Procedural Content Generation in Relinquish

Modern Artificial Intelligence (Autumn 2020): ***KGMOARI1KU***
14/12-2020

Francesco Frassineti
fraf@itu.dk

Magnus Rubin Peterson
magp@itu.dk

Ting-Yu Kuo
tiku@itu.dk

Abstract—**Procedural Content Generation (PCG) is a vital part of any roguelike game such as *Relinquish*. The first implementation of the *Relinquish* prototype did not include PCG. Therefore, in this paper we design, implement, and evaluate two agent-based PCG algorithms to generate the floor plan of the levels in *Relinquish*. We evaluate these algorithms according to several properties.**

1. Introduction

Our project consists of introducing procedural content generation (PCG) to *Relinquish*, a roguelike game that Francesco's group developed with the Unity Engine during the second semester of the MSc. in Games at IT University of Copenhagen. As it belongs to the roguelike genre, *re-playability* is expected to be a key aspect of the design [1]. Due to time constraints, the game was submitted without any implementation of procedural content generation and each level was completely hand-crafted by the designers. The goal of our work is to improve the *replayability* of the game by designing, implementing, and evaluating different procedural content generation solutions to generate the floors of the game using the rooms that were hand-crafted by the designers as its building blocks.

2. Background

2.1. Procedural Content Generation

Procedural Content Generation, often abbreviated as PCG, refers to the algorithm creation of game contents with limited or indirect user input [2]. As the demand for game contents keep rising and the developmental period of games are shortening, the importance of PCG becomes more significant. One of the benefit of generating content algorithmically is that it reduces the workload of artists and game designers. PCG allows games to be developed faster, cheaper, and with more diversity. Besides generating game content completely by algorithms, mixed-initiative methods can help designers with their tasks, empowering small teams with limited resources. PCG also plays an essential role in roguelike games. In fact, the whole genre of game would not exist without PCG. An ideal PCG algorithm should have the following properties:

1) **Speed**: the speed of content generation.
2) **Reliability**: the generated contents should be above certain quality and should not violate the gameplay experience.
3) **Controllability**: the content generator is easy to control by a human user.
4) **Expressivity and diversity**: the content generator is able to generate a diverse variety of content.
5) **Creativity and believability**: in most cases, the generated contents needs to look like they have been designed by human designers.

We evaluate our content generators based on the properties listed above.

2.2. Anatomy of PCG

Although there are a large variety of games, a previous review article proposed six main classes of game contents that could be generated by PCG algorithms [3]. The classes are not only limited to contents inside the game but also derived contents designed to immerse players. The classes were structured as a pyramid, in which classes of content closer to the bottom could be the foundation of the classes closer to the top. We listed the pyramid bottom-up here:

1) **Game bits**: elementary units, textures, sound, vegetation, buildings. fire, water, stone, etc.
2) **Game space**: the environments: indoor maps, outdoor maps, etc.
3) **Game systems**: biological ecosystems, road networks, NPC behaviors, etc.
4) **Game scenarios**: puzzles, story, levels, etc.
5) **Game design**: rules, player goals, system design, world design
6) **Derived content**: side-products of the game, such as game news, broadcast, etc.

In the current article, we focus on the level of *game space* and search for a suitable algorithm for game map generation.

2.3. Evaluation of PCG

Evaluating a content generator is important yet complex. Establishing good standards of a good content generator highly depends on the purpose and settings of the game [2]. Here, we listed a top-down and a bottom-up evaluation method, which are also the methods we considered applying to our generator:

2.3.1. Expressive Range (top-down). Expressive range refers to the space of potential levels that the generator is able to create [4]. Often, a large amount of content is generated, evaluated according the metrics, and then plotted in a heatmap. Heatmaps easily help to identify biases in the generator. Comparisons of heatmaps also show how different sets of input parameters affect the controllability of the generator.
The expressive range of a generator could be represented as an N-dimensional space, where each dimension is a metric defined for evaluation. The metrics for a generator are vary based on the game domain. For example, previous research on a 2D platforming game *Infinite Mario Bros.* utilized metrics as ”linearity” and ”leniency”, but the same metrics might not suit other games (e.g. *Relinquish*).

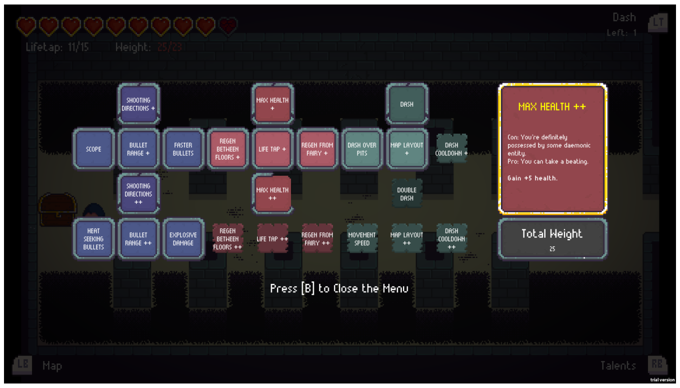
2.3.2. Self-report and questionnaire (bottom-up). One of the possible approaches to evaluate game content is to gather self-report from players. Play testers can be either a small but dedicate group, or regular users recruited from crowdsourcing platforms. One of the advantages of self-report is directness. Play testers could give direct feedback on all aspects of the game. Questionnaire, on the other hand, can be considered as a simplified way of self-reporting with a ranking structure.
In the current project, we collected qualitative feedback and results from Game Experience Questionnaire (GEQ) [5], which is a commonly used tools when it comes to game evaluation.

3. Game Mechanics

3.1. Overview of the Game - Relinquish

Relinquish is a 2D roguelike with twin-stick shooter gameplay [1]. A mysterious elevator is your only path to the surface. To use the elevator, the weights carried by the player must not exceed a certain value per floor. The core gameplay of Relinquish centres around an inverse progression system, where the player is forced to give up various perks - represented as weights - to progress through the levels of the game (see Fig. 1). To give up perks, the player must explore the level to find the various chests scattered throughout. Once the player has given up enough perks, they must return to the elevator which takes them to the next level. The primary goal of the game is to complete the final level, which can only be reached once the player has given up all perks.

The game implements permanent death. Various enemies and environmental obstacles block the player’s path, some of which are only surmountable if the player has not yet given up certain perks. The primary example of this is the ”pits” obstacles that the player cannot traverse unless they possess a specific perk (see Fig. 2).



Con: You're definitely possessed by some daemonic entity.
Pro: You can take a beating.

Gain +5 health.

Figure 1. Perk Graph in *Relinquish*. When the game starts, the player possesses all the available perks in the game. In the figure, some perks are already dropped.

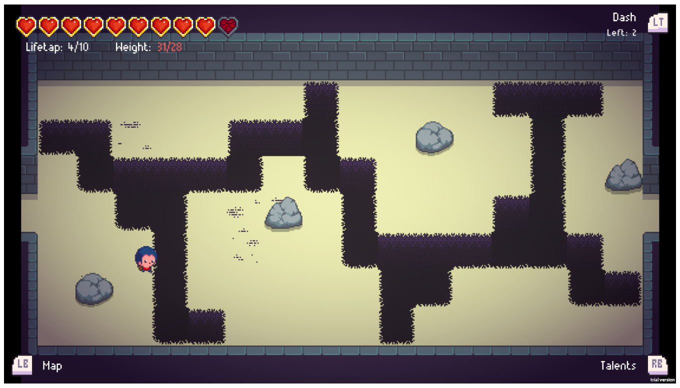


Figure 2. A room with pits. Pits are the main environmental obstacle in *Relinquish*.

3.2. Layout of the Floors

Each floor in *Relinquish* is made of a set of rooms that are placed on a grid and are directly connected to each other. Each room is individually saved as a prefab. A room prefab contains information about the placement of enemies, the tilemap, and the possible exits. Level designers can place random enemies inside a room by placing appropriate game objects with a script that converts them into actual enemies at runtime by randomly selecting an enemy from a pool of pre-defined prefabs.

There are different types of rooms: *common*, *elevator*, *boss*, *treasure*, and *fairy* rooms. The last four rooms will be referred as *special rooms* in this article. As a design requirement, each floor is made of a total of 15 rooms and there needs to be exactly one of each special room. Special rooms should be distanced from each other. Also, except for the *elevator* room, they should ideally be connected to only one other room. A summary of these requirements can be seen in Fig. 3.

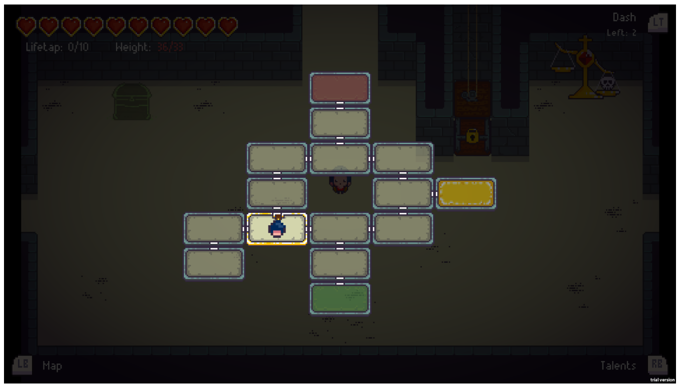


Figure 3. Example of an ideal floor layout in *Relinquish*. On the map, the boss room is in red, the treasure room is in yellow, and the fairy room is in green.

4. Methods

In this section, we describe our implementation of two Agent-based techniques [2] for generating floor layouts in *Relinquish*. The first algorithm performs a multi-agent 2D random walk. The second algorithm is an adaptation of Prim's maze generation [6] that was inspired and adapted from the procedural content generation in Eldritch [7]. The two algorithms both perform the following three high-level steps until a valid floor is generated or the max amount of attempts is reached:

1) A floor plan that specifies the positions and the connections of the placed rooms is generated. For each room position, some requirements are specified.
2) For each room position, a random room is selected by querying the room system with the requirements specified in the previous step.
3) The floor is validated against a set of floor rules. Each floor rule is encoded into a ScriptableObject of type *FloorRule* and is injected into the algorithm through the inspector. If any of the floor rules are not respected, the floor is discarded and the algorithm performs another attempt starting from step 1. An example of a floor rule is a rule that ensures the existence of a path between the elevator room and each other special room through non-optional rooms.

Both algorithms rely on the newly developed dynamic room system for the second step. *Relinquish* already had a basic room system, but we extended it to allow the procedural generation algorithms to select rooms from an internal room database by using simple queries.

4.1. Dynamic Room System

Floors in *Relinquish* used to be hand-crafted by the level designers by nesting the room prefabs inside of a floor prefab. This meant that the designers were the ones making sure that the rooms were properly connected and that the floors respected the requirements. In order to enable the procedural content generation algorithms to select rooms with desirable properties, we extended the previous room system by implementing a room database that can be queried to retrieve a list of rooms with the desired features. The code for the dynamic room system is mainly stored in *RoomListUtility.cs*. The list of rooms is saved as a table in *Assets/Resources/rooms.txt* and it is updated every time a room prefab is created, renamed, edited or deleted. Each line represents a room. Each column represents one of the following parameters:

1) **Path**: the path of the room prefab in the file system (All rooms are placed in *"Assets/Resources/Rooms/"* and its sub-folders).
2) **Door Configuration Regex**: the regular expression that encodes all of the possible configurations of the doors. For more information, refer to the Door System subsection.
3) **Type**: the type of the room (Common, boss, elevator, treasure or fairy).
4) **Is Optional**: if a room is flagged as optional, it means that there might be circumstances that prevent the player from crossing it. For example, a room with pits cannot be traversed without a specific perk and this might lead the player to get stuck on a non-compleatable floor. The procedural generation algorithms need this information to generate valid floors.

The room system can be queried with the following two utility methods:

1) **GetRoomTemplateInfos (requirements)**: returns the list of all the rooms in the database that satisfy the given requirements.
2) **GetRandomRoom (requirements)**: returns a random room from the list of all the rooms in the database that satisfy the given requirements.

The requirements are defined by a struct (*RoomTemplateRequirement*) with 4 different fields. A field is considered by the above mentioned methods only if a value is assigned to it. The four fields are:

1) **ignoreTheseRoomPaths**: a set of strings containing the paths of rooms that should not be selected (e.g.: to avoid duplicate rooms on the same floor).
2) **doorConfig**: the string that encodes the required configuration of the doors. For more information, refer to the Door System subsection.
3) **type**: the type of the room
4) **onlyNonOptional**: if true, filter out optional rooms.

4.1.1. Door System. This section will clarify what is meant by *"door"* and how doors are handled by the room system. In *Relinquish*, a door defines whether a room is supposed to be connected to another room along a direction (open door) or not (closed door). Every room has 4 doors, one for each cardinal direction. In each room prefab, each door can be in one of the following states:

1) **Always Open**: the room always needs to be connected to another room through the door.
2) **Always Closed**: the room cannot be connected to another room through the door (there is always a wall).
3) **Optional**: the door can either be open or closed, depending on the requirements defined by the procedural generation algorithm. If a room with an optional door is selected by the PCG algorithm, the door state is set to *open* or *closed* accordingly. This allowed us to increase the expressivity of the generators without the need to create multiple versions of the same room for every possible door configuration.

For each room, the possible configurations of the doors are encoded into the room list as a regular expression with a length of 4 characters. Each character represents a door state: 'o' stands for *always open*, 'x' stands for *always closed*, and '.' stands for *optional*. Each character index represents a cardinal direction: north (0), east (1), south (2), and west (3). See Figure 4 for an example.

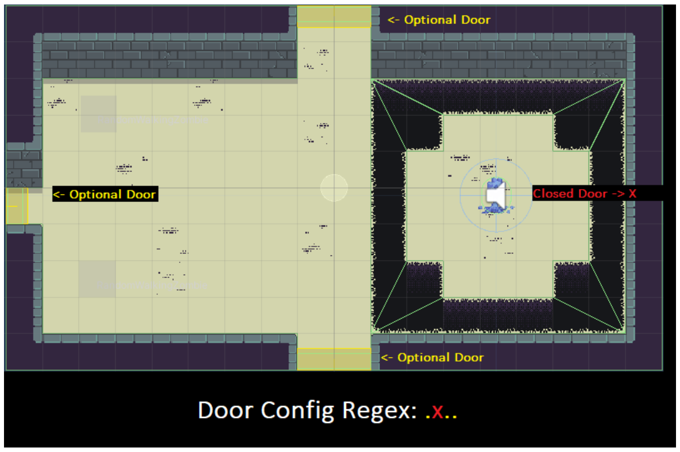


Figure 4. Example of a room prefab and its door configuration regex. The east door is always closed. The north, south, and west doors are optional. Therefor, the door configuration regex is ".x..".

When querying the room system, the rooms whose non-optional doors match the required configuration of the doors are returned. The possible configurations of the doors of a room are encoded with a regular expression because the problem of finding rooms whose possible configurations match the given door's configuration can be translated into the problem of finding the rooms whose regular expressions match the string that encodes the required door's configuration. The special character '.' was strategically chosen to represent optional doors because it matches any single character. A visual summary of the door system and examples of regex matching can be seen in Fig. 5.

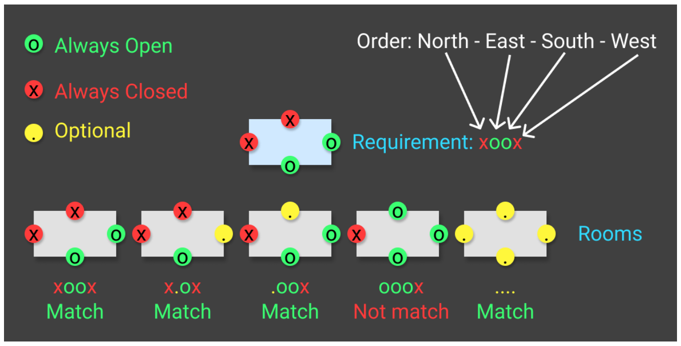


Figure 5. Visual summary of the door system. Given a string representing the required door configuration, the rooms whose regular expression don't match the input string are filtered out.

4.2. Random Walks Algorithm

Our main PCG approach for generating the floor plan of a level uses an agent-based approach. It is implemented in *AgentBased1FloorGenerator.cs*. There are two phases of floor plan positions generation. During the first phase, all of the positions for the common rooms are decided. This is done by having an agent do two-dimensional random walks as described by Brian Bucklew [8]. This means that an agent starts at a specific location and then steps in a random direction each cycle. However, in order to determine if a potential room is in a valid position from the random walk, we use a rule-based system to validate positions. If the agent fails to decide a position for a room after a set amount of attempts, they will be transferred to another random room position and try again. However, if this fails another set amount of times then the algorithm will give up and try again from scratch. This is done in order to not get stuck in a situation where there are no possible positions where a room can be placed. The agent will keep finding positions until they have decided positions for the number of rooms specified by the designer. After the agent has decided positions for all the rooms, the next phase of the algorithm starts. During this phase, the positions of the special rooms are decided. The position for a special room is decided by setting one of the positions of a common room to be the position of the special room instead. The method to figure out the position of a special room is done in two steps.

1) First, each room position is validated to check if it could potentially be the position for the special room. This is done using the same rule system as for the common room positions, but using a different rule set specific for this special room.
2) After finding all of the potential rooms positions, each position is given an evaluation score using an evaluation system. The position with the highest score is then chosen for the special room.

In case that the algorithm fails at finding a position for a special room, it will start over from scratch again.

The algorithm also allows for having multiple agents acting at the same time. This allows for the algorithm to expand in multiple different directions at once in order to increase the diversity of the resulting floor layout, therefor increasing the expressivity of the generator. Furthermore, the algorithm has a feature which allows the designer to set a chance to force the algorithm to start with a ”big loop” as the one seen in Fig. 3.

4.2.1. Room Rules. The rule-based system consists of a set of room rules that a designer can choose to add to the floor-generator. Each of these rules are then checked whenever the agent tries to place a room using the random walk. There exist rules for each type of room. This way a designer can more easily control how each different type of room is placed in the floor.

A room rule is a scriptable object of type *RoomRule* which requires a single function: Validate. The validate function has the responsibility of determining if the rule is followed or not. It takes a position and a dictionary over already decided positions for other rooms.

An example of a rule could be a rule that dictates that there must not be more than e.g. 6 rooms in a 3x3 square around the room. This rule would make it so there would not be as many rooms clustered together.

4.2.2. Room Evaluations. The evaluations system, much like the rule-based system, consists of a set of evaluations that a designer can choose to add to the floor-generator. Each of these evaluations are then run when the generator tries to determine the position of the associated special room.

A room evaluation is a scriptable object of type *RoomEvaluation* which requires a function: Evaluate. The evaluate function has the responsibility of evaluating how good a given position would be for the special room. It takes a position, a dictionary over already decided positions for other rooms, and a dictionary over connections. Furthermore, an evaluation also has a weight that designers can use to tune the importance of the evaluation itself.

An example of an evaluation could be an evaluation which uses the distance to the *elevator* room. This can be used to place e.g. the boss room as far away from the *elevator* room as possible.

4.3. Prim’s Maze Algorithm

Our secondary PCG approach for generating the plan of a floor is implemented in *PrimMazeFloorGenerator.cs*. It is based on a furtherly adapted version of the Prim’s Maze-based [6] algorithm used for generating levels in *Eldritch* [7]. The standard Prim’s Maze algorithm can be summarized into the following three steps:

1) Start from a single visited cell.
2) Randomly open a path to any adjacent unvisited cell.
3) Repeat until all cells are visited.

In *Eldritch*, the algorithm is slightly modified in the following way:

1) The expansion is biased from the most recent room position.
2) Walls are randomly knocked down to introduce cycles.

Like the Random Walk approach, the Prim-based algorithm consists of two phases: first all of the positions for the common rooms are decided using the above mentioned techniques. Then, after they are decided, some of the common rooms are replaced with special rooms using rules and evaluations in the same way as the Random Walk Algorithm.

4.4. Tests with Players and Designers

To test the reliability and believability of our procedural generation solution, we planned supervised tests with players. We prepared 3 different builds and equally assigned them to testers:

1) **Version A**: No procedural generation (hand-crafted levels from the previous prototype).
2) **Version B**: Random Walk algorithm.
3) **Version C**: Prim’s Maze-based algorithm.

During the playtests, we recorded the qualitative feedback to the game. We mostly focused on aspects related to the layout of the floors, the rooms, and the placement of the enemies. At the end of tests, the players were asked to answer questions from GEQ (See section: Game Experience Questionnaire (GEQ)).

To test the controllability of our algorithms, we also planned tests with the original designers of the game. Specifically, we were interested to see whether they could easily achieve the results they wanted by tweaking the parameters that are exposed in the Unity inspector. We were also very interested to see if they appreciated the possibility to add and remove room rules, room evaluation, and floor rules directly in the inspector without the need to know how to code. Lastly, we wanted to hear more about their needs in order to identify possible extensions to our solution.

We considered to evaluate the expressive range [4] of the generators, but we decided not to do it for the following reasons:

1) We did not know how to properly determine relevant metrics for describing floors
2) We lacked time to implement the actual code to analyze the expressive range.

Lastly, we are interested to see whether the algorithms we implemented manage to generate a new floor within the expected time (the duration of the animation of using the elevator to move to the next floor).

4.5. Game Experience Questionnaire (GEQ)

The Game Experience Questionnaire (GEQ) [5], [9] is a questionnaire for evaluating the gaming experiences of players. GEQ is a 5-point scale and consists of three different modules, including the core module, the social presence module, and the post-game module. The core module assesses players' game experience in seven dimensions: Competence, Sensory and imaginative immersion, Flow, Tension/ annoyance, Challenge, Negative affect, and Positive affect. The social presence module tends to look into psychological and behavioral involvement with other social entities: in-game characters, other players online, or co-located. The post-game module evaluates how players felt when they stopped playing. For the current study, we applied only the core module of the GEQ, which contains 33 questions. Besides the core module, we decided to add six extra questions regarding the game map of *Relinquish*. The questions are the following:

1) The game maps are repetitive
2) The length of each level is just right for me
3) The map length of each level is unbearable
4) I keep repeating the same path while playing
5) The game maps contain a high variety
6) In general, I think the game maps are nice and interesting

Question 1, 3, and 4 are reversed questions. Therefore the answer have been reversed when encoding. At the beginning of the questionnaire, players were asked which version they played.

5. Results

5.1. Game Experience Questionnaire (GEQ) results

We distributed GEQ to 19 playtesters and gathered their feedback. All players were tested on the core module. On top of the seven dimensions of the GEQ, we decided to add an extra "map" dimension, which allowed users to evaluate the map layout of *Relinquish* (for more details, see Methods). We compared the eight dimensions across the three game versions. The results are shown in Fig. 6.

We found that human-designed levels (version A) yielded stable and in general high score. It also got the best scores in the sensory and challenge dimensions. The random walks algorithm (version B), on the contrary, yielded inconsistent and in counter-intuitive results. It scored the best on flow, tension/ annoyance, meanwhile the highest in negative affect dimension. However, the "flow" experience refers to the positive mental state in which one is fully immersed in the activity one performs, related to a feeling of focus, pleasure, and full involvement [10]. The Prim's maze algorithm (version C) yielded scores in low challenge, flow, tension, and negative aspects. Further, Prim's maze algorithm yielded the worst results in six among all eight dimensions.

5.2. Qualitative Feedback from Playtesters

In general, players did not express strong feelings regarding the floor layout. Some testers with version A (handcrafted floors) assumed that the floors were procedurally

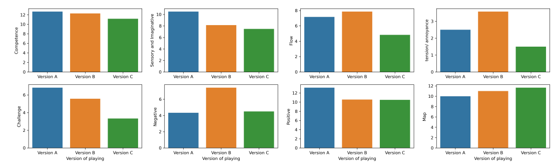


Figure 6. This graph illustrates comparisons among the three game versions and eight evaluating dimensions. **Version A**: human-designed levels; **version B**: random walk algorithm; **version C**: Prism's Maze algorithm

generated while some people with version B and C said that the floor layouts looked believable. In general, it appears that the layout of the floor does not affect the gameplay experience too much. Some people with version C (the Maze algorithm) complained about how compact the floor layout is by saying that it is harder to read on the map, it's harder to navigate, and it's less interesting. In general, people complained about backtracking and showed appreciation for maps with loops and many small branches. There were a lot of negative comments about the lack of variety in the content of the game: there are only 3 types of enemies and identical rooms were often spawned multiple times on the same floor. Players usually noticed that enemies were spawned in fixed positions and some would have preferred a more dynamic system. Players noticed that, unlike version A, in version B and C the boss rooms are not introduced by a corridor. We did not notice that this aspect of the floor design was a requirement and therefor we did not consider it while designing the PCG algorithms. Players with version B and C noticed that the special rooms were placed in terminal positions and appreciated the choice by saying that their placement looked natural. The multi-agent random walk algorithm (version B) is the one that received more praise both in terms of quality and believability.

5.3. Qualitative feedback from designers

The designers expressed that they were generally satisfied with the layout of the floor generated by the 'Random Walks Algorithm'. Furthermore, they appreciated the possibility to be able to tweak parameters in the inspector. However, they would like to be able to tweak the behavior of the agent itself. E.g. being able to give the agent a higher bias for making turns instead of going straight. They also mentioned that they would like to be able to set animation curves, based on carried weight, for the parameters instead of static values.

The designers found the Rule and Evaluation systems to be intuitive and appreciated the ability to drag and drop them without the need to code. However, they would like to have more agency over which specific rooms are placed next to other specific rooms. E.g. having a boss room always be anticipated by a corridor.

For the Room System, they appreciated the automatic update of the room database by simply adding or editing the room prefabs. They would, however, like to be able to specify a weight to be used when selecting a random room in order to make specific rooms rarer than others.

Lastly, they also commented on the lack of a check to prevent the same room from being used multiple times on the same floor. This, however, is caused by the algorithm not properly utilizing the functionality of the room system since there are not enough different rooms created by the designers yet.

6. Discussion

The insights from the tests with the players and the designers provided us a lot of feedback to better choose the best candidate among the two proposed PCG solutions and to design future extensions.
However, after analysing the results of the tests, we noticed that there were inconsistencies between the questionnaire and the qualitative feedback provided by the play testers. We assumed that the lack of data could be the main reason for the inconsistencies. Besides that, one previous research which reviewed 73 publications about GEQ, suggests that the reliability (Cronbach's alpha) of different dimensions within GEQ contain high variations [11], making it is difficult to determine which scores are robust. Therefore, we decided to focus more on the qualitative feedback rather than the questionnaire.

The comments about the believability of the floor layouts did not change much from version to version. Some players with Version A assumed the content was procedurally generated even if it was not. The same situation happened to Version B and Version C as well. We thus conclude that the believability of the floors is not particularly worse in Version B and C, compared to Version A. This could indicate that our algorithms improve the replayability factor of the game without compromising the quality of the floors too much.
We are satisfied by the speed of both algorithms. They both generate a new floor within the natural duration of the elevator animation.

The Random Walk approach has been particularly appreciated by the testers for its more frequent loops and small branches which made the map more interesting while reducing the amount of backtracking. The Prim's Maze-based algorithm has a tendency to generate overly compact maps with too many walls between the rooms, which could make the navigation of the floor uncomfortable for the player. The initial idea behind the approach was that maze-like structures would have been perceived as interesting for their branching factor, but the playtests confuted the hypothesis. Initially, we considered to keep both algorithms and to randomly choose between them at run-time, but since Random Walk has been appreciated much more than Prim's, we decided to move forward with the former approach over the latter.

The feedback provided by the game designers inspired possible future aspects of our PCG solution:

1) The ability to have more control over the behaviour of the agents by exposing more parameters (e.g. the probability of turning during a random walk).
2) The expressivity of the generator could be improved by dynamically adjusting the parameters with an animation curve based on player progression (e.g. amount of dropped perks).
3) The believability of the algorithm would improve by implementing the check for duplicate rooms.
4) Each hard-coded generation step could be converted into an injectable step encoded by a ScriptableObject. This would make it more modular, more designer-friendly, and more extensible (similar solution to the rules and evaluations).

The room system could also be upgraded by:

1) Introducing a weight to each room to specify its rarity.
2) Developing a dynamic enemy spawning system.

Having a dynamic enemy spawning system would improve the expressivity of the generator, but would not fix one of the main design problems of *Relinquish*: the lack of content. In order to address this issue, more enemies, and rooms need to be created in collaboration with the designers.

References

[1] J. H. Eiholt, A. C. Elsberg, J. S. Faber, F. Frassineti, and M. Wahlers, "Unlike roguelike: Inverting traditional progression systems in the context of the rogue-like genre.," 2020.

[2] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural content generation in games*. Springer, 2016.

[3] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, pp. 1–22, 2013.

[4] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pp. 1–7, 2010.

[5] K. Poels, Y. A. de Kort, and W. A. IJsselsteijn, "D3. 3: Game experience questionnaire: development of a self-report measure to assess the psychological impact of digital games," 2007.

[6] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[7] D. Pittman, "Level design in a day: Procedural level design in eldritch." https://www.gdcvault.com/play/1022110/Level-Design-in-a-Day. Accessed: 2020-12-12.

[8] T. X. Short and T. Adams, *Procedural Generation in Game Design*. USA: A. K. Peters, Ltd., 1st ed., 2017.

[9] W. A. IJsselsteijn, Y. A. de Kort, and K. Poels, "The game experience questionnaire," *Eindhoven: Technische Universiteit Eindhoven*, pp. 3–9, 2013.

[10] M. Csikszentmihalyi, "Beyond boredom and anxiety: The experience of play in work and leisure," 1975.

[11] E. L.-C. Law, F. Brühlmann, and E. D. Mekler, "Systematic review and validation of the game experience questionnaire (geq)-implications for citation and reporting practice," in *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play*, pp. 257–270, 2018.