

Creation of a dungeon through Procedural Generation algorithm

Francesco Autera

19 luglio 2021

1.Index

2. Actors and Uml.....3

3. Generation Algorithm Introduction.....6

4. Room generation under constraints.....8

5. Replacing unsuitable rooms.....13

6.Results.....15

7.Conclusions.....17

Bibliography17

2. Actors & UML

The method on which the algorithm is based is the one on an agent, where however the agent has construction constraints since each room must be at a fixed distance from the generating room. This method has two major advantages

- Effective because it allows the agent to create a complete dungeon given the inputs.
- Given an initial and final room, there will always be a road connecting them.

The main disadvantage is that the system does not allow much flexibility in room placement since their position depends only on the generating room.



Figure 1. Dungeon created through the proposed algorithm

2.1 Actors

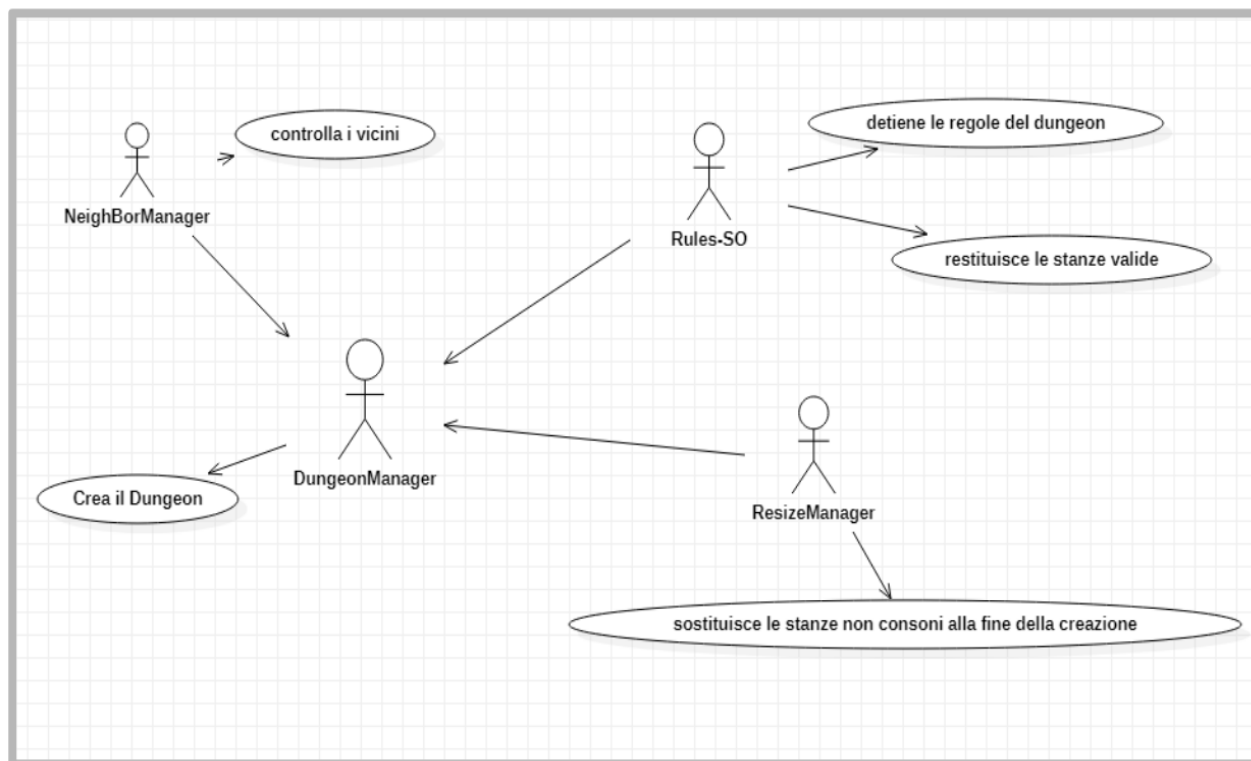


figure 2. System Actors

As we can see from the image above, our agent is the `DungeonManager`, which is the one who is in charge of instantiating rooms and corridors according to the rules dictated by the system. `DungeonManager` receives the input of how many rooms must be generated by Rules; this is a Scriptable Object in which it holds the creation rules, including the maximum number of rooms that must be generated, and it also has another important function, that is to return the valid rooms (this point is discussed in more detail in the following chapters). The remaining two actors have two very specific tasks which, like the previous one, we will analyze in the following chapters.

2.2 UML

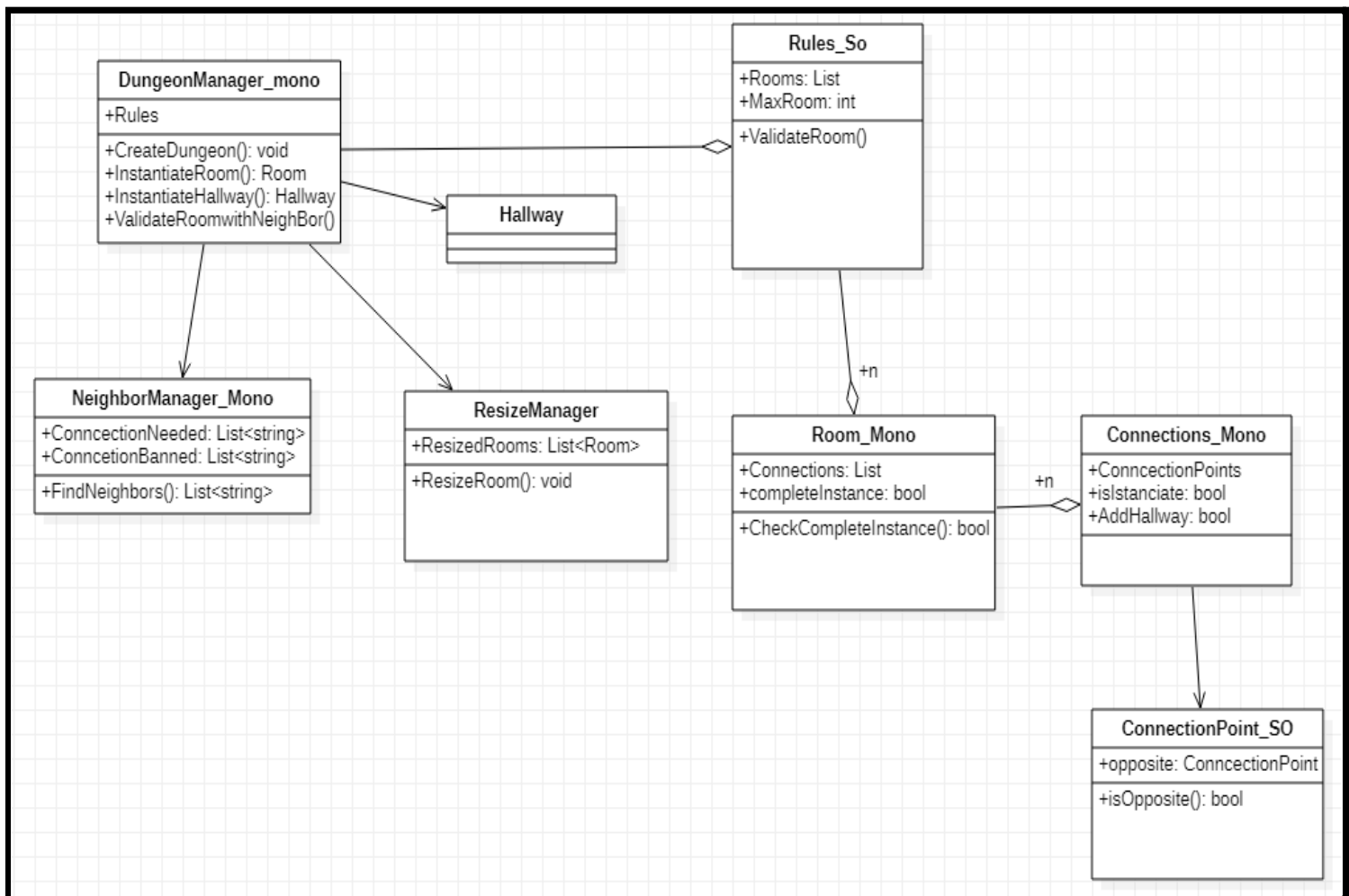


Figure 3. UML

As shown in the image above, Room is composed of a series of openings, called Connections, which inside has a connectionPoint that knows which opening it is and knows which is its opposite so that its neighbors must have that particular opening.

Rules in addition to holding the game rules holds all the possible combinations of rooms in the system.

DungeonManager checks that list to determine which room should be generated.

3.Generation Algorithm Introduction

This chapter will introduce the high-level logic flow of how the algorithm behaves; parts of this flow will be analyzed in the next chapters.

3.1 Rooms

As mentioned in chapter 2.1, rooms have openings each opening knows which is the opposite opening. Each room, as shown in figure 4, is a Unity gameObject of size 3x3 consisting of Tiles and walls of size 1x1. Tiles are gameObjects that represent a tile on the floor of the room, while the wall represents the outline of it.

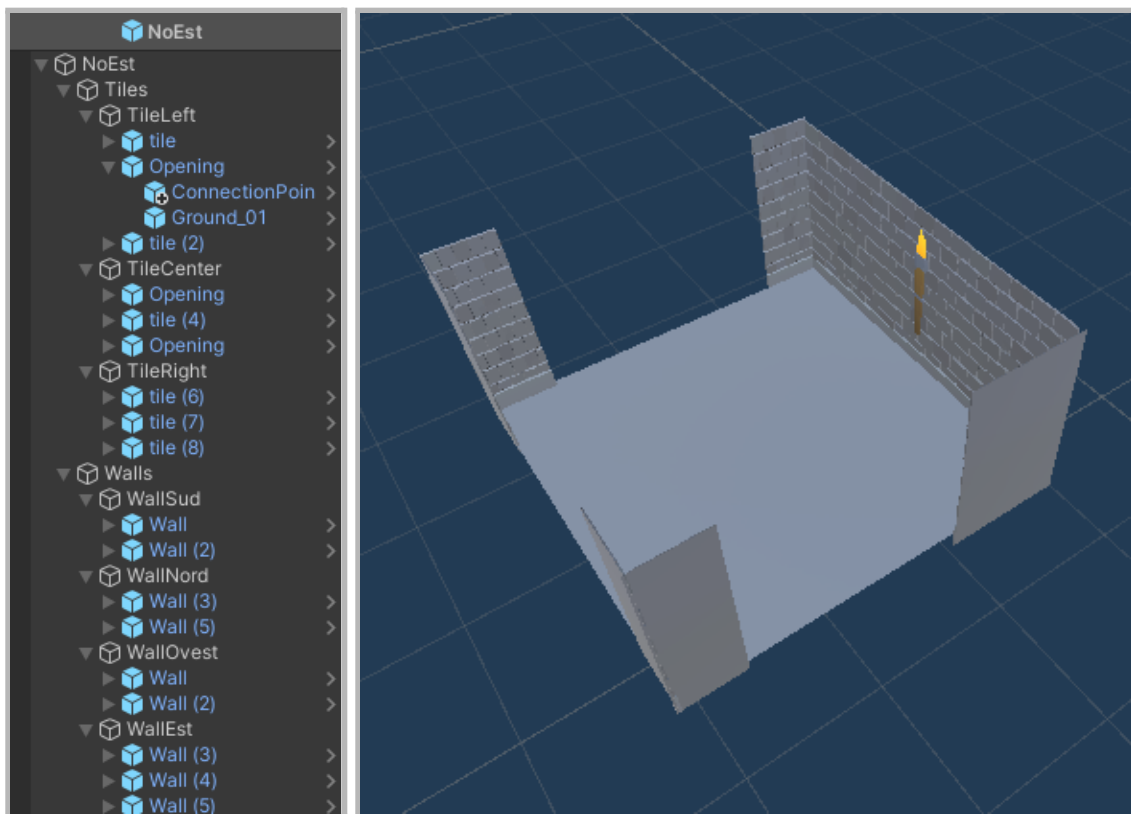
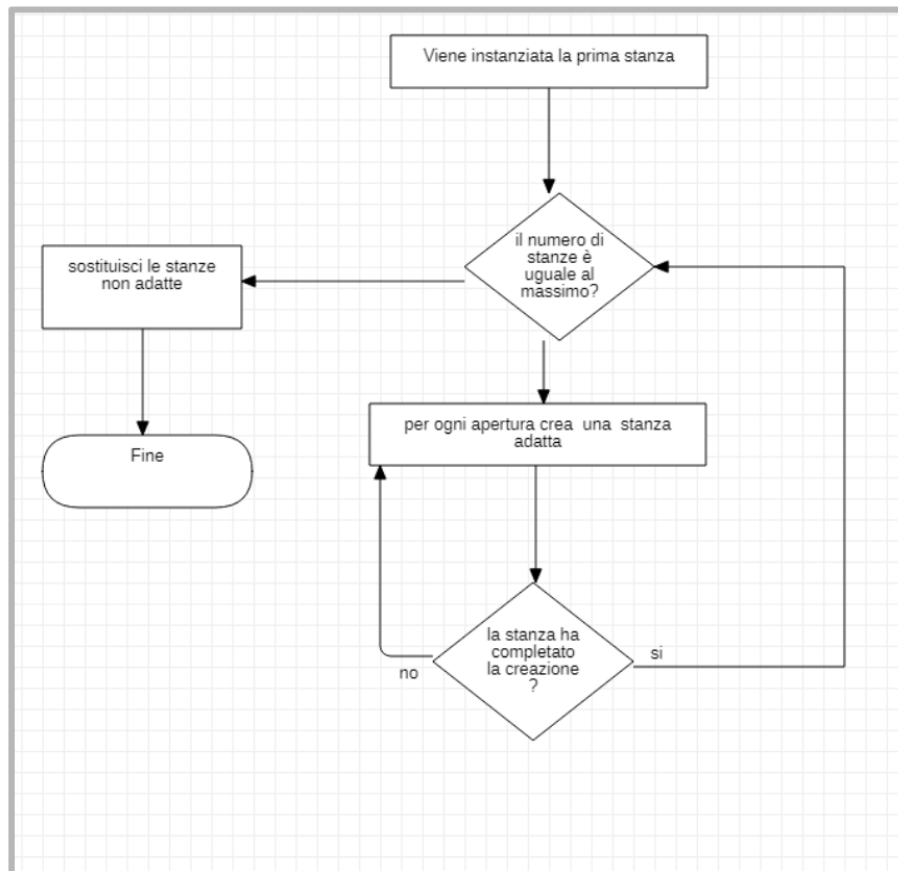


Figure 4 . Depiction of a Room and its Hierarchy.

3.2 Dungeon Generation Flow

The algorithm begins with the creation of the first room in position (0,0,0); this first room is taken randomly from one of the Rules rooms. Once instantiated, it is checked if the maximum number of rooms has been reached, otherwise it is passed to create, for each room opening, a valid room (that satisfies the conditions) and consequently the corridor in the middle between them; since each room has a fixed distance from its neighbor, the corridors have fixed size; each instantiated room is inserted in a list called Created Rooms. When the room x has completed creation, i.e. each Connection has created a room, it is checked if the number of rooms has reached the limit, otherwise, the index is increased and the cycle is repeated until the maximum is reached. Once the maximum is reached, the replacement is implemented.

figure 5 .Dungeon Generation Flow



4. Room generation under constraints

The algorithm proposed in chapter 3 obviously does not take into account many aspects such as, for example, the case in which a room to be generated has neighbors, or, the fact that some Connections of the room already have a neighbor. This chapter focuses on solving these problems.

4.1 Generating a room in the presence of neighbors

4.1.1 The Problem

As has been explained in previous chapters, rooms are instantiated at a fixed distance from the generating room; this leads to a new problem, namely the situation that a new room has neighbors. The neighbors are rooms that have a distance between them and the new room equal to the distance of the generating room with the new room. The system is obviously stable and must be able to reach the exit from every room, so the new room that must be created must satisfy not only the constraint imposed by the opening of the generating room but also the constraints imposed by the openings that face the new room. neighboring room: $\text{distance}(\text{neighboring room}, \text{new room}) = \text{distance}(\text{new room}, \text{room which created it})$

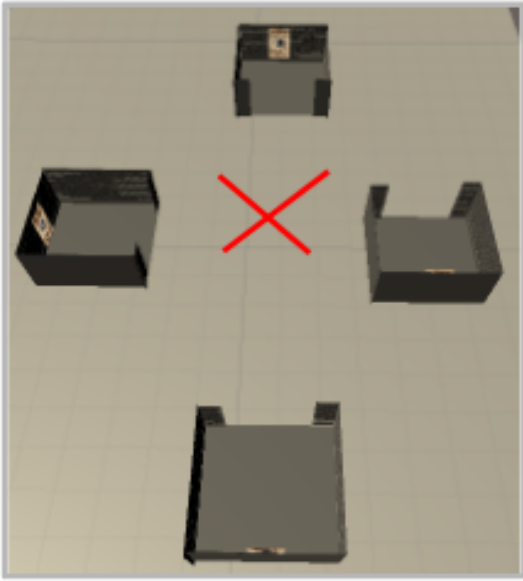


figure 6. The new room(represented by X) has three neighbors.

4.1.2 Rooms Matrix

To solve the problem in the previous section, a matrix of rooms is inserted. This is an $N \times N$ matrix of Rooms; each pair (x,y) represents whether a Room is present at that location or not; the first room is generated in $[N/2, N/2]$. This method is possible because the rooms are at a fixed distance from each other, therefore, rooms are generated at a given position given the generating connection (e.g., the room is generated on the right if a room has the opening to the right). Thanks to this matrix it is possible to know in a very short time ($O(1)$) who the neighbors are and how they are placed with respect to the new room.

tabella 1. Matrice di stanze della figura numero 8.

| x/y | 25 | 26 | 27 |
|-----|------|-------|------|
| 24 | null | South | null |
| 25 | Est | null | Nord |
| 26 | null | Nord | null |

4.1.3 Finding a Suitable Room

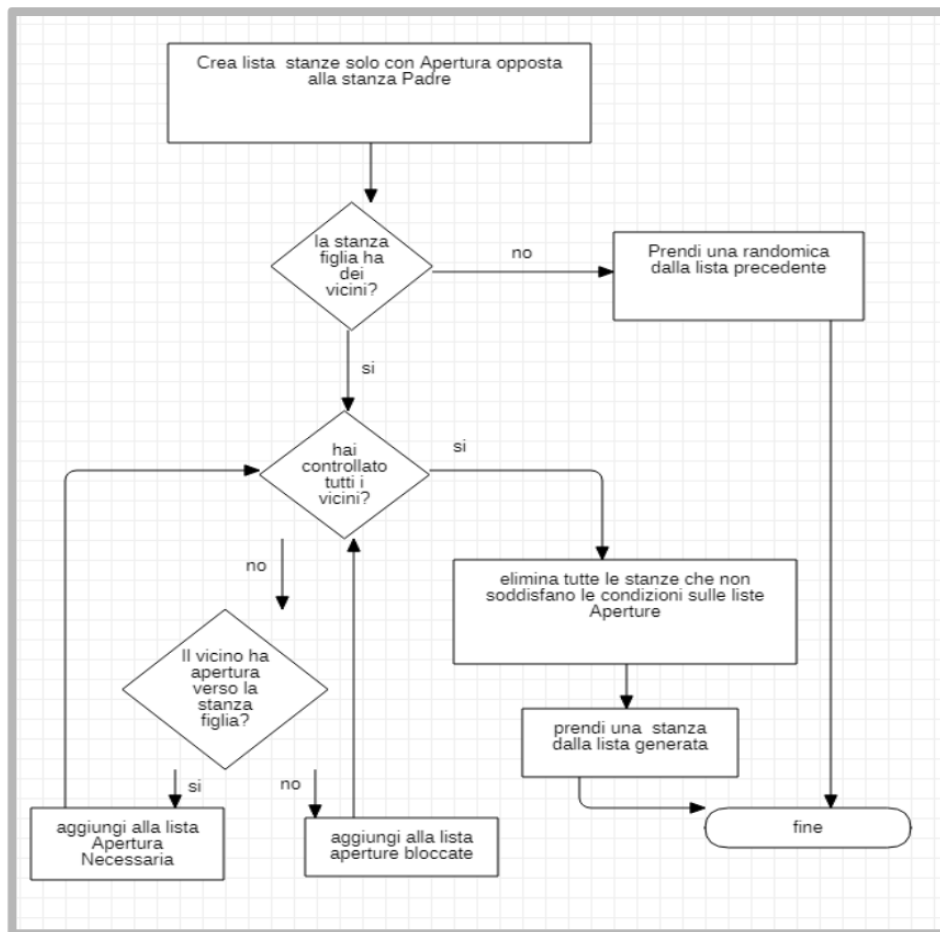


Figure 7.Flow of generating a suitable room.

The image above refers to Figure 5., more precisely in the part of for each opening find a suitable room. Thanks to the matrix, we can now find a suitable room to instantiate.

The process starts that, given a connection, it is compared with all rooms in rules and only the rooms that satisfy isOpposite() method of ConnectionsPoints are selected; these rooms are added to a List. Next, the new room is checked to see if it has any neighbors and, if none are present, it is instantiated from the previously created list because they are valid.

If not, the NeighBorManager has the task of checking whether or not the neighbors have an opening facing the new room. If they answer affirmatively then the opposite of the opening that the neighbor has is added to the Connection Needed list (e.g. if the neighbor has the opening facing north, south is added); if not, it will be added to the Banned list. Once these lists are created, starting from the previously created Room List, all rooms that do not meet the conditions in Connection Needed and Connection Banned are eliminated. The remaining rooms are the valid ones, so one of them will be taken and instantiated.

Example:

Again taking Figure 8 as reference:

1. Opposite Room
List:[West, Southwest, Northwest, Eastwest, NoEast, NoSouth, NoNorth, All].
2. Necessary Openings List: [North, South]
3. Blocked Openings List:[East]
4. List of Final Rooms:[NoEst]
5. Room to be created : NoEst

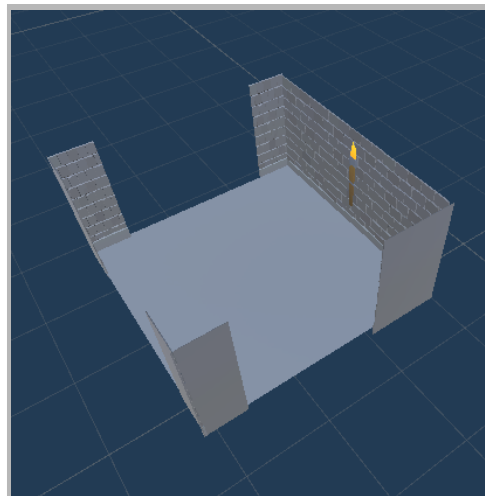
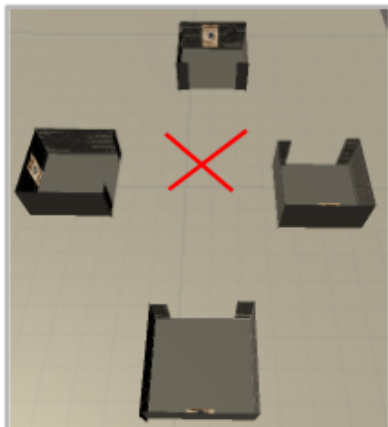


figure 8. On the left an example of generation in the presence of neighbors. On the right the room generated considering the constraints

4.2 Control of a Connection that has its neighbor instantiated

So far, we have only considered the case where a connection in a room does not already have a neighbor. In the case where a connection already has its neighbor, derived from the fact that another room has instantiated it, this is not a problem because the neighboring room generated before had to check its neighbors and as we have seen, the openings to the room are checked. Those openings, however, are marked as having already instantiated their room, so they only need to instantiate the corridor between them if not present.

4.3 All rooms instantiated but rooms not reaching the required number

In certain construction cases, it can happen that the construction stops before the required number is reached; this is due to the fact that many rooms with few openings are instantiated, which obviously means that the creation is not long-lived.

In this case, the `ReplaceRoom()` function of `DunegonManager`:

takes the first room in the `CreatedRooms` list that has only one neighbor and has only one opening is replaced by a room that has four openings

the algorithm presented in section 3 restarts from this last room

5.Substitution of unsuitable rooms

In this chapter the last part of the algorithm will be presented, that is the replacement of unsuitable rooms with valid rooms. By unsuitable rooms we mean the rooms that when the maximum number of rooms is reached, they have not completed creation (i.e. every opening of that room has not completed creation); in this case, the task is given to the ResizeManager which takes care of replacing these rooms.

As we can see from figure 10, the ResizeManager checks whether the created rooms have completed their creation or not. If not, the rooms are replaced; first, the connections that have completed creation are put into a list; second, the room is replaced with one that has only the connections from the list.

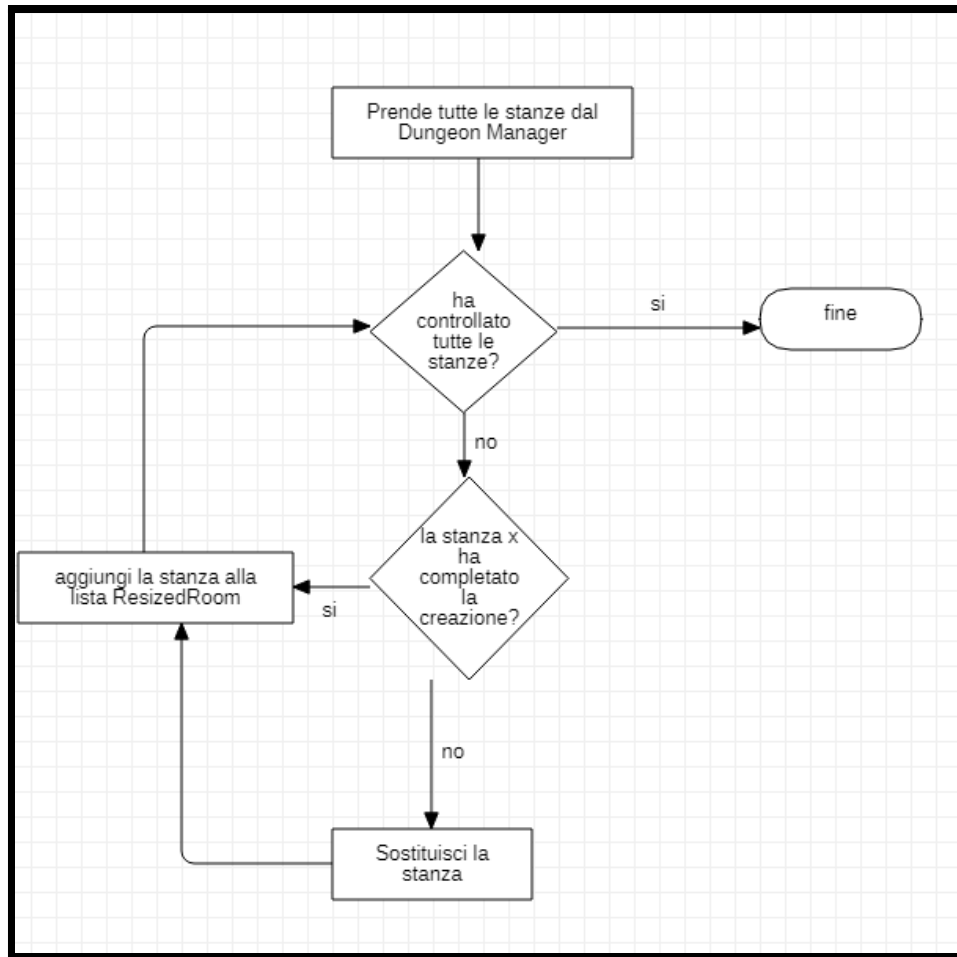


figure 9. ResizeManager Flow for Room Replacement

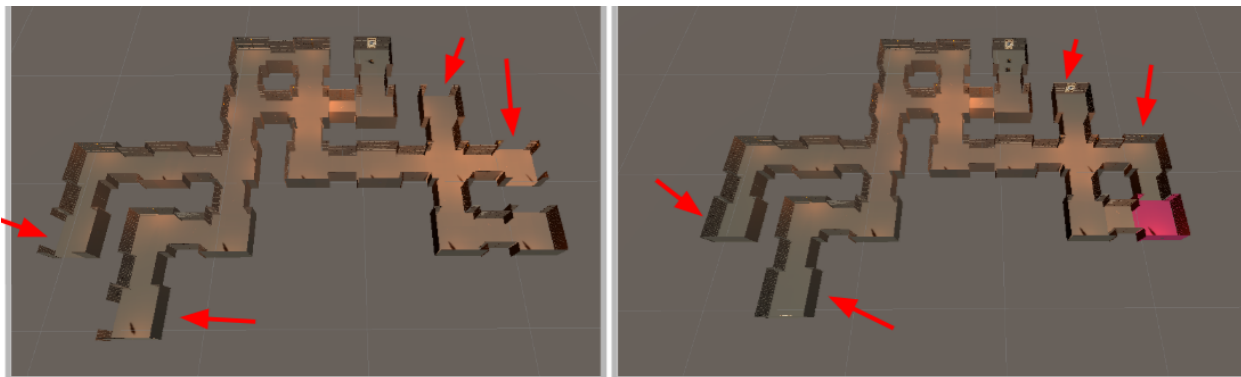


figure 10. In the left figure the dungeon before the substitution. In the figure on the right the dungeon after the substitution.

6.Results

This chapter will show the algorithm creation time based on the number of rooms generated.

Estimation of Algorithm Time:

return valid rooms from Rules: $O(n)$; with n =number of rooms in rules;

search for neighbors: $O(1)$;

eliminating rooms that do not satisfy neighbors: $O(m)$ with $m \in [1, n]$;

instantiate rooms for each opening: $O(n*m*a)$; with $a \in [1, 4]$;

instantiate all rooms : $O(\max*n*m*a)$ with \max =number of rooms created;

replace valid rooms : $O(\max)$ with \max =number of created rooms;

dungeon creation $O(2\max*n*m*a) \sim O(2\max*n) \sim O(2\max)$ with $\max \gg n$;

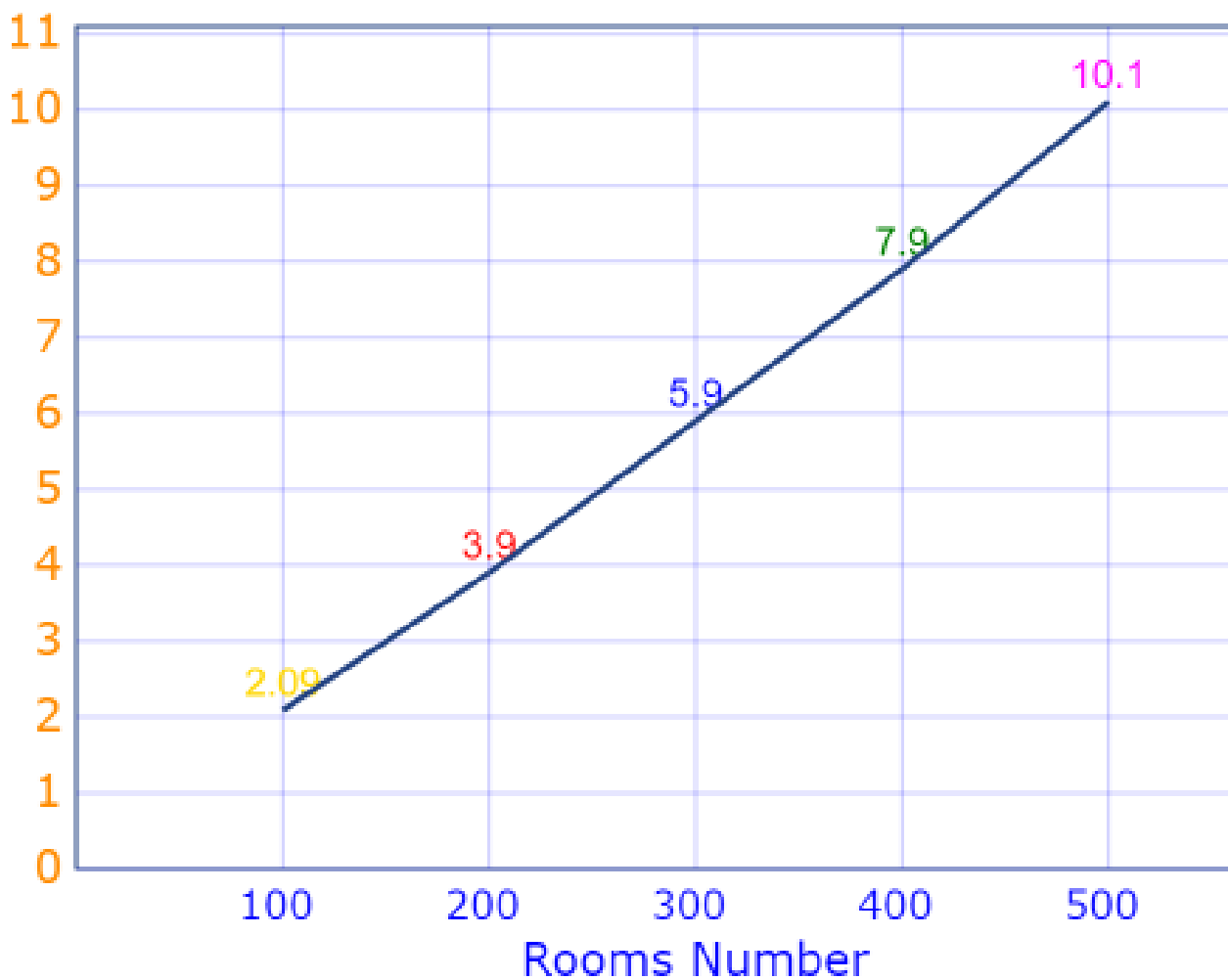


figure 11 Creation time when varying the number of rooms

Figure 11. shows the creation time as the number of rooms varies, and as we can see, the time increases linearly as the number of rooms increases; these results are an average after 50 trials for each number.


The results are consistent with what I expected because the algorithm is only affected by the amount of rooms it has to instantiate and not by other factors dependent on the number of rooms.

7. Conclusion

Tests were performed on an MSI-GL7595E with Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz processor, 16.0 GB Ram. The project was implemented on Unity platform version 2019.4.17 f1 with Windows 10 operating system.

In conclusion, this system allows to quickly obtain a dungeon of rooms that can be traversed in its entirety, without having blocked roads; this system is made possible thanks to a creation control placed on the agent.

8. Bibliography

1. Shaker N., Liapis A., Togelius J., Lopes R., Bidarra R. PCG Book: Chapter 3 Constructive generation methods for dungeons and levels (DRAFT)
2. Baghdadi W., Shams Eddin F., Al-Omari R., Alhalawani Z., Shaker M., Shaker N.: A Procedural Method for Automatic Generation of Spelunky Levels
3. <http://pcg.wikidot.com>
4.  Dungeon Crawler Using Procedural Generation in Unity : per realizzare l'algoritmo