
Anahtar Kelimeler: Yöntemsel içerik üretimi, Yazılım etmenleri, Ayrık kümeler

Procedural Generation of Multiple Dungeon Levels Using Software Agents

Kaya Oğuz[0000–0002–1860–9127] and Jane Lameski

Izmir University of Economics
Department of Computer Engineering
Sakarya Cad. No:156, 35330, Balçova, İzmir, Turkey
kaya.oguz@ieu.edu.tr, jane.lameski96@gmail.com
Abstract. Procedural content generation (PCG) is the algorithmic creation of game content. Dungeons are commonly used in role playing games (RPG) as closed and confined areas in which the player, the non-player characters (NPCs), and monsters can interact with the environment and with each other. Structurally, dungeons are made up of rooms of various sizes, connected with corridors that provide a means of exploration and navigation. The rooms have the same base level position and the connectedness of the rooms is limited to the rooms within the level. These levels can be generated by several PCG algorithms. However, software agents are rarely used in this regard. Using software agents helps the identification and isolation of concerns while encapsulating these concerns in discrete entities. It also gives greater control on the generation process. We have two main contributions; we consider the rooms on all levels while creating a dungeon of connected rooms; and we propose a method for the generation of levels using software agents for not discrete but connected levels of dungeons.

Keywords: Procedural content generation, Software agents, Disjoint sets

1 Introduction

Procedural content generation (PCG) is the go-to solution for developers to improve the replayability of their game while reducing the time to create the content manually. PCG has become commonplace because of the ever growing and competitive nature of the game industry where several games are published for personal computers, game consoles and mobile devices.

In its essence, PCG is the algorithmic creation of content. One of its early implementations can be found in games such as Rogue [25], and its derivatives, Diablo [4], in which the algorithm creates a new set of levels, or the so-called dungeons in Rogue-likes, along with items, monsters, treasures, and anything else that is required. While PCG is mostly used to create indoor [13] and outdoor levels [6], it has also been used to create quests [9,8], buildings [15], virtual worlds [22], cities [28,24], and stories [23]. A survey of PCG on all fronts is available [11], so are a textbook [20] and a more focused survey on procedural dungeon generation [13].

Dungeons are commonly used in role playing games (RPG) as closed and confined areas in which the player, the non-player characters (NPCs), and monsters can interact with the environment and with each other. Structurally, dungeons are made up of rooms of various sizes, connected with corridors that provide a means of exploration and navigation. In most cases, the dungeons are made up of separate levels that are connected via a group of connecting tiles, such as stairs or a passage that loads another dungeon level. Because of this single point of connection, it is relatively trivial to connect two or more separately generated but adjacent dungeon levels.

The floor plan of a dungeon level is basically a two-dimensional representation while a three-dimensional representation can be used in the game. The rooms
have the same base level position and the connectedness of the rooms is limited to the rooms within the level. In this study, we have extended this approach to three-dimensions, while maintaining the trivial structural generation of the level.

There are several approaches that are commonly used in procedural dungeon generation, and they will be discussed in the following section. However, software agents, entities that can sense their environment and act accordingly [18], are rarely mentioned as a method. Using software agents in dungeon generation helps the identification and isolation of concerns while encapsulating these concerns in discrete entities. It also gives greater control on the generation process. Although the purpose of the PCG algorithm is to minimize human intervention in the generation of the content, a set of parameters to control the output is always welcome. Our study contributes to the existing literature by proposing a method that uses software agents that consider all rooms in all levels for connectivity during the generation of dungeon levels.

The remaining paper is structured as follows. The following section discusses the existing approaches to generate dungeons. Our approach which incorporates software agents to generate levels that are connected in three dimensions, without the limitations of a single level generation process is detailed in Section 3. The results are given in Section 4. The paper concludes with Section 5 which discusses the results and the future work.

2 Related Work

Dungeon or level generation is one of the most studied and commonly used branches of PCG algorithms. The purpose of this study is not to cover all of them; therefore the foremost methods will be mentioned. Interested readers are kindly directed to the relatively recent survey on procedural dungeon generation by Linden et al. [13] as a starting point.

A typical straightforward yet effective approach to generating dungeon levels is to place rooms randomly and connect them with corridors. Nystrom places randomly generated rooms in a confined space and then connects the rooms via a maze-like approach [16]. Another approach is to use space partitioning, such as binary space partitioning and store the connections in a space-partitioning tree so that the connectivity of the rooms can be arranged [19].

Although it does not have the structure of a typical dungeon, cellular automata have been used to generate levels with an organic and natural feel, rather than a man-made synthetic feel [12,1]. Cellular automata are the application of a set of rules to a grid of cells that have states that change according to these rules. Cellular automata is mostly associated with Conway’s “Game of Life” [10], but they can also be used as a method that have very few parameters and a loose control on the output of an organic cave-like indoor levels.

While they are mostly used to create language and code parsers, generative grammars can be used generate game levels if the alphabet of the grammar refers to specific parts of the level, and the rules refer to the ways these parts can be
Dormans used generative grammars to combine the generation of missions with the generation of levels to create the two vital but highly coupled parts of the gameplay. Grammars have become a common approach to generate levels, such as the bachelor thesis of Adams [2] and the dungeon crawler by Linden et al.[14].

Evolutionary methods, such as genetic algorithms, are also popular in the procedural generation of dungeons. Genetic algorithms require the representation of the level as a chromosome, and a fitness function to evaluate the randomly generated chromosomes. Depending on the outcome of the fitness function, chromosomes can be used to produce new chromosomes, until a satisfactory fitness value is reached. Valtchanov and Brown use a tree structure for the representation of the level, and a fitness function that favors tightly packed clusters of rooms that have efficient connections [26]. Other studies are by Ashlock et al. which focuses on maze like levels [3], by Brown et al. in which they apply evolutionary algorithms to generate levels for the recently popular game Hotline Miami [5], and by Ruela and Delgado who also have worked on graph-based procedural content generation [17].

Software agents have been used to generate terrains [7] and 2D platformers [27], both of which have significantly different structures than dungeons. An application of software agents to procedural level generation is given in the third chapter of PCG book by Shaker et al. [19]. This approach uses an agent that digs tunnels and create rooms sequentially. Recently, Sheffield and Shah have improved this approach by incorporating reinforcement learning algorithm to the agent [21].

3 Proposed Method

There are several advantages for using software agents in the procedural generation of dungeons. The foremost advantage is the separation of concerns by encapsulating each to a single discrete entity. For each of these agents with different concerns, a different set of parameters could be given, improving the control of the developer over the creation process. It is also possible to switch agents with different approaches for the same problem.

With all these advantages in mind, we have come up with several agent types that operate on different parts of the dungeon with a loose hierarchical system. As shown in Figure 1, the dungeon agent is at the top, because it calls other agents sequentially to create the dungeon.

3.1 Dungeon Agent

The dungeon agent (DA) is responsible for the generation of dungeon levels, therefore it accepts parameters for the number of these levels, $n$, the width $w$ and height $h$ of each floor, space size limit $s$ which depends on $w$ and $h$, and probability value $p$ for partitioning probability, $b$ for blocking some of the regions for variety.
The agent creates a space of size $w \times h$ and passes $w$, $h$, $s$, $p$ and $b$ to $n$ number of spawned floor agents (FA). The partitioned spaces from all floors are retrieved and DA spawns a room agent (RA) for each of these spaces. DA then creates a connection agent (CA) and passes the spaces to retrieve the connections and spawns a corridor agent (CRA) and passes the rooms and connections to retrieve the corridors that connect these rooms. While this is the general flow of DA, each agent works on its own by considering the entities created by other agents.

### 3.2 Floor Agent

The floor agent is responsible for partitioning the space for each individual room. It uses a simplified binary space partition algorithm which randomly divides the defined area into two parts and continues to work on these parts until a threshold is reached. The spaces are not saved in a binary tree, because their connection will be handled by another agent. The flow of FA is detailed in Algorithm 1. The listing contains utility functions such as `Area` and `RandomValue` which would be available on any programming language library, or could be implemented trivially.

As the Algorithm 1 shows, FA defines a simplified binary space partition method and uses it recursively on generated spaces. If the area of current space is greater than the `SpaceLimit`, then the method finds the longer edge of the space. It randomly selects a point $T$ on that edge to create two subspaces, $S_1$ and $S_2$. FA rolls the dice for partitioning these spaces further by using the simplified binary space partitioning method recursively or stop and add the subspace to $L$. At this point the probability value $p$ is reduced to $p \times p$, making it more likely to partition the space earlier in the recursive calls. This is a preference that can be reversed, since we favored larger rooms to be available in the generated levels.
Algorithm 1: The Floor Agent uses a simplified binary space partitioning approach to generate spaces.

Input : w, h, s, b, p  
Output : L

Create space S at point (0, 0) with width w, height h  
Create L, an empty list for generated spaces  
SpaceLimit = Area(S) x s  
Define and call SimplifiedBinarySpacePartition(S, p):

If Area(S) > SpaceLimit:
    Find the longer edge of S, either its width or length  
    Randomly select a point T on that edge  
    Use T to create subspaces S1, S2  
    If RandomValue() > p:
        SimplifiedBinarySpacePartition(S1, p x p)
    Else:
        Append S1 into list L  
    If RandomValue() > p:
        SimplifiedBinarySpacePartition(S2, p x p)
    Else:
        Append S2 into list L
Else:
    If RandomValue() > p:
        Append S into list L
L contains a list of spaces

If the area of the space S is not greater than the SpaceLimit, then the method checks the blocking probability and adds S to L.

3.3 Room and Item Agents

For each space generated by the floor agent, dungeon agent spawns a room agent which works on the space given to it by generating a room that can fit into it. It then spawns an item agent (IA) which randomly places items on the floors of these rooms. IA receives a parameter m which limits the number of items it can generate for a room.

The agents for room and items are implemented more straightforward than other agents, although they can include a wide range of options depending on the context of the game. In this case, the focus was on the demonstration of agents, therefore a rectangular area and random spots on the floors were all that was needed.

3.4 Connection and Corridor Agents

Connection agent is vital for the generation of connected rooms among all levels of the dungeon. CA is spawned when the FA are done. It receives the spaces on
**Algorithm 2:** The Connection Agent uses disjoint sets a bounding box collision detection to locate the adjacent spaces.

<table>
<thead>
<tr>
<th>Input : List of spaces L</th>
<th>Output : List of connected spaces C</th>
</tr>
</thead>
</table>

For each space S in L:
   For each space T in L - {S}:
      If S and T are on the same level:
         If S and T share an edge:
            Set T as a neighbor of S
      Else:
         If T is on the level above:
            If Bounding Box Collision detects an overlap:
               Add T as a neighbor of S
               Add S as a neighbor of T

For each space S in L:
   If S has no neighbors:
      Remove S from L

Let C be the list of connections
Set all spaces to a set of their own
Until there is only one set left:
   Randomly pick S from L
   Randomly pick N from neighbors of S
   If S and N are NOT in the same set:
      Find the set of S, S1
      Find the set of N, S2
      Perform a union operation on S1 and S2
      Store the connection between S and N in C

all levels and finds the adjacent spaces on the same level, or the levels above or below. For the spaces on the same floor, it looks for the edges that are shared among these spaces to define them as neighbors. For the spaces on different floors, it uses a simple bounding box collision check to determine if two spaces overlap.

The flow of CA is given in Algorithm 2. As the algorithm shows, the connection agent first determines the neighbors of each space. This is done by going through the list of all spaces and comparing them to the list of remaining spaces. If two spaces, S and T are on the same level, then the agent checks if they share an edge. To do so, it runs a list of checks on the top left corner of the spaces, along with their width and height. If they share an edge, then T is set as a neighbor of S. The neighbors are stored as a list, so a space can have many neighbors. If these spaces are not on the same floor, then it is possible that they overlap.
Bounding Box Collision can be used to find overlapping spaces and set them as neighbors.

At this point, there might be some spaces that don’t have any neighbors because of the blocked spaces. Connection agent finds and removes them from the list of spaces.

In a regular level, all rooms are expected to be connected, providing accessibility to each. In our study, the rooms on all floors are considered for connection. This creates rooms that are inaccessible from their own floor, but another path through other rooms on other floors is generated. To make sure that all rooms are connected, the final part of the CA in Algorithm 2 uses the disjoint set approach with the union and find methods. It initializes all spaces on a set of their own, and continues to apply the union function to the sets until there is only one set left. It does so by first randomly picking a space $S$ from the list of spaces $L$, and randomly picking one of its neighbors $N$. If these spaces are not in the same set, then they are joined by the union operation. Also, the connection between them are stored in the list of connections $C$.

Once the CA is done, the disjoint set algorithm ensures the connectedness of all rooms. These connections, along with the list of rooms, are passed to the corridor agent. The corridor agent operates on the rooms in the connected spaces. For each room $R$, it finds the room $Q$ in the connected space. If they are on the same floor, it finds their relative positions and draws a corridor from $R$ to $Q$. If they are on separate floors, it assigns a space in both rooms as stairs that connects them together.

As with the room and item agents, the corridor agent is designed to serve its purpose for the demonstration of agents. Several different approaches can be used to extend its capabilities; such as using path finding algorithms to build the corridors.

4 Results and Discussion

The time complexity of the algorithm is $O(n^2)$ where $n$ is the number of spaces generated in all levels. The algorithm compares each space with the other ones in order to find the neighbors. Since the number of spaces are moderately low, the agents complete their work within seconds. On the other hand, the generation is usually done offline, so the current time complexity of the algorithm is well within accepted values.

The agents are implemented using Python and the resulting maps are generated as image files. A sample output of the three levels of a dungeon with ten levels is given in Figure 2. The parameters were: $1024 \times 1024$ is given as the size of each level, room size limit is set as the 0.05 percent of the total area, space partitioning probability is set to 0.3 and the room block probability is set to 0.05. To keep the maps simpler, the items are not printed on the images. Since the outputs are in two-dimensions, the stairs are shown as gray squares inside the rooms. As the maps show, there are some rooms that are not accessible from the rooms within the level itself, however the connections to the other rooms
above or below the level provide access to them. The effect of space partitioning can be seen in the level on the left and right, while the room blocking probability can be observed in the level on the left and the level in the middle.

The levels generated by this study look similar to the ones in the existing literature. However, we introduce the multi-level connectivity of the rooms and the realization of several software agents to generate the level. While we put forward and implement these contributions, we would also like to discuss the apparent improvements and existing weaknesses in this section.

Since the agents are kept relatively simple and straightforward in this preliminary study, they were able to work in harmony without any apparent conflicting issues in their linear fashion. The agents could be improved by adding variety to their behaviours. A typical example would be incorporating random turns in long corridors for the corridor agent, and using geometric shapes other than rectangles for the room agents. More agent types could also be added; the room agent could make use of more agents in their design, such as the inclusion of an agent that can handle the lighting of the rooms, another one to position objects.

The agents could be redesigned so that they could be started simultaneously. However currently this would not produce good results because in their current form because the connection agent would require the generation of all rooms to create paths that make use of rooms on other levels.

5 Conclusion and Future Work

We have defined a set of agents to handle the procedural generation of a dungeon. Most dungeons provide several dungeon levels, each of which are isolated from the remaining levels except their single point connections to the levels just above or below them. The rooms on each level are connected and accessible within the level. Besides using agents, our solution considers the rooms in all levels at the same time and all of them for creating a connected set of rooms, on multiple floors.
The lack of agents on the procedural content generation front is surprising, because they provide several advantages; such as the separation of concerns and the ability to replace agents with ones that have other purposes if need be.

The agents appear to generate the levels rather quickly even on an interpreter such as Python. A test run which included 100 levels, with room size limit and room block probability both set to 0.05, is completed within seconds. Compared to other agent based solutions our method provides more control on the generation process because of using agents for each concern.

This preliminary work is planned to be ported to a game engine, such as Unity, so that the dungeons can be immediately viewed in three dimensions, along with their textures and objects. This will also allow us to create more complicated agents that could consider more cosmetic challenges, such as the placement of lights and objects in the rooms. From an object oriented point of view, agents are also planned to define specific interfaces so that they could easily be extended.

References
