Artificial Intelligence for Video Games

End course project (A.A. 21-22)

Professors: Dario Maggiorini, Davide Gadia

Candidate: Alessio La Greca, 990973

**Index**

**Introduction**

**Chapter 1: The dungeon generation**

* 1.1: Parameters for customizing the dungeon
* 1.2: Data structures
* 1.3: The MonoBehaviour that builds the dungeon

**Chapter 2: The graph generation**

**Chapter 3: Flocking behaviour**

**Conclusions: what could have been done better**

**Introduction**

**In this documentation, you’ll find all the logic behind this project, that could be classified as:**

***Moving flock in a procedurally generated dungeon.***

**The main idea behind this project is divided in three steps, and each of them uses one (or more) of the many topics covered in the Artificial Intelligence for Video Games course of the University of Milan:**

1. **Procedural Content Generation of a dungeon using Space Partitioning Algorithm.**
2. **From Map to Graph: how to convert the dungeon in a graph, that will be used to find a path from a starting room to an end room. The search is done using A\* algorithm.**
3. **Use of a flock and combination of different Steering Behaviours to simulate bats that move in a cave.**

**The project has been developed Unity (the editor version is 2020.3.25f1), and this documentation should be considered both as a manual for Professors Dario Maggiorini and Davide Gadia, and as a general documentation for whoever is interested in taking a look at this project. All the magic happens in the directory Assets/LabyrinthPCG/LabyrinthV4, so all the paths I’ll mention will be relative to this one.**

**Chapter 1: The dungeon generation**

* 1. **– Parameters for customizing the dungeon**

The reason I chose to start this project with Procedural Content Generation, and in particular the Space Partitioning Tree algorithm, is that I’m a fan of Roguelikes myself, and captivated by algorithms that generate random dungeons at each run, like the ones in *The Binding of Isaac*. But what I also like about algorithms is generality: that is, the possibility to tweak some input parameters to get valuable outputs. A good example of this is the qsort algorithm of the standard C library, which allows the programmer to specify a function used to compare two elements.

So, I wanted to achieve a good degree of freedom when generating a dungeon, specifying a lot of things that would have allowed me to obtain different dungeons not only because of the random seed used, but also because of the different input parameters. So, I came up with the following parameters, that can be customized in the Unity Editor.

The scene in which the dungeon generation (and actually the whole project) takes place is PCGLabyrinth4.unity, in which we have an object called LabyrinthGenerator4 with a MonoBehaviour attached, that is LabyrinthGenerator4Animated.

Furthermore, keep in mind that for this project the coordinates work as follows: the horizontal axis is the Z axis, while the vertical axis is the X axis, and the (0,0) coordinates are the “upper left point” of our hypothetical plane. So, for example, to reach point (5,3) from (0,0), we have to move 5 steps to the right and 3 down.

So, the parameters are:

-Floor: a pointer to the floor asset we will be using for the floor and roof of the dungeon.

-Unit: a prefab cube used to build the walls of the dungeon.

-WallsMaterial: the material of the walls.

-HeightOfWalls: the height that the walls must have. This changes the Y scale value of the Unit.

-UnitScale: how big the Unit has to be. This changes its X and Z scale values, in order to have bigger dungeons.

-Z0 and X0: the (z,x) coordinates in the game world that represent the upper left corner of the dungeon.

-Width and Height: specifies how many Units the dungeon will have along the Z and X axis.

-SmallestPartitionZ and SmallestPartionX: the dungeon will be generated using a Space Partitioning Algorithm (from now on, SPA), that will divide the given space in two partitions, then four, then eight, and so on. Those two parameters specify the minimum width and height that a partition can have. In this way, we make sure that each room will be generated in a partition that is at least SmallestPartitionZ wide and SmallestPartitionX high.

-RoomsMustBeSeparated: normally, in a SPA the rooms can only be connected to other rooms via a corridor. But why should we limit ourselves to those scenarios? When this value is false, rooms can be generated everywhere inside the given partition space, and so we can have two rooms directly connected one to the other because of the lack of a wall between them. When true, instead, rooms can be connected only through a corridor, and the rooms will always be (at least) two units smaller than the given partition space.

-MinimumRoomZ and MinimumRoomX: the *desired* minimum width and height a room should have. Note the emphasis on “desired”: the algorithm will always prefer rooms bigger than those width and height, but if the given partition is not big enough to contain such a room, the generated room will be the biggest possible for that space.

-RandomSeed: to allow repeatability.

-Minimum/Maximum HorizontalCorridorWidth: the minimum width and maximum width a horizontal corridor can have.

-Minimum/Maximum VerticalCorridorWidth: the minimum width and maximum width a vertical corridor can have.

-DelayInGeration and Animated: if Animated is set to true, the dungeon creation will be done step-by-step, allowing the users to see how it is actually created. Each room and corridor is created *DelayInGeration* seconds after the previous one, so that one can decide the speed of the animation.

Given those parameters, it should be noted that not all the possible combinations are possible. In fact, I made some assumptions to simplify my work:

-MimimumRoomZ and MinimumRoomX must be greater than 3. This means that all rooms will be wide and high at least three units. I did this to make sure that the boids have enough space to spawn (even though, it’s still possible for a user to “break” this constraint. More details in chapter 3).

-MinimumHorizontalCorridorWidth must be less equal than MinimumRoomX, same goes for MinimumVerticalCorridorWidth and MinimumRoomZ. This ensures that it is always possible to dig a corridor from one of the walls of a room.

-MaximumHorizontalCorridorWidth must be less equal than smallestPartitionX, same goes for MaximumVerticalCorridorWidth and smallestPartitionZ. This ensures that, when we dig a corridor from the wall of a room, we don’t accidentally screw up the space given to an adjacent partition (with which we don’t want to connect right now).

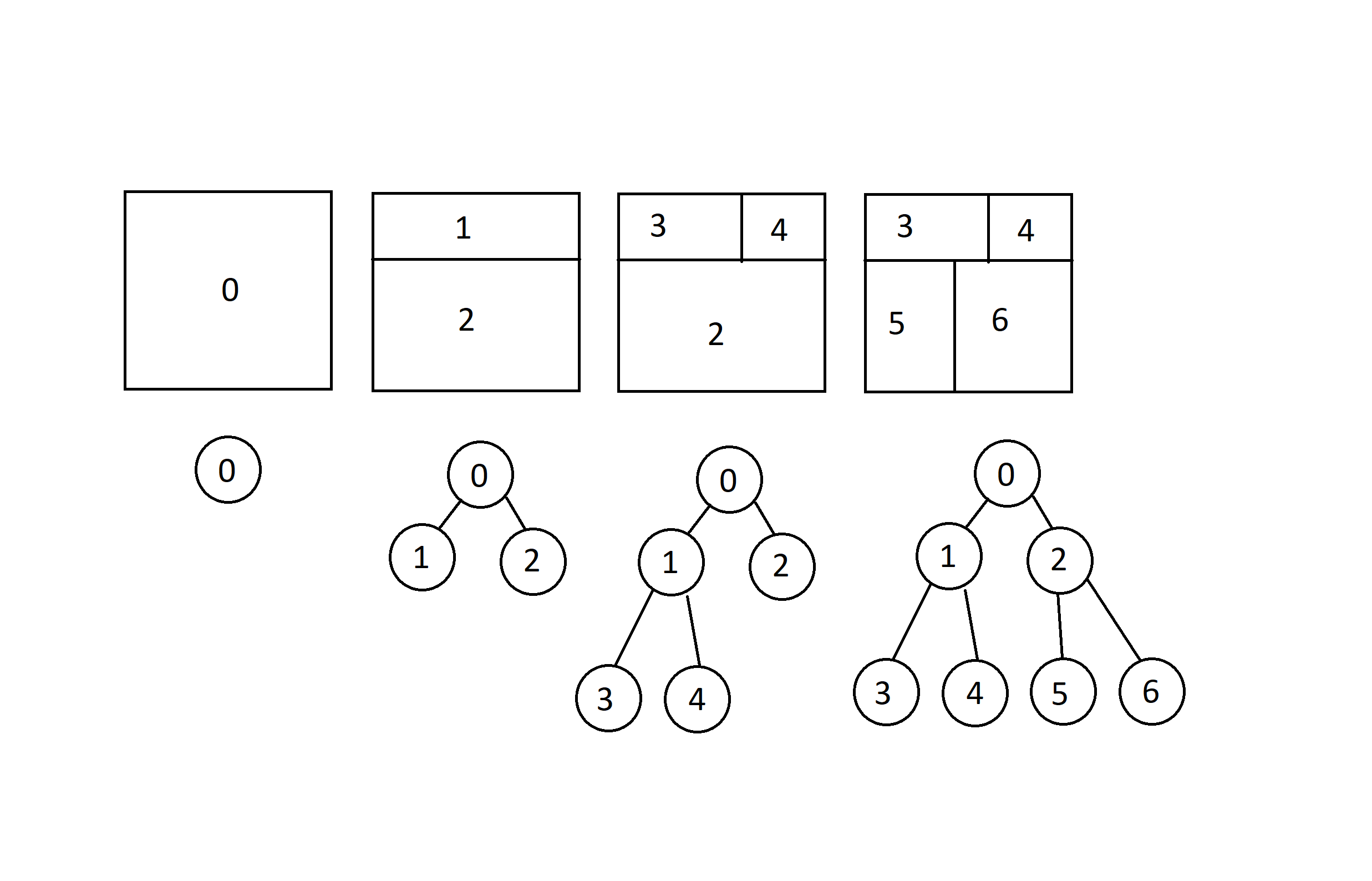
-Of course, the minimum width for both corridors must be less equal than their maximum width.

**1.2 – Data Structures**

In order to understand how the dungeon is created, we need to take a look at the data structures used in the process. They are located in DungeonGeneration/PartitioningTree4.

* Class PTConstants (Partitioning Tree Constants): just a class that has two integer constants that associate a horizontal cut and a vertical cut to an integer, where by “cut” we mean a line that will separate two partitions in the to-be-generated dungeon. I did this and not hard-coded those values to allow future generalizations, as an oblique cut.
* Class Square: a class that represents a square of the dungeon. With square, we literally mean a tile, since our dungeon will have a rectangular shape and will be made of small cubes (the Units), that will compose its walls, and empty tiles, which are the tiles for the rooms and corridors. This class has two fundamental properties: z and x, that together represent the square we are considering of the dungeon. For example, (7,4) represent the tile in the dungeon that is 7 steps to the right and 4 down in respect to the upper left. Also, note that those values have *nothing to do* with the space covered (or not) by the actual Units: the dungeon can have width = height = 100 and UnitScale = 5. This means that, in the actual space, the dungeon is 500 meters wide and 500 meters high, but for the computations that use squares it is still a dungeon made of 100x100 squares! The translation from this abstraction to actual game world coordinates will be covered in Chapter 2.
* Class Node: since we have to run a SPA, we will have to build a binary tree, that is composed of nodes. Each of them represents a partition. To represent those nodes, we use this class. In particular, this class holds the following attributes:
  + Square p1 and Square p2: the upper left and lower right squares that represent the space for this *partition*. Note that p2 is **exclusive**, so the actual partition goes from p1.z and p1.x and ends with p2.z -1 and p2.x -1.
  + Square room\_p1 and Square room\_p2: the upper left and lower right squares that represent the space of the *room* inside of this partition. Also in this case the second square is exclusive. It should be noted that, while in the leaf nodes this two points give an area that is surely empty, on all the other nodes those two points simply give the extremes of the area that contains some empty space (two or more rooms and the corridors that contain them) and some occupied space (walls).
  + Int CutOrientation: variable that holds the nature of the cut. In our project, can be only vertical or horizontal (actually, this variable can have one of the constant values of the PTConstants Class).
  + Int CutWhere: an int value that tells us where the cut was done. Depending on the value of CutOrientation, this will be interpreted as a coordinate on the z or x axis.
  + Node Parent: a pointer to our parent node, that is, the partition that was split in two to create this partition and another one.
  + Node left\_child and Node right\_child: pointers to the partitions originated by splitting this one.
* Enum Directions: an enum composed of four possible values, up, down, left and right, that will be used to know in which direction to dig for building a corridor.
* MyUtility: a class that contains a static function boolContains, that takes as inputs a Boolean array and a Boolean value and that returns true when the array contains at least one element with value “value”. It can be thought as the “if true/false in list\_of\_bools” statement of python.

Below an image of how a tree would be built. Each Node will “remember” the upper left and lower right coordinates (Squares) of its partition, and when a room will be created, also its two Squares will be conserved in each Node.



**1.3 – The MonoBehaviour that builds the dungeon**

The actual magic takes place in the LabyrinthGenerator4Animated script, attached to the LabyrinthGenerator4 object. It is also here that the parameters specified on Chapter 1.1 are located. The code is extremely (at least, I think it is) commented, and tries to go into the details of almost every line of code. So, to avoid duplication, I will explain here at a higher level what actually happens.

After checking that all the constraints of the simplifications are met, it instantiates the floor and the roof of the dungeon, properly scaled and at the right heights.

Then, a call to **generatePartitioningTree()** is made. This is a function that returns a Node, the root of the Partitioning Tree, by calling a recursive function, **generateNode()**. The first function, in fact, generates two Squares: one that represents the upper left Square of the dungeon, another that represents the lower right Square of the dungeon plus one (we always reason about the Squares considering the lower right one as exclusive, remember).

*generateNode()* takes as parameters the two Squares representing the current Node and a pointer to the parent Node. When the first call to this function is made, this last parameter is null, since this is the root Node.

Then, an object Node is generated, called *currentNode*, that will represent this partition, and both the width and the height of the partition are calculated.

Then it is checked if it is possible to do a horizontal or a vertical cut, comparing the half of width and height with the smallestPartitionZ/X. If none of them is possible, the left and right child of the currentNode are set to null and this Node is returned. If instead only a cut is possible, that one is chosen, and if both are possible, one of the two is chosen at random.

After choosing the direction of the cut (horizontal or vertical), a random coordinate (of the Z or X axis respectively) in which the cut will take place is chosen. The coordinate is chosen in such a way that the two sub-partitions will be bigger than smallestPartitionZ/X. The coordinate of the cut is then saved in the currentNode, along with its direction of course. At this point, the two children can be generated recursively: calculate the upper left and lower right Squares of the two sub-partitions and call *generateNode()*. In the end, return currentNode.

Now that we have the partitioning tree, we can create the rooms inside it. Before doing that though, we need a representation of the dungeon: I decided to implement it as a bitmap- well, technically, it’s not a bitmap, just a 2-dimensional array, but the only values it can have are 1 or 0. If bitmap[i,j] = 1, it means that in the Square of coordinates i,j there is a wall. Otherwise, that square is empty, either because it contains a tile of a room or a tile of a corridor. Together with this bitmap, I also initialize a matrix of GameObjects (initially set to null) of the same dimensions, that will allow me to have control over the generated Units that will represent the actual dungeon.

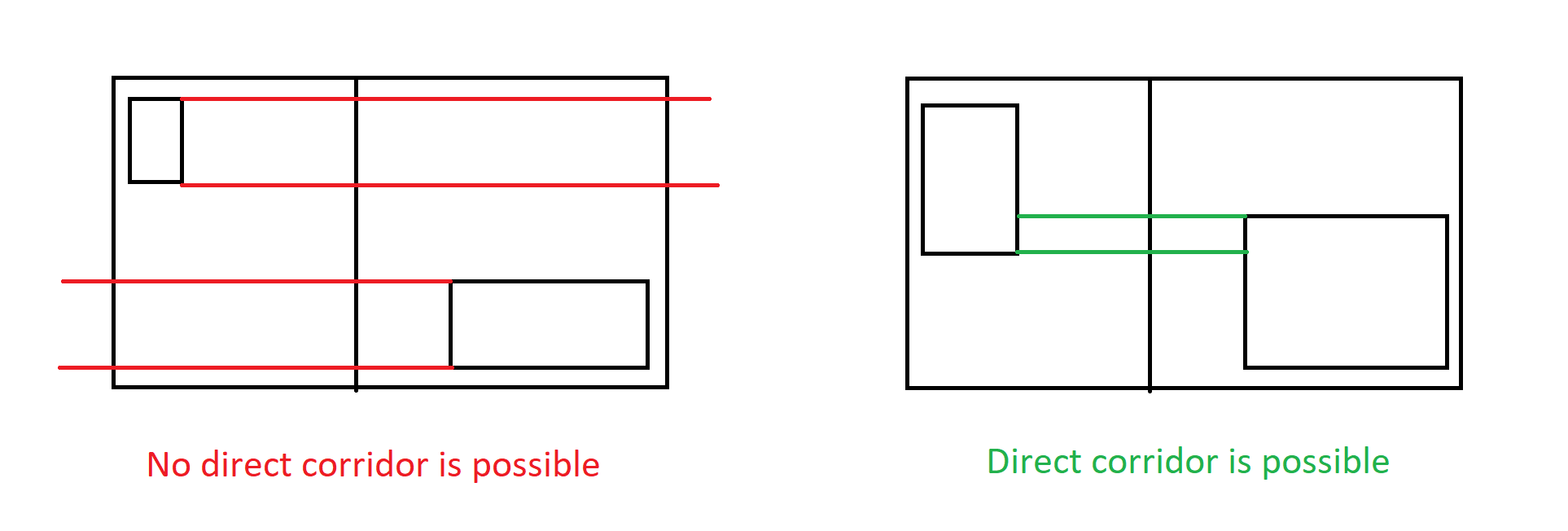
Then, a call to **generateRooms()** is made. The root Node of the previously calculated partitioning tree is passed as argument, and this function simply calls another function, **exploreNodeToGenerateRoom()**, that initially takes as argument the root Node of the tree.

*exploreNodeToGenerateRoom()* is a recursive function. The base case occurs when both children of the current Node are set to null (else the function is called recursively for both children). When that happens, it means we have found a leaf Node, that is, a partition in which a room must be generated.

To create a room, we need the following informations: the upper left coordinates of the room and its width and height. To get them, we perform the following steps:

1 We first take the width and height of this partition.

1. We subtract two to those values if the roomsMustBeSeparated flag is set to true, to ensure that the room is “wrapped” inside this partition.
2. Now we have to be careful. We have to imagine that this room will have to be connected thanks to a corridor to another partition. We assume that all the corridors generated need to be direct: we can’t have L-shaped corridors (I mean, *technically* we can, but good luck in doing that: I spent a lot of time trying to figure out how to do it in a meaningful way, but with no avail. Knowing my stubbornness though I’ll probably try to do it again in the future). So, to make sure that this direct corridor will “hit” the adjacent room, we need this room (and this applies to every room) to be at least big half of the given partition plus one. See the image below for a visual representation. At this point one might ask: so, even if I specified a minimum Width of 3 for a corridor, in some situations I’ll have to stick with a corridor large 1? No, but we’ll see this topic in a minute during when we’ll talk about how corridors are actually generated.



1. Then, the upper left coordinates of the room are generated using **obtainStartingCoordinateForRoom(),** that given the available space in the partition and the constraints specified, returns the z and x coordinates requested.
2. Last things to calculate are the width and height of the room. These are calculated by **obtainLengthForRoom()**, that similarly to the previous function uses the upper left coordinates and the constraints to calculate the actual dimensions of the room.
3. We save in the current leaf node the coordinates of the two points representing the room
4. We set to 0, in the bitmap, all the values between the two coordinates of the room.
5. We make a call to the function **addNode()**, that will save in another MonoBehaviour the *center* of this room. We’ll talk about this in Chapter 2.

Last but not least, we have a call to **generateCorridors()**, that once again takes the root Node as parameter