A fast approach for automatic generation of populated maps with seed and difficulty control

Pedro Sampaio Augusto Baffa* Bruno Feijó Mauricio Lana

PUC-Rio, Departamento de Informática, ICAD/VisionLab, Brazil

ABSTRACT

Procedural Content Generation (PCG) is the programmatic generation of game content using a random or pseudo-random process that produces an unpredictable stretch of gaming spaces. Many games use this technique to increase the length of gaming, and some of them rely greatly on the procedural content generation. This paper aims to detail a fast and direct PCG approach capable of generating a diverse array of dungeon-like maps, developed to be used in an experimental game created by undergraduate students, that has shown very good results. The approach in question is capable of generating numerous unique maps while maintaining control over the generated levels through seeds. It also has the ability of designating different difficulties to the generated content, and uses this very feature to increase the variety of the procedural generation by taking advantage of the way seeds work.

Keywords: procedural generation for games, dungeon generation, game content generation, generated content difficulty scaling.

1 INTRODUCTION

The level design is a core element of a game. Dedicated designers are in charge of designing game levels through an extensive process of creation that undergoes a number of stages (i.e art conception, draft, rendering) and depends highly on the creativity of the designers. This process can lead to unique high quality game level design but in exchange it can borrow a lot of development time resources. Procedural Content Generation (PCG) steers to the opposite direction. As expressed in [22], PCG can be defined as the algorithmic creation of game content with limited or indirect user input. PCG algorithms can create a vast quantity of levels with little effort if compared to designing them manually one by one. This potentially reduces production and development costs of a game.

Although it may seem that procedural content generation is a recent trend in game development, Smith [19] affirms that the first uses of the PCP concept for digital games is dated to 1980s, and it has been used even before that if we consider analog games where, instead of a computer, a person is the agent of the procedural generation. Rogue [23] is one the earliest notorious game with procedural generation and there are numerous follow-ups of its style being published until the present day. Another example of game that uses procedural content generation is No Man's Sky [9], a game that achieved a lot of attention solely on the fact that its procedural generation algorithm was capable of creating eighteen quintillion planets, each one with its own flora and fauna.

Procedural content generation has received increasing attention in commercial games. Alongside with No Man's Sky [9], numerous other popular commercial games features procedural generation. Diablo [7] is an action role-playing hack-and-slash digital game featuring procedural generation for creating the maps and entities of the game, and has a large fan base since the first installment of the series released in 1996. Some other examples are Civilization IV [8], a turn-based strategy game that allows unique game experience by generating maps, and Spore [13], that lets people design their own creatures that are automatically animated using procedural animation techniques. As another example, Minecraft [14], one of the most popular games currently, features extensive use of PCG techniques to generate the game world and its content.

Procedural Dungeon Generation (hereafter PDG) is a type of PCG that explores particular aspects of Roguelike games, which are adventure or role-playing games (RPGs) characterized by a dungeon crawl in a labyrinthine environment, full of rooms, caves, and corridors. Also, Roguelike games are characterized by procedurally generated game levels, turn-based gameplay and tile-based graphics. PDG algorithms generate a 2D map composed of rectangular rooms connected by narrow corridors on a grid of vertices (x,y), where x and y are integers. Nowadays, PDG algorithms should generate more than maps of rooms and corridors they should have control of the gameplay. However, few games have PDG with gameplay-based control, where PDG parameters are related to gameplay data [24]. A general description of PDG algorithms can be found in [24] and [2].

In our work, we are interested in the simplest type of PDG algorithm, named Random Room Placement, which is a brute force room-generating algorithm [2]. The reason for our interest is because we claim that this is the most efficient way to have gameplay-based control. In this paper, we present a method to generate rooms, create corridors, populate the levels with actors (characters, powerups, and tokens), recover any previous level map, and control difficulty level based on two parameters only: the seed value and the level number. The Random Room Placement algorithm and the properties of the seed value (in pseudo-random processes) has been extensively used by the industry [15], but as far as we are aware no other work has the same level of control in real time as the one provided by our method.

2 RELATED WORKS

In order to achieve a better understanding about procedural content generation and how it can be used in gaming, several papers were consulted while we were designing our algorithm. Initially, the very own concept of what is and what is not procedural content generation in games was studied. The authors of [22] indicate the difficulties of coming up with a unique PCG definition that everybody agrees on, but it does a good job clarifying its concepts through contrasting it to other forms of content generation in games with which it could easily be mistaken. In [10], we are presented with a taxonomy of game content and PCG techniques for games that helps the comprehension of how PCG can be used in different areas of game content. It also surveys the state-of-the-art in PCG techniques for games and the use of these techniques in real games.

While resolving the type of game level to be generated by our algorithm, we stumbled upon a paper that showcased a specific type of game level as one of the most suited for an PCG approach. In [24], dungeon level type is highlighted as being well capable of demonstrating the benefits of PCG. Dungeons basically consist of several rooms connected by corridors, but, in games, it may also refer to caves, caverns, or human-made structures. The concept of

*e-mail: abaffa@inf.puc-rio.br

rooms connected by corridors matched our intentions for the map design of our experimental game. In the same paper, a number of dungeon generation methods is described while pros and cons of each approach is discussed. In particular, the method that uses cellular automata [11], a model studied by Stephen Wolfram since the 1980s [25], stood out with results that were visually pleasant and had a natural and chaotic feel to the generated map, but, as stated in [24], there is a lack in the direct control of the generated map. Additionally, connectivity between any two generated rooms is not guaranteed by the method.

An in-depth analysis on dungeons and procedural generation is done in [5], where patterns for procedurally generating dungeons are presented, along with a classification for the different types of dungeons encountered in released games.

An interesting approach to procedural content generation was detailed in [17], where answer set programming (ASP), a form of declarative programming oriented towards difficult search problems, explained in [12], was used for PCG in games. But, as stated in the paper, a naive implementation of this approach can be very costly in terms of time, specially when dealing with large problem spaces.

A hybrid solution is proposed in [21], where a combination of different PCG methods is presented. The paper goal is to study an approach that retain the advantages and avoid the disadvantages of the combined methods.

When it comes to the level population, where entities of the level are the object of the PCG, The authors of [1] suggested the use of evolutionary algorithms, described in [20], to be able to position generated content on interesting locations based on the game universe, a trait that is indeed very important in game content generation, and necessary to match the design of our experimental game. The method described in [1] gives a good control over its generated content, but the positioning based on the fitness functions specified in the paper could lead to poor results, as stated in the paper.

The writers of [6] suggest to consider content generation as a duel process, separating it in a generation step to create variety and a resolution step to make the output generated into a coherent and useful configuration. It concludes that breaking down the content generation algorithms into these steps facilitates the design of PCG algorithms. Our approach has leaned to a similar division, where game spaces are generated at first, and then the algorithm proceeds to make it coherent to the experimental game universe.

A very interesting outlook on PCG future potential in gaming is expressed in [18]. The paper proposes several new research directions for PCG that require both deep technical research and innovative game design, exploring the future of PCG from five different perspectives: data vs. process intensiveness, the interactive extent of the content, who has control over the generator, how many players interact with it, and the aesthetic purpose for PCG being used in the game. It inspires important reflections for future researches on PCG uses on games.

Finally, Procedural Content Generation in Games: A Textbook and an Overview of Current Research [16] brings a good overview of PCG in games, and it is the first textbook about the subject, as stated by the authors. It covers a big range of contents related to PCG and is recommended for a in-depth study.

Having the PCG literature as background, we designed a direct approach that generates a large variety of singular maps with connectivity of rooms guaranteed, where we aimed to achieve similar results of existent methods maintaining a fast and direct approach to the generation. As opposed to the literature previous cited in this section, our approach includes the use of seeds combined with the play difficult of a level as a way of gaining control over the generated content without being restricted by the seed numerical limitation. This feature is better detailed later in this paper.

Figure 1: Experimental game to use our PCG implementation

3 Proposed Model

In this section, the modelling of our algorithm is detailed. In order to better elucidate the modelling, the experimental game characteristics that influenced the design of the algorithm are also specified, as well as their impacts on it.

3.1 Experimental Game

Figure 1 shows the experimental game developed, a 2D tile-based RPG game inspired in the popular MMORPG Tibia [4], and the subject of the implementation of our algorithm. For starters, there are some important points to cover about the game before we proceed to describe our map generation approach. Some key features of the game were responsible for altering the modelling of our algorithm, so it is important to clarify the mechanics of a level in our game. A level in the experimental game begins with our main character, Fabian, being spawned at the start location of the map. The map is populated by different entities ranging from collectibles to enemies. Tokens are one of the collectible types and is presented in three different grades: gold, silver and bronze. These grades represents the value and the difficult of obtaining the token, and there are only one of each per level. Fabian must find at least one token of any grade and take it to the level Totem, an entity that represents the final location of the map. That's all the information regarding the game necessary to understand the procedural generation implemented.

3.2 Our PCG Approach

Now it is possible to describe how the PCG for the experimental game was modelled, taking into account the level mechanics described above. To simplify the description, it is arranged into two categories: Level Skeleton, associated with the physical structure of the level, and Level Population, associated with the entity population of the level. Also, the control of the generation through seeds and the play difficulty is better detailed to clarify one of the key mechanics of the algorithm.

3.2.1 Generation Control

Seeds are integer values that are attached as a parameter of the generation. For a specific algorithm configuration, a seed will guarantee the same output for the generation always. With this feature, we were able to control the level generation with the assurance that a seed will give the same generated map in every execution of the algorithm in a given configuration. Although using seeds enabled the control of generated levels, it also limited the quantity of distinct levels generated by the algorithm. Since seeds are often stored as integers and each level generated is attached to a seed, the integer limitation becomes the limit of levels generated. To increase this limit, we adopted a strategy that took advantage of the way that

seeds work. A seed will return the same output always only for a particular configuration of the algorithm. If the configuration is to be changed, the result of the generation will no longer be the same. We combined that notion with the designation of play difficulty of generated content. A factor that scales the difficulty of the generated level was used to scale the map size and other elements of the experimental game universe i.e enemy strength, giving a particular difficulty to each generated map. The trick is that, by altering this factor, the algorithm configuration changes, and, consequently, the generation output is not the same anymore. This means that a seed does not need to be necessarily attached to only one generated map. Instead, a seed now have different generations according to the difficulty factor, increasing vastly the number of possible levels to be generated by the algorithm considering that even minor variation on the difficult factor will result in a distinct generation.

3.2.2 Level Skeleton

A dungeon structure was preferred to be the model of the game's physical level structure, which consists of rooms interconnected by corridors. The process of creating the level structure is fairly simple: rooms are created with arbitrary sizes and are randomly placed throughout the level space allowing superposition of different rooms, and then randomly shaped corridors are generated to connect random locations of two different rooms until all rooms are connected. The corridors shape is created by randomly alternating directions on the path to the location to be connected. It is important to point out that the entrances are generated through the corridors generation when a room boundary is reached. Superposition of rooms has shown itself to be beneficial to the level singularity. By enabling the superposition of rooms, different and singular room shapes were allowed to be generated resulting in a larger array of possibilities for our generation.

3.2.3 Level Population

After creating the level structure, it is necessary to populate the level with all map entities according to the description of the game. Fabian, enemies, the level Totem, tokens and other collectibles must be placed in the generated level structure. To generate a playable level that obeys to the game specifications, there are some particularities to pay attention. The major particularity to consider is that tokens have different values and the difficult of obtaining them varies. With that in mind, the positioning of tokens of different grades must have at least a little reasoning. While others entities would surely benefit from an intelligence placement on the level, there is no game reason to do it, and with the goal of keeping it simply, they are simply randomly placed along the free locations of the level. With the same modus operandi, Fabian and the Totem are also placed in the level. Tokens are then placed in the map accordingly to the difficulty of getting them. The difficulty is measured acknowledging the number and the types of entities around each field candidate to contain a token as well as the distance of each field between the starting and final location.

4 Proposed Solution

This section will explain the practical implementation of the model proposed and detail its steps to shed a light on the operations and procedures that were taken to obtain the results exposed in this paper.

4.1 Pseudocode

The psesudocode 1 presents the algorithm structure and course exposing each procedure executed to generate the map in the order that they are conducted. Furthermore, each component of the pseudocode 1 is detailed to facilitate the understanding of the algorithm implemented.

Algorithm 1 A fast approach for automatic generation of populated maps with seed and difficulty control

Require: float : difficulty
Require: integer : seed
Ensure: generated map (grid)
1: SetGenerationSeed(seed)
2: SetGenerationDifficulty(difficulty)
3: *grid* ← GridGenerator()
4: *roomVector* ← RoomVectorGenerator(grid)
5: RoomPlacer(grid, roomVector)
6: MakeAllRoomAvaliable(grid, roomVector)
7: *grounds* ← RetrieveGrounds(grid)
8: WallGenerator(grid, grounds)
9: MapPopulator(grid, grounds)
10: SpawnPointGenerator(roomVector, grid)
11: *tokenGrounds* ← CalcGroundCost(grid, grounds, squareSize)
12: TokenPopulator(grid, tokenGrounds)
13: **return** $grid$

The following definitions aim to detail the map generation through each step that was taken. All functions, structures and data are detailed one by one in spite of better explaining how the generation works.

4.1.1 Seed

Seed is an integer value that generates a unique map. This value represents the initial state of all instances of Random method calls in the generation. Once defined, a seed generates the same map always, and different seeds generates different maps.

4.1.2 Difficulty

Difficulty is a float value that defines the difficulty factor for the map generation. All further steps of the generation acknowledge this value as the current map generation difficulty and uses it as a scale factor for the content generated when necessary.

4.1.3 Room

Room is a structure that represents a room in the generated map. It contains the x and y coordinates, and also the height and width of the room. It also contains a flag that informs the accessibility of the room generated.

4.1.4 SetGenerationSeed(seed)

SetGenerationSeed is a function that sets seed for the map generation by attaching it to the Random method initialization. It is important to point out that the current implementation uses C# Random methods that offers seed support.

4.1.5 SetGenerationDifficulty(difficulty)

SetGenerationDifficulty is a function that sets the difficulty factor for the map generation, altering all necessary parameters that are affected by the difficulty factor, including grid maximum and minimum size, rooms quantity and dimensions, enemy quantity and strength, and other entities quantities.

4.1.6 GetRandomNumber(min, max)

GetRandomNumber is a function that returns a pseudo aleatory integer between min (inclusive) and max (exclusive), based on the seed that was previously set. All values to be randomized will be randomized by this function.

4.1.7 GridGenerator()

GridGenerator is a function that returns a grid (bi-dimensional array) populated with blank values (non-defined fields), of random size that is influenced by the current difficulty factor. This grid represents the data structure of the world, and all generated map will be embodied in this grid.

4.1.8 RoomVectorGenerator(grid)

RoomVectorGenerator is a function that returns an array containing all the rooms (as defined by the structure Room) to be generated in the map. The number of rooms and each room's attributes (x, y, width, height) are random defined through the function GetRandomNumber(min, max), where the random generated room must be within the grid (parameter of this function) boundaries. The intersection of rooms is appropriate, since it gives the generated map more uniqueness. In the current state, all generated rooms are square shaped. As stated before, parameters of this generation are influenced by the difficulty factor.

4.1.9 RoomPlacer(grid, roomVector)

RoomPlacer is a function that populates the grid with the generated rooms. As of now, all room fields are considered grounds, so the function should fill the grid with a symbol that represents ground fields. Walls are going to be generated later. This function simply iterates through the array of rooms and fills the grid with grounds in the area that represents each room. Room intersections may generate different shapes other than simple square, increasing the map distinctiveness.

4.1.10 RoadGenerator(initialPoint, finalPoint, grid)

RoadGenerator is a function that makes a road between two points (x,y) in the grid. This function connects two points in the grid by making random grounds between the two points received in the parameter. In each step, this function randomly chooses a coordinate (x or y) to increase or decrease its value (depending on the direction between initial and final point). When a coordinate reaches its final value, only the other coordinate is considered, tracing now a straight line to the final point. Thus, the road always reaches the final point, but it still makes each generated map more unique when used to connect rooms, despite being of very naive implementation.

4.1.11 MakeAllRoomsAvailable(grid, roomVector)

MakeAllRoomsAvailable is a function that makes all rooms in the roomVector array accessible. It prioritizes the singularity of the map by randomly choosing the rooms to make accessible. The way it works is that one room is randomly defined as the first accessible room, and another room (inaccessible) is chosen to be connected to the first accessible room, making both rooms connected and accessible. After that, it randomly gets one accessible room to connect with the next inaccessible room, and it repeats this procedure until there are no more inaccessible rooms in the roomVector. The connection between two rooms is made by using the function RoadGenerator(initialPoint, finalPoint, grid), and the points (x,y) are chosen randomly in both rooms.

4.1.12 RetrieveGrounds(grid)

RetrieveGrounds is a function that returns a collection of all ground-type fields in the grid, with the goal of reducing future algorithm costs.

4.1.13 WallGenerator(grid, grounds)

WallGenerator is a function that generates walls around grounds existent in the grid. The WallGenerator function does its job by iterating through each ground to find suitable walls for the map. A suitable wall is a non-defined field that is adjacent to a ground field.

4.1.14 MapPopulator(grid, grounds)

MapPopulator is a function that populates the map with entities. The map is randomly populated by enemies, boxes and spikes. Enemies disturbs the player from progressing in the level. Spikes simply hurt the main character when they get stepped on. Boxes can contain a series of different items inside, from ammunition to power-ups, so they can be seem as a beneficial entity, as opposed to the others. The notion of beneficial and harmful entities is going to be important when it comes to the point of populating the map with tokens. The distribution of the entities is affected by the difficulty scale factor, likewise the strength of the enemies is also affected.

4.1.15 SpawnPointGenerator(roomVector, grid)

SpawnPointGenerator is a function that defines two specific locations: the start location for the main character, and the final location, representing the final goal of the generated map. The final goal is the place where the main character must bring the token. Fixing these points beforehand will help the population of tokens in the map.

4.1.16 GroundCostCalculator(grid, grounds, squareSize)

GroundCostCalculator is a function that calculates the cost of all the remaining non-populated grounds, candidates of being locations for tokens. The purpose of these costs is to give a metric to the difficulty of visiting each ground. Each ground cost is calculated taking into consideration the following information: the entities around the ground and the distances from the ground to the start and final location of the map. To define the area size to be taken into account, the function receives, as a parameter, a value that defines a square area around each ground. Given that value, the function scans for entities in the square area around each ground, increasing or decreasing each ground cost depending on the types of entities found in the elected area. Naturally, different entities should weight differently on a ground cost i.e enemies have a greater impact on the main character if compared to spikes, so they should have a bigger weight on the ground cost. After all non-populated grounds are associated with costs regarding the entities within the square area around them, it is also important to add a cost that represents the distances to the start and final locations of the map. The distance is calculated using Manhattan Distance [3].

4.1.17 TokenPopulator(grid, tokenGrounds)

TokenPopulator is a function that populates the map with tokens. Tokens are the collectible objects that the main character needs to achieve his objective. There are three different types of tokens: gold, silver and bronze. The gold token is the most valuable one, while the bronze is the least. The main character must find one of the tokens and proceed to the final location of the map to complete it. It is desirable that the tokens are positioned into challenging locations. Also, the hierarchy of token's value should mirror the difficulty of retrieving them. This function receives the collection of grounds that are candidates to be locations of tokens. Within each ground of the collection, there is an appended cost of visiting it. With that in mind, the only thing this function needs to do is to choose one ground for each token based on the ground's cost. This is done by sorting the collection and, having defined indexes for each type of token, associating the token to a ground. These indexes are related to the difficulty of visiting the grounds. Having sorted the collection in ascending order of cost, the last index is the hardest location to visit and possibly the best to position a gold token within the given options, for instance.

4.2 Token Population Analysis

Lets analyze the non-trivial part of the algorithm. It is desirable that the tokens are positioned into challenging locations. Also, the hierarchy of token's value should mirror the difficulty of retrieving

them. To be able to achieve those requirements, a metric must be associated to the hardship of visiting a location. After having calculated all costs to the tokens possible locations, designation of the tokens locations is trivial.

The algorithm was designed aiming to obtain a solution adequately close to an optimal solution in a reasonable time. Putting into context, a level generated with challenging token locations where the most valuable is seemingly at one of the hardest locations to visit while the least valuable is at a less challenging location compared to the others but still challenging to get would be a suitable level for the experimental game. The strategy adopted to calculate the costs of each possible ground for a token is quite simple and is done by the function GroundCostCalculator(grid, grounds, squareSize). Basically, each ground cost is calculated taking into consideration the following information: the entities around the ground and the distances from the ground to the start and final location of the map. To define the area size to be taken into account, the GroundCostCalculator function receives, as a parameter, a value that defines a square area around each ground. Given that value, the function scans for entities in the square area around each ground, increasing or decreasing each ground cost depending on the types of entities found in the elected area. Naturally, different entities should weight differently on a ground cost.

Now let's analyze the effort generated by function GroundCostCalculator(grid, grounds, squareSize): let n be the number of grounds in the map and let mxm be the size of the square area defined by squareSize parameter. Also, let the grid structure be of size SxS. Consider $n <= S * S$, since $S * S$ represents all game space including other types of fields that aren't grounds. For each ground the algorithm scans mxm other grounds, which indicates that for all the procedure there would be $n * m * m$ operations. At first sight it seems to be of high effort, but let's take a better look at it. Asymptotically, the time complexity seems to be $O\{n * m^2\}$, but consider that the growth of m follows very lightly the growth of n, which is a fair assumption because the scan area mxm does not need to be increased by a large margin when the map gets bigger, since the weight of the distance from the ground and the initial and final locations is also taken into consideration.

Testing the generation with a map of size $100x100$ with a scan area of $5x5$ results in a suitable map for the experimental game and if evaluated, $n <= 10000$ while $m = 5$. In this case, a maximum of 250000 operations were performed or $25 * n$ operations, which seems pretty acceptable, having only a small impact in the overall performance of the algorithm. In fact, generated maps are satisfactory even with proportionally bigger differences between n and m. With all costs calculated, all that remains to be done is to choose a ground for each token and that is the job of the function TokenPopulator(grid, tokenGrounds).

Sorting the collection of grounds by cost is, inevitably, of $O\{n * logn\}$ complexity, while choosing the grounds from the sorted list is of $O\{1\}$. That is exactly what TokenPopulator function does. The grounds in the sorted list are directly related to its difficulty by the list indexes, transforming the levelling of token positioning in a mere task of accommodating the right index for the desired token. For instance, the last index of the list would attach a convenient ground to place the most valuable token, if the list is sorted in an ascending way.

5 EVALUATION

The results achieved by our PCG approach were rather satisfactory. It produced a vast range of singular environments improving the quality of our game experimentation and enabled the reallocation of development resources to other branches without downgrading the level design to a degree that falls below expectations. The use of seeds gave the desired control over the content generated allowing us to manage the levels in our experimental game in a controlled manner. The use of the difficult factor in combination with seeds increased the capacity of the generation, and gave a new possible trait to the experimental game: levels that are generated from the same seed have a similar structure in comparison to levels generated from other seeds, meaning that, with minor changes in the difficult factor, we can create groups of similar maps that can represent sections of the game. This feature gives a new layer of control to the generated content, as we can easily populate a section by preserving the seed on multiple generations, and create new ones by altering the seed of the generation. To illustrate this behaviour, in Figure 2, Figure 3 and Figure 4 we expose our results in a graphic grid made of characters that represents maps generated by the algorithm proposed

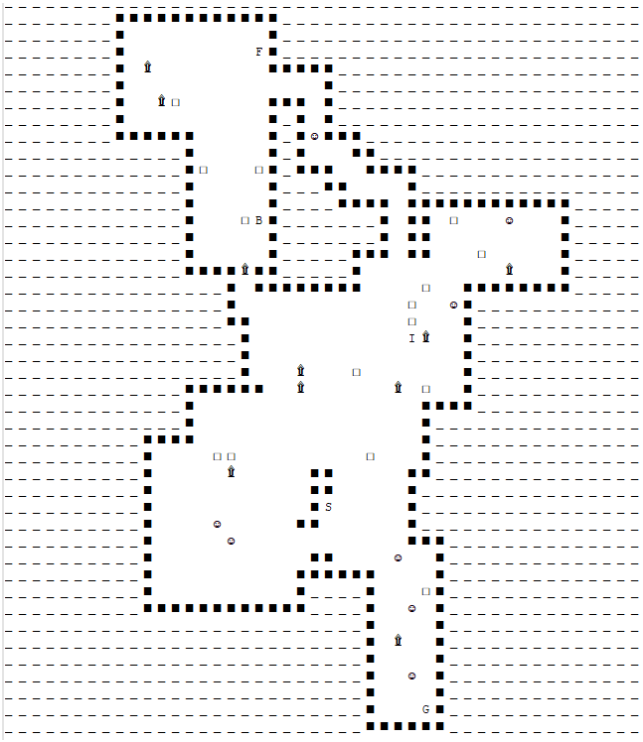

Figure 2: Map generated by our algorithm

5.1 Proof of Concepts

From Figure 2, it is possible to observe how room superposition gave more singularity to the generated map. As of now, rooms are square shaped only, but different shapes can be easily be added and would certainly improve the map eccentricity. It is also possible to perceive how the tokens are positioned accordingly to their value - characters "G", "S" and "B" represents gold, silver and bronze tokens respectively. Other entities displayed in Figure 2 refers to enemies, boxes and spikes. A generated map of the same seed with a different difficulty factor is shown in Figure 3. For a better visualization it is recommended to zoom in the figures.

Note that the structure of the generated map shown in Figure 3 is very similar to the map shown in Figure 2, and that is because they have the same seed, but different difficult factors. The difference in the difficulties is of a tenth, thus they are very similar. Finally, a new seed is used to generate the next displayed map in Figure 4.

Clearly the map generated shown in Figure 4 differs a lot in the structure if compared to the ones generated in Figure 2 and Figure 3, and that is the result of using a different seed.

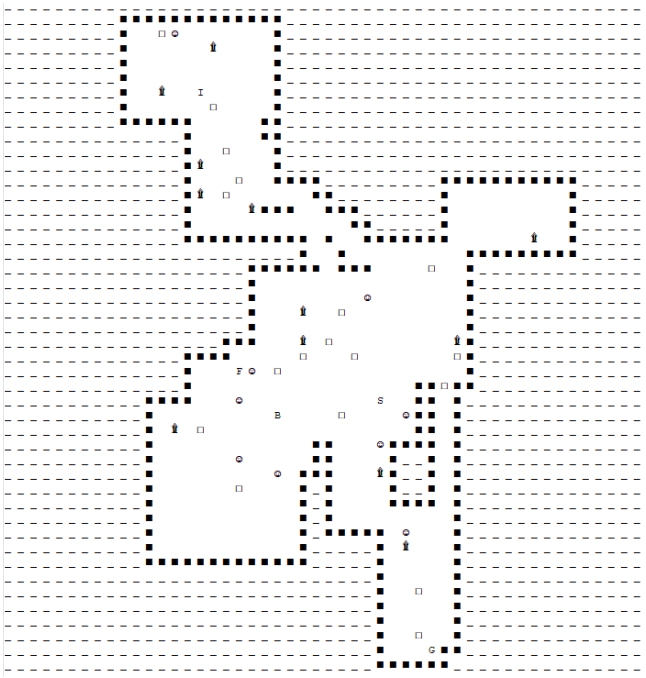

Figure 3: Map generated using the same seed as Figure 2 generation with a difference of 0.1 in the difficult factor

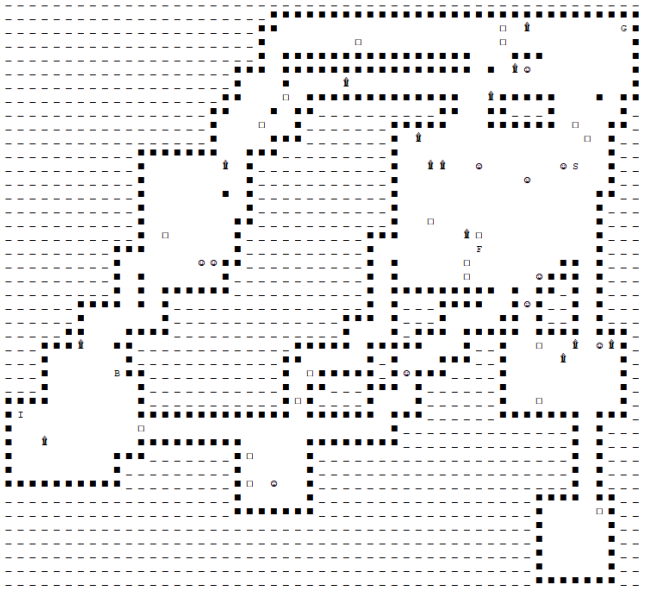

Figure 4: Map generated with a different seed

5.2 Performance Evaluation

With a direct implementation of the suggested PCG approach, we obtained a decent performance from the algorithm, even with a single-threaded approach and no clipping optimization, meaning that the whole map is generated on a execution of the algorithm.

Performance tests were performed in a machine with the following specifications: Windows 10 Enterprise 64-bit, Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 16384MB RAM, NVIDIA GeForce GTX 950.

The table 1 shows test results for different levels in the experimental game:

Level	Time	Grid Size	Entities	Memory Usage
1	1ms	1764	72	32768B
100	4ms	5476	103	125848B
500	19ms	29584	158	657632B
1000	28ms	62001	192	1459872B
5000	343ms	1590121	348	41988864B
10000	1310ms	6446521	411	122681112B
20000	5070ms	18267076	489	447736560B
30000	15842ms	45873529	557	1104960320B
50000	56210ms	156975841	635	3421365800B
100000	194069ms	454968900	723	9544367168B

Table 1: Performance and memory usage tests (ms for milliseconds and B for Bytes)

The loading time of a level was very satisfactory, specially considering that the whole map structure and population is generated in real-time. As seen in the table above, extremely high levels that probably would not be achieved by a normal player were loaded in a very adequate time. The same statement can be made about memory usage, as it was fairly acceptable for normally unreachable levels as well.

If we were to compare results with the answer set programming approach, [17] gives us a time of several seconds on much smaller game spaces only in the first phase of the generation, reinforcing that [17] approach can be very costly in terms of time.

6 Final Conclusions

This paper presents a fast and direct PCG approach for generating complete and singular levels for dungeon-like game maps, using seeds as a way of controlling the generation, and applying a difficulty factor to the generation in a way that expands the possible outcomes of the generation.

We were able to control the level generation knowing that a seed gives the same generated map in every execution of the algorithm in a given configuration. That trait combined with the use of a difficulty factor helped in designing the play difficulty curve of the experimental game, and widened the possible outcomes for the generation.

Square-shaped structures were used in the map generation, allowing superposition of structures to increase the singularity of the generated level. A logical next step would be the use of different shapes in the generation to improve the map uniqueness.

The strategy adopted to surpass the numerical limit acquired with the use of seeds produced interesting results. By using different difficulty factors to expand the possible generations of a single seed, numerical limitation of possible generations using seeds was indeed surpassed, but it created a side effect. Levels generated within the same seed are very similar in its structure. In our experimental game, that trait was appreciated as it did not hurt our intentions for the game design, but it might be not desired for all. An investigation on how to increase distinctness of levels from the same seed

should be headed in the future, for cases that similarity on generated content from the same seed is not desired.

Although obtaining reasonable results in terms of algorithm effort, diverse techniques can be used to ease the generation loading time. An obvious one is to generate the level content whilst the player is still playing on a previous level. Another method is to procedurally generate smaller sections of the level, which should be quicker to produce, and piece them together as the player enters that part of the level. A more interesting approach, albeit a bit trickier, is to generate level content accordingly to the precise exploration of the level, where content is only generated in the imminence of entering player's area of visualization, reducing the generation to a much smaller section in each iteration. A further option is to increase the algorithm performance by adapting it to a multi-thread approach, which can be done in several ways, such as dividing the map in smaller sections and designating each section to a worker thread.

In conclusion, the effort into applying our procedural content generation method in our experimental game was worth it and fulfilled our expectations, as it expanded the possible environments in the game universe to an unreachable degree if we were to design them manually one by one, while maintaining the level design value and a good control over the level planning and difficulty curve balance.

ACKNOWLEDGEMENTS

This work was partially supported by CNPq (National Council for Scientific and Technological Development, linked to the Ministry of Science, Technology, and Innovation), CAPES (Coordination for the Improvement of Higher Education Personnel, linked to the Ministry of Education), FINEP (Brazilian Innovation Agency), and ICAD/VisionLab (PUC-Rio).

REFERENCES

[1] D. Ashlock and C. McGuinness. Automatic generation of fantasy role-playing modules. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[2] J. R. Baron. Procedural dungeon generation analysis and adaptation. In *Proceedings of the SouthEast Conference*, ACM SE '17, pages 168–171, New York, NY, USA, 2017. ACM.

[3] P. E. Black. Manhattan distance. *Dictionary of Algorithms and Data Structures*, 18:2012, 2006.

[4] CipSoft. Tibia (digital game), 1997.

[5] S. Dahlskog, S. Björk, and J. Togelius. Patterns, dungeons and generators. 2015.

[6] J. Dormans and S. Leijnen. Combinatorial and exploratory creativity in procedural content generation. 2013.

[7] B. Entertainment. *Diablo (Digital Game)*. Blizzard Entertainment, 1996.

[8] F. Games. Civilization iv (digital game), 2k games & aspyr. 2005.

[9] H. Games. No mans sky (digital game), 2016.

[10] 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)*, 9(1):1, 2013.

[11] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 10. ACM, 2010.

[12] V. Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.

[13] Maxis. Spore (digital game), electronic arts, 2008.

[14] Mojang. Minecraft (digital game), microsoft studios, 2011.

[15] RIXGAMES. Dungeon grind procedural dungeon generation tutorial, 2013.

[16] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Springer, 2016.

[17] A. J. Smith and J. J. Bryson. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*, 2014.

[18] G. Smith. The future of procedural content generation in games. In *Proceedings of the Experimental AI in Games Workshop*, 2014.

[19] G. Smith. An analog history of procedural content generation. In *FDG*, 2015.

[20] F. Streichert. Introduction to evolutionary algorithms. *paper to be presented Apr*, 4, 2002.

[21] J. Togelius, T. Justinussen, and A. Hartzen. Compositional procedural content generation. In *PCG@ FDG*, pages 16–1, 2012.

[22] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, page 3. ACM, 2011.

[23] M. Toy, G. Wichman, K. Arnold, and J. Lane. Rogue (digital game). 1980.

[24] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[25] S. Wolfram. Computation theory of cellular automata. *Communications in mathematical physics*, 96(1):15–57, 1984.