Artificial Intelligence for Video Games

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

Professors: Dario Maggiorini, Davide Gadia
Candidate: Alessio La Greca, 990973

Index

- 5.1: Quadtree to have less walls
- 5.2: A zip around the dungeon
- 5.3: Help lost boids find their way (edit: solved!)

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 (or just a flock) that move in a cave.

The project has been developed in 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

1.1 – 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 one 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 4).

-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 with 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.

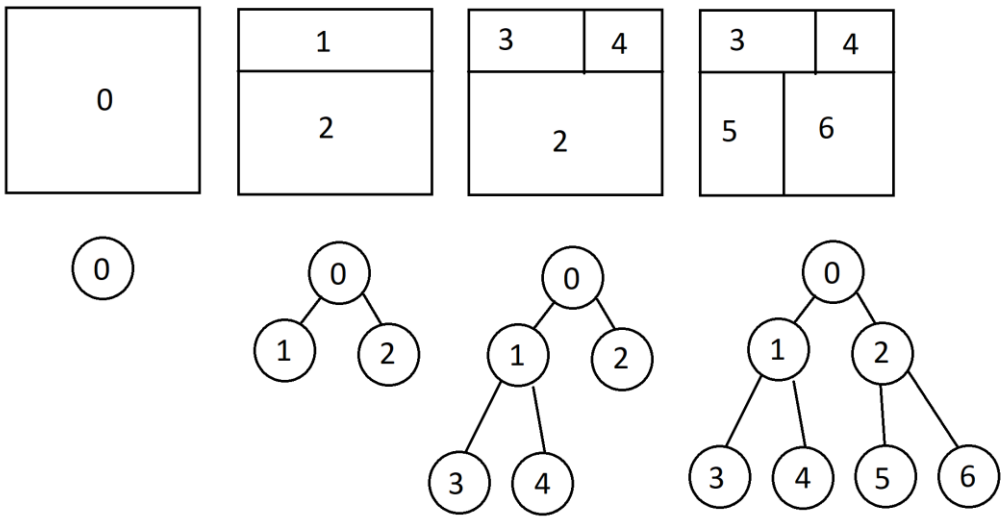
- Class PTConstants (Partitioning Tree Constants): just a class that has two integer constants that associate a horizontal cut and a vertical cut to an integer, where by "cut" we mean a line that will separate two partitions in the to-be-generated dungeon. I did this and not

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 connect 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()* for both. 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.

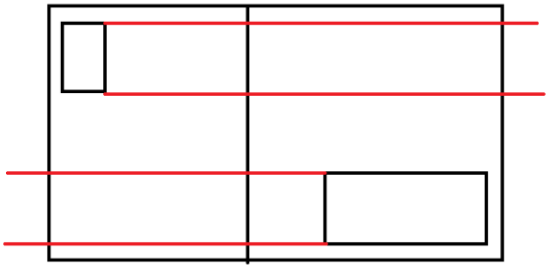
exploreNodeToGenerateRoom() is a recursive function. The base case occurs when both children of the current Node are set to null (else the function is called recursively for both children). When that happens, it means we have found a leaf Node, that is, a partition in which a room must be generated.

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:

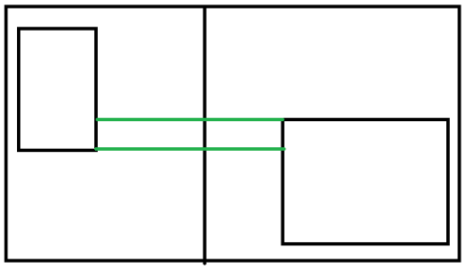
1 We first take the width and height of this partition.

2 We subtract two to those values if the roomsMustBeSeparated flag is set to true, to ensure that the room is "wrapped" inside this partition.

3 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 when we'll talk about how corridors are actually generated.



No direct corridor is possible



Direct corridor is possible

4 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.

5 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.

6 We save in the current leaf node the coordinates of the two points representing the room

7 We set to 0, in the bitmap, all the values between the two coordinates of the room.

8 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.

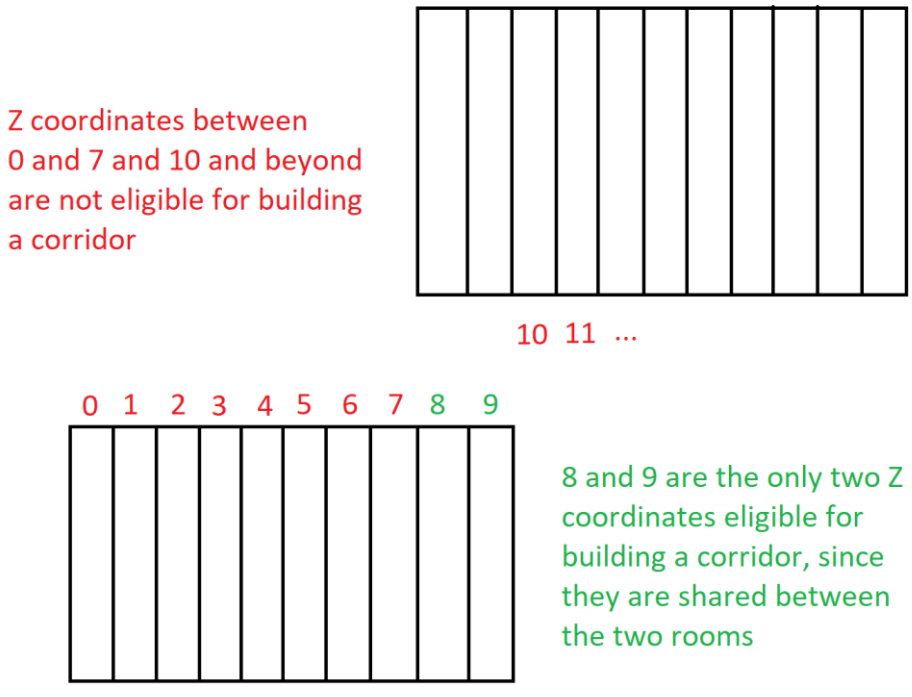
Last but not least, we have a call to **generateCorridors()**, that once again takes the root Node as parameter. This time though what it returns is an array of Squares, that will always contain two Squares, that represent the upper left Square and lower right Square of the newly generated room (newly generated because it is the union, thanks to a corridor, of two rooms).

Base case: the current Node passed as parameter is a Node representing a room (so it's a leaf Node), then no corridor is needed. In this case, the two Squares representing this room are returned.

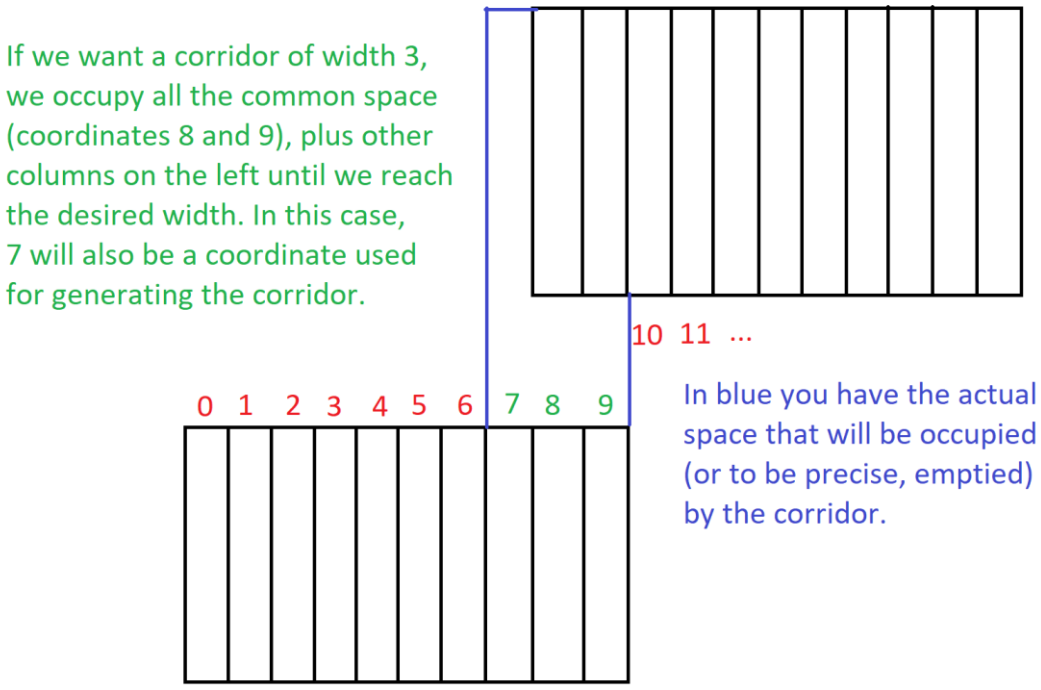
In the recursive case, instead, two recursive calls to *generateCorridors()* are made. The reason behind this is that, in order to generate a corridor between two rooms, we need to know the coordinates of these two rooms, that can be obtained through two recursive calls to the left child and right child.

Then the following steps occur:

1. The top left and lower right Squares of the sum of these two rooms are identified and saved in the current Node.
2. The CutOrientation of the current Node is identified (horizontal or vertical). For this explanation, we will assume that the cut was horizontal, and so a vertical corridor must be generated. The case in which the cut is vertical follows the same idea.
3. The width of the corridor is determined by using the constraints specified in the editor.
4. Then the two rooms are "scanned" through the Z axis, and a set of "common coordinates" are identified by using the **getZIntersections()** function. The "common coordinates" refer to all the values of the Z axis that can be found in at least one Square of both rooms, see image below.

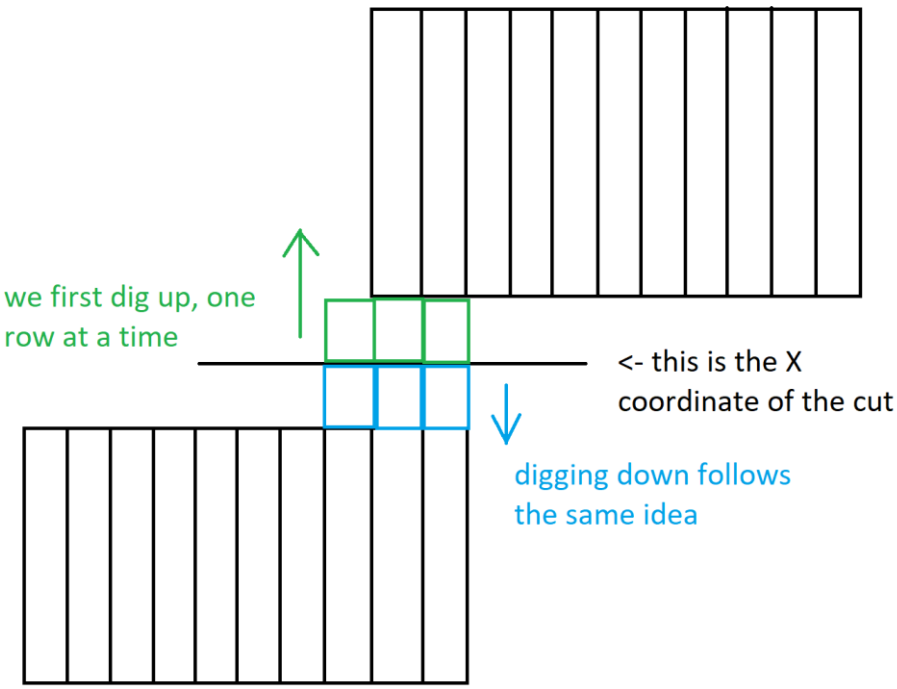


5. A call to **generateDirectCorridorBoundCoordinates()** is made, that provides two coordinates on the Z axis: one that represents the left wall (inclusive) of this corridor, and the other that represents its right wall (exclusive). To calculate those values, the function takes into account the right wall of the room “on the left” (see image above), the required width that the corridor must have (clamped in such a way that it will be smaller than the room width) and the common coordinates of the two rooms. What it tries to do is to generate a corridor of the required width in the common space. But what if the common coordinates are just 8 and 9 like in our example and the required width for the corridor is 3? This might happen if the minimumVerticalCorridorWidth is 3.
   Well, in cases like this, another approach is taken: the corridor will occupy all the common space between the two rooms, plus some columns on the left: as many as necessary to have the requested corridor. See image below for a visual representation.



6. Now that we have the left wall and right wall of the corridor, we actually have to build it. To do it, a call to

generateVerticalCorridorFromCut() is performed, that calls two times another function: **digVerticallyInSearchForRoom()**, once asking to dig up, another asking to dig down.
But let's step back a bit: how are the corridors actually built? The main idea is the following: since we have the X coordinate of where the cut was done, and we know that one room is above the other, we can start "digging" the corridor from the cut going upwards and downwards until we find the two rooms. We dig one row at a time, and based on what we have in front of us we decide what to do.



Let's consider more in detail the *digVerticallyInSearchForRoom()* function, assuming we want to dig upwards. First, before actually "digging", that is, setting to 0 the appropriate values on the bitmap (which is performed by the function **Dig(z,x)**), we check the row that we want to dig, to see if there is a wall in front of us, a completely empty space or some of both.

- The row is completely empty? Then we have in front of us a room, and no more digging is required. Also, a call to **addNode()** is performed, that will save the *center* of this row, that please note that is a Square inside of the room that leads to a corridor.

- The row is partially empty and partially full? Then, it means that we have found a room, but in order to have a corridor of the required width, we still have to dig some more in the occupied space. We then remember that now we are digging on the edge of a room. Also in this case a call to **addNode()** is performed, just like before.
- Are we digging on the edge of a room and we have found a row that is completely occupied? Then we have finished digging.

If none of the above conditions apply, a call to **Dig()** is performed, and the currently checked row is dig, meaning all its bitmap values are set to 0.

Since this is done recursively, in the end all the rooms will be connected in some way, and we will have our dungeon in form of a bitmap.

To translate the bitmap in actual gameObjects, the coroutine **drawLabyrinth()** is executed, that calls the function **fromBitmapToDungeon()**. This function simply checks all the values in the bitmap and spawns in the corresponding position (with respect of course to Z0, X0, unitScale and heightOfWalls) a Unit if bitmap[i,j] = 1. This means that the number of gameObjects spawned (and conserved in a matrix of gameObjects, *wallsArray[,]*) are a O(width*height). In the conclusions we will discuss a better approach that could be taken.

The very last function invoked is **generateBitmapsForGraphGenerator()**, that calls **generateCorridorsBitmap()** in order to generate a new bitmap starting from the one representing the final dungeon. The generated bitmap, *corridorBitmap[i,j]*, is simply the bitmap of the final dungeon where all the Squares belonging to a room are set to 1 instead of 0. It's basically a bitmap where the only "empty" spaces are those occupied by a corridor. This bitmap, together with the one representing the final dungeon, are passed as arguments to ***setBitmaps()*** of the

GraphGeneratorAnimated component, belonging to the scene object called *GraphGenerator.* We will now discuss this in the following Chapter.

Chapter 2: The graph generation

2.1 – Data Structures

Since we want to abstract our dungeon as a graph, that will allow us to find a path with A*, we need some classes to represent nodes, edges and the graph itself. They are located in the directory GraphGeneration, and all of them are based on the code provided by Professor Dario Maggiorini and Professor Davide Gadia:

- Class GNode: a class to represent a point in space. Note that, while before we talked about bitmaps that abstracted the effective width, height and scale of the dungeon, in this case the z and x attributes of the class represent the actual z and x coordinates of the GNode in the unity world! Other than the coordinates, this class is composed of three Booleans:
    - Is_room: set to true if this GNode represents the center of a room.
    - Is_corridor_entrance: set to true if this GNode represents a point in a room that, once crossed, takes us into a corridor.
    - Is_intersection: set to true if this GNode represents a new point that we “think it’s important” to have a meaningful graph. More details later.
- Class GEdge: a class to represent an edge connecting two GNodes. It is composed of two GNodes, called “from” and “to”, that represent the two GNodes that this GEdge connects, and a weight, that represents the distance from one GNode to the other (or whatever you want this weight to be).
- Class Graph: a class to represent the graph. It is equipped with a Dictionary<GNode, List<GEdge>>, and some methods to query or write some values of the dictionary:

- AddEdge: to add a GEdge to the Graph
- getConnections: to get all the outgoing GEdges from a certain GNode.
- getNodes: to get all the keys of the dictionary, that is all the GNodes of this Graph, as an array.
- isNodeAtCoordinates: Boolean function that returns true when there is a GNode at the specified coordinates in the Graph.
- getNodeAtCoordinates: function that returns the GNode of the graph at the given coordinates.
- areNodesConnected: function used to know if two GNodes are already connected by a GEdge. Note that the Graph that we will build is a directed Graph where, if two GNodes are connected, then two GEdges are instantiated: one that goes from the first GNode to the second and another that goes in the opposite direction (to simulate an undirected graph basically), and both of them have the same weight. This function, in fact, is only used to speed up the animation process.

2.2 – The addNode function

Let's now talk of how the graph is actually generated. The code I'll be referring to is located in the MonoBehaviour *GraphGeneratorAnimated*, attached to the scene object *GraphGenerator*.

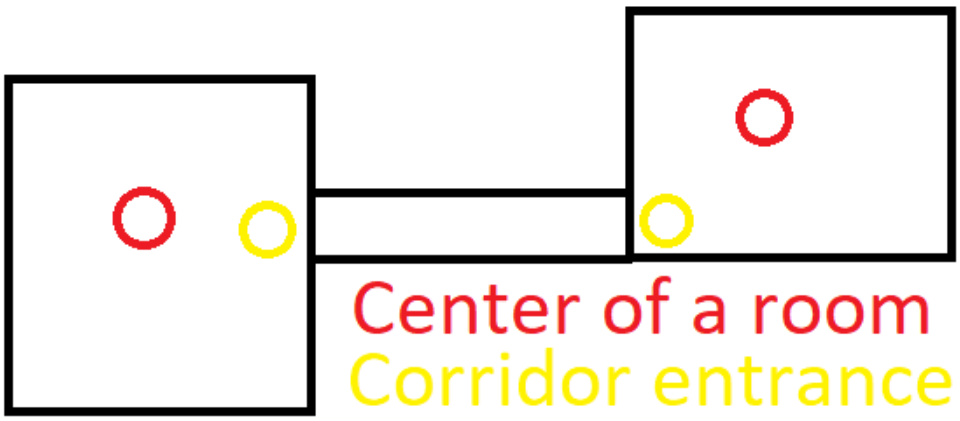
We have mention a few times, in the last Chapter, the *addNode* function, invoked **when a room is created or when a corridor has been completed**. Let's now see what it actually does, and keep in mind: this time, the graph we are going to build will have as (z,x) coordinates in its GNodes the actual game world coordinates of that point.

addNode takes three input parameters:

1. The z coordinate of this GNode.
2. The x coordinate of this GNode.

3. The type of this GNode, that is represented as an int. 0 = those coordinates refer to the center of a room, 1 = those coordinates refer to a corridor entrance.

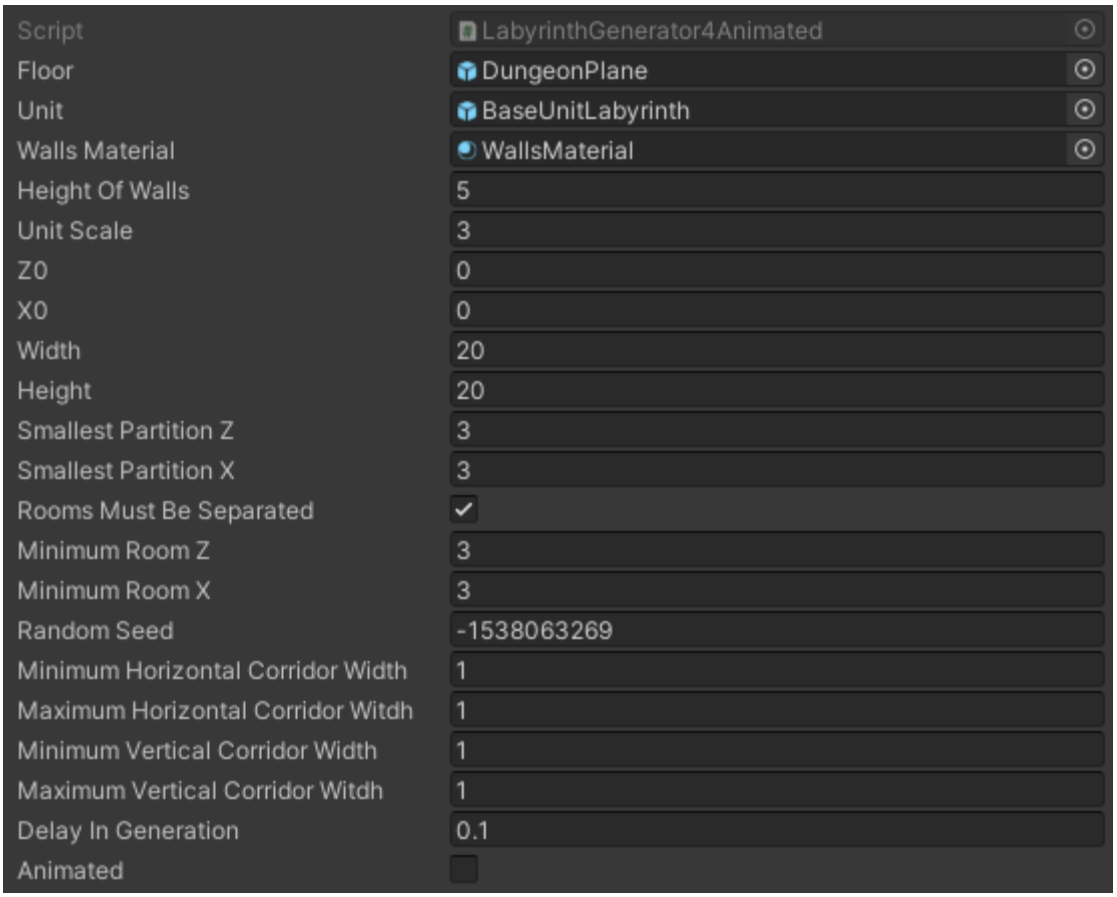
With corridor entrance, we mean a point that is just in front of a certain corridor, see image below for details.



When the function is invoked, the (z,x) coordinates that it receives as inputs refer to the original bitmap. So, the first thing to do is to convert them in world coordinates. This is done using the unitScale value of the *LabyrinthGenerator4Animated* component. Now, depending on the type, we have different behaviours:

- If it is 0, a new GNode is created with those coordinates, and the flag is_room is set to true. Note that ALL the GNodes that represent the center of a room are created BEFORE all the GNodes representing a corridor entrance. This is important.
- If it is 1, it means we are asked to add a GNode representing a corridor entrance. There are some cases though in which a corridor entrance might coincide with a room center. This can happen when the generated rooms are extremely small, and in those scenarios, we don't want two nodes in the same position. So, we first check if there is already a GNode in the graph with those coordinates. If

there is, we simply take that node and set the is_corridor_entrance flag to true. Else, we build a new GNode and add it to the Graph (this also has is_corridor_entrance set to true). In the animated version of the algorithm, I decided to use Red as the color of room nodes, Yellow for corridor entrances and Orange for the GNodes where both flags are true. A seed where this happens:

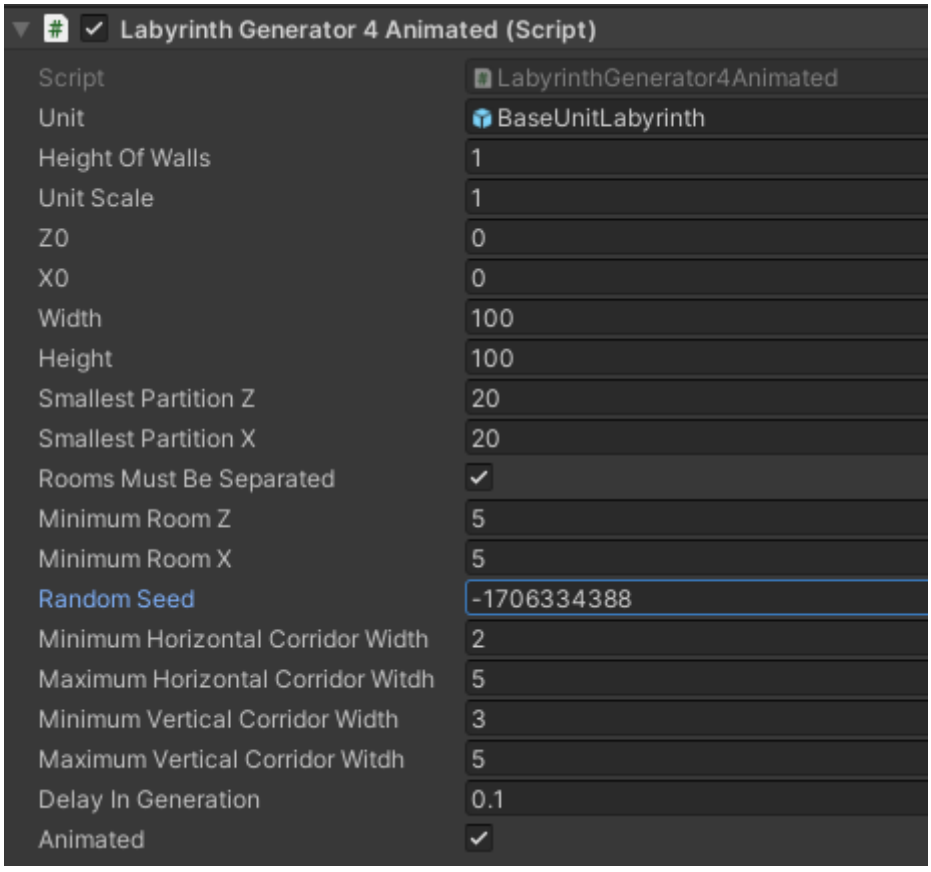


(seed = -1538063269)

2.3 - The setBitmaps function and the intersections

The last thing we did when we created the dungeon was calling the *setBitmap()* function of this component. In this way, we can pass to this script both the bitmap representing the built dungeon and the bitmap that only contains the informations regarding the corridors. Now, the function **findIntersections()** is called. What does it do? It loops on the width and height of the bitmap (doesn't matter which one since they have the same dimensions), and for each cell checks:

- In the corridorBitmap, at the coordinates (i,j), is there an empty space? That is, are those the coordinates of part of a corridor? If yes, check also if it is already occupied by a corridorEntrance (yes, a corridor entrance can be inside a corridor, since two big partitions might be united thanks to a corridor that intersects other corridors).
- If it is not, then check how many GNodes are visible, in the four cardinal directions. To do this, use the dungeonBitmap (the one that represents the actual dungeon) and some support functions (**lookAround(), lookUp(), lookDown(), lookRight()** and **lookLeft()**). In this way, we'll see how many room nodes, corridor entrance nodes and other intersection nodes are visible from here. If this number is >= 3, then a new GNode is created, with the current coordinates and the is_intersection flag set to true. One might ask: is this really necessary? Yes, because else some meaningful GEdges would be lost. See, as an example, the seed below, and imagine the graph without the green GNodes on the left (Green is the material for the intersection nodes).



(seed: -1706334388)

It is also possible to add to the animation process the discarded points of the corridors (those where an intersection wasn't necessary). To do that, de-comment lines 160 and 199 (should I have moved the code and the lines aren't there anymore, just ctrl+f "TO SEE CORRIDORS" and you're good to go). It should be noted that the cubes created in this way do not affect the Graph generation: in will just be harder to see the edges connecting the nodes.

At this point, we have all the nodes, and if the flag Animated has been set to true in the script, they will appear inside the dungeon. We now have to reason about the edges. They are created when the call to **buildEdges()** is performed, inside the coroutine **drawNodes()**.

The generation of the GEdges is pretty straightforward: each GNode of the graph is considered and a Raycast is set toward each other GNode (except for the current node, else we would have that each node is connected to itself through an edge). If a wall is encountered, the GEdge is not generated. Else, two new GEdges are created and saved in the Graph, one that goes from the current GNode to the other one, and viceversa. To make it work I used some LayerMask for both the prefab used for the walls of the dungeon and for the cubes representing the nodes.

After creating some lines in the game world that represent the edges, if the flag Animated was set to true, three seconds elapse, allowing the user to visualize the Graph created. Then, after destroying the 3D objects representing the edges and nodes, a call to *myInitialize* of the component *FlockAStar* is made, passing as argument the Graph created.

Chapter 3: My take on Combining Steering Behaviours

3.1 - Preliminary study

When I first approached the problem of creating a combination of steering behaviours, I started studying the Flocking implementation of Professor

Dario Maggiorini and Professor Davide Gadia (it can be found by cloning this repo: git://pong.di.unimi.it/classes/aivg/2122). Their implementation is based on the work of Craig Raynolds, that in 1986 proposed that the flocking behaviour should be implemented as the blending of three steering behaviours, each of them applied to a single instance of the flock that we will call, from now on, “boid”. The behaviours are:

- Alignment: tries to steer the boid towards the average direction of the local flockmates.
- Cohesion: tries to steer the boid towards the average position of the local flockmates.
- Separation: tries to steer to avoid crowding local flockmates.

In the given implementation, these three satellite steering components are given and blended together each frame for every boid. This results in a flock that moves in a believable way inside a closed room. I toyed around a bit in the scene, and found out that, after some time, especially with high speeds, the boids tend to compenetrate the walls and escape the cage (please Professors don’t hit me). I spent some time trying to understand why, and came up with the following conclusion:

In the separation component, an array of colliders is given, and, given the distance that the current boid has in respect to all those close colliders, a “repulsion” vector is returned to the blending component. The problem is that this code gives the same importance to the other boids and to obstacles, that in our scenario are the walls of the room. This means that, if a boid is close to a wall, but it has a lot of boids around him, this returned vector could suggest the blender to move the boid even closer to the wall, simply because in the opposite direction there is a big set of boids. This can cause the boid to escape the room, and the align component might suggest to its neighbours to do the same.

3.2 – The BoidWallAvoidance component

So, in order to avoid this kind of behaviour, I thought of implementing another satellite component, **BoidWallAvoidance**, that acts the same as

BoidSeparation except it considers only the colliders around the boid that aren't other boids, and extracts a steering. In the same spirit, I modified the BoidSeparation component in such a way that the only colliders considered are the ones of other boids, since BoidWallAvoidance takes care of the obstacles.

3.3 – Avoid the wall at all costs

Still, though, this wasn't enough, and sometimes the boids would still escape the room. Why was that? Because, even if we now have another satellite component that takes into account only the obstacles and blends its contribute with the other satellites, we might still have some situations in which this suggestion is extremely less important than the others, and the wall is traversed anyway.

So, let's step back a bit and ask ourselves what we really want to achieve: we want the flock to behave as expected when a wall is far away, but, when extremely close, we want to give priority to the wall avoidance component in spite of the others. This is a concept that I also learned about during the Artificial Intelligence for Video Games course, for which this project has been made. In fact, when combining steering behaviours, two paths can be followed: Blending, the one we have been talking about, and Arbitration, that given the suggestions of the satellite components, takes one (or few) of them in consideration, based on different factors. One doesn't have to stick with just one of those concepts, of course: usually an in-between solution is what we are looking for, like in this case.

So, in order to implement the idea "when too close to a wall, evade it, no matter what", I added two more parameters to the *BoidShared* script (that is, together with all the other components regarding this topic, in the Flocking directory). Those parameters are *WallThreshold* and *WallRepulsion*. The first one is used by the *BoidWallAvoidance* component, the second one by the *BoidBlending* component.

- In *BoidWallAvoidance*, along with the calculations already mentioned, a check is done against all the collider obstacles passed as arguments to the function *GetDirection()*: is at least one of those obstacles closer than the *WallThreshold* value? If yes, set to true the variable *tooCloseToWall.* This variable is really just a way to notify the fact that a wall is extremely close to this boid, and so the steering returned by this component should have a special treatment.
- In *BoidBlending*, after getting the suggestions of all the satellite components, a check is done: is *tooCloseToWall* set to true? If so, multiply the *BoidWallAvoidance* steering component by *wallRepulsion*. If *wallRepulsion* is higher than 1, we have that this steering suggestion will be considered much more important than the others when a wall is close. Of course those parameters should be tweaked depending on the application and, for example, the boid speed, since higher speeds mean less controllable boids. But, nonetheless, I was able to "imprison" the boids inside the room in this way. Also note that, in the class *BoidShared*, the *AvoidComponent* and *WallRepulsion* properties could be merged, but I preferred to leave them separated to emphasise my ideas (still, it's something to work on in the future).

So, are we done yet? Not quite…

3.4 – "Please keep a minimum distance of…"

Now the problem was that the contribute given by *BoidWallAvoidance*, when *tooCloseToWall* was set to true, was so strong that affected the flock even when the wall was not that close anymore. This would cause the boids to, over time, get closer and closer to each other (you can repeat this step of the development by setting a high speed for the boids and setting the *SeparationThreshold* value to 0 in the *BoidShared* component, which is attached to the gameObject *Spawner* in the scene "Flocking"). To solve this, I simply applied the ideas of a Threshold and Repulsion to the *BoidSeparation* component as well: when there is at least one boid closer than *SeparationThreshold* to the current boid, a Boolean flag is set to true,

which in the *BoidBlending* component has the effect of multiplying the separation steering component by *SeparationRepulsion*. This guarantees that all our boids will try to stay at a certain distance one from another when a wall is not near.

[**Trivia**: originally, in the BoidSeparation component, the "if" hadn't the statement "gameObject != neighbors[i].gameObject" in *logic and* (&&) with the other condition. In this way, each boid was convinced that, at each frame, it was too close to another boid, resulting in a completely independent behaviour: what he didn't know was that that goddamn stalker boid was none other than himself! lol].

Chapter 4: Moving Flock and A*

4.1 – A* implementation

The code covered in this Chapter is located the FlockEscape directory.

The implementation of the A* algorithm is given in the class AStarSolver, and has been developed by Professors Dario Maggiorini and Davide Gadia, nothing more, nothing less.

4.2 – The FlockAStar component

As mentioned on Chapter 2, once the Graph has been built, a call to **myInitialized()** is performed, a method of the component *FlockAStar*, attached to the scene object *FlockEscape*. What it does is taking the Graph previously generated and using The *Solve* function of AStarSolver to find a path from a starting node to an end node. These two nodes are found in the following way:

- The exit node will always be the room node representing the top rightmost room, that is the room whose center coordinates are closer to the upper right corner of the dungeon.
- The starting room is chosen at random from the set of the rooms that have their center on the left part of the dungeon. If, for some reason

(for example when the dungeon is extremely high but not quite wide), there are no rooms on the left part of the dungeon, a completely random room is chosen. Note that in this case the starting room might coincide with the end room. But as long as the dungeon is “normal”, you’re good to go.

After finding the path from start to end, the coroutine **spawnBoids()** takes care of spawning a number of boid as specified in this class. Actually, let’s take a look at (part of) the parameters we can specify in this script:

- Count: how many boids do we want to spawn in this room. Keep in mind the performance, since too many boids can cause serious problems. On my machine I was able to do fine executions with flocks composed of one-hundred boids, more become problematic.
- Boid: the prefab that represents the boids.
- Radius: when we spawn the boids, their z and x coordinates are decided randomly from a pool of coordinates taken from a circle with this radius, and center the center of the starting room. Their y value, instead, is a random value between the floor and the ceiling. This is why I would not advice to create dungeons with rooms that are too small or that are too low on height. In the first case, we risk that the boids are created inside walls, in the second the boids might be created below or above the dungeon, and that’s now what we want. **If you have problems with the spawn of the boids when generating dungeons with extremely small rooms or that are too low on height, that is your responsibility.** *Let me just tell you that if I wanted a maze for dwarfs I’d have done a different project*.
- StopAtFirstHit: Boolean used by A*. Allows to stop the search after the goal has been found at least one time.
- HeuristicToUse: the heuristic used by A*.
- NodeReachedThreshold and SeekComponent: values used from the new satellite component we’ll discuss in the following paragraph.

4.3 – The BoidSeek component

We want our boids to follow a certain path, and in order to do that we can add a new satellite component for our composed steering behaviour. That component is *FSeek* (Note that all the satellite components used in the dungeon are called *FSomething* to distinguish them from the components of the directory Flocking).

This component takes into account the path found by A*, and does the following:

1. Consider the next node we want to reach, that is a point in the dungeon.
2. Is this node the last one of the path AND am I closer to it than a certain threshold value called *NodeReachedThreshold*? If so, we don't have to seek anything anymore, because we have arrived to our destination. Here I decided to simply always return Vector3.zero from this component when the destination is reached, so that the flocking can still act as such even when its goal is met. One can also simply destroy the boid or whatever it prefers, of course.
3. If the previous condition hasn't been met, it means we are still on our way toward the destination. So, check: is our target node closer than *NodeReachedThreshold*? If so, take the next node of the path as the target node and return a Vector3 that suggests the blender to move toward that direction. As with all the other components, this vector is first normalized and then multiplied by *SeekComponent*, a value between 0 and 1 that tells the importance of this contribute to the blending component.

Great, now we just have to plug this component in the blending and we are done! …no, not really. When I first did this, the boids started to have parkinson. But why?

4.4 – Refactoring of FBoidWallAvoidance

In the flocking implementation given in the Flocking directory, the neighbour colliders considered by the separation and wallAvoidance components were the same: in the former only other boids were

considered, in the latter only the obstacles. But let's look at the implementation. Those colliders are found using the method:

Physics.OverlapSphereNonAlloc(transform.position,BoidShared.BoidFOW, neighbors);

This makes sense for "sensing" other boids: the one we are considering has a certain Field Of View, that can be higher or lower depending on what we want to achieve. Basically, we create a sphere around our boid and see the boids colliders inside this sphere. Let's also assume, as an example, that the FOW is of 5 meters (this value can be set in the *FBoidShared* script). This means that our sphere has a radius of 5 meters. So far, so good.

The same is true for the *FBoidWallAvoidance* component, that takes the same set of colliders as argument, but this time considering only the obstacles, such as walls. This means that, even if a walls is 5 meters far away, it will be considered by the blending component as something to avoid. Still no problem though: it's one wall against dozens of boids, so the flocking behaviour will prevail over the wallAvoidance one, right?

Yes. But the problem arises when we don't just have 6 walls in total in our scene, like in the Flocking scene, but a lot more! In fact, our dungeon is nothing more than a 1-1 translation of a bitmap, where, when (i,j) = 1, we spawn a gameObject. So, since each gameObject has its own collider, our *FBoidWallAvoidance* component will sense sometimes even tens of walls all around him, even if they are 5 meters away from it. The resulting effect is that the boid will feel "oppressed" by a multitude of walls even when there is a lot of free space around it.

To solve this, I changed the array of colliders that the *FBoidWallAvoidance* component has to consider: the sphere considered by this component has a different radius, *wallThreshold*. This is a complete refactoring of the class, since now it doesn't need any more a Boolean to notify when a wall is too close: if it is, its contribute will be considered by the blending component (and still multiplied by *wallRepulsion*), else, it will just be 0. This solves the parkinson.

That said, the rest of the code is as expected: the blending component takes suggestions from the satellite component, blends them together and produces a behaviour for each boid, that takes them from the start to the end.

Chapter 5: Conclusions, aka what could have been done better

5.1 - Quadtree to have less walls

In the process of generating the actual gameObjects that will make up the dungeon, each cell of the bitmap is considered, and each time a 1 is encountered, a new gameObject is spawned. This means that we have a O(width*height) of gameObjects in our scene, which becomes problematic performance-wise when those numbers are high. Just 100x100 is a “heavy” dungeon for our implementation. A workaround to this problem would be converting the bitmap into a quadtree, and then translating the quadtree to gameObjects. In fact, if each leaf node is converted into a gameObject or an empty space, we would generally have much less objects to deal with. This is certainly something I’ll change in the future.

5.2 – A zip around the dungeon

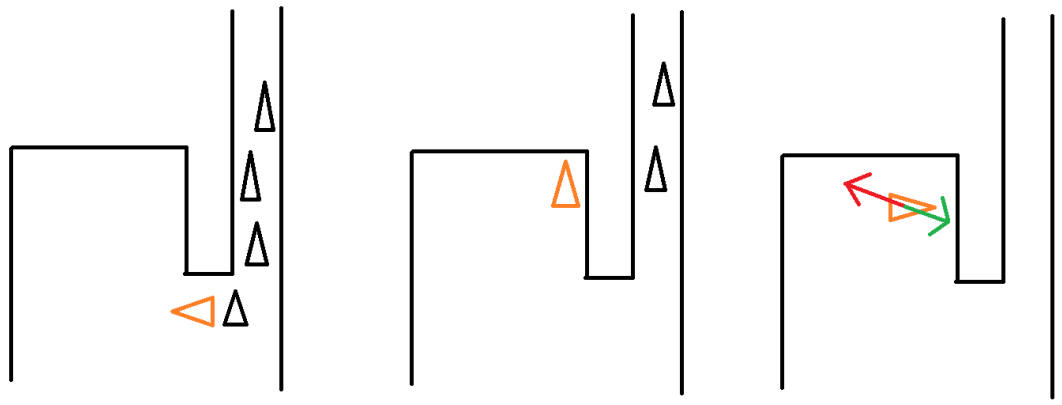
The RoomsMustBeSeparated flag in the LabyrinthGenerator4Animated component allows us to have rooms directly connected by the absence of a wall, rather than with a corridor. Sometimes though some of these rooms can spawn on the very edges of a dungeon, allowing the boids to escape it. It would be useful, to solve this, adding 4 full walls on the edges of the dungeon, or something like that.

5.3 - Help lost boids find their way (edit: solved!)

Sometimes, the crowding of the boids in some points, like the entrance of a narrow corridor, might suggest to a boid to steer away, rather than

entering that corridor. If a boid ends up in a situation like this, he might enter an adjacent room that is separated from the corridor by a wall. If that boid remains alone, the only satellite component giving him advices are the wallAvoidance one and the seek one. The seek one suggest to move towards the corridor, but on the way there is a wall. The wallAvoidance one suggests to go back. The result is that the boid is stuck in a loop, as shown in the image below.

The orange boid is our test subject. After entering (against the will of the seek component) in the room on the left, the seek component tries to take him back to the corridor entrance, but the wallAvoidance component suggests going in the opposite direction.



This is of course a problem, because some boids might get lucky and eventually find their way towards the target, but others can get stuck forever.

Edit: I solved it!

I'm writing this after having written all the documentation above. After a bit of reasoning I came up with a solution.

In the *FSeek* component, a coroutine is started each time a new target of the calculated path from A* needs to be reached. This coroutine, given the distance between the actual position of the boid and the one of the target, calculates the distance between the two. Since we also have access to the speed of the boid, we can calculate how much time would it take to go to

the target from the current position if we could proceed without obstacles. This time is called *timeExpected.* But ***we know*** that some obstacles exist, like the walls and other boids, and because of them rarely we'll get to the target in the expected time. So, we give each boid a certain tolerance for that time: in particular, we multiply *timeExpected* by *timeFactor*, a private variable I just introduced in the boids (its value can be changed of course).

Now we have two scenarios:

- The target (or at least the tolerance area of the target, remember in fact that a target node is considered reached if the boid is closer than *NodeReachedThreshold* to it) is reached before *timeExpected * timeFactor* seconds. In this case, the coroutine is aborted, the next target is calculated and another coroutine starts.
- If after *timeExpected * timeFactor* seconds our boid still isn't in range of the target, the coroutine resumes, and assumes that it got lost like in the example provided before. It can be stuck, it can be "late", doesn't matter: the boid must be updated on the path he must traverse. To do that, a new node is added to the graph, which coordinates are represented by the current position of the boid. At the same time, this node is provided of all the edges that can take him to all the visible nodes of the graph (see the new function *addNodeAndFindEdges()* in the *Graph* class). Now, a new path from this position to the final destination is calculated in the following way:
    1. We consider our new target node *not* the old target node, but its successor. Why is that? Because another strange behaviour I found out was that, if the boid doesn't get closer than *NodeReachedThreshold* to the target, but it surpasses it thanks to the align component of its neighbour boids (that basically force our boid to go forward, right to the next node), what he tries to do is to go back, "touch" that target and then proceed his course. This is of course not what we want: if you surpassed that target, just ignore it and go for the successor!
    2. We use A* to find the best path from our current position to the new target.

3. We concatenate this path to the path that starts from the new target and goes to the final destination (that we already have, since it's a subset of the original path).

Of course, the higher the *NodeReachedThreshold*, the better results we have. Still, I'm pretty satisfied with this implementation: now the boids that get lost are much less, and even those eventually find their way to the end.

And they lived happily ever after