computers

MDPI

Article

Creating a Newer and Improved Procedural Content Generation (PCG) Algorithm with Minimal Human Intervention for Computer Gaming Development †

Lazaros Lazaridis and George F. Fragulis *

Department of Electrical and Computer Engineering, University of Western Macedonia, 501 50 Kozani, Greece; dece00049@uowm.gr

* Correspondence: gfragulis@uowm.gr

† This paper is an extended version of our paper published in the 4th International Conference, HCI-Games 2022, Virtual Event, 26 June–1 July 2022.

Abstract: Procedural content generation (PCG) algorithms have become increasingly vital in video games developed by small studios due to their ability to save time while creating diverse and engaging environments, significantly enhancing replayability by ensuring that each gameplay experience is distinct. Previous research has demonstrated the effectiveness of PCG in generating various game elements, such as levels and weaponry, with unique attributes across different playthroughs. However, these studies often face limitations in processing efficiency and adaptability to real-time applications. The current study introduces an improved spawn algorithm designed for 2D map generation, capable of creating maps with multiple room sizes and a decorative object. Unlike traditional methods that rely solely on agent-based evaluations, this constructive algorithm emphasizes reduced processing power, making it suitable for generating small worlds in real time, particularly during loading screens. Our findings highlight the algorithm's potential to streamline game development processes, especially in resource-constrained environments, while maintaining high-quality content generation.

Keywords: procedural content generation (PCG); game development; replayability; algorithmic map generation; computer games; resource management; dynamic content creation

Citation: Lazaridis, L.; Fragulis, G.F. Creating a Newer and Improved Procedural Content Generation (PCG) Algorithm with Minimal Human Intervention for Computer Gaming Development. *Computers* **2024**, *13*, 304. https://doi.org/10.3390/computers13110304

Academic Editors: Carlos Vaz de Carvalho, Hariklia Tsalapatas and Ricardo Baptista

Received: 3 September 2024
Revised: 23 October 2024
Accepted: 1 November 2024
Published: 20 November 2024

1. Introduction

Procedural content generation (PCG) is known for its algorithmic generation of data, a method used to create random and streamlined content for several purposes such as maps, loot, item attributes, occasionally lore, etc. [1], in contrast with manual and static creation. Games are often evaluated in terms of their replayability, how elaborate their content is, play time, etc. Game content of high quality usually requires manual generation and significant effort by a large team that includes designers and developers, which considerably increases expenses and is also very time-consuming. Both wealthy studios and publishers have the required resources to support and invest in such concepts. On the other hand, this sumptuousness cannot be afforded to independent (indie) developers [2], so an alternative path must be found. PCG content has deep history in electronic gaming, and it has been relied upon by many games, particularly those heavily based on replayability to maintain the player's interest. Several popular games that utilize PCG methods are *The Binding of Issac* [3], which randomly generates rooms (see Figure 1), and *Minecraft* [4], in which its world is procedurally generated, with each component is uniquely arranged every time a new game is started, ensuring that no two players' worlds are alike. In *APEX Legends* [5], the weapons' spawn locations are completely randomized, every map is divided into subareas, and each subarea has a different spawn ratio of special or powerful equipment and power-ups, making some zones more desirable than others. In the case of generated maps in a 2D space, the same ones can also be reused in the game itself; for example, in

a real-time strategy (RTS) game, the generated map can be used both as a real-scale map and as minimap with fewer details. The same concept can also be used for facial creation purposes based on players' preferences. In most RPGs (role-playing games), which allow a player to create their own unique hero, this process is performed manually, and, in rarer cases, a player can choose a randomly generated character based on a basic feature, e.g., race or clan. Modern games such as Grand Theft Auto Online [6] and Dark Souls 3 [7] offer very detailed character customization, from face details to body parts, by the user, but this procedure is considered laborious and time-consuming if the user wants a completely customized character. Recent works suggest methods for automatic face creation either by setting some features, e.g., skin color, nationality, class, etc., or by inputting a single photo [8,9]. Although faces cannot be produced by strict PCG algorithms, as the player needs their character to remain the same throughout a playthrough, there are situations where the hero is wounded, so a random scar can be depicted on a random part of the hero's face, or the hero may become older after several game years [10]. A positive effect of PCG methods is the fact that the demanded disk capacity requirements of the final game are significantly reduced, as the content generated by its game engine is not stored anywhere on the disk but is created on the fly, leading to improved resource management. However, content created on the fly demands more processing power, ideally before the game loads, but it turns out that this concession is worth the effort.

This paper's contribution is the presentation of an improved algorithm [11] that can create a top-down outdoor-level map filled with rooms of three basic sizes, which can make it quite congested, with a fountain in a random arrangement that changes every time the map is loaded. The three basic rooms can also be slightly changed during loading scenes in order to create a map with rooms that vary even more in size rooms. The same algorithm can be used in several other applications such as the inside of a dungeon or a room [12,13]. In other instances, it can be used to randomly generate loot and/or weapons with random range of properties or to add details such as vegetation, rocks, clouds, waterfalls, and so on [14]. Nevertheless, such methods are supposed to be used as helper applications rather than replacing jobs in the illustration field [15].

This paper is organized into five sections as follows: Section 2 provides an overview of fundamental methods and strategies related to procedural content generation (PCG). Section 3 addresses issues concerning content quality. Section 4 presents an analysis of the rules that the algorithm must adhere to, along with a detailed explanation of the algorithm itself. The results of our study are discussed in Section 5, followed by a discussion of the algorithm and its characteristics in Section 6. In Section 7, we offer a comprehensive conclusion on PCG, along with insights into future trends. The Appendix A contains a detailed presentation of the algorithm.

Figure 1. Binding of Isaac is a game that generates and decorates all of its rooms randomly.

2. Concepts of PCG

Over the years, multiple PCG methods have been developed, each differing majorly in its approach used to achieve content generation [16]. Some methods generate game content during the loading phase (*Offline*) [17], while others, such as *Online* ones, though less common, create content dynamically during gameplay based on miscellaneous factors, like the player's performance. The main difference between them is whether the implemented algorithm is classified as constructive or not. *Constructive* algorithms [18,19] do not require any evaluation as their outcome is considered unconditionally playable, in contrast to *generate-and-test* algorithms [20,21] where an agent must be present to test, for instance, if a

game level leads to a dead-end. Nevertheless, special attention is needed for *softlocks* [22] as they lead to dead-end states even if they are not always undesirable, depending on game logic [23,24]. Consider a state in a Super Mario [25] game, as shown in Figure 2. If the player makes a successful jump, they can complete the level by reaching the pole where a flag is hung, whereas if they fail to avoid the intermediate gap, they will permanently become stuck between walls, a *softlock* state (see Figure 2).

The improved spawn algorithm belongs to *Offline* ones in which the map is being created just before the game level begins. In many cases, an agent should be present to examine in detail if the final map is playable. In this case, it is not necessary as it is a constructive one and it relies on a predefined ruleset that overcomes any undesirable states leading to dead-ends while the final content is being created in each playthrough literally from scratch [26]. Furthermore, although it relies heavily on its strict generative rules to construct diverse level maps every time a game level is loaded, it is not regarded as *random seed* [27] entirely as it can be accepted as a minor input from developers for complexity purposes by defining some *parameters* depending on how dense or sparse a level is desired to be.

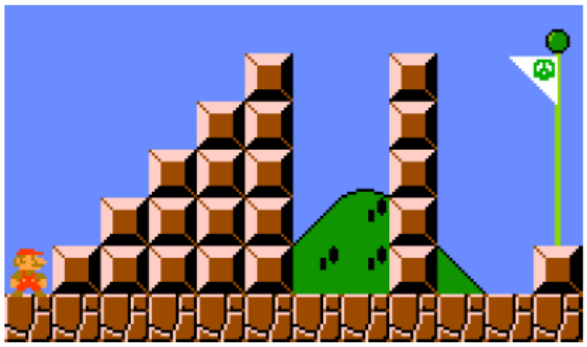

Figure 2. A manually constructed level example in Super Mario that contains a possible softlock (if a player falls into the gap between walls, it is impossible to jump out, leading to a dead-end). Since there is nothing to kill the hero, the player must either manually reset the level or quit from it in order to abandon this state, abruptly losing any progress, as in this game there no saving points or autosaves.

2.1. The Role of Algorithms in PCG

PCG methods are capable of building a complete game, taking into consideration the needs of each asset (real-time difficulty adaption, special class loot rarity, map, room decoration, equipment etc.) that can be generated randomly in each load based on several rules by tracking the player's progress. Specifically, the most well-known methods they use are the following: (i) *Markov models* [28], which are considered particularly fast in PCG generation. (ii) *Cellular automata* [29,30], that consults a predetermined rule-set on a grid map area and explores if the adjacent cell can be occupied for a suitable asset or not. This method is commonly used for cave-like creations. (iii) *Generative grammar* [31] is essentially based on a grammatical rule system and then a parser undertakes the role to decide if an action can be applied or not. It is mostly used in games with complex lore where their progress is closely related to the user's actions and, depending on his choices, either a quest will be terminated or additional actions are required so as to be successfully accomplished, or a non-playable-character (NPC) team member will decide whether he will follow or reject the player, etc. (iv) *Machine learning* algorithms progressively learn and store all past actions, or they begin from a ready-to-use dataset [32] and extend with new data accordingly. Although they are very fast, they are characterized by their unreliability, as they do not guarantee that the final outcome will be playable, especially in narrow or dense areas such as a room interior, so an agent is necessary. However, a number of solutions have been proposed that subdue this behavior with specialized methods such as generative adversarial networks (GANs) [33,34], reinforcement learning (RL) [35–37], and

deep learning [38]. (v) *Evolutionary Algorithms* [39]: although they are not yet widely used with PCG, they are preferred in 3D landscape modeling [40] to optimize game maps in strategy games that can accommodate massive armies and assist in dungeon modeling [41]. They present some failures to natural representation along with some minor conflicts between objects. Therefore, in specific operations they thrive with excellent results. Our algorithm is based on cellular automata (CA) with grid base as its kernel component.

2.2. PCG via Machine Learning

Procedural content generation via machine learning (PCGML) is considered as the generation of game content by using methods that have previously been trained on existing content from other instances [32,42]. It can be applied anywhere in a game where random content must be generated such as maps, items and their attributes, weapons and their features, character dialogues, cosmetics, etc. Machine learning can be effectively used to produce visual material that is closer to the user's preferences [43], for example, cosmetics for their equipment or building an initial character based on rudimentary questions or previous experience, but in case of levels, maps, or quests things become severely more complicated as other factors also take place. Especially on maps, machine learning methods should evaluate the final result to examine if the map is playable or if a character can jump on any permitted floor or be able to reach the exit. Such issues can be solved, but not entirely, by applying data augmentation methods [44], where the variety is increased in a given dataset, not by collecting more ready-to-use data but by adding modified versions of the already existing data [45]. In another case, a map can be represented as a set of puzzle pieces where each piece portrays a single element on a map. The puzzle pieces can also be shuffled to form another view of the same map or even to create a completely new one, while all pieces must fit seamlessly so as to avoid discontinuities (see Figure 3). The Bioshock collection [46] used this method to create myriad hacking puzzles with varying levels of difficulty.

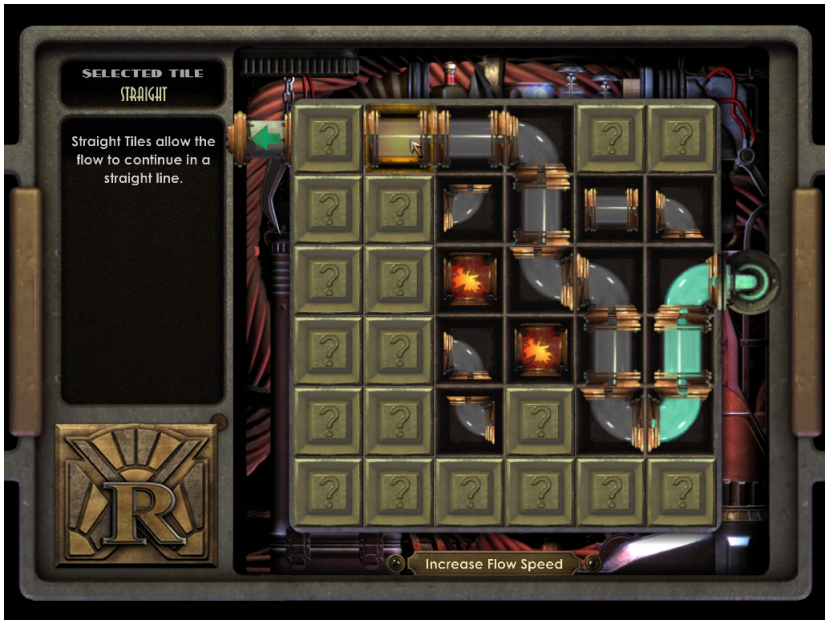

Figure 3. A mini hacking puzzle game for a number of Bioshock alarm systems. The generated puzzle games are random, whereas the pieces of the puzzle are more than enough so that the puzzle can be solved in several ways.

2.3. Generative Adversarial Networks

This architecture is considered a special extension of machine learning. It can be assumed as an adversarial game between a generator and a discriminator. Initially, it generates synthetic data from a dataset and then exhibits similar characteristics to the real

data; at the same time, a discriminator strives to classify if the generated data are considered fake or not. Generative adversarial networks (GANs), among all domains that have been applied, work better with image processing such as applications that involve human faces or handwritten characters. In terms of game levels, and the fact that GANs are based on machine learning methods [47], they encounter the same problems as they demand an initial set of known and functional maps in order to produce more of them. Additionally, due to the fact that such methods demand a lot of processing power, they are not suggested for real-time content generation [48]. Nevertheless, a study [49] managed to create several level maps for the well-known DOOM game from an initially created dataset suitable for training GAN from over than 1000 DOOM [50] levels by using two models—conditional and unconditional—where the conditional model uses several features as an input that are extracted from real levels, whereas the unconditional model uses only images from the given dataset (see Figure 4).

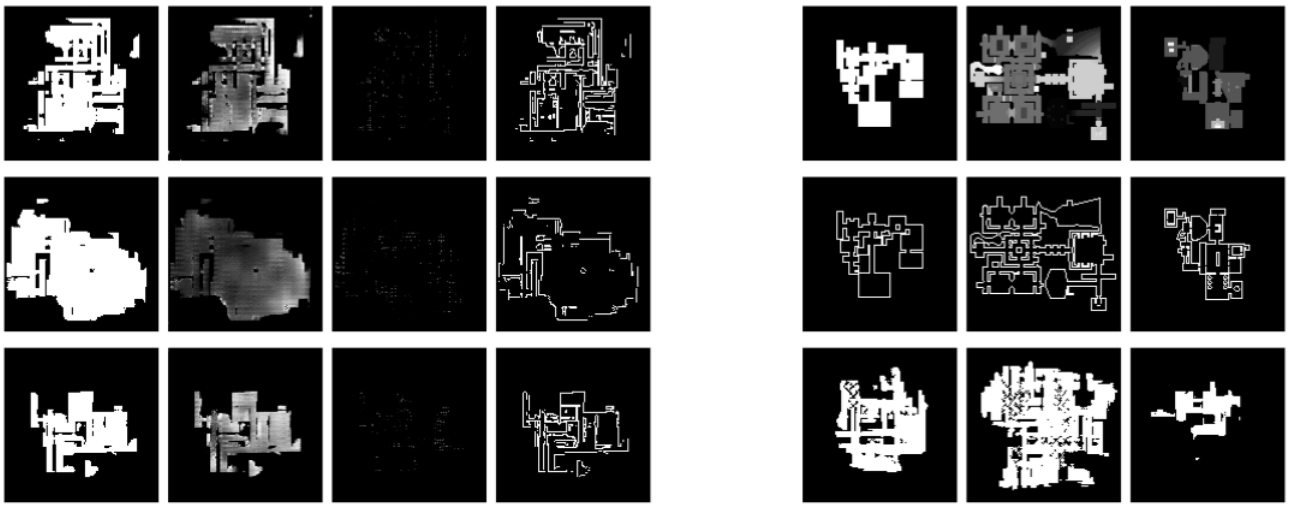

Figure 4. Left image shows maps that were created from an unconditional network while the maps in the right image were created from a conditional network.

2.4. Scenario Needs

Although there are a number of PCG algorithms to choose from, not all of them are suitable to implement any scenario. There is a big difference in creating random face portraits from scratch to produce playable terrains depending on different circumstances. Contemporary algorithms that use machine learning (ML) of GAN methods seem very promising but they are not used extensively for creating area maps of room interiors. In this paper, the proposed algorithm creates 2D area maps filled with varied-size rooms or caves and a decorative item, particularly a fountain. Each map is created from scratch without any previous experience and it can be configured so as to create maps with more or less room density depending on the desired difficulty, while the area around each room is considered as free roam state, and there are no specific paths. It uses the *cellular automata* technique, which fits perfectly for our cause as the whole map is based on a grid area where all objects are placed upon it, while at the same time the algorithm examines if there is enough space among them in order to avoid any collisions or blockages. The improved algorithm can generate a much larger number of varied-size rooms for better quality and a more diversified content. At this point, further analysis must be conducted for quality diversity and to determine how this content fits appropriately in a generated environment; either it is an open-air area with different climate conditions and terrain, or indoor areas such as caves or chambers.

3. Content Quality

Although random generated maps and environments are what we expect from a procedural content generation algorithm, the final outcome is not always as good as we would like it to be. This is not because of some failed object placements that overlap each other or an unexpected dead-end [51], but because the final scene does not seem natural or sensible, as the selected environmental objects do not match to a particular terrain or the co-existence of some objects does not make sense to the same place or map. Some practices define complex rules before the execution of an algorithm. Other solutions suggest a mixed approach where a PCG algorithm creates several templates [52], and at the end the designer chooses any or all of them that are compatible with their goals.

However, there are cases where some features are not disastrous; on the contrary, they enrich the complexity of a game or a level depending on what we would like to achieve. For example, roads on a strategic map should occasionally overlap each other, so crossroads, T-roads, or any other junction can be created. This method can convert a simple road network into quite a complicated one and it is very useful and easy to define the difficulty level of a game, e.g., if you are in early stages or in later ones. Also, the same technique can be used for defining the difficulty in real time, for instance, if a player's score is very high in one level, the next one will be much harder and vice versa. Another case is if a player is having a hard time passing a level so after a few failed tries on the same level, the next one may become easier. A study that uses roads to create new maps was applied to the *Kindom Rush: Frontiers* [53], a web-based tower defense game in which new maps were created with different road networks and random tower places with minimum distances between them all over the map, as overlap and close proximity in this situation must be avoided (see Figure 5). The Kullback–Leibler (KL) divergence was applied for the cover distribution:

$$KL(P||Q) = \sum_{x} P(x) \log(\frac{P(x)}{Q(x)}) \quad (1)$$

which defines a standard distribution for the whole map, which gives quite a natural appearance. At this point, for comparison purposes, our algorithm is based thoroughly on the asset number that is defined from the designer and it chooses if the selected asset can be placed or not on the map, based on the distance or location constraints giving a natural environment in a different way.

Other methods suggest a top-down approach where a game level has an entry point at the top of it and an exit point at the bottom; in other words, the player is always moving down. A known game that follows this strategy is Spelunky [54], where each map that represents a level is divided into 4×4 rooms and a path is planned throughout these rooms, as shown in Figure 6. At this point, it is important to mention that it is not necessary for all rooms to be used, which is something that easily defines how long or short a level can be. Each box is called a chunk and they are replaced by a random number; finally, the algorithm defines which number will be at the start and which one will be at the end [55]. The next step includes the room placement for each chunk, selected from quite a large set of templates, and the set of templates used depends on the area the level is in and whether the room falls on a path. For instance, according to Figure 6, beginning from the start chunk, a room with a corridor that has an exit point to the right of it is needed in order to continue in the room where the corresponding arrow indicates, so the suitable template set is the one that has exit points on the right side of the rooms, etc. Therefore, the room in Figure 7 fits in the second chunk as both entry points are located on the left of it and the exit point at the bottom. If we combine all of the above with decoration, obstacles, and monsters, this process leads to a game that generates unique levels in such a way that each game is always different and, at the same time, keeps the game fresh and exciting.

Based on the same principle is the game Diablo [56], which is separated into four stages where each stage includes four levels by using the top-down approach. Here, the player can access each level by finding an entrance to the lower one. In fact, all of the stages are dungeons with different names and design, and polished with appropriate themes.

Figure 5. Three random levels are created in the *Kingdom Rush: Frontiers* with roads, tower places, and monster generation in consecutive waves. Also, in each wave, monsters are grouped based on their kind.

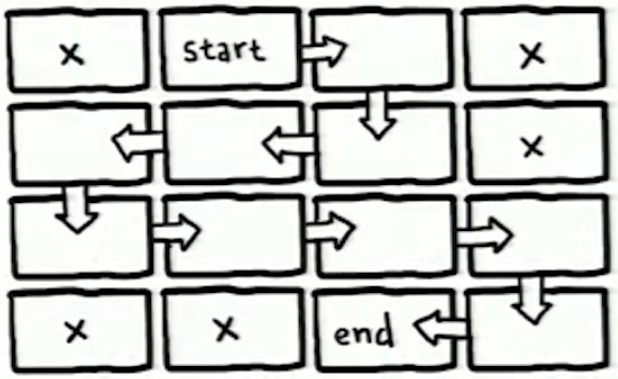

Figure 6. All levels in *Spelunky* game use this structure, with a preplanned top-down path, while all levels have the same size of 4×4 rooms. The start and end points are randomly generated in any room.

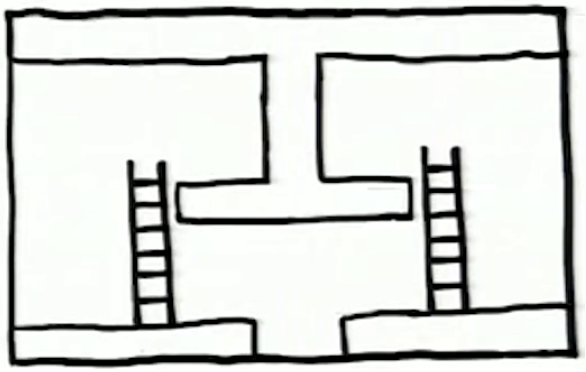

Figure 7. Example room design in *Spelunky* in its primary state, without any added decoration. Note that this room has two entry points, one from the left and another one from the right.

Everything in a level has a fully randomized arrangement at such a point that even the exit spots that lead to the lower levels are randomly placed throughout, but in this case an extra control is applied and the level is recreated from scratch if the exit spot cannot be placed due to the lack of space. On the other hand, extra attention is given both to the rooms that are located inside a level, which must be connected through corridors or other room entrances, and to the room entrances where their directions must not face a wall, something that we also check on our spawn algorithm. In Figure 8, a map is generated for the Cathedral level in Diablo, clearly showing the rooms' placement, corridors, and the exit point to the lower level.

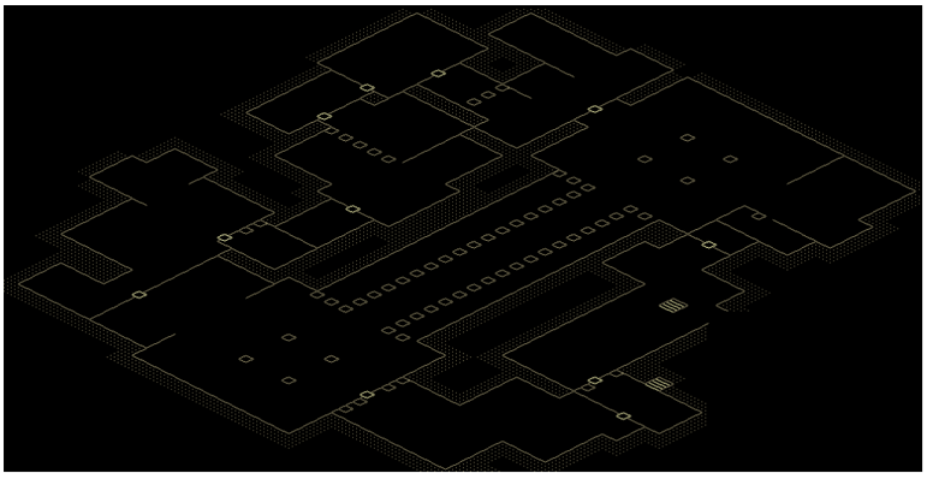

Figure 8. A randomly generated level of the Cathedral stage in *Diablo*.

3.1. Quality Diversity

Content quality not only refers to asset placement but also leads to quality diversity in order to make a game more enjoyable. As is well known, many games lack multifarious levels or stages so that a player can discover patterns from a point onward. Providing diverse levels to players adds extra value to entertainment since they need to deploy different strategies under specific circumstances [57]. Map diversity can also be combined with other game elements such as decoration objects, obstacles, statistics over offense/defense weapons, or rewards based on specific criteria. For example, in *Spelunky*, after the room selection, the next step comprises choosing obstacle and trap placement in certain points of each map; finally, object and monster placements are selected in random locations of each map, as shown in Figure 9.

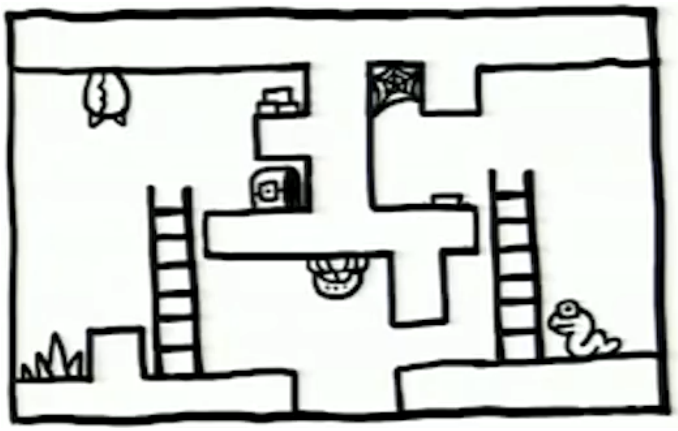

Figure 9. After room selection in *Spelunky*, as a last step, randomized decorations, obstacles, and monster placement takes place.

Different parts of an environment need different manipulation from an algorithm. For example, in the case of decoration, the first thing needed is a very large object dataset; at the same time, this dataset should be divided into several categories depending on place, i.e., if it is outdoor, indoor, rooms, corridors, etc. Irrespective of the concept we would like a map to contain, it is of critical importance for a sequence to be kept. For instance, rooms are usually placed first on maps, then a road/corridor network to connect all or part of them or adjacent rooms that are connected with a door, and, finally, decoration comes last. Each step comprises different strategies and can be implemented by different algorithms, or one algorithm can also be used by configuring several settings. As long as a step belongs to the late stages in the chain of action, more restrictions are applied. For instance, it is important that the placement of decoration objects is predefined on certain spots that will be used as static objects in order for unnecessary obstacles to be avoided; otherwise, both motion and visual attributes will probably be hindered, except for cases where someone would like to hide a valuable object, treasure, contraption, etc. The improved Spawn algorithm keeps the same functionality as it is designed to be used in several scenarios that include both outdoor and indoor actions.

To keep the uniqueness of an environment, several techniques are developed, and this is where artificial intelligence (AI) shines, as special uses of it produce diversity with rich environments. One of the most reliable and fast techniques from the modern AI field used to create content on maps is reinforcement learning (RL), where environments are usually modeled by Markov decision processes (MDPs), a mathematical formulation that is used to study optimization problems. MDPs are often represented by the type (S, A, P, R, γ), where S is state space, A is the action space, $P_a(s, s') = P_r(s_{t+1} = s' | s_t = s, a_t = a)$ defines the transition probability from a state s to s' by executing an action a at time t, $R_a(s, s')$ consists of the immediate reward earned from the current transition, and, finally, $\gamma \in [0, 1]$ is the discount factor, which computes how many future cumulative rewards are used compared to the current one [58]. Due to the fact that RL is still adding complex computational tasks [47], it is somehow difficult to used in real-time implementations, but, on the other hand, it fits perfectly in turn-based games where the real-time actions exist at a minimum, mainly in nonbattle events. In addition, as Figure 10 shows, RL algorithms agents are necessary to use, and, in order to ensure that the quality will be high, a feedback system is required to monitor the overall process step by step [36] by comparing the previous state and the current updated state by using a reward calculator for a particular game. Depending on earned rewards, RL agents can generate random playable maps producing infinite unique designs, increasing both replayability and winning strategies. A recent study proposed stage creation in two phases, battle and nonbattle events [59]. Here, an evolutionary method is proposed, as in machine learning (ML) algorithms, a sufficient amount of content is required for training, but in this case, everything can be generated from scratch by learning *online* behaviors, either from the player or ready-to-use environments which represent levels or stages. In other words, it applies self-learning by interacting with the environment, something that can lead to unsupervised learning, which significantly reduces the overall processing time.

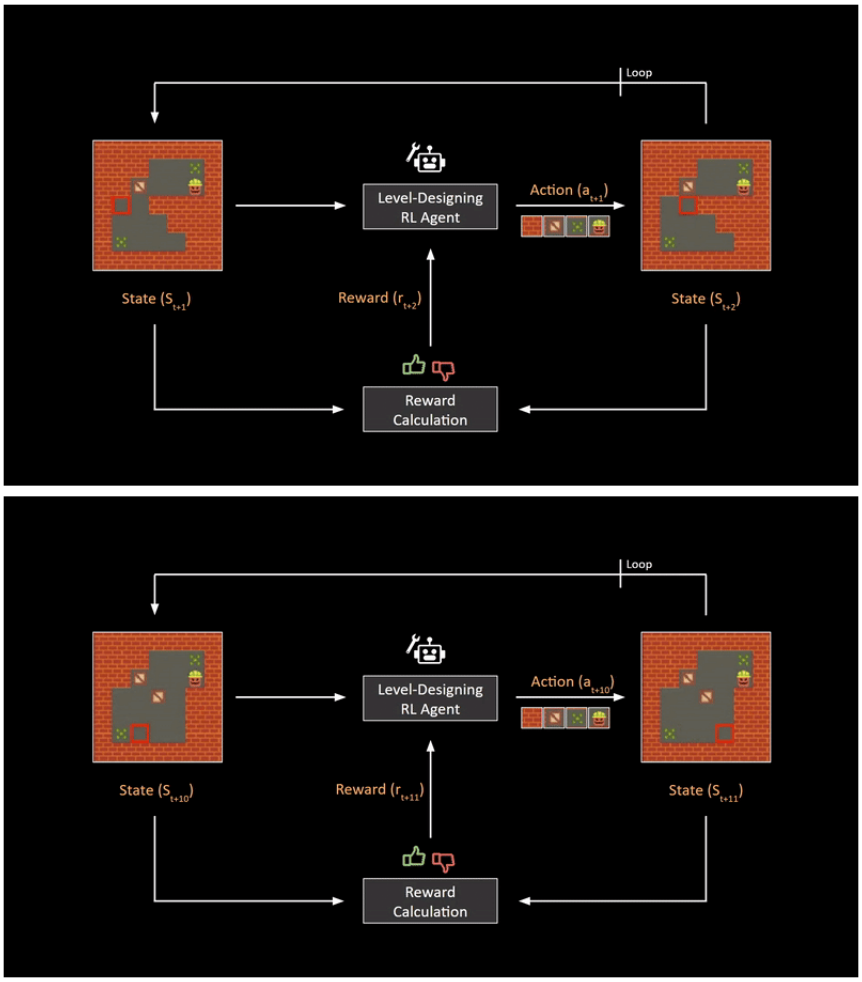

Figure 10. The first image shows the beginning of an RL agent and how it is interacting with the stage, while in the second image, the RL agent has finished its iterations. A reward calculator is present as it is needed to advise the agent if the proposed changes can lead to a dead-end or not, or if they can be accessed in general.

3.2. Intelligent Diversity

Except for environmental diversity, the replayability of a game also depends on its playstyle. Content diversity usually refers to static objects such as terrain, trees, roads, or weapons with stats that are randomly found in a treasure chest or after defeating enemies. The same static environment can also change the course of a battle by changing the location in which a fight takes place. For instance, for a boss that is vulnerable in ranged attacks and is easily defeated in one attempt, in another playthrough, they could be placed in a location that does not favor ranged attacks, so another approach must be developed by the player in order to defeat the same adversary. In other cases, bosses could be different. Along with their properties and stats, adding extra challenges in terms of the unknown of what one can encounter negates the same strategy being used every time a new game starts. In particular scenarios, if a difficulty option has been added, and a variety of heroes with completely different playstyles are present, e.g., characters who specialize in ranged attacks, melee combat, mages, summoners, etc., in the highest difficulty levels, every winning condition is totally different, as special combined strategies are required. In Diablo II [60], especially the *Hell* difficulty, levels, the playstyle of each player character must be changed as the player's resistances to all elements are dramatically dropped, even below zero, and all enemies have immunity at least to one kind of attack with different bosses. In Figure 11, a mini boss has two immunities, to physical and magic damage, so a player must use other kinds of attacks, e.g., poison, to defeat him.

Figure 11. *Hell* difficulty in Diablo II adds several random immunities to all foes; as a consequence, all of a character's properties must be reassigned in such a way to enhance a combination of two or three dissimilar attacks that can damage anyone in the battle field.

As a result, any player must redistribute any earned skill points in such a way so that his character becomes specialized in several attacks that better fit his playstyle, and if he belongs in a party with a specific role, such as a damage dealer or a defender. Furthermore, changing the stats only is not enough as it is mandatory for a player to also replace his weapons and armor as the game in this difficult level spawns even more advanced objects by piling up properties and giving the opportunity to players to customize their equipment even more depending on their playstyle. All of them inarguably add more challenge to the game by entirely changing its perspective, as if a different game is created.

To make a game more entertaining, a mechanism could be used to evaluate the diversity by measuring a possible satisfaction factor with the help of entropy, which is used as a base reward received by an agent. It is used to estimate the possible amount of information that a scenario has, for example, the number of segments that seem to display possible repetitions. The greater that number of repeated segments is, the lower the entropy value will be, and vice versa; this is how the diversity of a scenario is evaluated and measured. In [61]'s study, the entropy is calculated using the formula

$$H(x) = -\sum_{i}^{n} p(x_i) log(p(x_i)) \quad (2)$$

where $p(x_i)$ defines the probability occurrence for each event of the variable x, which in this case is represented by each segment, and the $log(p(x_i)$ calculates the amount of gathered information for each segment. Another study promoted a solution where it used the KL-divergence in order to quantify the similarity between the segments [62] in an RL algorithm to be able to generate endless playable levels in the Super Mario [25] game. In more detail, a level is divided into segments, then multiple RL agents evaluate the degree of diversity; finally, they are concatenated to form a full level. The KL-divergence was also used for the same Mario game, but in the study of [63], it was implemented asymmetrically,

generating complex and rich environments, as shown in the Figure 12 level, but, on the contrary, a large dataset was required by the algorithm.

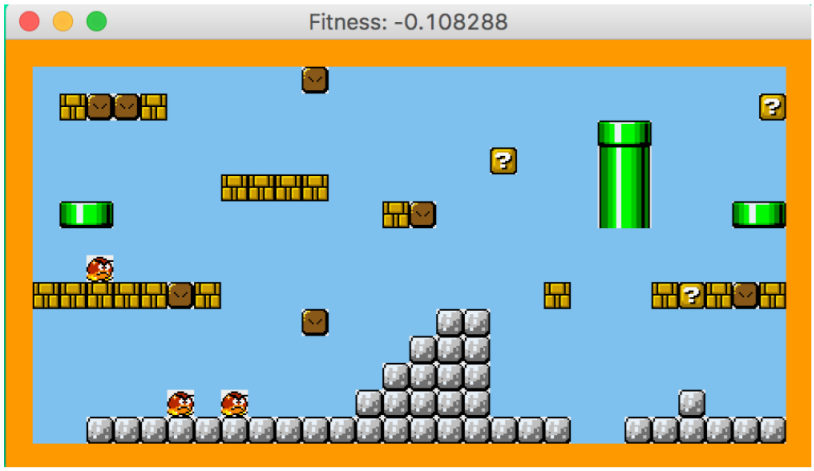

Figure 12. This level was generated by asymmetrical KL−Div with the use of a considerable size of a training sample where any novel pattern that did not exist in that sample is subjected as a candidate for use in the level generation.

In Table 1 we can see a summarized of the methods used to produce PCG environments.

Table 1. Methods used by applications to produce PCG content.

	Training Sample Required	Space	Predefined Levels	Asset Rotation
Minecraft	No	3D	No	No need
Binding of Isaac	No	2D	Merely	No
Spelunky	No	2D	Merely	No
Diablo	No	2D (map overlay)	Yes	No
Super Mario	Yes	2D	Merely	No need
Doom levels	Yes	3D	No	No
Kingdom Rush: Frontiers	No	2D	No	No
Our Spawn algorithm	No	2D	No	Merely

4. The Improved Spawn Algorithm

In general, the algorithm produces three room sizes that are placed randomly on a 2D grid-based map. Large and medium rooms have four standard entrance directions, specifically north, south, east, and west, while the small rooms' entrances can rotate everywhere in 360°. The entrance rotation of each room is also randomly generated and special attention is given to the fact that all entrances cannot face any wall side, as a minimum distance from it is estimated. The number of rooms that can be placed also remains the same, so the maximum number of each type is as follows: one large, two medium, and three small rooms. The fountain is considered as a decoration item so its size remains the same. Each item on the map has its own hitbox for collision detection purposes, where AABB [64] fits perfectly for rooms and a spheroidal for the fountain, as it is very important to distinguish boundaries to be set for future expansions.

The map uses the cellular automata (CA) technique, a method that is very suitable for building maps with ready-to-use places upon them. All grids are grouped in 5×5 groups to form larger ones and all rooms are placed into them without the obligation to fit exactly,

as shown in Figure 13. The final result visually remains the same and comprises a mini-map that can later be translated to a full-scale one. In terms of improvement, this version expands the overall number of room sizes in each type. In particular, large rooms in this version have 11 sizes, medium rooms have 31 sizes, and small rooms have 51 sizes. Each room size of each type cannot be scaled in a way that overlaps another type, e.g., a large room cannot be shrunk in a size equal to or less than a medium room. As a consequence, the end result becomes even more varied, with possible unique combinations based on type

$$C(n,r) = \binom{n}{r} = \frac{n!}{(r!(n-r)!)} \tag{3}$$

where r represents the maximum number of rooms that are likely to exist on the map, while n is the possible size of a room. Therefore, the combinations are as follows:

- Large rooms: 11 ($r = 1$ and $n = 11$);
- Medium rooms: 465 ($r = 2$ and $n = 31$);
- Small rooms: 20,825 ($r = 3$ and $n = 51$).

The overall possible combinations exceed the 213 million, a number that cannot actually be achieved in reality, so the possibility of two maps totally matching is dramatically diminished. Table 2 summarizes the differences between the initial and improved algorithm.

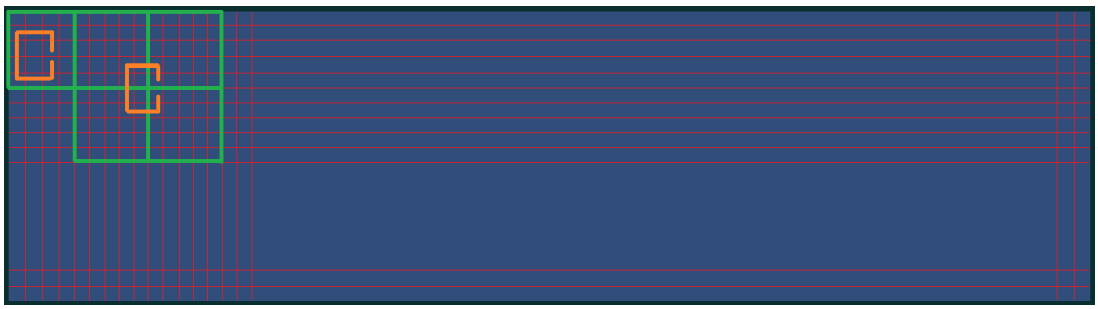

Figure 13. The grid map is divided into larger grids and each one includes 5×5 small grids. The first room fits exactly in the large grid while the second one is placed in the corner, essentially occupying four large grids. This placement constitutes the worst-case scenario, something the algorithm takes into account and acts on accordingly.

Table 2. Key differences between two algorithm versions.

	Method	Trained Dataset Required	Room Sizes (Total)	Map Size
Initial algorithm	Cellular Automata	No	3	10×3 cells
Improved algorithm	Cellular Automata	No	93	10×3 cells

4.1. Algorithm Complexity

The algorithm divides the creation procedure into several steps, where each one executes a small part of the overall process; specifically, the first one is the creation of a blank grid map with rows and columns that are given before the algorithm's execution. For this instance, we decided that fifteen (15) rows and fifty columns (50) are enough for the algorithm to work, consisting of the smallest grids on the map. While rooms and any decorations occupy more than one tile, right afterwards, the algorithm divides the grid into larger areas containing part of the tiles; specifically, each large area contains five (5) rows and ten (10) columns of small tiles: a total amount of fifty (50). In this space, any object from small rooms to large ones and any of the decorations can fit from any angle, but it is decided that only small rooms can use this feature for simplicity reasons. This is a standard process that also demands a fixed time that takes about 45×10^{-11} seconds.

All other steps include loops, as their basic goal is to decide if and where a room will be placed as long as there is a free place. As is shown in Appendix A, large rooms are placed first as they are considered the most difficult because of their size. In general, bigger rooms are placed first, if selected, and then smaller ones, in order to minimize the exclusion possibility of bigger rooms not being placed at all due to lack of space. There are four loop stages in total, where each is used for creating a room type and the fourth is used for decoration placing. Each stage includes two nested loops; the outer one examines the large columns while the inner loops scrutinize each row of the selected column to determine if the chosen room can fit. All four loops are executed at least one time, and an extra compromise is taken into account where, if a large room is present in a large column, no other rooms in that specific column can be placed except for decoration. This compromise prevents the probability for a small part of the map to become populous and the rest of it being underpopulated, as, for vision purposes, it is optimal for all objects to be placed all over the map as much as possible.

In terms of computational load, all four loops execute almost the same calculations and they are executed only if a special condition is true. First of all, each loop chooses a random number that corresponds to an asset, e.g., if the chosen value is zero (0), that means a large room is selected. If the chosen value is within the correct loop, then the process of finding a random suitable place on the map begins by scanning all the large rows within the corresponding large column. Also, minor actions take place, such as if the maximum number of an asset is reached or determining which direction the entrance of a room faces; at this point, the loop ends. The maximum computation cost can be estimated as follows: ten loops are used for scanning the large columns, multiplied by three loops for each asset along with their simple computational costs; in other words, there are thirty loops. This means a total time based on $\log(n)$ calculations, specifically 9×10^{-11} seconds, translated in a few milliseconds.

4.2. Worst-Case Complexity

The maximum possible loop number cannot exceed thirty in this instance; in fact, it is impossible to reach this value because of the fact that the maximum number of assets that can be placed on a map is seven, as shown in Figure 14. A loop is only executed for a valid asset if it is chosen by a random generator, so the maximum loop number becomes seven, where in this case the algorithm searches for a free place to set it on. The worst-case scenario in this instance is the selection of all seven rooms and the fountain with their placement to take place at the end of the map, meaning the right part of the map, which adds a tiny fraction of time to the overall process. As a result, the total time becomes $\log(n)$ calculations, specifically 3×10^{-11} seconds, plus the time for the grid map creation.

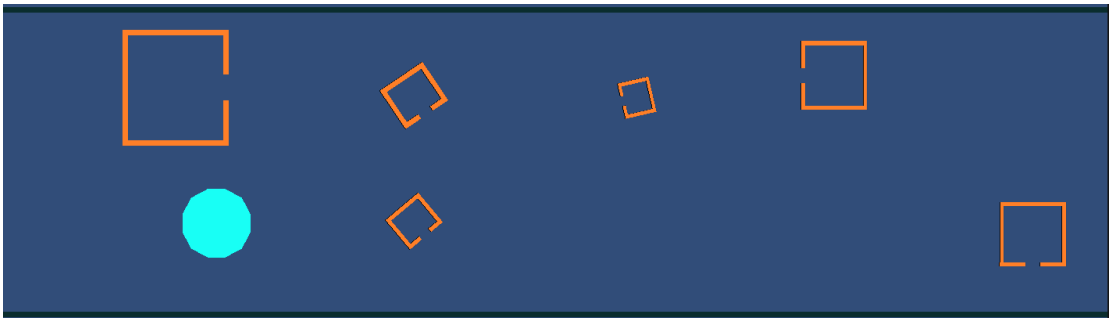

Figure 14. A random example with full rooms in the new improved algorithm.

5. Results

The overall modifications achieved a better result while the differences, in terms of size among rooms, are easily detected visually. As shown in Figure 14, all rooms, even those that belong to the same category, have obviously dissimilar sizes. The two medium rooms have few differences, while the small ones have more obvious varied sizes. The

large room is an exception as the algorithm produces only one, so there is no comparison measure, but in the end, if we compare it with Figure 15, in this instance it is a little smaller. The newer algorithm was tested thoroughly by executing it multiple times, where each time a completely new map was created.

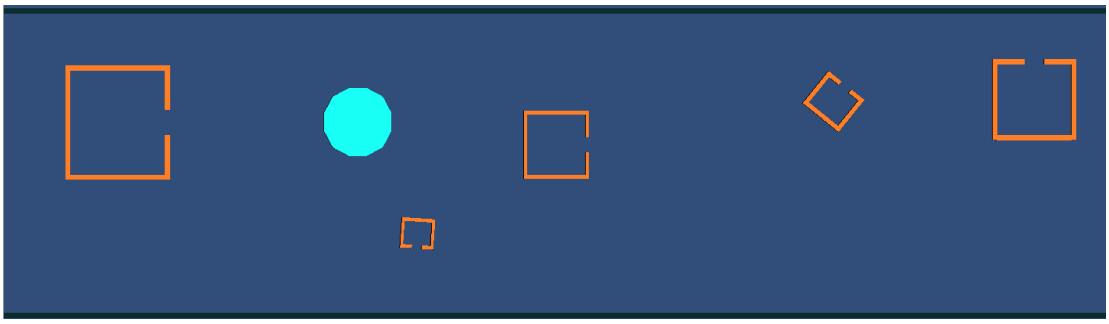

Figure 15. Another instance focusing on the large room which is a bit smaller.

Nevertheless, according to algorithm rules, there are situations where the fountain is not placed at all, producing an environment that lacks any decoration. Although this is not considered as a bug, a plain or barren map is not attractive at all except if it is used for special occasions like a small secret treasure room or a trap that leads into a room that the player must escape. On the other hand, despite the modifications, the spawn algorithm was executed seamlessly without showing any crushes or critical errors, and in terms of performance, the extra additions almost did not affect it at all. In the end, the final product was accomplished by producing a considerably more varied map without increasing the utilization of computing resources. These updates help an application to remain both replayable and lightweight.

6. Discussion

The algorithm was tested several times; specifically, 500 maps were created in a row, and all output results were within limits that were defined during the software development. Before the improvement, there was a rare situation in which two medium rooms collided by overlapping each other, but in this version, everything is corrected. The graphics engine remained the same, in particular Unity, and C# was used as the programming language as it is embedded with a very friendly development environment and is also very versatile, offering complete integration and full compatibility with any previous versions. The generated maps are quite different from each other, but because the room sizes can vary by a vast number, only half of them can be visually distinguished by the human eye. The algorithm generates a map in milliseconds, even for fully loaded ones, but it could become even faster if some conditions are removed, as in some cases they are not necessary. A case was observed in which a large room was placed at the top of a column, and while no other object can be placed underneath, the algorithm continues to check if something can be placed in the same column. This behavior was detected recently, and while it does add a negligible working load, in larger-scale maps this could become a bigger problem. But on the other hand, why should there be extra effort when it can be avoided? Because of the fact that the Spawn algorithm is rather lightweight, it can easily be used for web-based applications as it does not demand significant computational resources and its output is small in size so it can be downloaded without much effort, even over slow connections.

Its uniqueness relies on the fact that in each load a new map is generated from scratch, building a basic layout, but in the future this feature is going to change a bit as a road network will be added, at least among rooms. There are many games that use PCG methods for generating levels and maps but they use several tricks to display different outputs, and two of the best games in this matter are *Binding of Issac* [3] and *Spelunky* [54]. Especially in Spelunky, there is a large level dataset in which several of them are selected and then obstacles are placed in a random manner to change the overall view while all levels are

generated from the beginning. On the other hand, *Kingdom Rush: Frontiers* [65] was used by a revolutionary RL algorithm to generate maps with several characteristics which were defined for use in such a way to add extra value in difficulty modes. Most of the 2D maps, despite the technique they use, have a common place as they are strictly grid-based, because this kind of arrangement offers a very convenient way to place everything in a determined fashion and it is also proven to be very fast if the video game relies on levels, especially if even the secret ones are considered distinct rooms. Grid-based methods are also capable of creating not only small-scale maps but huge worlds which represent a full game playthrough, but there are various methods that use other algorithms to achieve unique generated worlds, such as the *Terraria* [66], which generates uniquely random places by adding noise per pixel and then applies multiple scans each time the algorithm adds something, such as dirt, cavities, water, flora, etc.

7. Conclusions

In this study, we explored the development of an algorithm designed to enhance the procedural generation of game environments, particularly for role-playing games (RPGs). The aim was to create more dynamic and immersive game worlds by varying environmental structures and conditions, reducing the need for human intervention in the early stages of large-scale game projects. Our findings demonstrate that while many games, including dungeon crawlers, maintain a basic pattern across playthroughs, our algorithm can introduce significant variability by altering the placement of structures within environments, such as replacing a fountain with a market in a village. Additionally, the algorithm has the potential to create expansive open worlds with distinct regions characterized by unique environmental conditions. This ability to generate diverse and complex environments automatically enhances the depth and replayability of RPGs, pushing the boundaries of procedural content generation [54]. The core program is very lightweight as it does not demand a lot of resources, about 60 MB on disk and 70 MB on RAM as a final build in Unity, since all levels/maps are created on the fly during the loading stage and, based on worst-case complexity, the loading time is negligible.

The algorithm's capacity to modify game environments on a larger scale presents significant implications for game design, particularly in the context of RPGs. By automating the placement and variation of game structures, developers can focus on higher-level design elements while ensuring that each playthrough offers a unique experience. This approach not only increases replayability but also opens up new possibilities for creating more intricate and interactive game worlds. The potential to apply the algorithm to both outdoor and indoor settings, as well as to multilevel maps, further emphasizes its versatility and relevance in modern game development. Special attention was given to softlock state avoidance where a dead-end is revealed without implying that is a bug. This problem was not present in any cases of our tests, as we predicted, first of all, enough space between rooms so that a road (in future version) can be added. This behavior prevents a player from becoming trapped in adjacent rooms as the character is also placed randomly. All this effort is additionally enhanced by meticulous collision detection methods, especially those applied in small rooms that are rotated in 360 degrees, where a rotated corner can dangerously narrow an already tight space. In conclusion, the algorithm represents a significant step forward in the procedural generation of game environments, particularly for RPGs and dungeon crawlers. By minimizing human intervention and automating the creation of diverse and interactive game worlds, this approach has the potential to revolutionize game development.

Future Trends

Despite the advancements presented, the algorithm still lacks certain features necessary for practical application in real-world software. For example, while the algorithm effectively varies the placement and size of rooms, it currently does not address the integration of cosmetic assets or points of interest, such as hidden treasures. Future research

should focus on these areas, particularly on connecting room entrances with paths that avoid overlapping with other map elements, and on decorating room interiors with a broader range of assets. Additionally, enhancing the algorithm to support the creation of multilevel maps with entrances and exits would further extend its capabilities. The final intention is for the same algorithm to also be used for decorating room interiors, as the asset arrangement between outdoor and indoor settings could be similar with similar restrictions, but in the case of the interior, the number and the variety of elements that will be chosen to input will be rather larger than outdoor ones. For replayability purposes, in order for the maximum result to be achieved, outdoor maps will vary in terms of environmental conditions while decorating them with proper assets. At the same time, the room interior will vary depending on randomly chosen themes that will be created on the fly by using suitable rules attached to the current one. Finally, a special addition will be the placement of an entrance or exit, or both, in each map in the form of a ladder, upwards or downwards, in order for multilevel maps to be created by moving back and forth, a feature that adds extra playable time and difficulty. The ultimate goal, though, is for the same algorithm, with minor adjustments for each case, to be able to create a playable game level from scratch without any human intervention.

Author Contributions: Conceptualization, L.L. and G.F.F.; methodology, L.L.; validation, L.L. and G.F.F.; writing—original draft preparation, L.L.; writing—review and editing, L.L. and G.F.F.; supervision, G.F.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No research data available.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

Input: 15 rows × 50 columns grid area, where maximum number of large, medium, and small rooms is set to 1, 2, and 3, respectively.

> **2 dimension vectors:** map[x, y] **Variables:** i, j as counters.
> **Random values:** 0 → large room, 1 → medium room, 2 → small room, 3 → no room
> **Maximum elements number:** large rooms → 1, medium rooms → 2, small rooms → 3, decorations (fountain) → 1

–Grid Map creation–

```
1. for i = 1 to maxColumns(50) do
2.    for j = 1 to maxRows(15) do
3.       map[i, j] = new Vector2(x, y)
4.       y = y + 1 (+1 tile in the row)
5.    end for (j)
6.    y = 0 (initiate the row tile)
7.    x = x + 1 (+1 tile in the column)
8. end for (i)
```

–Large column loop–

```
9. for i = 1 to largeColumns(10) do
```

–The nested "for j" loops chooses in which large row the rooms and decorations will be placed–

–Large room loop–

```
10.    for j = 1 to largeRows(3) do
11.       randomGenerator = randomValue 0 to 3
```

```
12.       if randomGenerator == 0 and maxNumberLargeRoom != 0 and noPresenceOfAn-
otherLargeRoom
13.          choose a direction other than no face wall
14.          create a large room as Vector3(map[i,j].x, map[i,j].y, direction)
15.          choose a random scale
16.          reduce the maxNumberLargeRoom by 1
17.       end if
18.    end for (j)

      –Medium room loop–
19.    for j = 1 to largeRows do
20.       randomGenerator = randomValue 0 to 3
21.       if randomGenerator == 0 and maxNumberMediumRoom != 0 and noPresenceO-
fAnotherLargeRoom
22.       choose a random direction
23.       create a medium room as Vector3(map[i,j].x, map[i,j].y, direction)
24.          choose a random scale
25.          reduce the maxNumberMediumRoom by 1
26.       end if
27.    end for (j)

      –Small room loop–
28.    for j = 1 to largeRows do
29.       randomGenerator = randomValue 0 to 3
30.       if randomGenerator == 2 and maxNumberSmallRoom != 0 and noPresenceOfAn-
otherLargeRoom
31.          choose a freely random direction
32.          create a small room as Vector3(map[i,j].x, map[i,j].y, direction)
33.          choose a random scale
34.          reduce the maxNumberSmallRoom by 1
35.       end if
36.    end for (j)

      –Decoration (fountain) loop–
37.    for j = 1 to largeRows do
38.       randomFountainGenerator = randomValue 0 to 1
39.       if randomFountainGenerator == 1 and maxNumberFountain != 0
40.          create a fountain as Vector3(map[i,j].x, map[i,j].y, direction)
41.          reduce the maxNumberFountain by 1
42.       end if
43.    end for (j)

      –Move to next map large column–
44. nextPointerLargeGridX = 0
45. nextPointerLargeGridY = nextPointerLargeGridY + stepY (10)
46. end for (i)
```

References

1. Viana, B.M.; dos Santos, S.R. Procedural Dungeon Generation: A Survey. *J. Interact. Syst.* **2021**, *12*, 83–101. [CrossRef]
2. Barriga, N.A. A short introduction to procedural content generation algorithms for videogames. *Int. J. Artif. Intell. Tools* **2019**, *28*, 1930001. [CrossRef]
3. Edmund M, F.H. The Binding of Isaac. Available online: https://bindingofisaac.fandom.com (accessed on 16 August 2024).
4. Persson, M. Minecraft. Available online: https://www.minecraft.net/en-us (accessed on 16 August 2024).
5. Electronic-Arts. APEX Legends. Available online: https://www.ea.com/games/apex-legends (accessed on 16 August 2024).

6. Games, R. Grand Theft Auto Online. Available online: https://www.rockstargames.com/gta-online (accessed on 16 August 2024).
7. Namco, B. Dark Souls III. Available online: https://en.bandainamcoent.eu/dark-souls/dark-souls-iii (accessed on 16 August 2024).
8. Shi, T.; Zou, Z.; Shi, Z.; Yuan, Y. Neural rendering for game character auto-creation. *IEEE Trans. Pattern Anal. Mach. Intell.* **2020**, *44*, 1489–1502. [CrossRef] [PubMed]
9. Shi, T.; Zuo, Z.; Yuan, Y.; Fan, C. Fast and robust face-to-parameter translation for game character auto-creation. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 1733–1740.
10. Zhao, J.; Cheng, Y.; Cheng, Y.; Yang, Y.; Zhao, F.; Li, J.; Liu, H.; Yan, S.; Feng, J. Look across elapse: Disentangled representation learning and photorealistic cross-age face synthesis for age-invariant face recognition. In Proceedings of the AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; Volume 33, pp. 9251–9258.
11. Lazaridis, L.; Kollias, K.F.; Maraslidis, G.; Michailidis, H.; Papatsimouli, M.; Fragulis, G.F. Auto Generating Maps in a 2D Environment. In Proceedings of the International Conference on Human-Computer Interaction, Virtual Event, 26 June 26–1 July 2022; pp. 40–50.
12. Freitas, V.M.R.d. Procedural Generation of Cave-Like Maps for 2D Top-Down Games. Bachelor's Thesis, Universidade Federal Do Rio Grande Do Sul Instituto De InformáTica Curso De Engenharia De ComputaçãO, Porto Alegre, Brazil, 2021.
13. Viana, B.M.; dos Santos, S.R. A survey of procedural dungeon generation. In Proceedings of the 2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), Rio de Janeiro, Brazil, 28–31 October 2019; pp. 29–38.
14. Minini, P.; Assuncao, J. Combining Constructive Procedural Dungeon Generation Methods with WaveFunctionCollapse in Top-Down 2D Games. In Proceedings of the SBGames, Recife, Brazil, 7–10 November 2020.
15. Lai, G.; Latham, W.; Leymarie, F.F. Towards friendly mixed initiative procedural content generation: Three pillars of industry. In Proceedings of the International Conference on the Foundations of Digital Games, Bugibba, Malta, 15–18 September 2020; pp. 1–4.
16. Gellel, A.; Sweetser, P. A hybrid approach to procedural generation of roguelike video game levels. In Proceedings of the International Conference on the Foundations of Digital Games, Bugibba Malta, 15–18 September 2020; pp. 1–10.
17. De Kegel, B.; Haahr, M. Procedural puzzle generation: A survey. *IEEE Trans. Games* **2019**, *12*, 21–40. [CrossRef]
18. Green, M.C.; Khalifa, A.; Alsoughayer, A.; Surana, D.; Liapis, A.; Togelius, J. Two-step constructive approaches for dungeon generation. In Proceedings of the 14th International Conference on the Foundations of Digital Games, San Luis Obispo, CA, USA, 26–30 August 2019; pp. 1–7.
19. Liapis, A. 10 Years of the PCG workshop: Past and Future Trends. In Proceedings of the International Conference on the Foundations of Digital Games, Bugibba, Malta, 15–18 September 2020; pp. 1–10.
20. Gisslén, L.; Eakins, A.; Gordillo, C.; Bergdahl, J.; Tollmar, K. Adversarial reinforcement learning for procedural content generation. In Proceedings of the 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 17–20 August 2021; pp. 1–8.
21. Song, A.; Whitehead, J. TownSim: Agent-based city evolution for naturalistic road network generation. In Proceedings of the 14th International Conference on the Foundations of Digital Games, San Luis Obispo, CA, USA, 26–30 August 2019; pp. 1–9.
22. Mawhorter, R.; Smith, A. Softlock Detection for Super Metroid with Computation Tree Logic. In Proceedings of the 16th International Conference on the Foundations of Digital Games, Montreal, QC, Canada, 3–6 August 2021; pp. 1–10.
23. Cook, M.; Raad, A. Hyperstate space graphs for automated game analysis. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
24. Chang, K.; Aytemiz, B.; Smith, A.M. Reveal-more: Amplifying human effort in quality assurance testing using automated exploration. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
25. Nintendo Ltd. Super Mario Bros. Available online: https://www.nintendo.com/en-gb/Games/NES/Super-Mario-Bros-803853.html (accessed on 17 August 2024).
26. Bontrager, P.; Togelius, J. Learning to Generate Levels From Nothing. In Proceedings of the 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 17–20 August 2021; pp. 1–8.
27. Summerville, A. Expanding expressive range: Evaluation methodologies for procedural content generation. In Proceedings of the Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference, Edmonton, AB, Canada, 13–17 November 2018.
28. Snodgrass, S.; Ontanón, S. Learning to generate video game maps using markov models. *IEEE Trans. Comput. Intell. AI Games* **2016**, *9*, 410–422. [CrossRef]
29. Adams, C.; Louis, S. Procedural maze level generation with evolutionary cellular automata. In Proceedings of the 2017 IEEE Symposium Series on Computational Intelligence (SSCI), Honolulu, HI, USA, 27 November–1 December 2017; pp. 1–8.
30. Flores-Aquino, G.O.; Ortega, J.D.D.; Arvizu, R.Y.A.; Muñoz, R.L.; Gutierrez-Frias, O.O.; Vasquez-Gomez, J.I. 2D Grid Map Generation for Deep-Learning-based Navigation Approaches. *arXiv* **2021**, arXiv:2110.13242.
31. Thompson, T.; Lavender, B. A generative grammar approach for action-adventure map generation in the legend of zelda. 2017. In Proceedings of the 7th International Symposium for AI & Games, Artificial Intelligence and Simulation of Behaviour, Bath, UK, 18–21 April 2017.
32. Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A.K.; Isaksen, A.; Nealen, A.; Togelius, J. Procedural content generation via machine learning (PCGML). *IEEE Trans. Games* **2018**, *10*, 257–270. [CrossRef]

33. Gutierrez, J.; Schrum, J. Generative adversarial network rooms in generative graph grammar dungeons for the legend of zelda. In Proceedings of the 2020 IEEE Congress on Evolutionary Computation (CEC), Glasgow, UK, 19–24 July 2020; pp. 1–8.
34. Torrado, R.R.; Khalifa, A.; Green, M.C.; Justesen, N.; Risi, S.; Togelius, J. Bootstrapping conditional gans for video game level generation. In Proceedings of the 2020 IEEE Conference on Games (CoG), Osaka, Japan, 24–27 August 2020; pp. 41–48.
35. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
36. Khalifa, A.; Bontrager, P.; Earle, S.; Togelius, J. Pcgrl: Procedural content generation via reinforcement learning. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Online, 19–23 October 2020; Volume 16, pp. 95–101.
37. Delarosa, O.; Dong, H.; Ruan, M.; Khalifa, A.; Togelius, J. Mixed-initiative level design with rl brush. In Proceedings of the International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar), Virtual Event, 7–9 April 2021; Springer: Cham, Switzerland, 2021; pp. 412–426.
38. Liu, J.; Snodgrass, S.; Khalifa, A.; Risi, S.; Yannakakis, G.N.; Togelius, J. Deep learning for procedural content generation. *Neural Comput. Appl.* **2021**, *33*, 19–37. [CrossRef]
39. Alvarez, A.; Dahlskog, S.; Font, J.; Togelius, J. Empowering quality diversity in dungeon design with interactive constrained map-elites. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
40. Silva, R.C.; Fachada, N.; De Andrade, D.; Códices, N. Procedural generation of 3D maps with snappable meshes. *IEEE Access* **2022**, *10*, 43093–43111. [CrossRef]
41. Gravina, D.; Khalifa, A.; Liapis, A.; Togelius, J.; Yannakakis, G.N. Procedural content generation through quality diversity. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
42. Yannakakis, G.N.; Togelius, J. *Artificial Intelligence and Games*; Springer: New York, NY, USA, 2018; Volume 2.
43. Juliani, A.; Berges, V.P.; Teng, E.; Cohen, A.; Harper, J.; Elion, C.; Goy, C.; Gao, Y.; Henry, H.; Mattar, M.; et al. Unity: A general platform for intelligent agents, 2018. *arXiv* **1809**, arXiv:1809.02627.
44. Risi, S.; Togelius, J. Increasing generality in machine learning through procedural content generation. *Nat. Mach. Intell.* **2020**, *2*, 428–436. [CrossRef]
45. Werneck, M.; Clua, E.W. Generating procedural dungeons using machine learning methods. In Proceedings of the 2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), Recife, Brazil, 7–10 November 2020; pp. 90–96.
46. Levine, K. Bioshock. Available online: https://2k.com/en-US/game/bioshock-the-collection/ (accessed on 23 August 2024).
47. Park, K.; Mott, B.W.; Min, W.; Boyer, K.E.; Wiebe, E.N.; Lester, J.C. Generating educational game levels with multistep deep convolutional generative adversarial networks. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
48. Volz, V.; Schrum, J.; Liu, J.; Lucas, S.M.; Smith, A.; Risi, S. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In Proceedings of the Genetic and Evolutionary Computation Conference, Kyoto, Japan, 15–19 July 2018; pp. 221–228.
49. Giacomello, E.; Lanzi, P.L.; Loiacono, D. Doom level generation using generative adversarial networks. In Proceedings of the 2018 IEEE Games, Entertainment, Media Conference (GEM), Galway, Ireland, 15–17 August 2018; pp. 316–323.
50. id Software. Doom. Available online: https://www.idsoftware.com/en (accessed on 25 August 2024).
51. Alvarez, A.; Dahlskog, S.; Font, J.; Holmberg, J.; Johansson, S. Assessing aesthetic criteria in the evolutionary dungeon designer. In Proceedings of the 13th International Conference on the Foundations of Digital Games, Malmö, Sweden, 7–10 August 2018; pp. 1–4.
52. Alvarez, A.; Dahlskog, S.; Font, J.; Holmberg, J.; Nolasco, C.; Österman, A. Fostering creativity in the mixed-initiative evolutionary dungeon designer. In Proceedings of the 13th International Conference on the Foundations of Digital Games, Malmö, Sweden, 7–10 August 2018; pp. 1–8.
53. Liu, S.; Chaoran, L.; Yue, L.; Heng, M.; Xiao, H.; Yiming, S.; Licong, W.; Ze, C.; Xianghao, G.; Hengtong, L.; et al. Automatic generation of tower defense levels using PCG. In Proceedings of the 14th International Conference on the Foundations of Digital Games, San Luis Obispo, CA, USA, 26–30 August 2019; pp. 1–9.
54. Yu, D. Spelunky. Available online: https://spelunkyworld.com/original.html (accessed on 24 August 2024).
55. Lee, N.; Morris, J. A Procedural generation platform to create randomized gaming maps using 2D model and machine learning. In Proceedings of the CS & IT Conference Proceedings, Jakarta, Indonesia, 16 February 2023; Volume 13.
56. Entertainment, B. Diablo. Available online: https://us.shop.battle.net/en-us/product/diablo (accessed on 29 August 2024).
57. Pereira, L.T.; de Souza Prado, P.V.; Lopes, R.M.; Toledo, C.F.M. Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players. *Expert Syst. Appl.* **2021**, *180*, 115009. [CrossRef]
58. Nam, S.; Ikeda, K. Generation of diverse stages in turn-based role-playing game using reinforcement learning. In Proceedings of the 2019 IEEE Conference on Games (CoG), London, UK, 20–23 August 2019; pp. 1–8.
59. Nam, S.G.; Hsueh, C.H.; Ikeda, K. Generation of game stages with quality and diversity by reinforcement learning in turn-based RPG. *IEEE Trans. Games* **2021**, *14*, 488–501. [CrossRef]
60. Entertainment, B. Diablo II. Available online: https://diablo2.blizzard.com/en-us/ (accessed on 27 August 2024).
61. Dutra, P.V.M.; Villela, S.M.; Neto, R.F. Procedural content generation using reinforcement learning and entropy measure as feedback. In Proceedings of the 2022 21st Brazilian Symposium on Computer Games and Digital Entertainment (SBGames), Natal, Brazil, 24–27 October 2022; pp. 1–6.
62. Shu, T.; Liu, J.; Yannakakis, G.N. Experience-driven PCG via reinforcement learning: A Super Mario Bros study. In Proceedings of the 2021 IEEE Conference on Games (CoG), Copenhagen, Denmark, 17–20 August 2021; pp. 1–9.

63. Lucas, S.M.; Volz, V. Tile pattern KL-divergence for analysing and evolving game levels. In Proceedings of the Genetic and Evolutionary Computation Conference, Prague, Czech Republic, 13–17 July 2019; pp. 170–178.
64. Lazaridis, L.; Papatsimouli, M.; Kollias, K.F.; Sarigiannidis, P.; Fragulis, G.F. Hitboxes: A survey about collision detection in video games. In Proceedings of the International Conference on Human-Computer Interaction, Virtual Event, 24–29 July 2021; Springer: Cham, Switzerland, 2021; pp. 314–326.
65. Ironhide. Kingdom Rush: Frontiers. Available online: https://www.kingdomrush.com/kingdom-rush-frontiers (accessed on 29 August 2024).
66. Re-Logic. Terraria. Available online: https://terraria.org/ (accessed on 29 August 2024).