can not, in the short amount of time allowed to this thesis, count on developing an AI agent that would actually play the game; I need to develop an agent-free model. As far as I know, there is no existing work on an agent that plays *Celeste* that could be used in the context of this project.

Existing methods using Machine Learning to that end consist of two elements: a representation of the data structure as well as a training algorithm type. A survey [Summerville et al., 2017] describes different types of data structures used in PCG via Machine Learning and the five training algorithms



(a) Horizontal gameplay (b) Vertical gameplay

Figure 1: Two rooms of *Celeste* with different gameplay

under consideration. The workflow that I have developed (cf. section 4.2.1 to 4.2.5) makes me think that the grid representation, *id est* representing rooms using arrays/matrices, is the most straightforward and efficient way to work with *Celeste* rooms. However, some papers using other representations are very interesting and could be worth spending some effort on replacing arrays with other data structures. In section 2, I will present some of these different studies related to level generation and how it could connect to PCG in *Celeste*.

2 Related work & existing methods

After a quick review of around 40 papers, no unsupervised method seemed really promising for such complex level generation (complex in the sense that there is no chance that a random generator consistently creates a playable level for *Celeste* - there is a need for structure). Also, it would be a bit of a loss not to exploit all the available data. I decided to focus on methods that would put an emphasis on finding the underlying structures in existing levels which would be the foundations for some brand new AI-generated rooms. The core idea here is that I really need a robust model that can grasp what makes a playable level so that it would be easier to tweak it to explore a bit more the search space of possible levels rather than having a model that is really creative but struggles at making clearable levels. Procedural Content Generation (PCG) encompasses a variety of techniques for creating game content algorithmically, and it has been applied extensively to level design in both 2D and 3D games. This state-of-the-art review is organized by level representation methods, including grid-based, grammar-based, and other hybrid techniques.

2.1 Grid-Based Representation

Grid-based representations involve levels being modeled as a matrix of cells or tiles, where each cell represents a specific type of terrain or object. This approach is commonly used in both 2D platformers.

2.1.1 Random Generation with Constraints

Some games like *Baba Is You* use a combination of handcrafted levels and procedural generation to create puzzles. The procedural aspects involve random generation of puzzle elements within certain constraints to ensure solvability of created levels [Charity et al., 2022]. Another great example is the work from [Kazemi, 2008], that employs a random dungeon generation approach with predefined rules and constraints applied to the 2D platformer *Spelunky*. The grid-based levels are generated using Perlin noise and random number generators, ensuring each playthrough offers a unique experience while maintaining playability. This study inspired me the skeleton generation for room assembling (discussed in 4.2.2).

2.1.2 Markov Chains based methods

A good model is a model that can learn what makes a *good* level through patterns detection in existing levels. Markov Chains are useful to model probabilistic transitions between different states and are

also used in level generation in 2D video games like Mario Bros. [Snodgrass and Ontanon, 2013] Levels are then represented as 2D arrays and the probabilities of each tile/entity are computed from a set of existing maps to generate new ones. In another paper from the same authors dealing with Markov models-based level generation [Snodgrass and Ontañón, 2017], they applied their models to other games like *Kid Icarus* that present more vertical/mixed gameplay and even tackled the playability issue. That method seems very promising for *Celeste* level generation but needs to be handled with care because of the game specificities: because *Celeste* has rooms that are meant to be played vertically, mixing up all the levels in the process could harm the performance of the model. The same thing goes for the difficulty; some rooms are way more difficult than others, and it might be really interesting to split the training rooms by difficulty levels or some other parameter, hence creating several well-performing models rather than one that mixes up everything.

2.1.3 Cellular Automata

Cellular Automata have been used to model and generate complex structures in grid-based environments. Due to its low complexity and runtime, the work from [Johnson et al., 2010] makes great use of the self-organization characteristics of cellular automata to generate interesting, playable, and efficient 2D maps.

2.2 Grammar-Based Representation

Grammar-based representations utilize formal grammars to generate levels by applying production rules to a starting configuration. This approach allows for the creation of levels with specific structures and constraints.

Initially developed for natural language, generative grammars have been adapted to procedural content generation for video games. Graph grammars are adapted to model dungeon levels, where nodes represent rooms and edges represent connections. Adams [Adams, 2002] used this approach to generate FPS levels, focusing on topological control but facing limitations due to hard-coded rules. Dormans [Dormans, 2010] extended this by introducing mission grammars to generate adventure game dungeons, adding gameplay-based control. Van der Linden et al. further refined this in [Linden et al., 2013] with gameplay grammars, allowing designers to create generic graph-based dungeon layouts tied to player actions, demonstrated in Dwarf Quest.

2.3 Metric-centered methods

Some studies focused on trying to generate coherent and playable levels using some smart hand-crafted metrics like the ones defined in *A Comparative Evaluation of Procedural Level Generators in the Mario AI Framework* [Dahlskog et al., 2014], where for example *leniency* is basically illustrating the ratio between platforms and gaps in a Mario level, hence a metric related somehow to the difficulty of a level. While this approach alone is incomplete to generate levels, it can be interesting to couple it with a level generator model to influence some characteristics of the created rooms, as in the very interesting paper *Intentional Computational Level Design* [Khalifa et al., 2019], where metrics are used in order to restrict/influence gameplay of the new levels.

2.4 Hybrid evolutionary approaches

A lot of studies around level generation in Mario used methods based on Generative Adversarial Networks (GAN). Among them, the work discussed in [Volz et al., 2018] is one of the most interesting I found, since it is well documented, the experiments are well described and available for reproducibility. This study presents a method for procedurally generating *Super Mario Bros.* levels by combining Generative Adversarial Networks (GANs) with evolutionary algorithms. A GAN is first trained on existing levels to learn their structural patterns, creating a latent space that represents various level features. Evolutionary algorithms are then used to explore this latent space by optimizing latent vectors based on objectives like playability and complexity, effectively evolving new and diverse level designs. This approach successfully generates novel, playable levels that adhere to the game's style, demonstrating the potential of integrating deep learning and evolutionary computation for advanced procedural content generation applied to game design. This work goes even beyond the scope of GANs

by showing additional methods to enhance GAN-generated levels. It definitely is a robust method that shows a lot of potential for video games level generation, even though a lot of efforts is required to ensure a decent playability rate and an accurate capture of what really makes a level challenging or enjoyable to play.

Similarly, one could mention the paper from [Thakkar et al., 2019] using autoencoders and Evolutionary Algorithms applied to level generation in video games. An autoencoder, a type of neural network used for dimensionality reduction, is first trained on existing *Lode Runner* levels to capture and compress their essential features into a lower-dimensional latent space. This latent space is then explored using an evolutionary process similar to what was done in [Volz et al., 2018]. This work was definitely interesting because of the similarities between levels and their representations in *Lode Runner* (the game studied in the paper) and *Celeste*.

If the evolutionary characteristic of the above method lied in the exploration of the latent space of *Super Mario Bros.* levels, some approaches use evolutionary methods as the main component of their PCG model. For example, [Balali Moghadam and Kuchaki Rafsanjani, 2017] explores the use of genetic algorithms to automatically generate levels for 2D platformer games. The method involves representing game levels as sequences of tiles, which can be manipulated through typical genetic operations like mutation and crossover to evolve towards new levels over successive generations. The fitness function, which guides the evolution, evaluates levels based on gameplay-based criteria like playability, difficulty, and diversity. This approach aims to create levels that are both playable and varied, and ultimately demonstrates that genetic algorithms can effectively generate engaging and playable platformer levels, offering a flexible and adaptive method for PCG.

2.5 Summary of the SOTA and choice for *Celeste* PCG

After reviewing various approaches on PCG applied to game design and more specifically to level generation, it became evident that unsupervised methods are not suitable for generating playable levels in complex games like *Celeste* . Instead, methods that leverage existing level structures offer more promise. Of these, Markov chains methods stood out due to their ability to efficiently learn structures from existing levels in grid-based representations, which aligns well with how *Celeste* levels are structured and handled from a coding perspective. This inspired the choice of a Markov chains-based PCG model for *Celeste*, as it provides a robust and flexible - yet rather simple - framework for generating structured, and hopefully playable levels.

3 Research question

I initially started this project with the intent of making it a modding tool usable by the community; that is, players should be ultimately able to use this tool to procedurally generate random, unique, and most importantly, playable levels. Building an entire workflow able to generate a level from a few entry parameters is already a challenge of its own, given that nearly nothing exists to that end besides a Julia repository originally developed for a visual map editor project [Cruor et al., 2018]. However, the player-oriented aspect is very important to me in this project and I definitely want to push this project a bit further than basic generation. Once the generator workflow is operational, I want to evaluate its performance through several aspects which, to me, are critical when it comes to player experience in a platformer, like the complexity, the interestingness, or most importantly the playability of a level. The evaluation will provide insights into the effectiveness of the tool and its impact on player experience and how the generated levels meet criteria for these aspects, with special care for playability. A level can be incredibly interesting and amusing; at the end of the day, if the game is not clearable, the player leaves with nothing but frustration.

In this thesis, I aim to develop a comprehensive, configurable procedural content generation pipeline applied to the 2D platformer *Celeste*, while being able to evaluate the produced levels, hence enabling a finer control on the characteristics of the output of this pipeline. I hope this study will eventually contribute to the field by offering a revised tool for level generation and providing foundations for a new AI framework for assessing procedural content applied to *Celeste*.

4 General idea and setup

Due to its important and active modding community, a visual map editor *Ahorn* has been developed to help people create their own levels in *Celeste*. For my project, I adapt some of the Julia code behind this interface (a project named *Maple* [Cruor et al., 2018]) to generate rooms and levels through some lines of code, making room and level generation automation possible.

This wrapper code actually allows to read the binary files containing the original levels of the game. I was hence able to notice that a level is nothing more than a set of rooms assembled in space using the right coordinates. All the interesting information and all the details of a level were contained and divided in each of the rooms. A room consists of many things, and I can actually summarize its most important components below (I will ignore what I call *decals* in the room data, which is purely aesthetic and will not be considered for the scope of this project):

- a foreground: the set of tiles which composes the ground, ceiling, and walls of a room
- the entities: all additional items that the player can interact with, can be helpful, neutral, or even deadly depending on the entity (just like a mushroom or a Goomba if I was to find the equivalent in *Super Mario Bros.*)
- a background: the set of decorative tiles that are in the background, only aesthetic, does not interact with the gameplay
- metadata: special settings of a room, like wind (will alter considerably gameplay) or a specific music track (will not change gameplay but can be chosen accordingly with level difficulty for example)

Knowing this, the initial scope of my project consists in generating foregrounds and sets of entities that are well structured and make sense in their interaction when it comes to playability. Those are the base requirements for room generation in *Celeste*. Backgrounds and metadata like custom music tracks can be added at a later point and are not a difficult part of the workflow once a running pipeline is established, but I want to keep it as simplified as possible to introduce the general idea of my solution. As I will discuss in a subsection dedicated to the PCG model below, background can be generated the same way foreground is some ideas about music integration, like choosing music according to the difficulty of a level (epic tracks for epic rooms of course), will be discussed at the very end of the project, when the relevant quantities, naming, and metric are introduced. My focus is definitely on foreground and entities so far, since both elements constitute the core of each room and make it playable, or not. Integration of additional elements will be discussed later at relevant times or in the conclusion as potential further improvements.

4.1 General idea: model and workflow

As stated in the section 2 dedicated to studies dealing with procedural content generation applied to the generation of levels of video games, I need structure. Structure is even more important in platformers than in other *genres*, where level structure and design is critical at both high- and low-levels as it impacts directly the gameplay experience, as showed in [Sweetser and Johnson, 2004] which conducts an extensive study on correlations between level design and player engagement. I chose to implement a model heavily inspired by [Snodgrass and Ontanon, 2013] and [Snodgrass and Ontañón, 2017]: the representation of a room as an array whose elements are symbols corresponding to specific tiles or entities makes perfectly sense when considering *Celeste*, and by essence, Multi-dimensional Markov Chains are a powerful mathematical tool which takes structure into account as it will be discussed in the subsection dedicated to the PCG model. But then, another problem soon arises: even though I can generate a plethora of rooms, connecting them is nothing but easy: indeed, the exits of a room are not part of the data or metadata of a level; they are given by the general structure of a level and how the rooms are placed at a high-level. I solved this issue by adopting an approach inspired from [Kazemi, 2008]: rather than trying hopelessly to recombine rooms in space, I first generate the high-level layer of a level, its *skeleton*. The generation therefore comes in two parts:

- I first generate the high-level layer, the skeleton of a level. This object was specifically created for this project and displays some interesting characteristics: the skeleton of a level consists of a

given number of boxes organized in space such that all boxes are connected to at least one other box through "exits" and there is no overlapping at all between any pair of boxes. The skeleton generation process, input parameters and characteristics will be detailed in the subsection 4.2.2 dedicated to the *Celeskeleton* module. To sum it up, the first stage of the level generation pipeline outputs a random, empty level: I have at disposition a set of empty boxes/rooms and their respective exits well-organized in space, such that there exists a high-level path from the starting to the ending room, no matter the input parameters chosen.

- At this point, what is left is to fill the skeleton previously generated: for each room composing the skeleton, I use the Markov chains based model (briefly mentioned above, and detailed in the subsection 4.2.3) to generate a set of foreground tiles and entities matching the size of the room to fill. Of course, playability is no longer ensured at a low-level after the filling, but after this two-step generation the output is a set of rooms, filled with foreground tiles and entities, and connected in 2D space through exits without any overlap, which is both necessary and sufficient to be converted into a playable binary file.

Beyond the generation part of the project, I clearly identified and split this project into steps, into modules, on which I worked separately in order to build a continuous, streamlined, and relatively tunable workflow. I will share a brief overview of the different steps composing the pipeline before diving into each module in details. The first step is to extract the information from the game and create a "database" with all levels from *Celeste* as training data to feed the Markov chains based model. A first module is the *Data loader* described in subsection 4.2.1. Then, moving towards the generation; I described the two-part level generation right above. The subsection 4.2.2 is focusing on the procedural generation of the skeleton object freshly introduced, and on the elaboration of two *Python* classes designed to that end: *Room* and *Celeskeleton*. The subsection 4.2.3 describes precisely the model used to generate the data filling the rooms of a level skeleton. The fourth module described in 4.2.4 is deeply interconnected with the model module, as it deals with the post-processing of every single generated room, like adding necessary entities to make the level launchable by the game, or assessing the playability of a generated room in accordance with the exits defined by the skeleton. Finally, when the generation is final - skeleton created, filled, and each room post-processed - I want to convert this output data into a single binary file, such that the game can process this freshly generated data into a level one can play. The *room encoder* described in subsection 4.2.5 is the last module of this project, and serves this purpose.

4.2 Modules

4.2.1 Data loader

Thanks to the existing *Julia* wrapper, *Maple* [Cruor et al., 2018], I was able to extract the data from the level files. I used and adapted some parts of this code originally made to encode data into playable binary files, to decode existing files containing hand-designed levels. I thought it was really interesting to make it a real customizable module instead of a one-time work, as this allows me to consider a dynamic database; *Celeste* is a video game which benefits of a very active modding community which designs brand new and original hand-crafted levels frequently. At this point, it is important to understand that, from the game point of view, a level is equivalent to a vector of rooms, and each room is an object made of several attributes like foreground/background tiles, entities, necessary metadata like the room size and origin in the 2D grid of the level. Some additional information can be also extracted, like the music track, the wind force, gravity, number of dashes available, etc.

Given a level file, I designed a function in *Julia* that decodes and returns for each room the foreground tiles, the background tiles, and the entities. The tiles are simply represented by 2D arrays of symbols stored into *.csv* files and the dictionary of entities is stored into a *.json* file.

Since there is no overlapping possible between any entity and the foreground, the choice of 2D arrays/matrices is very reasonable for level representation in *Celeste*. Essentially, the representation of a room including background is done using 3D matrices for this project (or equivalently a collection of two matrices of the same size - one for the foreground/entities mix, the other for the background). After the extraction, a second step is therefore to recombine the foreground and the entities within a single matrix. I extended an existing dictionary in *Maple* to map entities from the game to symbols

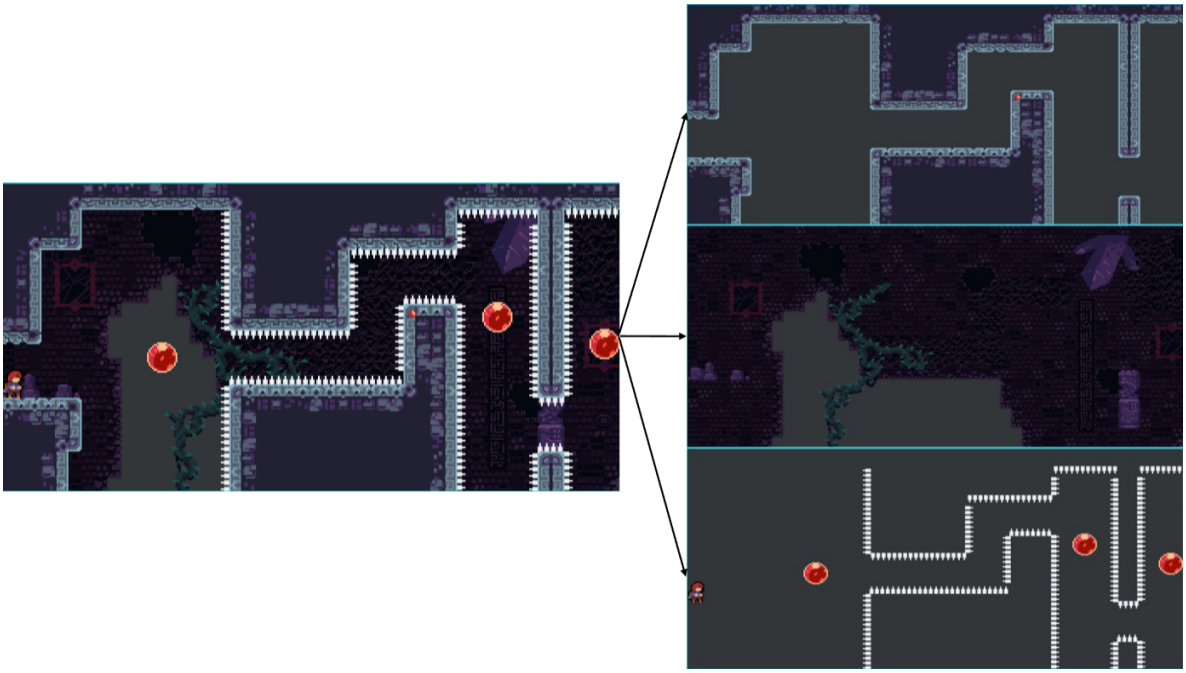


Figure 2: A typical *Celeste* room, is shown on the right when split between foreground, background, and entities in that order

to represent them in the arrays. An entity is so far represented by its name, its origin, and potentially its size (amongst other attributes specific to each entity that I will not present here). In the matrix representation, entities can be 0D - just one tile - when entity size is *None*, 1D for entities whose size has only one component, or 2D for the entities having a non-zero size along both axis. Furthermore, while positions and sizes of the entities are in pixels, *Celeste* rooms are measured in tiles, which are simply 8×8 pixels squares. An entity i can be represented simply by

$$(\mathcal{S}_i, x_{0i}, y_{0i}, w_i, h_i)$$

where $\mathcal{S}_i$ is the symbol assigned to entity i, (x_{0i}, y_{0i}) is its origin and (w_i, h_i) its width and height. Let $\mathcal{M}$ be the matrix containing the foreground tiles information of a room. For all entities i, and for all $(x, y) \in \{x_{0i}, \dots, x_{0i} + h_i\} \times \{y_{0i}, \dots, y_{0i} + w_i\}$, I simply set $\mathcal{M}_{x,y} = \mathcal{S}_i$. Since there cannot be any overlap between any entities and the foreground tiles, I am assured that this step fills at most empty tiles (symbol 0), and therefore I am able to recombine all the foreground tiles and entities within a single 2D array.

First of all, I wanted to check whether I could use existing maps as training data or not. Since the source code of *Ahorn* is designed to encode data into playable rooms, I wanted to use it to decode rooms into data. And that is exactly what the Import Data Function (IDF) does. Given a file of a level, I designed a function that decodes and returns for each room its data: the foreground tiles, the background tiles, entities, and even extra information like its coordinates in the level, but also music, wind, gravity, or the number of dashes (you can have up to 2 dashes in some levels). Actually, I even extract both foreground and entities of a room in a **single** matrix since tiles and entities do not overlap as detailed right above, making the use of 2D arrays really convenient for room representation.

This means that I can constitute a database of existing rooms - not only the 700+ rooms from the game but also any level designed by the community - that can be used for training a PCG model.

4.2.2 Room & Celeskeleton

As mentioned above when describing the general workflow idea, the level generation is a two-step process, and generating its global structure, its skeleton, is the first important part of it. This skeleton of a level has to obey a certain set of rules and specifications in order to be considered as a potential

candidate for future valid levels. Decomposing levels in sub-parts is interesting from a diversity perspective, but soon a major question arises: how to assemble generated rooms in a whole level such that the resulting arrangement makes sense from a game design point of view? The *Room* and *Celeskeleton* objects are *Python* classes I introduced and designed to address this central point.

When encoding a level (*id est* converting data into a binary file that can be interpreted by *Celeste*), the coordinates of each room need to be specified so that the game can correctly place the room in a 2D space. Each room is basically a rectangle that can be described by four different attributes:

- Origin: (x_{room}, y_{room}) indicates the position of the bottom-left corner of a room.
- Size: (w_{room}, h_{room}) indicates the width and height of a room.
- Data: a 2D-array of size $w_{room} \times h_{room}$ containing symbols representing all the tiles and entities filling the room.
- Exits: a 4-keys dictionary (one per side) indicating the coordinates of the different exits present in the room.

The first class I designed, the *Room* class, serves this sole purpose of storing all the information related to a room object, but I also extended it to enable some post-processing and playability functionalities discussed in 4.2.4. Overall, this class allows to create, store and modify room objects that will be used for level generation.

Now concerning the assembling of the rooms: to design a level, I am facing two unavoidable requirements.

- First, two rooms are said to be *connected* if they have juxtaposed exits. That being said, each room composing a level needs to be connected to **at least** one room.
- Rooms can **never** overlap.

Simply put, a level is a set of *Room* objects meeting the requirements stated above. And that is where the *Celeskeleton* object intervenes. The *Celeskeleton* is a collection of *Room* objects, augmented with information about which rooms are the starting and ending room. When I experimented and created my own levels through code, I made the calculations and hand-crafted the coordinates of each room so that they would be well assembled and the resulting level would hence be playable. But of course, I wanted to automatize this process and it definitely is the reasoning behind the creation of the *Celeskeleton* class. This approach was deeply inspired by looking at *Celeste* levels using the debug mode, as one can see in figures 3 and 4.

Algorithm 1 Celeskeleton generation algorithm

```
Initialize Celeskeleton object
Set initial room size (from input or random)
Initialize empty Room object and add first room to skeleton
while number of rooms in Celeskeleton < nb_rooms do
    Generate new Room
    Choose connection side and choose Room to connect with (with probabilities p_1, p_2)
    Attempt to connect and place new room
    if no overlap then
        Add Room to Celeskeleton
    end if
end while
Set start and end rooms
Return the skeleton
```

4.2.3 Markov Chains-based model

As mentioned in the section 2, the model I designed for Procedural Content Generation applied to level generation in *Celeste* has been inspired from [Snodgrass and Ontanon, 2013]. Since I needed to build the whole AI framework for *Celeste* , I wanted the PCG core model to be simple and flexible. I

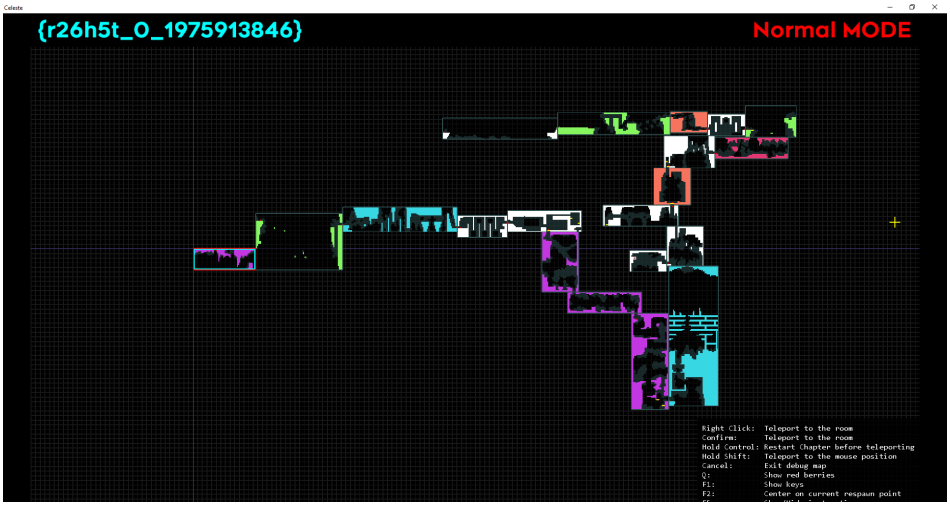


Figure 3: Example of a level seen in debug mode

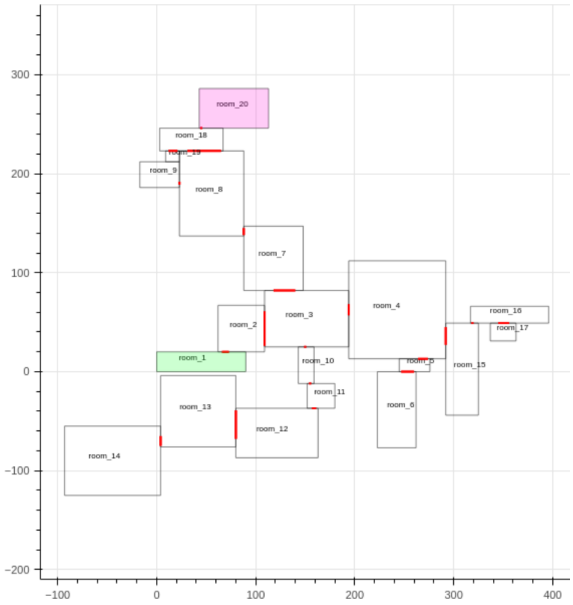


Figure 4: Example of a Celeskeleton object generated with 20 rooms

think this Markov Chains-based approach that I adapted to *Celeste* was the best compromise I could find with respect to these constraints I set. I will present in this part all the details related to the model designed for *Celeste* PCG.

Multi-dimensional Markov Chains (MdMC) To completely understand how the procedural generation works in my work, I will start with a short mathematical parenthesis about Markov Chains (MC). A MC is a stochastic process that transitions from one state to another within a finite or countably infinite set of states; for this study, the set of states is the set of possible tiles and entities in the grid representation of *Celeste* rooms and is hence finite. The key property of a first-order MC is that the probability of transitioning to the next state depends only on the current state and not on the sequence of events that preceded it. Introducing some mathematical formalism, it gives:

Let $\mathcal{S} = \{S_i\}_{i=1}^{N}$ be a finite set of states, and P be conditional probability distribution such that $P(S_t|S_{t-1})$ represents the probability of transitioning to a state S_t given that the previous state was S_{t-1}. These two elements are what a first-order Markov Chain consists of. The Markov property for

first-order MC can be translated as:

$$P(S_t|S_{t-1}, S_{t-2}, \dots, S_0) = P(S_t|S_{t-1})$$

However, higher-order MC are definitely essential to the model I use. Let r be a strictly positive integer. An MC of order r therefore takes into consideration r previous states:

$$P(S_t|S_{t-1}, S_{t-2}, \dots, S_0) = P(S_t|S_{t-1}, \dots, S_{t-r})$$

To generalize the basic concept of Markov Chains to a 2D-space, let $R = \{R_{i,j}\}_{(i,j)\in\{1,\dots,N\}\times\{1,\dots,M\}}$ be a 2D room of shape (N, M), and $\mathcal{S} = \{S_i\}_{i=1}^{T}$ be the set of the possible states, accounting for the T possible types of tiles or entities one can find in *Celeste* . For any pair of coordinates (i, j), the tile $R_{i,j}$ is in a given state S_k, where $k \in \{1, \dots, T\}$. Simply put, a room is represented by a 2D-array of given states among all the existing tiles and entities within *Celeste*.

Learning with MdMC Let's consider a MdMC representing the probability of a tile in a room according to the surrounding tiles. This set of surrounding tiles used for learning probabilities will be called a *configuration* and can be represented using a 3×3 configuration matrix $\mathcal{C}$:

$$\mathcal{C} = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & 2 \end{bmatrix}$$

where $C_{22} = 2$ is by default representing a tile $R_{x,y}$ to learn, and for all other (i, j) pairs in $\mathcal{C}$, $C_{ij} = 1$ means that the probability of the tile $R_{x,y}$ depends on the tile $R_{x+i-2,y+j-2}$; else $C_{ij} = 0$. Let's consider an example. The configuration matrix

$$\mathcal{C}_{000011012} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

considers the tiles on top, on left, and on top-left of the tile considered. Each configuration will be identified by a unique 9-digits sequence corresponding to the values of its matrix read line-by-line. The example above represents the 000011012 configuration.

For the sake of clarity, I will introduce the concept of *adjacent* tiles. For example, I consider the configuration 001001112 whose matrix is given by:

$$\mathcal{C}_{001001112} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

and a example 5×5 room given by:

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & D & D \\ 1 & 0 & 0 & D & D \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

For a given position (x, y) in the room R, the tiles of interest given by $\mathcal{C}_{001001112}$, the *adjacent* tiles, are the two tiles above, and the two tiles on the left of $R_{x,y}$. These adjacent tiles, being read from left to right, then from top to bottom, constitute a n-gram that will prove useful for practical training of this MdMC-based model. Therefore, if I look at $R_{4,5} = D$, the adjacent tiles are represented by the n-gram made of $(R_{2,5}, R_{3,5}, R_{4,3}, R_{4,4}) = 0D0D$.

Now for the learning process: let $\mathcal{C}$ be a configuration matrix, and $\mathcal{R}$ a dataset of rooms R that are considered for the training. The MdMC model is learned in two steps:

- The method begins by counting the occurrences of each tile type w.r.t. the adjacent tiles in the whole training dataset, with adjacent tiles being defined by the configuration matrix $\mathcal{C}$. These counts are called *absolute counts*, and are defined for each tile type S_i by

$$N(S_i|S_{i-1},\dots,S_{i-t})$$

  where t represents the number of 1's in $\mathcal{C}$.

- Using the absolute counts, *transition probabilities* are calculated. These probabilities represent the likelihood of a tile type following a specific arrangement of adjacent tiles, the tiles considered for this arrangement being once again defined by the chosen configuration $\mathcal{C}$. The transition probability of a tile type S_i is computed as:

$$P(S_i|S_{i-1},\dots,S_{i-t}) = \frac{N(S_i|S_{i-1},\dots,S_{i-t})}{\sum_j N(S_j|S_{j-1},\dots,S_{j-t})}$$

In practice, I coded a function which, for a given configuration and a given training dataset, builds a dictionary whose keys are all the possible n-grams of adjacent tiles encountered in the training set, and whose values are also dictionaries, where this time the key/value pairs are given by all the possible tile types and their associated transition probability computed following the above algorithm. Training a model is therefore equivalent to building this dictionary of probability transitions, and depends of course on the choice of both the configuration and the set of rooms considered for the training.

Classification of rooms is at least suggested in order to train such a model. I will also experiment and try to figure out if a single model training on the whole set of rooms is efficient enough, but my first intuition is that *Celeste* presents by essence a great diversity in terms of level design, and mixing altogether this plethora of well-designed levels would probably result in a sub-optimal pattern extraction and to a confused level generation. A simple example would be that some levels are meant to be more vertical than others (one could think of rooms that have exits on their top and bottom exclusively), and therefore the underlying structures display more verticality as well. When it comes to the gameplay, it can be a tough task for some rooms that present dynamics that are neither totally vertical nor horizontal. I could think of creating a third class to label rooms with mixed gameplay. A simple way to classify rooms is to consider the level to which they belong; as one would except some continuity within a same level, I believe that training a Markov Chains-based model on separate levels could enhance the pattern extraction and consequently make it really good at understanding what composes a good room, at least for every single level. On the other hand, I would definitely assume that training one model on the whole set of rooms would be counterproductive, as patterns and gameplay-defining structures might get mixed up, and the probabilities of given entities that appear on specific levels only would get drowned, leaving the floor to only a couple omnipresent entities like spikes.

Some extension of this study would be to find some alternative training data splitting, like classifying the rooms depending on their difficulty. To that end, one could think of using the extensive classification made for the *Randomizer* mod introduced earlier. Although it is a subjective classification, it is a good estimate of how hard a room is to clear and could be a good starting point to split existing rooms into several training sets.

Room generation Once the model has learned a given MdMC for a pair configuration/training dataset, meaning that I have built a dictionary of probability transitions (DPT), I create an room array full of zeros R that I update using an iterative process, starting from the top-left tile $R_{2,2}$ (accounting for the size of the configuration matrix), the following steps are done for every single tile, from left to right, and then top to bottom:

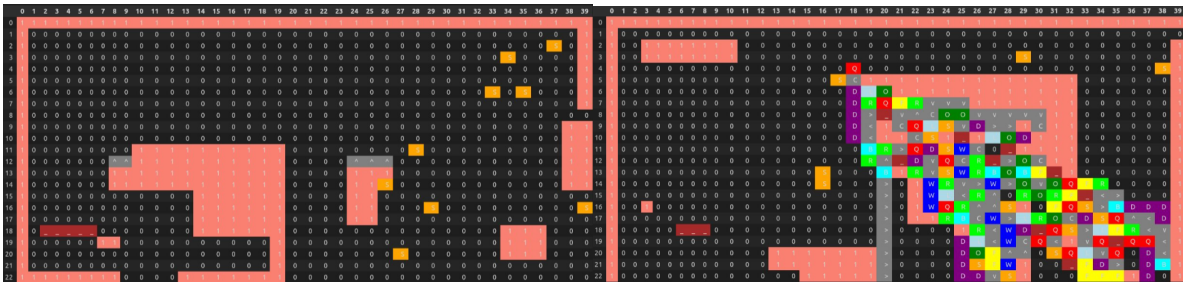
- I extract the adjacent tiles n-gram
- The corresponding probability distribution in the DPT is
- The next tile is picked randomly among this probability distribution

However, a major issue encountered using this process in [Snodgrass and Ontanon, 2013] was the handling of unseen states, preventing this method from being used in-game as the PCG model would

eventually fail. I adapted this process so that the room generation would go through; if the model ever encounters an **unseen state** (*id est* a n-gram that has not been faced during the training step), I introduced a backtracking possibility: the generation process goes back, and tries generating another tile until the updated n-gram is known. If all the tile types have been tried unsuccessfully, the process goes one more step back until one of the two outcomes is reached:

- A combination of tiles is finally found such that the model no longer faces an unseen state, the generation process resumes normally
- There is no possible combination of tiles avoiding an unseen state and the model reached the backtracking maximum depth (I set a limit such that room generation does not become indecently time consuming): I solved the unseen state issue by simply setting a random tile type

One could think that backtracking is therefore useless and that I could simply solve any unseen state issue through random generation; however, experiences confirmed that this would inevitably lead to frequent degeneration as random tile generation often leads to some more unseen states, and an apparently local problem results in a global generation failure. Combination of both random generation and backtracking allows the model to achieve complete room generation while avoiding the generation of random diagonals.



(a) with backtracking (b) without backtracking

Figure 5: Comparison of room generation with and without backtracking

4.2.4 Post-processing and playability

After generating rooms for *Celeste* using a Markov Chain-based method, a crucial step involves post-processing the generated layouts to ensure that they are not only structurally coherent but also playable. This post-processing stage focuses on refining the generated rooms by creating accessible exits and adding mandatory gameplay elements like respawn points, which significantly enhance the playability rate of the rooms.

Creating Accessible Exits Given that the coordinates of the exits are predetermined during skeleton generation, the first post-processing step involves ensuring these exits are unobstructed and functional. This is achieved by clearing an small area around each exit, setting the corresponding tiles to 0 (air/empty space). By doing this, I guarantee that the player can seamlessly transition from one room to another **locally**, without encountering any obstacles that would render the exit inaccessible. Ensuring clear exits is critical for maintaining the flow of the game, as obstructed exits can lead to frustration and disrupt players experience. However, this is definitely not a guarantee of global playability, this measure should rather be seen as a safeguard that does not necessarily alter the global room structure as level design common sense tends to leave the exits rather clear and unobstructed. Still, since the PCG model has no reason to have any knowledge related to exits structure, it is a post-processing measure that proves itself quite necessary; otherwise, most of the generated rooms are simply unclearable just because the exits are obstructed.

Adding Respawn Points To ensure a room is readable by the game, it is mandatory to place respawn points, represented by the symbol P. These points are strategically placed within each room near the exits. Beyond their mandatory aspect because of *Celeste* game desgin, respawn points are also essential for player progression, as they define locations where the player will respawn after dying,

reducing the potential for frustration by minimizing the distance the player must travel after a failure if the player loses at the very end of a room by accident. In addition to respawn points, small platforms made of dream blocks, represented by the symbol D, are placed below each respawn point. Dream blocks serve as temporary platforms that allow the player to regain control and plan their next move after respawning. Dream blocks have also very convenient gameplay properties in *Celeste* , like the ability of being dashed through at no cost, making it a structure that is particularly safe and permitting. The inclusion of these elements ensures that the room is not only playable but also fair and engaging, as it provides players with a fallback structure and reduces the likelihood of them being stuck in difficult or unfair scenarios, or worse: stuck in a deathloop.

Ensuring playability To make sure a whole level is playable, I wanted to make sure that all rooms were independently playable; because of the way I created the *Celeskeleton* module, rooms are properly connected by design, and if all rooms are playable, it ensures the whole level is playable too. Playability details will be extensively discussed in section 5.1. As far as the PCG pipeline is concerned, if a room is determined as not playable during the generation process, then it is generated again until playability is finally achieved.

4.2.5 Room encoder

The Room Encoder module serves as the critical counterpart to the data loader module presented in 4.2.1. While the data loader decodes existing rooms into data matrices that are used for training, the Room Encoder performs the reverse operation: it encodes generated level data into a format that is interpretable and playable within *Celeste* . Essentially, this function takes the output from the level generation process—including every single room generated but also the metadata contained in the level skeleton mentioned in 4.2.2, necessary for the spatial arrangement of rooms—and converts it into a binary file compatible with *Celeste* 's engine.

The encoding process involves multiple components. First, the room's primary data, which includes the layout of foreground tiles, entities, and any interactive elements, is translated into a structured format. This ensures that the generated room maintains the gameplay mechanics and environmental interactions expected in *Celeste* . Additionally, the Room Encoder manages the metadata associated with each room through the encoding of the level skeleton. This includes the room's origin within the level, its dimensions, the placement and functionality of exits, ensuring that rooms are arranged and connected within the overall level, thus enabling seamless navigation from one room to another. Moreover, other less critical gameplay parameters like wind conditions, gravity, and music settings could be integrated here, leaving the door open to a more complete level generation.

The functionality of the Room Encoder has been validated through testing, where rooms and even entire levels generated via coding were successfully encoded and played in *Celeste*. This capability makes the Room Encoder module a central and indispensable component of the project, as it bridges the gap between abstract data and concrete, playable game content. Without it, the transition from a procedurally generated level design to an actual gaming experience would not be possible, underscoring its importance in the conception of an end-to-end workflow.

4.3 From CLI to playable binary files

The entire procedural content generation (PCG) pipeline described in the previous section can be initiated with a single, parametrizable command line. Here follows a brief summary of the input parameters that can be modified:

- `--config`, `-c` (`str`): Configuration matrix used to train the MdMC model
- `--training-dataset`, `-td` (`str`): Room dataset used to train the MdMC model
- `--nb-rooms`, `-nr` (`int`): Number of rooms to generate
- `--proba`, `-p` (`float`): Probability-p_2 introduced in algorithm 1-of generating a labyrinth-style level. 0 creates a pathway, 1 results in a completely random room order
- `--room-size`, `-rs` (`list of two ints`): Room dimensions, specified as width and height

- `--bt-depth`, `-btd` (`int`): Maximum backtracking depth for room generation
- `--tries-limit`, `-tl` (`int`): Maximum number of attempts to generate a **playable** room
- `--reset-skeleton`, `-r` (`bool`): Whether to reset and regenerate or not the entire skeleton if room generation fails

This command loads the necessary training dataset and builds the corresponding DPT based on the specified configuration. The DPT helps in structuring the generation process by defining how different tiles and entities transition within the game environment.

Once the DPT is established, the pipeline generates a level skeleton using the specified number of rooms and probability p_2, which outlines the layout and connections between rooms. The MdMC model then fills each room with appropriate gameplay elements, using the specified backtracking depth.

For each room generated, post-processing steps ensure that each room is playable, re-generating any rooms that fail to meet playability criteria (discussed in section 5.1). Sometimes, because of the randomness introduced in the Celeskeleton module, some room arrangements considerably hurts the likelihood of generating playable rooms; the last two arguments are actually introduced to make sure the level generation process avoids such loops and eventually even reset a level skeleton which seems to be problematic.

Finally, the Room Encoder module converts the entire level into a binary file that is ready to be played in *Celeste*, making the entire process from command input to a playable level both streamlined and efficient.

5 Evaluation of generated rooms

Generating rooms and levels is, of course, the main objective of this project, and that is what I achieved through the pipeline I built and described in the previous section. However, to create a truly effective PCG system, especially for a game like *Celeste*, having robust evaluation tools is essential. Even simple metrics or scores enable valuable feedback that can be used to fine-tune the various control parameters throughout the generation process. Evaluation is not just about model optimization; it's a crucial step for refining the output product and gaining a deeper understanding of how different parameters influence the results.

At this stage of the project, I can generate levels from a simple command line as presented in the subsection 4.3, but I only have a rough idea of how each control parameter affects the final output, let alone how to set them to create great, playable rooms. Thinking even ahead of this project, I built this pipeline leaving enough room for flexibility, especially when it comes to room generation. I believe that the PCG model I designed for this thesis is definitely far from being optimal, and I am convinced that other hybrid approaches mentioned in 2, while being more complicated to implement and adapt to *Celeste*, would produced more elaborated and coherent levels. My choice, as explained earlier, has definitely been highly motivated by the implementation subjective simplicity-I still had to write around 2000 lines of code for this project-and this part also aims to finding the limitations of such a "simple" PCG model.

But first things first: what makes a room 'great'? And what does it even really mean for a room to be 'playable'? To address these questions, I'm introducing three key metrics that I want to evaluate for every room generated: *playability*, *interestingness*, and *difficulty*. These metrics will guide the fine-tuning process and help determining to what extent generated rooms are not only functional but also engaging and appropriately challenging for players. In the following subsections, I will describe more precisely each one of these metrics and properly define them using some mathematical formalism, why and how I think they matter for the player experience, and I will present experiments set up to evaluate the current PCG model performance with respect to these three key aspects.

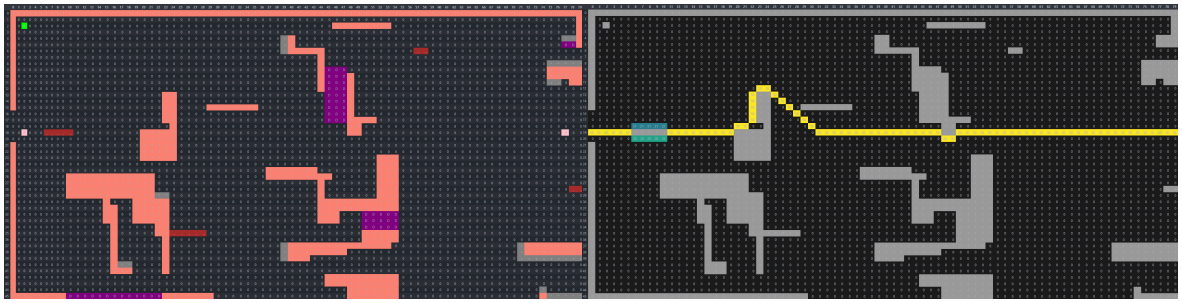
5.1 Playability

5.1.1 Definition & Justification

When introducing the research question in section 3, I emphasized the importance of playability in my project; many papers dealing with PCG applied to level generation in platformers present only a low playability rate in the generated levels - it has nothing to do with the quality of their generators, the approach is just completely different. As my project was initially thought of as a mod usable by the *Celeste* community, I initially wanted to achieve a decent playability rate to keep the generating time rather low. I still stand by this idea and the whole architecture of my generator has been heavily influenced because of this choice.

In the context of procedural content generation (PCG) for 2D platformers such as *Celeste*, assessing the playability of generated rooms is crucial for ensuring that they are navigable and engaging. The playability metric focuses on evaluating whether a room's exits are connected by a viable path, which is fundamental to the gameplay experience. In my analysis, I consider rooms as 2D arrays with exactly two exit points. My goal is to determine whether a path exists between these exits, adhering to the physics and movement constraints of the game.

To achieve this, I use an adapted A* pathfinding algorithm, tailored specifically for platformer games. Standard A* algorithms, while efficient for general pathfinding tasks, do not account for the unique physics dynamics of games like *Celeste* , where gravity and player movement significantly influence gameplay. Traditional A* may find paths that are technically valid but impractical due to the game's specific physics constraints, such as gravity affecting jump trajectories and platform interactions as shown in figure 6b.



(a) Example of room used for A* refinement (b) Heatmap of paths found, base A* - 50 runs

Figure 6: Base version of A* - no directional weighting and no custom heuristic

While being functional, this algorithm is definitely not suitable for playability assessment as the paths found lacks of realism. My custom A* implementation addresses these challenges by incorporating game-specific adaptations. Firstly, I introduced randomness in neighbor selection by shuffling potential next steps, which encourages diverse pathfinding outcomes and mimics the exploratory nature of gameplay. Additionally, I assign varying weights to movement directions to reflect the game's gravity. For instance, movements that go against gravity are penalized more heavily, making paths that align with the natural gameplay mechanics more favorable. These weights were then added to the heuristic, making the path generation way more diverse as one can see in figure 7. However, this is not completely satisfying as the variance is now too high and some paths are definitely not really aligned with the *Celeste* gameplay.

To constraint the paths to be more aligned with the gameplay, I added to the directional weighting a penalty score to the heuristic being the distance to the closest non-lethal entity (NLE), *id est* the closest tile one player could reach. Indeed, in a platformer, the player needs to go from a platform to another, sometimes reaching some in-between entities to get some help. A gameplay-accurate path could not reasonably be far away from all NLEs available in a room. The results of this final adaptation is shown in figure 8, and is very satisfying, as this adapted version of A* presents some variance in the paths found, while looking realistic in terms of feasibility from a player point of view.

This adaptation within the heuristics of A* aims to generate more realistic paths that are feasible

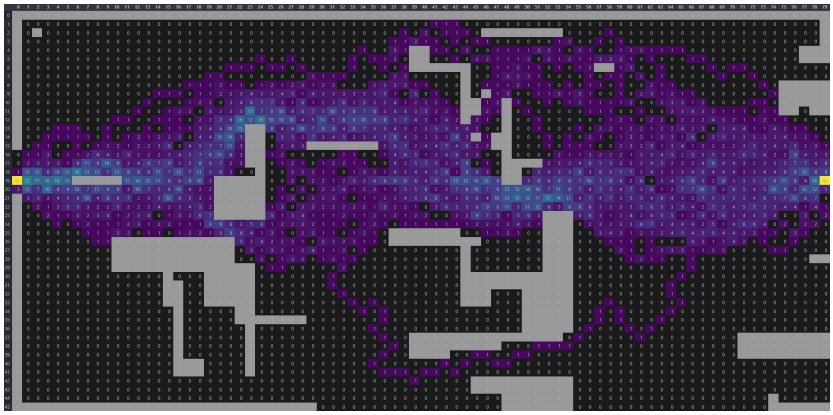


Figure 7: Heatmap of paths found, A* with weighted directions - 50 runs

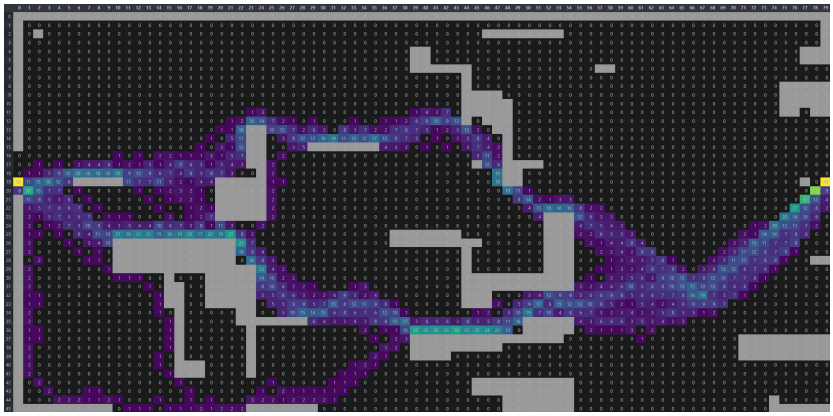


Figure 8: Heatmap of paths found, A* with weighted directions and adapted heuristic - 50 runs

within the game's physics constraints, but this approach can also become quite consuming both in terms of memory and time and hence requires to find some balance in the weights and the heuristic that are ultimately chosen, as suggested by the work from [Iskandar et al., 2020] on reviewing pathfinding algorithms for platformers. Ultimately, this project was aimed to be a real-time module for *Celeste*, so I want to keep the computing time as reasonable as possible. Figure 9 shows that, modifications in the pathfinding algorithm lead to exponential gaps in computing time, and this effect is even amplified with the size of the investigated room.

I ran all two versions of A* presented above, base-A* vs the final tailored-A* (let me call it *Celeste*-A* to avoid any confusion), on the same set of 150 rooms, with room sizes being 40×23, 80×46, and 120×69 (50 rooms for each size). Another conclusion came from this experiment. While each iteration of *Celeste*-A* is indeed more computationally demanding, I noticed that this version of the pathfinding algorithm required less iterations to actually find a path because of the more elaborated heuristic. It has little impact on small rooms, but as I experimented for larger rooms, I noticed that *Celeste*-A* actually even achieved better runtimes as the base A* was taking so many iterations. In fact, base A* often reaches the iterations limit (set by default to the area of the room) to solve the pathfinding problem in large, complex rooms filled with obstacles. In figure 9, the third experiment with 120×69 rooms displays this issue, with room 23 being the perfect example of what happens there. While it occurs rarely for smaller rooms, third plot shows that base A* is less efficient than *Celeste*-A* for around half of the rooms, I showed the explicit values for room 23 in 1. If high generating times for both algorithms show that the room is most likely unplayable, some rooms with seemingly complex gameplay show that *Celeste*-A* can find a room when base A* fails to do so. However, for rooms with simpler geometry, A* still works fine, if I let the path realism component on the side.

Beyond the need of gameplay-accurate paths for better room evaluation, it somewhat proves that the PCG generator does a decent job in capturing the global structure of a room, and manages to produce

Algorithm 2 A* Pathfinding Algorithm adapted to *Celeste*

```
Initialize start and end nodes
Create open_list (priority queue) and closed_list
Add start node to open_list
Define movement options and costs
while open_list is not empty do
    Remove node with lowest f from open_list (current_node)
    Add current_node to closed_list
    if current_node is the end node then
        Return path from start to end
    end if
    for each adjacent node do
        if node is within bounds then
            Calculate costs and heuristic for this node
            if node is not in closed_list or has a better path then
                Add node to open_list
            end if
        end if
    end for
    if iteration limit exceeded then
        Return None (no path is found)
    end if
end while
Return None (no path is found)
```

Room	Algorithm	avg. nb_iter	avg. time (s)
23	A*	30420	329.4
23	*Celeste*-A*	7384	15.3

Table 1: Comparison of base A* and *Celeste*-A* performances on a complex room - average on 50 runs

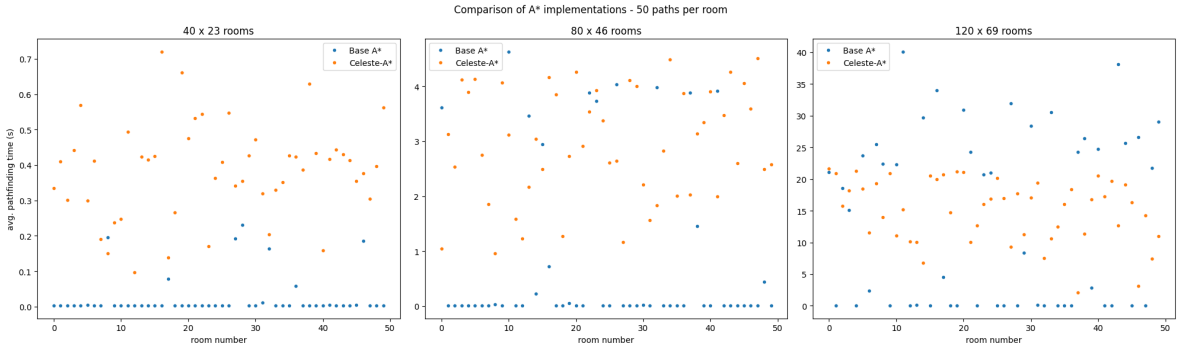


Figure 9: Running time of pathfinding algorithms and effect of the room size

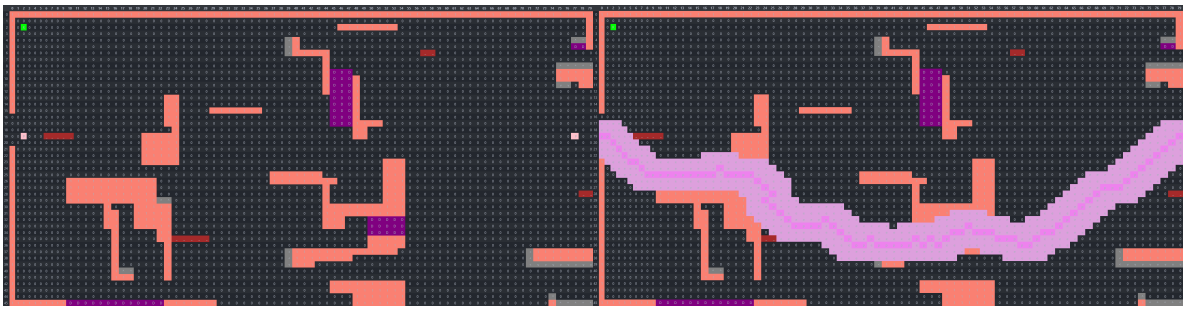
coherent rooms from a player perspective, as suggesting *Celeste* gameplay elements to the heuristic enhances pathfinding performances.

The pathfinding algorithm's output is not uniquely about finding a path but ensuring that it aligns with gameplay realism. Of course, not finding a path is definitely a good indicator of the unplayable nature of a room, but finding a path does not ensure playability. However, paths that respect the game's gravity and movement constraints are more indicative of a playable room. Furthermore, such paths can be used to compute metrics like interestingness and difficulty, as they are quite representative of room areas in which a player is expected to go through. To further evaluate playability, I introduce a path evaluation metric based on the proximity of the path to Non-Lethal Entities (NLEs). This metric is defined as:

$$s_{path} = \frac{1}{N} \sum_{i=1}^{N} Distance_i$$

where $Distance_i$ represents the distance from each point i on the path to the nearest NLE, and N is the total path length. This metric quantifies how well-supported a path is by nearby NLEs, which are essential for player progress in the game. A lower score indicates that the path is surrounded by NLEs, enhancing playability, while a higher score suggests a path that is less supported and potentially less playable.

As mentioned previously, I also used paths to evaluate other metrics than simply playability. To that end, I defined an *area of interest* which simply consists in all the tiles from path, enhanced with the n tiles below and above each tile of the path. It is an estimate of the area that a player will most likely visit to cross the room. Therefore, this zone, noted AOI_n, can be used as a mask to extract lethal/non-lethal entities, safe spots, or even holes that will be used in the next sections for evaluation purposes.



(a) Example of room used for reference (b) Same room with path and AOI displayed

Figure 10: Example of an AOI_n with $n = 2$

In summary, this playability metric combines advanced pathfinding techniques with game-specific adaptations to assess room connectivity and environmental support. By integrating these methods

and evaluating paths with respect to NLE proximity, I provide a robust framework for ensuring that generated rooms are not only navigable but also align with the gameplay experience of *Celeste*.

5.1.2 Maximizing the playability

When evaluating the playability of procedurally generated rooms in *Celeste*, it's essential to identify the parameters that significantly influence this aspect. Understanding which factors most impact playability will enable us to refine the generation process, ensuring that subsequent rooms not only meet aesthetic and design standards but are also functional and engaging for players. I am looking for parameters that are satisfying enough to generate baseline rooms for interestingness and complexity evaluation.

To achieve this, I will assess playability across several key parameters:

- Configuration: the matrix used to train the MdMC model
- Training Set: the source data from which the Markov Chain model learns, as the diversity and underlying structures in this data are likely to affect the generated rooms
- Backtracking Depth: to find a balance between stable generation and exploration of unseen states
- Room Size: from what I observed in figure 9, size might definitely affect the navigability of a player as more room means potentially more structures and difficulties to overcome

Configuration Among all possible configurations using a 3×3 matrix, they have not really been evaluated from a playability point of view in [Snodgrass and Ontanon, 2013]. The idea here was to verify if my intuition about the configuration 0000010112 was correct: configurations not taking all 3 tiles around the tiles of interest will eventually lack of coherence when it comes to generation. Let's think about 000000012, this configuration only considers one tile on the left, meaning that there cannot be any coherence between generated lines, and I would expect generated rooms to be chaotic, hence not playable. Then, I would expect the configurations considering too many tiles to be too restrictive, and mainly generating a lot of unseen states everywhere, resulting in an almost completely random generation, leaving only little room for playable rooms. I evaluated the playability here generating 100 rooms of two different sizes. Given the non-deterministic nature of *Celeste*-A* I ran the pathfinding algorithm 10 times per room, and considered a room to be playable as long as at least one of these occurrences succeeded in finding a path. Determining a real playability threshold will be done in a further section. As shown in figure 11, the only outstanding configuration is indeed 0000010112, validating our intuition. The only configuration I consider for the rest of the experiments is 0000010112. Looking at the generated rooms, I even validate the hypothesis one step further: most of the generated rooms for all other configurations are full of symbols randomly distributed.

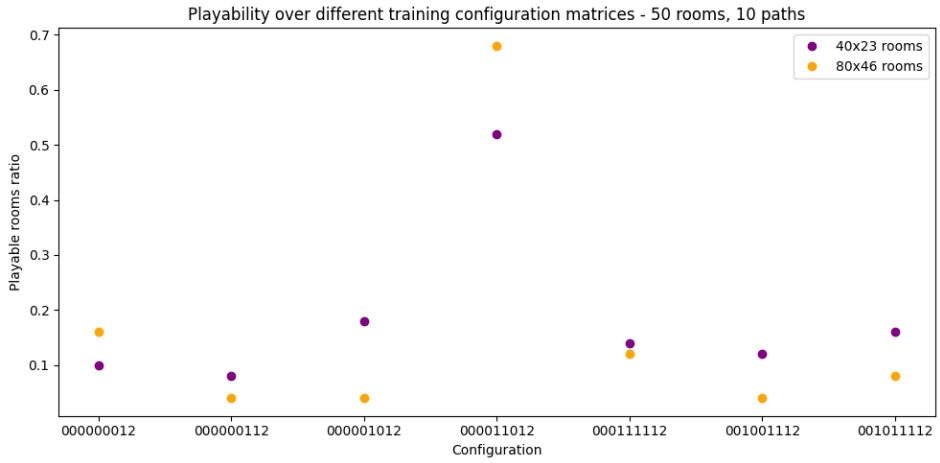


Figure 11: Playable rooms ratio achieved by different configurations over two generated sets of 50 rooms - 10 *Celeste*-A* iterations

Training Set As mentioned earlier in 4.2.3, I grouped data from *Celeste* by levels. All rooms from the first level are in the training set 1, and so on, for all 9 levels of *Celeste*, representing almost 800 rooms in the global training set. An assumption I had when mentioning this grouping was that, I think the model would most likely benefit from learning from a subset of the dataset. Indeed, as a player, I can see the continuity in the rooms of a same level. They are not identical, but present similar structure and elements of gameplay. While some levels could be interesting to combine, I think training the MdMC model on the whole dataset would be sub-optimal for two reasons:

- Some entities, like dream blocks, appear only in a fraction of *Celeste* levels. Training on all available levels would inevitably lead to a dilution problem; probability of rare entities would be so little that such elements would almost never be generated, let alone in a playable room. This reduces considerably the diversity of generated rooms. I would rather train several models, and use a different model for each room, to keep this diversity in the generation process.
- Some levels have completely different designs. From the pathway-like levels that could remind one of basic platformers like *Super Mario Bros.*, to the levels that have a very vertical progression (cf. figure 1), some levels even present a labyrinth-like design, mixing vertical and horizontal gameplay. From a player perspective, the structural elements composing such levels are very different and I believe mixing them altogether would bring some confusion in the pattern learning.

To experiment with training sets, I trained a model for each selected set, then generated 50 rooms per model. For each room, I ran *Celeste*-A* exactly 10 times and just like for configuration, considered a room as *playable* if at least one path was found among the 10 iterations. Room size chosen was 40×23, because it is the most common format in the game, as it is the largest size displayable at once.

For the training sets, I considered each individual level of course, but also all 2, 3, and 4-levels combinations; for the sake of understanding the experiment results, a training set made of rooms from levels 2, 3, and 8 will be named 238, so will the model trained using this dataset. Finally, I also trained a model on the global training set for the comparison. That makes in total 256 training sets considered, and as many models trained-and 128000 runs of *Celeste*-A* for this experiment alone. Each one of the 256 trained models has been evaluated on 50 *rooms,* 10 *paths*; I therefore computed 256 resulting playable rooms ratios. As representing all results in a plot or a table would be highly unreadable, I decided to group the results by levels, as seen below in figure 12.

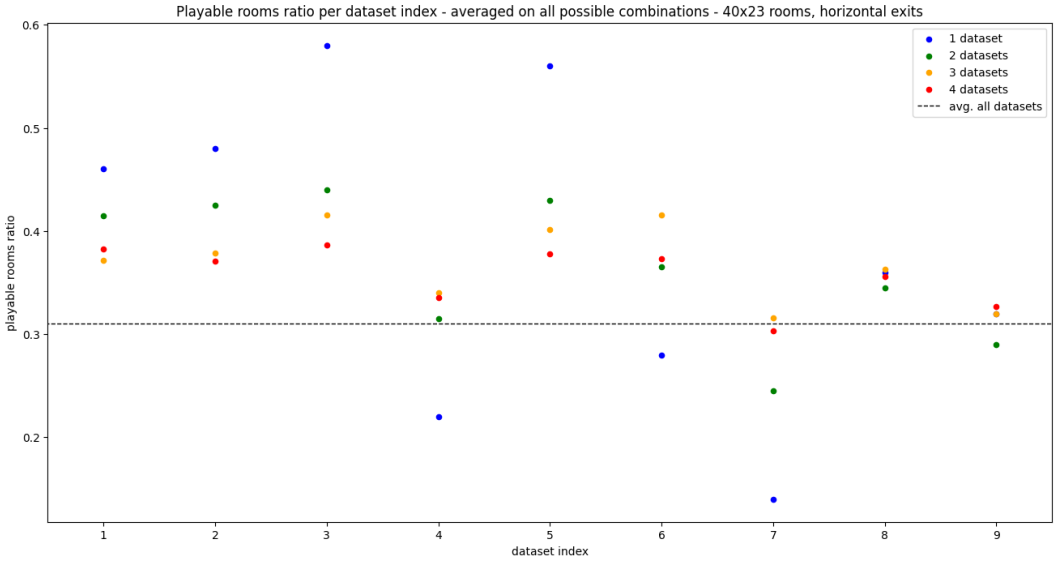


Figure 12: Impact of levels as training datasets in the playable rooms ratio - 10 *Celeste*-A* iterations - playability of model trained on all datasets: 0.31

Blue dots represent the performance achieved by training the PCG model on individual levels, the dataset index being the level number. For the other colors, some clarifications are necessary: let $n \in \{2, 3, 4\}$ be the number of datasets considered for the grouping. For each level $k \in \{1, \ldots, 9\}$,

I computed the average of the playable rooms ratios achieved by all models trained on n-datasets in which k appears. For example, the level 2 appears in 12, 23, 24, 25, 26, 27, 28, and 29; so the green dot for level 2 is computed as the average playability of these eight models. Therefore, what you can see in figure 12 is the average *contribution* of every single level when augmented with other datasets. The main takeaways of this experiment are the following:

Overall, it seems than models trained on individual models perform well in terms of playability. There are clear exceptions, but they are definitely expected:

Figure 13: Celeste spiky sixth level

- Level 4 and level 7 have a lot of moving entities that have been ignored since they are not easily compatible with the grid representation of rooms. But more importantly, there is a lot of vertical gameplay (exits are mostly on top/bottom instead of left/right). The learned structures are not well-adapted to horizontal gameplay, which explains the poor playability.
- Finally, level 6 features very complicated rooms with many spikes, requiring the player to adopt very precise gameplay. These rooms often contain floating spikes (unlike other levels where spikes are attached to a wall), biasing the model into placing walls and spikes everywhere, which likely lowers playability.

These single levels exceptions aside, it seems that training a model on the whole 9-levels set indeed lowers the achieved playability: with a value which seems to converge to 0.31 (cf. figure 14), it seems that models trained on smaller, coherent subsets are doing better. Models 1 and 2 achieve great results with a playability near 0.5, and model 3 and 5 even display playability above 0.55. Of course, it seems that combining these training sets harms their playability ratio, as results achieved with more datasets are a bit lower; however, it also accounts for "undesirable" combinations, like model 57, achieving a miserable playability rate of 0.28 due to the antinomic design of the two levels (and model 7 alone achieves merely 0.14), and, while playability seems to be affected negatively, combining training sets bring some diversity in generated tiles; e.g. model 25 enables dream blocks (specific to level 2) to be associated with red boosters (specific to level 5), bringing some more diversity and potentially exclusive gameplay associations in the generated rooms. Sacrificing a little playability for the sake of diversity and originality seems to me a fair price to pay.

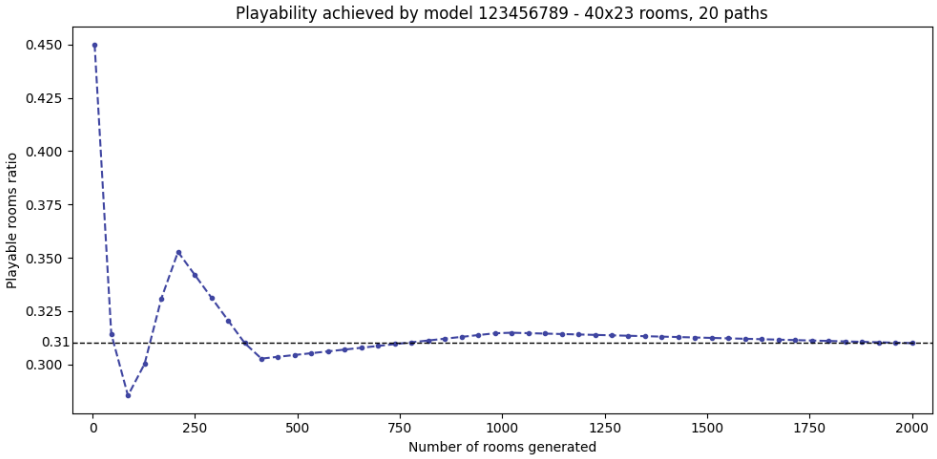


Figure 14: Convergence of the playability ratio for the model trained on all nine levels data

One more thing I had a look at is the path evaluation defined above as the average distance of a path

to the closest safe tile, the closest NLE. The results are shown in figure 15; please note that I made the same grouping than the one explained for the figure 12.

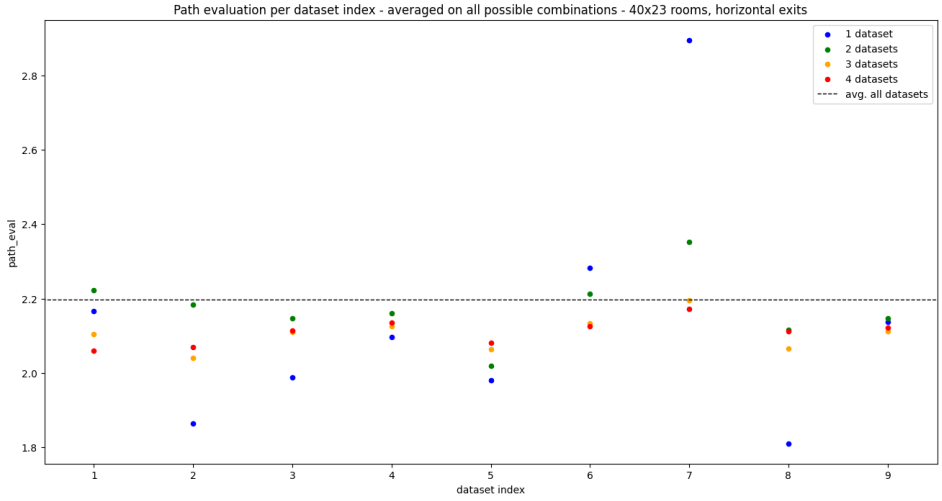


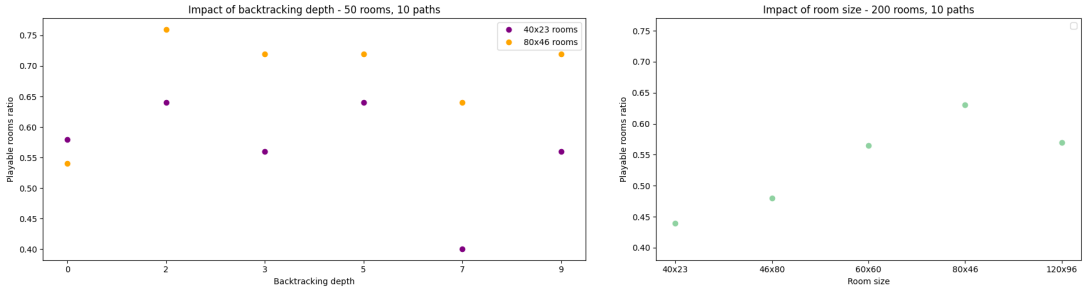
Figure 15: Impact of levels as training datasets in the average path evaluation - 50 rooms, 10 *Celeste*-A* iterations; average on all found paths

From a data point of view, the training set 2 looks very promising: the playability ratio was satisfying, and now the average path score is by far one of the lowest, meaning that in average all rooms generated using this model are well supported by NLE, providing safe spaces for the player to progress through such rooms, this is the training set I chose for the backtracking and room size experiments. Model 2 produces simple, yet coherent rooms, and is perfect as a baseline model to experiment the impact of other parameters.

Backtracking depth & room size When investigating the impact of backtracking on room generation, it did not appear that the value really mattered as one can see in figure 16a. For rooms that are bigger, the absence of backtracking starts to have a given weight: degeneration like the one shown in figure 5 happens more frequently, and if it happens in one of the first rows in the generation process, chances are this randomly generated diagonal prevents any path from existing. This experiment confirms that the use of backtracking is necessary to generate coherent and playable rooms, but there is no clear sign of a trend giving an optimal value. I would keep it to 2, to still allow for some more variety in the generation.

However, what really strikes me in figure 16a is the fact that playability was consistently higher for bigger rooms. I hence decided to generate a few rooms of different sizes. As displayed in figure 16b, playability seems to increase with room size until a certain point. As a room gets bigger, it is definitely more complicated to prevent the existence of paths, the ways and means of completing a room are exponentially increasing. But remember figure 9, when *Celeste*-A* runtime seems to dramatically increase. This loss of playability does not come from the room size itself, but rather from the limitations of the *Celeste*-A* algorithm. A decent fraction of the 120×69 rooms that were marked as unplayable would have been, in fact, marked as playable had I given enough budget to the pathfinding algorithm to find a path.

Mathematically, the bigger a room, the less plausible it is that exits are truly not connected. Indeed, the probabilities learned by a model make very unlikely that it creates a structure splitting the room in two, making the exits completely disconnected, because this is not happening in the training set. More specifically, it *can* happen because of the randomness underlying in such a generative stochastic process, but it is rare. And the bigger the room the model tries to fill, the bigger this "unlucky" splitting structure needs to be, and therefore the less it will happen.



(a) Effect of backtracking depth

(b) Effect of room size

Figure 16: Impact of backtracking depth and room size on playability

5.1.3 Discussion about playability

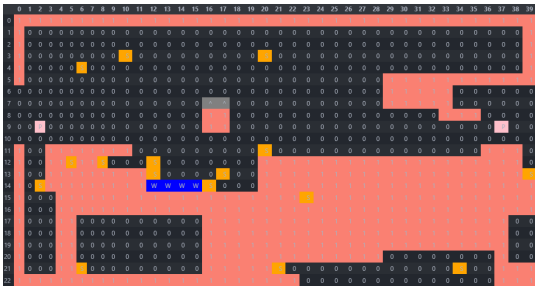
It seems that I found a setup which produces results that are in phase with the announced objectives of this part. Training my PCG model using the configuration 000011012 and training set 2, if I consider generating 80×46 rooms using a backtracking depth of 2, my PCG generator achieves a playability ratio of 0.76, and still 0.64 for 40×23 rooms, which looks great in theory.

However, I want to be precise here. This result looks fabulous, 76% is great! Still, it is a little misleading because of the very definition of *playability* I adopted for the experiments. For such rooms, the number I computed is an *upper bound* of the *true playability*. Indeed, I tried to adapt the behaviour of *Celeste*-A* using a hand-crafted heuristic and directional weighting to make paths more accurate; but when the rooms are truly unfeasible from a gameplay point of view while the exits are theoretically not disconnected, paths will be found anyways. For very large rooms, it is even worse; since the limitations did come from the budget allocated to the pathfinding algorithm, it is impossible to affirm that the playability then computed is an upper bound too. Supposing I could spend all the time and memory I want, then it would eventually find a path for very large rooms and the playability thus computed would be an upper bound for any room generated.

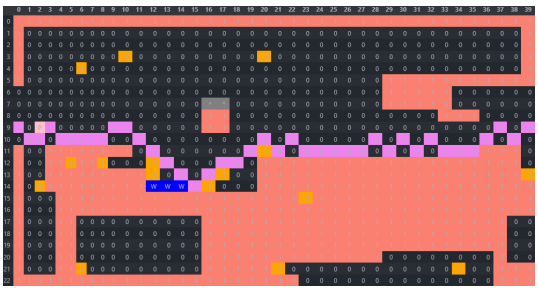
How to determine the true playability? This question is very hard to answer without a proper AI-agent able to test-play generated rooms. Path evaluation is a first good idea, but is not enough in its actually form. This path scoring I computed for figure 15 only takes into account the average distance the path to the closest NLEs. I tried to find a relevant threshold above which a path is not considered as viable anymore through manual testing (*id est* playing the rooms myself), but in practice this score is not characterizing enough to find a decent rule of thumb. The reason for this is, a path can have very close NLEs locally, and then a section where literally nothing can help the player. And while the average distance to the closest NLEs can be rather small because of local structures, a single section where there is no close support at all and the whole path viability can be endangered.

One of my last progresses in this thesis was to add another score to the path evaluation. In addition to the mean, I started considering the variance of the distance of a path to the NLEs. The underlying idea is that the variance gives some indication concerning the distribution of this distance path/NLEs. A small variance coupled to a reasonable average distance is almost the guarantee of having a safe path as it is well and *consistently* supported. And, while the average distance never gets too large because of the limited room sizes I considered, when the variance is abnormally large, it is often a good signal that there is a zone where the path is definitely not supported, which should not happen because of the way *Celeste*-A* is implemented, unless, there is effectively nothing to support such a path.

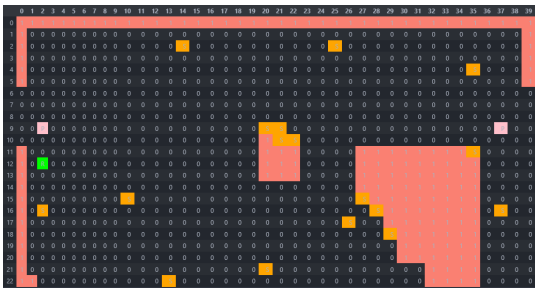
While extreme variance cases seem trivial to assess, this really gets complicated for paths whose variance are near the median; and what about edge cases, like a floor full of spikes? Of course it is unplayable, however the path evaluation metrics would be pretty good. And in the context of such a project where one of the base ideas was to generate levels for real players, I would not mind so much dropping a false negative, I classify a room as unplayable when it is, in fact, playable. However, a false positive would be an absolute disaster. At the moment, the rooms which display a variance above twice the median are usually getting very complicated to clear and a first rule of thumb would be to



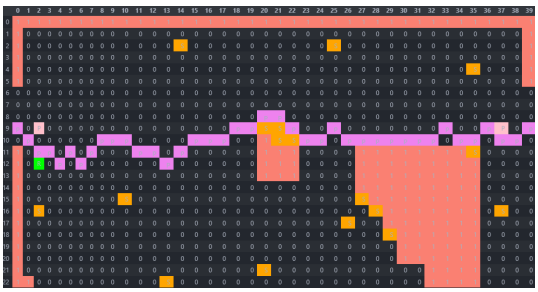
(a) mean: 1.23 / var: 0.22



(b) Path is consistently supported



(c) mean: 3.0 / var: 5.25



(d) Path is not consistently supported

Figure 17: Example of extreme variance paths found during the training of single level models - median mean: 2.00 / median var: 1.65

exclude them. However, this approach does not put an emphasis on the precision but rather on the recall, which is not really what is suited here.

It is still a decent result to be able to define these upper bounds, and I would definitely like to extend this part of the work through bringing some more advanced AI/ML techniques, and to develop and train a classifier aiming to assess rooms' playability, with a special dedication for precision, as ideally I would not let any false positive go through. Some deep learning approaches could prove interesting here, and way more realistic than developing an AI-agent for *Celeste* which is performing well-enough to be assigned to the room playability characterisation. The latter approach would definitely be a good way not to let any false positive go through, but would have to come a long way before completing interesting and challenging rooms. That is definitely a project of its own.

5.2 Interestingness

5.2.1 Definition & Justification

In the context of procedural content generation for 2D platformers like *Celeste*, the interestingness metric serves as a critical measure for evaluating the quality of generated levels. This metric is designed to quantify the level of engagement and variety within a room, reflecting how likely the room is to captivate and challenge players. At its core, the interestingness metric considers several key factors: the density of Non-Lethal Entities (NLEs) and the diversity of these entities.

I define the interestingness I of a level as:

$$I = w_1 \times d_{NLE,global} + w_2 \times d_{NLE,local} + w_3 \times s_{diversity}$$

where:

- $d_{NLE,global}$ represents the global NLE density, calculated as the ratio of the total number of NLEs N_{total} to the room total area A:

$$d_{NLE,global} = \frac{N_{total}}{A}$$

- $d_{NLE,local}$ represents the local NLE density, calculated as the ratio of the number of NLEs in the $AOI = AOI_2$ (defined in section 5.1.1) $N_{AOI,NLE}$, to the area of interest AOI:

$$d_{NLE,local} = \frac{N_{AOI}}{AOI}$$

- $s_{diversity}$ is the diversity score of NLEs, which can be computed using Shannon entropy, with p_i being the proportion of the NLE i in a given room:

$$s_{diversity} = -\sum_i p_i \cdot \log(p_i)$$

and w_1, w_2, and w_3 are tunable weights.

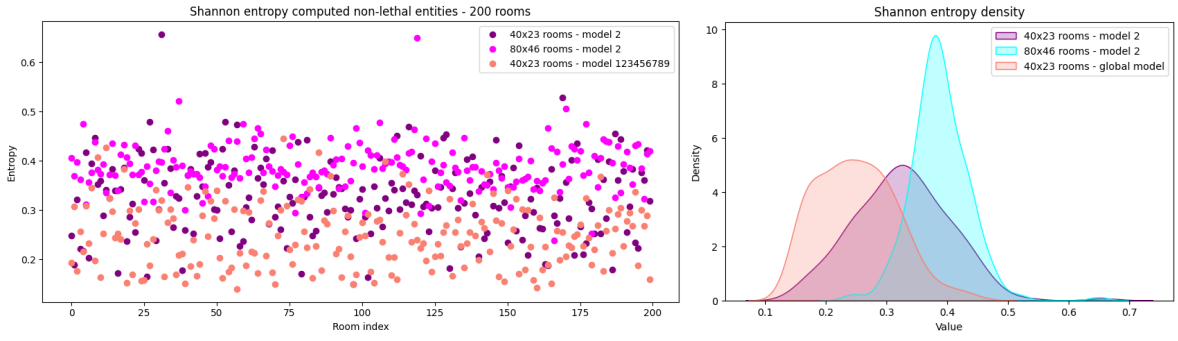
The global NLE density is a foundational component, assessing the overall concentration of interactive elements throughout the room. A high global density suggests that the room is rich in potential interactions, encouraging exploration and offering multiple gameplay opportunities. Complementing this is the local NLE density, which focuses on areas where the player is most likely to spend time. Prioritizing local density ensures that key gameplay zones are populated with engaging elements, directly influencing the player's experience.

To further enhance the metric, I introduce a diversity score that measures the variety of NLE types present in the room. Diversity is crucial in maintaining player interest, as it prevents monotony and ensures that the gameplay feels dynamic and varied. These combined components make the interestingness metric a comprehensive tool for assessing room quality, guiding the PCG system towards generating rooms that are not only playable but also engaging and enjoyable.

This approach to evaluating interestingness aligns with previous research in PCG for video games, where the balance between content richness and player engagement is emphasized. For instance, [Shaker et al., 2010] explored similar metrics in the context of Super Mario Bros., assessing among other criteria the influence of content variety and placement on player satisfaction. Additionally, [Togelius et al., 2011] highlighted the importance of diversity in PCG systems, noting that varied content is essential for sustaining long-term player interest. By grounding this metric in these principles, I ensure that the evaluation framework not only reflects established best practices but also adapts them to the specific challenges and opportunities presented by *Celeste*.

5.2.2 Experiments

To begin with, if it is rather clear that $d_{NLE,local} < d_{NLE,global} < 1$, it is maybe less easy to visualise the diversity score through Shannon's entropy. It is not the first time it is used in a context of diversity evaluation, as a study from [Masisi et al., 2008] uses entropy as an indicator of structural diversity. I wanted to have an overview and comparison between this entropy measures across different models.



(a) Entropy measurements for model 2 vs. global model

(b) Entropy densities

Figure 18: A peak into Shannon entropy to measure entity diversity - 200 rooms per model

As expected, while having more potential tiles in the training set, the global model has such little probabilities for these special tiles to appear that there is definitely less diversity than in model 2,

where less tiles are available but with a higher-one could even say *realistic*-probability distribution, ensuring more diversity in the output rooms. It also seems that bigger rooms mean more diversity through more opportunity for random generation and, once a rare tile happens, especially for tiles like dream blocks in model 2, other occurrences of the same tiles are way more likely to appear.

Technically, Shannon entropy is not limited. However, it is well known that situations of equiprobability actually maximize the entropy, and even if I considered having up to 100 entities uniformly distributed in a room, the entropy would be:

$$s_{diversity} = -\sum_i \frac{1}{100} \cdot \log \frac{1}{100} = \sum_i \frac{1}{100} \cdot \log 100 = 100 \times \frac{2}{100} = 2$$

This could be normalized by dividing by the maximum entropy achievable given the number of entities possible, but let's say it is included in the weight w_3; moreover, I is a score which is not necessarily normalized on $[0, 1]$.

I chose $w_1 = w_2 = w_3 = 1$ and ran an experiment to compare the interestingness of model 2 and global model. Technically, I am expecting to see more diversity in model 2, as well as a better overall capture of underlying structures in *Celeste* . I generated 200 rooms for each model and then computed the mean interestingness of each room over 5 paths-from 1 to 5, depending on how many pathfinding runs have succeeded. Figure 19 delivers again with our expectations; model 2 seems to produce overall more interesting rooms than the global model.

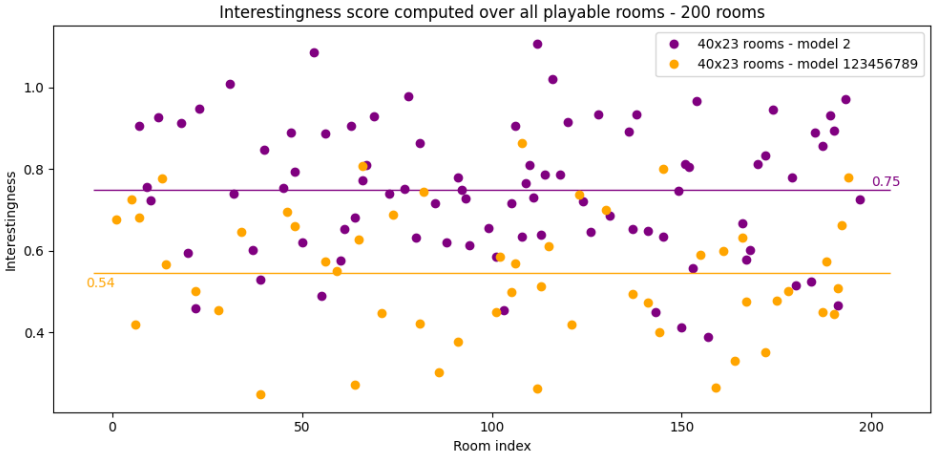


Figure 19: Comparison of the interestingness score between model 2 and global model - 200 rooms, 5 paths

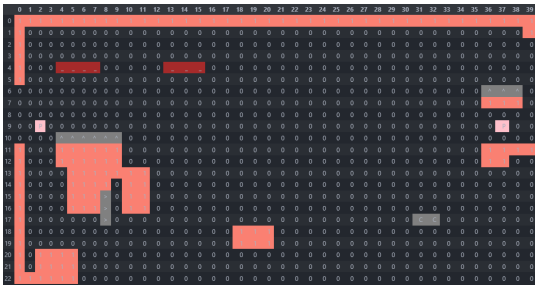
I asked a few players to play 4 rooms that were at the extremes in terms of the interestingness score, two per model, and asked them to give a grade from 1 to 10 to the room interestingness: their engagement with the room, whether they found it fun or not. The idea is to see if players' opinion align well on the score definition.

At first glance, I was not very satisfied, as a player, of the scores for the model 2 rooms, had I been asked which score was paired to which room, I would have swapped both. But room in figure 20b seems to be the most achieved after all; gameplay is clear but allows for some freedom, nice diversity in the tiles; while room 20a appears almost unfinished, like a work in progress, with only a few tile types. On the opposite, the scores seemed totally justified for rooms 20c and 20d, the diversity both in the structures and in the apparent gameplay have been taken in consideration.

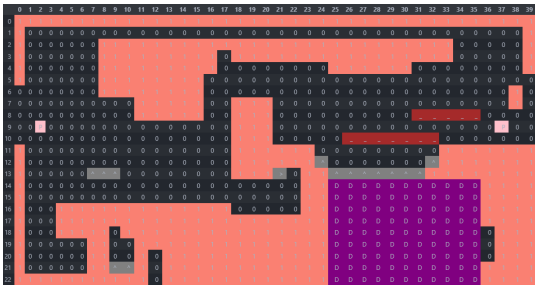
5.2.3 Results & Discussion

A panel of 12 players played and evaluated these rooms. From experts who spent thousands of hours and created their own levels, to absolute beginners, I had a nice diversity within the profiles.

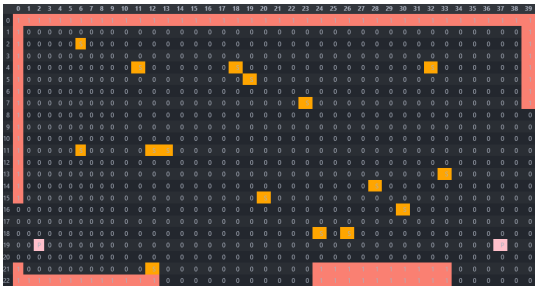
I grouped together the intermediate and expert players in the advanced group, as their evaluation were similar. 8 players belong to this group, while the 4 others never played *Celeste* before.



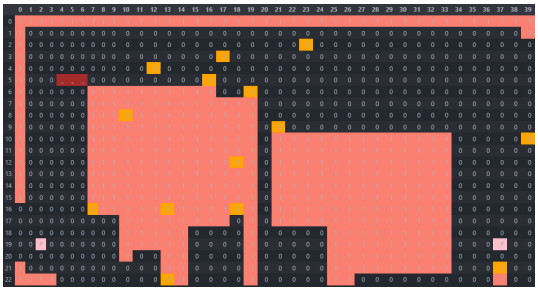
(a) Model 2, $I = 0.41$



(b) Model 2, $I = 1.08$



(c) Model global, $I = 0.33$



(d) Model global, $I = 0.81$

Figure 20: Four rooms tested and evaluated by players; two rooms per model, one among top scores, one among worst scores

	Count	Grade room a	Grade room b	Grade room c	Grade room d
Advanced players	8	6.75	3.25	5.38	5.25
Beginner players	4	4.25	7.75	2.25	7.50
All players	12	5.92	4.75	4.33	6.00

If I look at the results, there is one surprise to me: the fact that the expert group gave a similar note for room 20c and room 20d. While room 20c was basically unbeatable for the beginners group as it requires some advanced mechanic (explaining therefore the evaluation from their group), I was indeed expecting the advanced group to find it more interesting from a gameplay point of view. The biggest issue with this survey concerning experienced players is that since the level is in a brute format-no music, no background-they had a hard time focus on anything else than the gameplay. I acknowledge that at first glance I shared the advanced group's opinion about rooms 20a and 20b, but if I look at it from the beginners point of view, it seems this group is quite aligned with the interestingness definition I came with for this study.

I would have two points that are to me worth discussing; first, even though the beginners group is definitely too small to be statistically relevant (so is the advanced group), I believe that the observed trend is no coincidence: the metric I came with only accounts for NLE density and diversity within the tiles generated in a room. These are only very high-level consideration, and this does not really account for more advanced elements of gameplay like advanced mechanics, precision of the trajectory, and variety in the movements and actions. It seems quite realistic to me that a beginner would have this high-level vision on a video game he never played before, while an experimented player has a different approach when it comes to level design. It needs to be interesting, through originality, novelty, and innovation. Finally, I would say that a next step is to gather more testers, and generate rooms of distributed interestingness (not only top and bottom) and see if that metric is robust to a beginner audience, and coming with a refined version of the metric targeted at an advanced public. This way, it would enable the PCG model to tailor level generation to the target audience, adding one more layer before even mentioning difficulty.

5.3 Difficulty

5.3.1 Definition & Justification

The difficulty metric is another critical component in evaluating the effectiveness of a PCG system for generating levels in *Celeste*. Difficulty in this context is not merely a measure of challenge but a multifaceted assessment of how the generated rooms test the player's skills, manage risk, and ultimately contribute to the overall game experience. This metric considers various factors that contribute to the challenge posed by a room, including the presence of hazards such as holes and lethal entities, the scarcity of NLEs, as well as the spatial distribution of lethal entities.

I define the difficulty D of a level as:

$$D = z_1 \times H_f + z_2 \times d_{LE,local} + z_3 \times s_{scarcity}$$

where:

- H_f represents the hole frequency, calculated as the ratio of the number of holes H to the path length L:
$$H_f = \frac{H}{L}$$
- $d_{LE,local}$ represents the local LE density, calculated as the ratio of the number of LEs in the AOI defined in section 5.1.1 $N_{AOI,LE}$ to the area of interest AOI:
$$d_{LE,local} = \frac{N_{AOI,LE}}{AOI}$$
- $s_{scarcity}$ measures the scarcity of NLEs, which can be computed as the inverse of the local NLE density:
$$s_{scarcity} = \frac{1}{d_{NLE,local}}$$

and z_1, z_2, and z_3 are tunable weights.

A significant factor in the difficulty metric is the frequency and placement of holes—areas that cause the player to fall to their death, often necessitating a room restart. The presence of lethal entities, which confronts direct threats to the player, further contributes to the challenge, as navigating around or through these hazards requires precise timing and skill. Additionally, the scarcity of NLEs is a critical consideration; rooms with fewer safe zones or movement aids inherently demand greater precision and increase the risk of failure, consequently elevating the room's difficulty. By comprehensively evaluating these factors, the difficulty metric provides a nuanced understanding of the challenges posed by the generated levels, enabling the PCG system to fine-tune its algorithms to produce rooms that are appropriately challenging but not unfairly punishing.

Additionally, considering dynamic hazards, such as moving platforms or timed obstacles, further heightens the difficulty by introducing elements of unpredictability and requiring the player to adapt quickly, and could be also considered to enhance this metric; *Celeste* is full of dynamic entities, and even though I did not include such entities in the training because of representation incompatibilities, this is one more potential improvement to explore further. One could even think of considering adding the number or required inputs or the gameplay complexity, had I an AI-agent able to play *Celeste*.

The importance of difficulty metrics in PCG has been widely recognized in the literature. The work of [Sorenson and Pasquier, 2010] on dynamic difficulty adjustment in game AI underscores the role of difficulty in maintaining player engagement and satisfaction. The concept of adaptive difficulty is also becoming more and more central in recent video games, where the system adjusts the challenge based on player performance, highlighting the need for accurate difficulty assessment if one wants to integrate relevant adaptive difficulty within a PCG model. By integrating these insights, the difficulty metric is designed not only to measure the challenge level but also to ensure that it aligns with the intended player experience in *Celeste*.

5.3.2 Experiments

The current design of the difficulty metric seemed to be incompatible with the playability requirements; there was only a limit of difficulty achievable using this metric, and the results were very inconsistent. It was not really possible to run the same experiment than the one designed for playability. I ran again an experiment generating 200 rooms, for each one of team I computed each component of the difficulty score D, as well as the path metrics, like s_{path} and path variance. Due to the very unstable nature of the scarcity, the difficulty score explodes alongside the scarcity, which I tried to investigate in 21. However, a very high scarcity means a very low local NLE density, hence a path which is not supported at all. I displayed the scarcity over the rooms generated from model 2 and model global to investigate the behaviour of this measurement. Moreover, I tried to look into potential correlations between difficulty and path characteristics in figure 22.

5.3.3 Results & Discussion

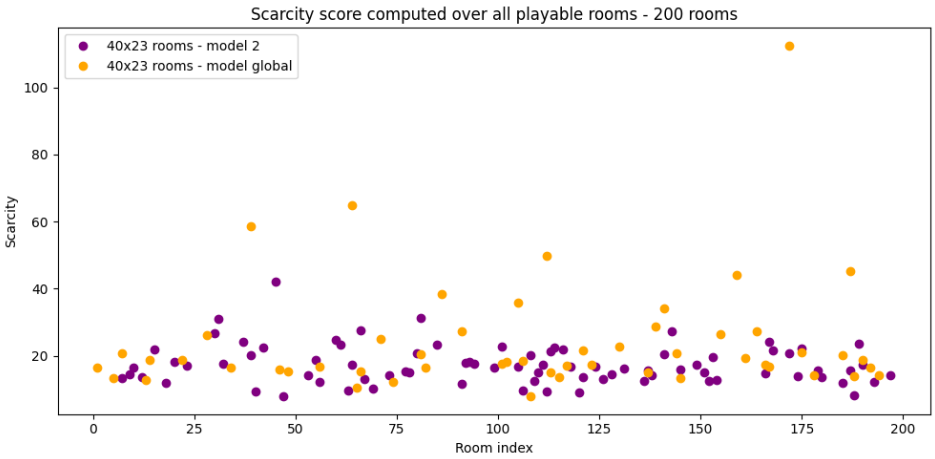


Figure 21: Scarcity score investigation

One can see that the scarcity values really explode for some room from the global model, while it seems more stable for model 2. As mentioned previously, a high scarcity is a marker of very low local NLE density, and by extension, a decently high s_{path}. I looked into most of these rooms manually, and they were simply unplayable. Maybe scarcity, among other metrics, could be of help to characterise better *true playability*. At the end of the day, such rooms should not even be considered for the difficulty/interestingness evaluation, but it requires again a *precise* playability assessment algorithm.

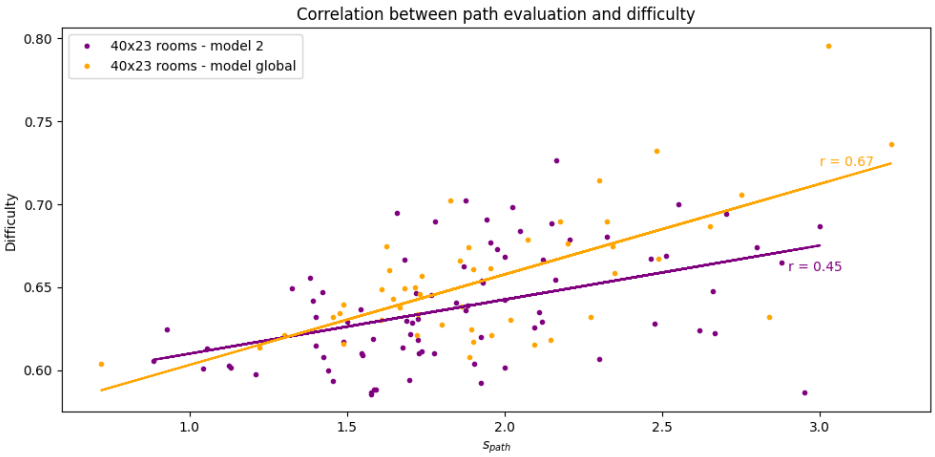


Figure 22: Correlation investigation between difficulty and s_{path} - 200 rooms, 5 paths

Finally, I discovered that with the current difficulty metric, difficulty and s_{path} are moderately positively correlated, and that, to me, is a problem of definition/conception of the difficulty metric. I mentioned it earlier for interestingness, but I think it makes even more sense to consider integrating advanced mechanics and precise gameplay within this concept, as those two elements are truly the foundations for a player progression framework. I think that it might be beneficial to further pursue this study by classifying training rooms not only by levels, but also by difficulty. Therefore, I might be able to train models able to extract patterns related to difficulty, instead of the structural continuity that I was looking for by classifying rooms by levels.

6 Conclusions & Discussion

This thesis presents the development and implementation of a robust procedural content generation pipeline specifically designed for the 2D platformer *Celeste*. The primary focus of this work was to create a method capable of generating playable, challenging, and engaging levels that adhere to the design principles of *Celeste*. Leveraging a Markov Chain-based model, I rather successfully captured the underlying structures and gameplay dynamics of existing levels, allowing the generation of new rooms that maintain both the aesthetic and functional qualities of the original game.

The Markov Chain-based approach, while not revolutionary in itself—having been previously applied to games like Super Mario Bros.—was tailored here to address the specific challenges and nuances of *Celeste* and proved effective in modeling the probabilistic transitions between tiles and gameplay elements, enabling the creation of rooms that are structurally coherent and varied. Post-processing steps, including exit refinement and the strategic placement of respawn points and platforms, were crucial in enhancing the playability of the generated rooms. These steps ensured that the rooms not only met the functional requirements for *Celeste* but also provided a fair and enjoyable challenge to players.

In terms of evaluation, the generated levels were assessed using several metrics, including playability and interestingness mainly, while the difficulty component was tough to define properly. The results, coupled to manual testing, indicated that the rooms generated by the pipeline were for the most part successful in replicating the desired gameplay experience. The playability metric, in particular, showed a high success rate, with most rooms being fully navigable and meeting the expected challenge levels. The complexity of the generated rooms varied, with some rooms exhibiting novel configurations that introduced new challenges, while others closely mirrored traditional level designs. Player engagement and interestingness, measured through playtesting, further validated the effectiveness of the PCG pipeline, as testers reported decently high levels of satisfaction with their engagement with the generated levels, in particular when addressed to a beginner audience.

However, despite the successes, there were some limitations observed. The Markov Chain model, while effective, occasionally produced levels that lacked the creativity and diversity seen in handcrafted levels. While the difficulty of the generated rooms was also for the most part satisfying, I realized that having a close control of the difficulty without taming the playability aspect totally was somewhat bold, and could have been anticipated in the difficulty metric design. On another note, the general level design was not so much matching experienced players' expectations. This suggests that further refinement, perhaps through the integration of additional generation techniques or hybrid approaches, could enhance the creative potential of the pipeline and potentially capture finer details that make the level design in *Celeste* so unique and appreciable. Additionally, the playability metric, although generally high, did reveal occasional instances of rooms where difficulty spikes or poorly placed obstacles detracted from the player experience. Addressing these issues may require several layers of room generation techniques, more sophisticated post-processing techniques, and/or the inclusion of machine learning methods to predict and correct potential playability issues during the generation process.

Looking forward, several avenues for future work have been identified. One potential direction is the exploration of more advanced AI techniques, such as Generative Adversarial Networks methods, to further refine the level generation process and improve the adaptability of the model to different gameplay styles and player preferences. Additionally, expanding the evaluation framework to include more comprehensive player feedback and integrating adaptive difficulty mechanisms could enhance the overall player experience. I would definitely use this revised room generator to produce a reliable

training set for a potential *Celeste* AI agent.

Moreover, the application of this PCG pipeline is not limited to *Celeste*; the underlying principles and methods could be adapted to other platformers or even different game genres entirely. By tailoring the Markov Chain model to the specific mechanics and aesthetic requirements of other games, this simple approach could be extended to generate content for a wide variety of gaming experiences.

In conclusion, while this thesis does not pioneer the use of Markov Chains for 2D platformer level generation, it does demonstrate their effective adaptation to the unique challenges of *Celeste*. While there is room for further improvement and innovation, the results indicate that this method is a promising tool for creating engaging and challenging game content that enhances the overall player experience. I consider this pipeline as a proof of concept; this thesis is laying the foundation stones for building what would be an equivalent to the famous Mario AI Framework, and hopefully adapt part of the work that has been proven successful for Super Mario Bros. to this more complex platformer, *Celeste*.

References

[Adams, 2002] Adams, D. (2002). Automatic generation of dungeons for computer games.

[Balali Moghadam and Kuchaki Rafsanjani, 2017] Balali Moghadam, A. and Kuchaki Rafsanjani, M. (2017). A genetic approach in procedural content generation for platformer games level creation.

[Charity et al., 2022] Charity, M., Dave, I., Khalifa, A., and Togelius, J. (2022). Baba is y'all 2.0: Design and investigation of a collaborative mixed-initiative system.

[Cruor et al., 2018] Cruor, Vexatos, Macho, J., and DemoJameson (2018). Maple: a thin wrapper for generating map files for the game celeste. https://github.com/CelestialCartographers/Maple.

[Dahlskog et al., 2014] Dahlskog, S., Horn, B., Shaker, N., Smith, G., and Togelius, J. (2014). A comparative evaluation of procedural level generators in the mario ai framework.

[Dormans, 2010] Dormans, J. (2010). Adventures in level design: Generating missions and spaces for action adventure games.

[Iskandar et al., 2020] Iskandar, U., Diah, N., and Ismail, M. (2020). Identifying artificial intelligence pathfinding algorithms for platformer games. pages 74–80.

[Johnson et al., 2010] Johnson, L., Yannakakis, G., and Togelius, J. (2010). Cellular automata for real-time generation of.

[Kazemi, 2008] Kazemi, D. (2008). Spelunky generator lessons. http://tinysubversions.com/spelunkyGen/.

[Khalifa et al., 2019] Khalifa, A., Green, M. C., Barros, G. A. B., and Togelius, J. (2019). Intentional computational level design. *CoRR*, abs/1904.08972.

[Linden et al., 2013] Linden, R., Lopes, R., and Bidarra, R. (2013). Designing procedurally generated levels. pages 41–47.

[Masisi et al., 2008] Masisi, L., Nelwamondo, F., and Marwala, T. (2008). The use of entropy to measure structural diversity. *CoRR*, abs/0810.3525.

[Shaker et al., 2010] Shaker, N., Yannakakis, G. N., and Togelius, J. (2010). Towards automatic personalized content generation for platform games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

[Snodgrass and Ontanon, 2013] Snodgrass, S. and Ontanon, S. (2013). Generating maps using markov chains. pages 25–28.

[Snodgrass and Ontañón, 2017] Snodgrass, S. and Ontañón, S. (2017). Learning to generate video game maps using markov models. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):410–422.

[Sorenson and Pasquier, 2010] Sorenson, N. and Pasquier, P. (2010). Towards a generic framework for automated video game level creation. volume 6024, pages 131–140.

[Summerville et al., 2017] Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A., and Togelius, J. (2017). Procedural content generation via machine learning (PCGML). *CoRR*, abs/1702.00539.

[Sweetser and Johnson, 2004] Sweetser, P. and Johnson, D. (2004). Player-centered game environments: Assessing player opinions, experiences, and issues. In Rauterberg, M., editor, *Entertainment Computing – ICEC 2004*, pages 321–332, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Thakkar et al., 2019] Thakkar, S., Cao, C., Wang, L., Choi, T. J., and Togelius, J. (2019). Autoencoder and evolutionary algorithm for level generation in lode runner. *2019 IEEE Conference on Games (CoG)*, pages 1–4.

[Togelius et al., 2011] Togelius, J., Yannakakis, G., Stanley, K., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3:172 – 186.

[Volz et al., 2018] Volz, V., Schrum, J., Liu, J., Lucas, S. M., Smith, A. M., and Risi, S. (2018). Evolving mario levels in the latent space of a deep convolutional generative adversarial network. *CoRR*, abs/1805.00728.