Generating simple dungeon

I°) Introduction

First of all, we create a blueprint actor. We will mainly define functions in this actor. In order to use them, the construction script page will be used. I will not detail how to use it as it is not the point of this tutorial. Instead, you will find here descriptions for creating the required functions.

The way this method works is the following:

1. Rooms are placed randomly
2. Edges between rooms are determined depending on their distances
3. Proper corridors are created for each link
4. The elements are spawned

Basically, the method presented here comes partially from the python script:
www.roguebasin.com/index.php?title=Dungeon_builder_written_in_Python
It has been heavily modified but the starting point comes from it.

II°) Creating the rooms

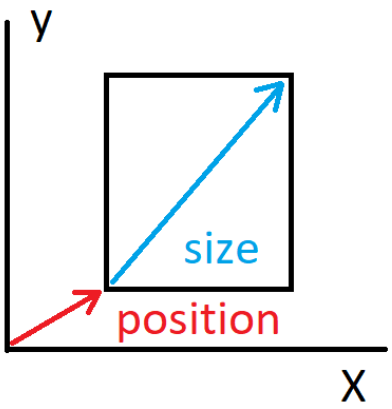
Here, we try to position rooms. We will consider that a room has rectangular shape. It might be possible to come with different shape but we will limit ourself in this report.

II - 1 °) Overall method

The point is to position random rooms randomly on the map. The method used here is :

1. Generate a room and place it somewhere
2. See if it overlap an existing room. We don't want overlapping here, so we consider that if it's not overlapping, it's successfully placed.
3. In the case of an overlapping, we get rid of this proposed room and generate a new one.
4. We stop if we have enough room or if we failed too many successive times.

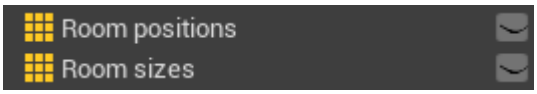
II - 2 °) Abstracting the room data type



First part is how we can store the room data. For now, a room is characterized by its position (xy coordinates) and its size (width, height). So basically, 4 numbers.

In order to simplify things for later on, we will consider these numbers to be integers (i.e rounded numbers). It will be very handy for generating floors, walls and other things: we are working on tiles. Because it can be useful to keep some possible data available (for example, which kind of room: office, hall, dormitory), we will be working with 3D vectors.

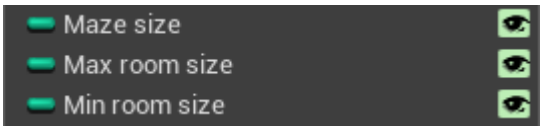
As we want to be able to generate a dungeon as big as someone desire (and computer allows to), we will store these vectors in arrays:



Therefore, the room number k will be placed at Room_positions[k] and with a size Room_size[k]. Both arrays are aligned (same index in both arrays represents the same room)

II - 3 °) Generating a room

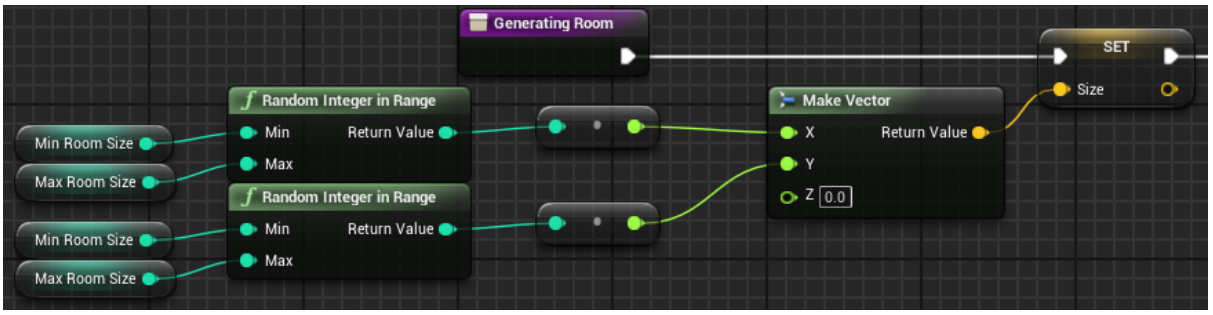
Our dungeon should be limited in space. We don't want two rooms that are 20 minutes walking distance. For that, we will define a map size. It shall be noted that all the numbers considered here are expressed in "tile-length" unit. For similar obvious reasons, we don't want our rooms to be too large nor too small. Therefore, we need to define 3 more variables:



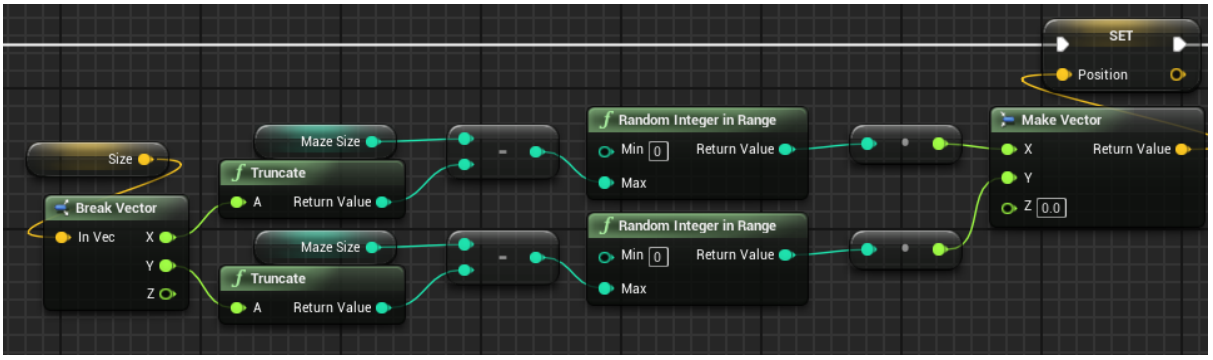
These variables (integers) are visible so, when you place your dungeon, you can directly modify them in the editor, you won't need to open the blueprint.

In order to generate a room, we will create a function "Generate a room". This function shall return 2 vectors: a position and a size. So, in the "Generate a room" function, we create two local variables: position and size, which are vectors. Then, we generate a random size. The size shall be between the "Min room size" and "Max room size" we define before. As they are global variable, they are visible inside functions. We won't use them as input but it could be useful if you want to generate room with drastically different properties.

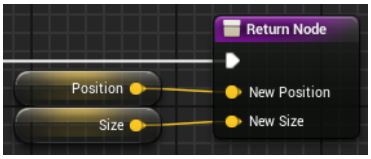
To define a size, we just pick random uniform numbers in range:



Now, we can define a random position. We will do the same for the position, picking a random number between 0 and “Maze size”... with one difference! As we don’t want our rooms to extend over the “Maze size” and because the position is the low left corner in the xy plan, we will need to bound our position between 0 and the previously defined size:

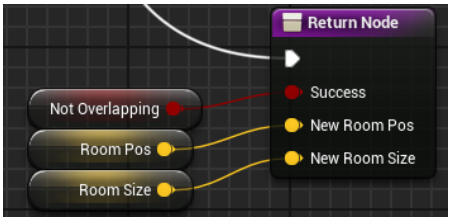


Now, we just have to return the position and size:



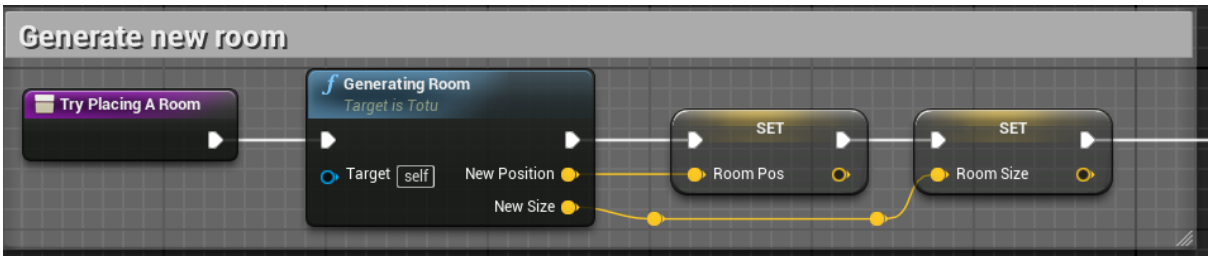
II - 4 °) Try placing a room

Now that we have a new generated room, let’s try to place it. For being able to place it, we must be sure that it is not overlapping any previously placed room.
In order to check that, we create a function “Try placing a room”. This function shall return a boolean (Did we generated a proper room?) and the room data (position, size).



We will need to define these as local variables. Don’t forget to initialize “not overlapping” as True. Indeed, once we start checking, it can only degrade to False once we find one overlapping room.

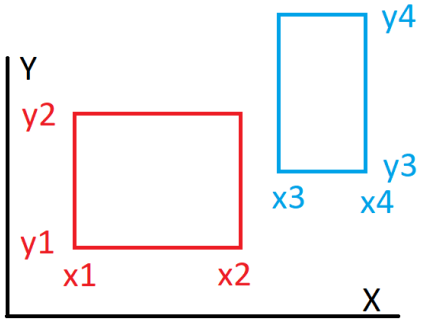
So, first thing you do once you define the function, you generate a room:



Now, we have to check with all the previous ones. Remember that we store the rooms data in the global variables “Room positions” and “Room sizes”. So, for each room, we check if we overlap:

Here, we pick all the rooms, so it’s a for loop. In UE4, you can loop directly on an array, so we use this. As we consider that the room number k is at the same position in both arrays, we need to verify the overlap between the room defined by : (Room_position[k]; Room_size[k]) and the one we just generated (Room pos; Room size). Then, we consider that if we didn’t overlap for this one and we didn’t overlap up to now, we still don’t overlap. Once we completed this loop, we can return.
Note: Take some time to understand well the for loop structure and the way it is used here. It will be used several times later on.

You notice the presence of another function here: “Overlapping Room?”. This function takes the information of 2 rooms and return a boolean. I won’t describe every blueprint function but you should be able to write it. Basically, it’s easier to check when they are not overlapping. It makes 4 conditions. If one of them is true, then they are separated. Don’t hesitate to make a drawing to see the conditions and return the not(separated).



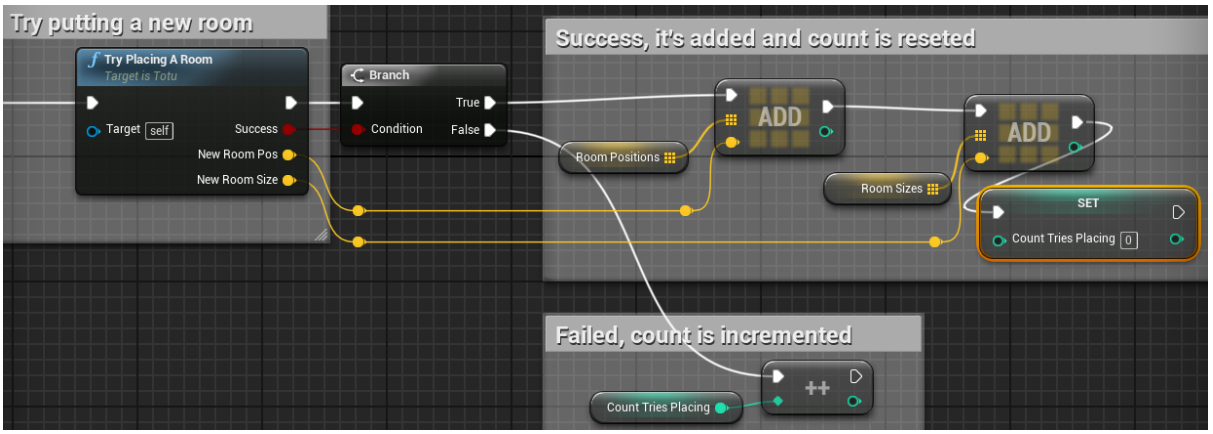
II - 5 °) Generating all the rooms

Now it is time to generate all the rooms. Basically, we will place room until we reach the desired number of room. So, this means we have another global variable also visible so we can edit it easily in the editor.

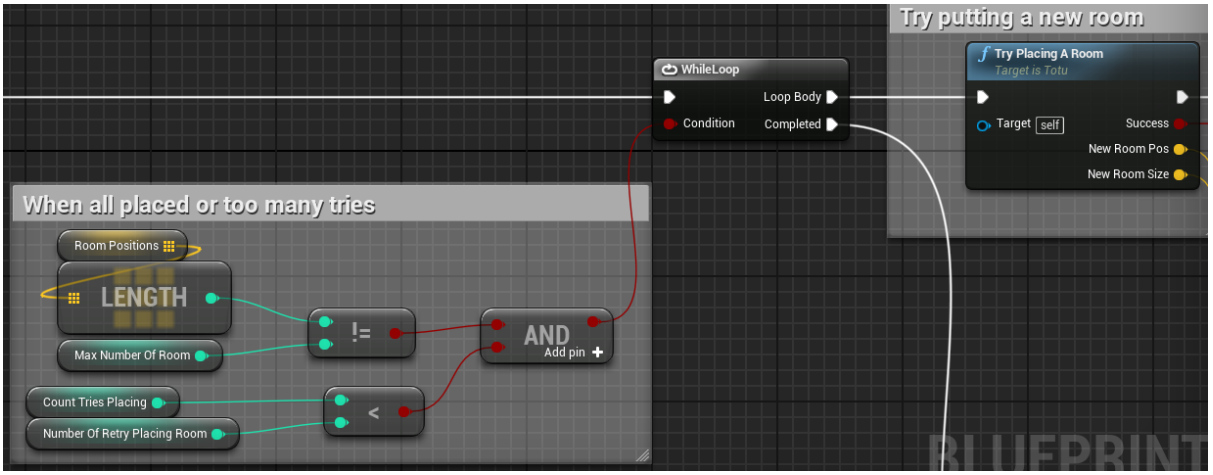
So, the idea is to try placing a room. If it succeed, we add it to our arrays “Room positions” and “Room sizes”. If not, we retry until we reach the proper number. Maybe you already have guessed but there is one problem with this method. What is there isn’t any place left and we still have room to create? We might be trying again and again… and never manage to finish it. In practice? The editor will freeze. So, we should avoid that. One way of doing it is we consider that we allow only a certain number of successive failure to place a room. After that, we just give up and consider that there it is no more possible to place a room. So, we create another global value:

Clearly, once we reach this value, it doesn’t mean there are no more place left. It means that it becomes too difficult to find it. If the variable “Number of retry placing room” is low, the dungeon will be sparse, rooms will be separated. If this variable is high, more rooms will find places and the dungeon will be more dense.

Basically, this function returns nothing as it just adds elements to some (2 arrays) global variables. It consists in a while loop that tries spawning rooms. If it succeed, the room is added to the arrays and the failure count (need to create a local variable for that) is reset. If it fails, the failure counter is increased. Here is what should be executed on each loop:



Only thing left to do is to write the stopping condition for the while loop. The loop should stop once we reach the maximum number of rooms or the counter for placing failure reach we previously defined limit:



Now, if we run this function, we end with the arrays of rooms properly filled with plenty of non overlapping rooms.

III°) Creating the corridors

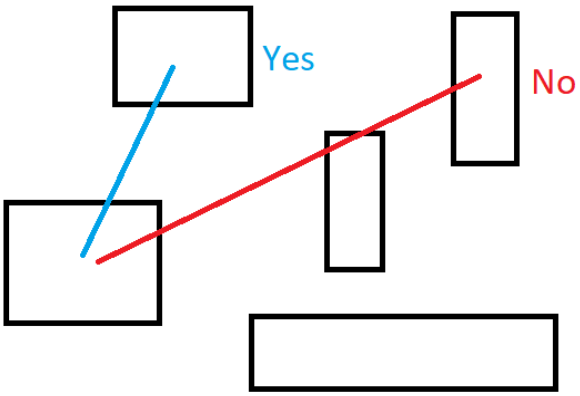
When talking about corridors, we are talking about a link between two rooms. The way the corridors will be built will depends

III - 1 °) Overall method

The method used here requires some graphs and trees knowledge. We will need that in order to choose which room is linked to which one. Basically, we want to be able to go from any room to any other. But we don't want a corridor for each couple of rooms (room 1 linked with all others, room 2 with all others, …) as it will very quickly become messy. Once all the needed links are known, we can start generating corridors in a more proper manner.

III - 2 °) Creating the minimal spanning tree

What we will want here is basically create a minimal spanning tree (check the wikipedia webpage https://en.wikipedia.org/wiki/Minimum_spanning_tree) . In other words, what corridors are necessary for linking all the room while creating only short distance corridors?

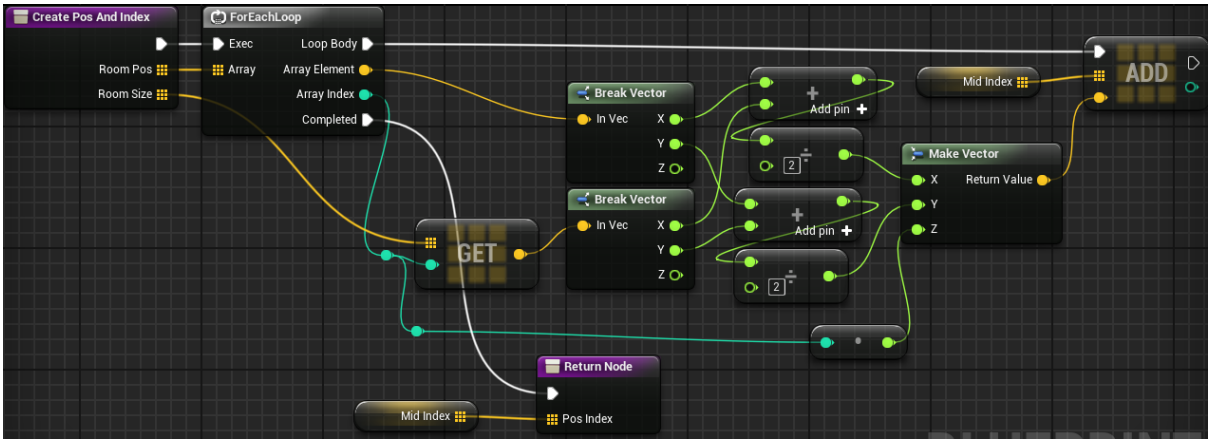


For example, in the previous image, the blue link is a good one as it's a link between close rooms, contrary to the red one.

For creating a minimal spanning tree, the Prim algorithm is some I found quite easy to implement. But before starting working on this algorithm, we will need to create another array of vectors. To create such a tree, we need to consider the distance between two different room. In this version, the distance I consider is the distance between the centers of each room.

III - 2 - 1 °) Creating Center position and index

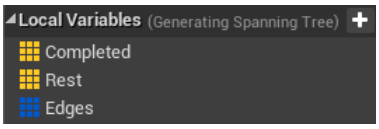
So, our first function will be a function that will return a vector array. The xy coordinates of the vector will be the center of the rooms, while the z coordinates will be the index of the designated room. This will be useful as we'll want to create a list of links between rooms (so, basically, between indexes).
This function is not very complex. You create a local array and for each element of the main arrays ("Room positions" and "Room sizes"), you compute the center of the room, get the index as last coordinate of a vector and add it to your local array. Return the local array at the end.



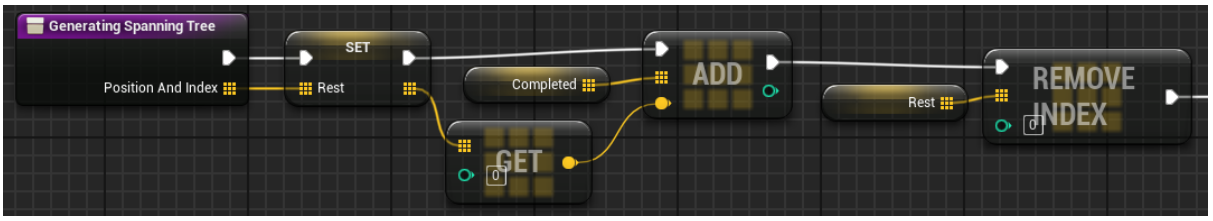
III - 2 - 2 °) Generating spanning tree

Based on the result of the previous function, we will use the Prim algorithm. The method might not be very time efficient but it will be very easy to understand. First of all, the split the rooms into two parts: what is linked and what isn't isn't. So, it means two local arrays, here

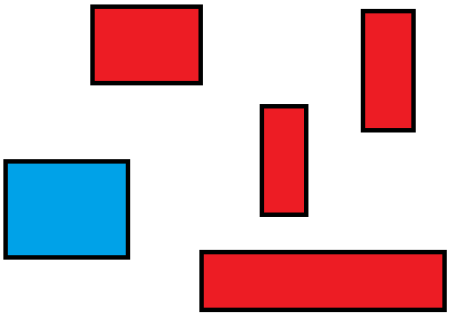
called “Completed” and “Rest”. In order to return the edges, we use an array of vectors, called here Edge (for this one, I used a integer array, as it’s not positions but only indexes).



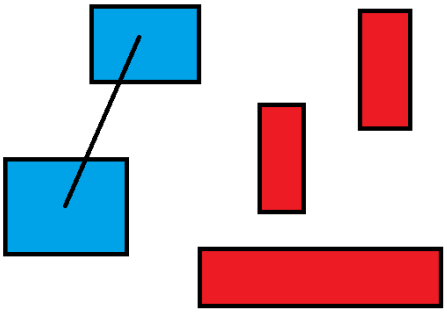
We start by saying that the first room is completed (it will be the starting point). So, we add it to the proper array, and remove it from the other



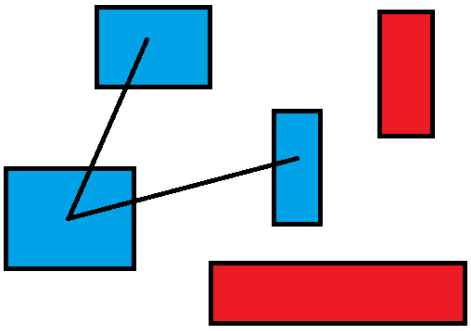
Now, the method is the following: as long as there is some rooms left, we find the non-completed room the closest to the set of completed room. Then, we create an edge between this non-completed and the closer completed. The non-completed is then added to the completed list and removed from the “Rest” array.
In the following images, the blue rooms are “Completed” while the blue is the “Rest” of the rooms:



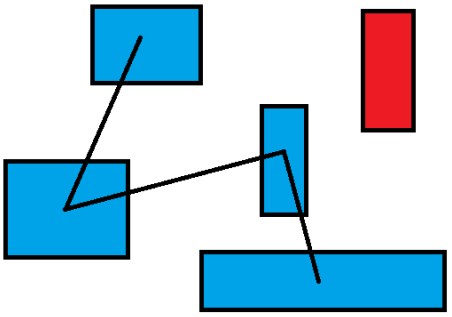
We initialize the first room as “Completed”.



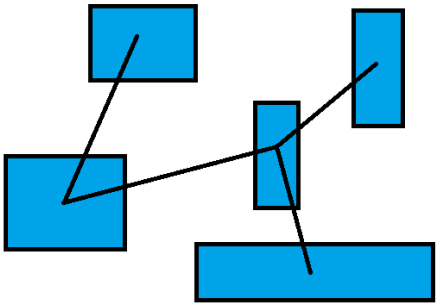
The closest red room is linked and added.



The closest set of red and blue rooms are linked and the red one is now in the “Completed” list.



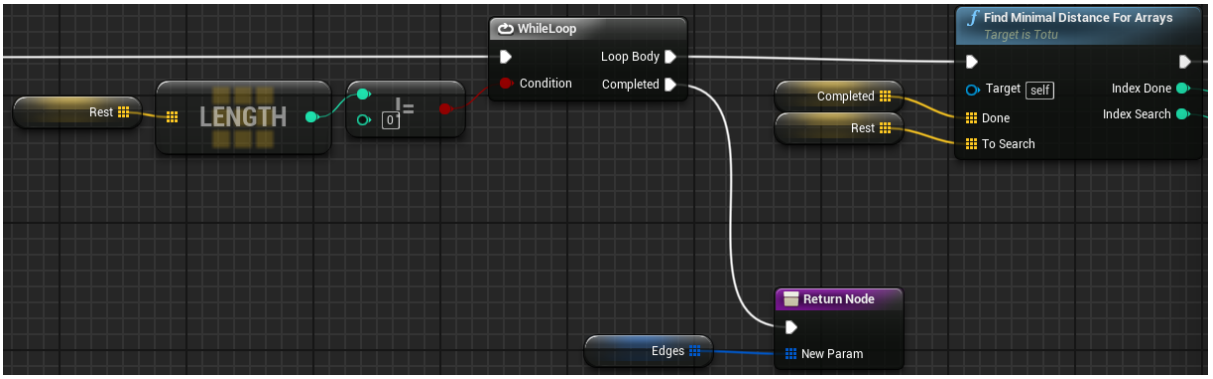
The closest one is the bottom room and it is close with the last added room in this case.



The last room is close to the one in the middle. We link them. As there are no more “Rest” rooms, we finished.

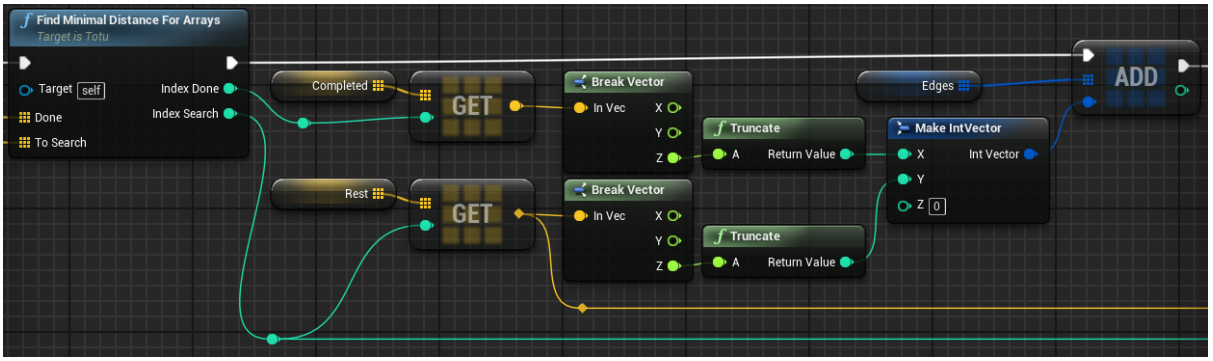
If the method used here isn’t clear, check the wikipedia webpage, it’s animation is very clear and easy to get.

So, in blueprints, the while condition is stated as:



The bloc “Find minimal Distance For Arrays” is used to determine which “Completed” room is the closest to which “Rest” room. We’ll describe it later on. What is important is that is return the indexes on each arrays (“Completed” and “Rest”)

Now that we have the room index, we create a link between them:

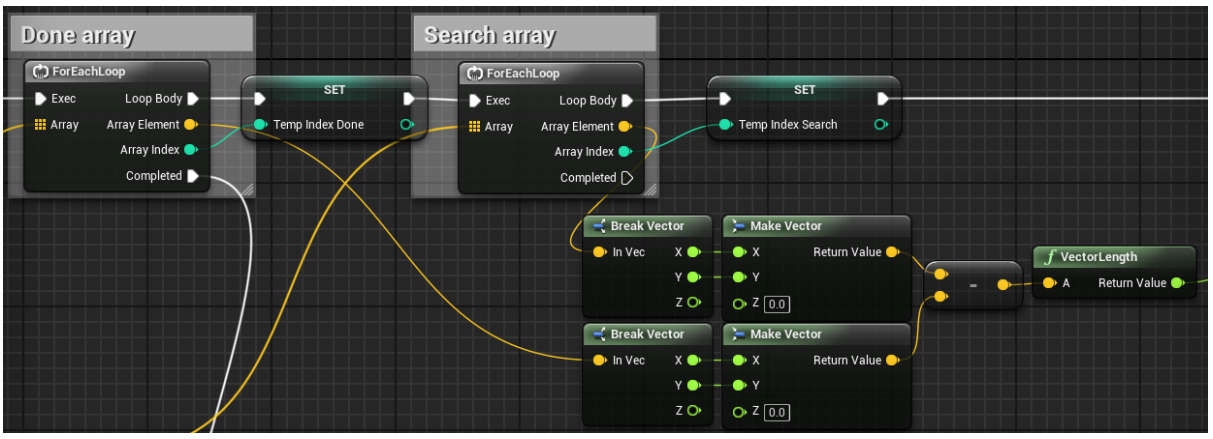


And we switch the room to the “Completed” part:

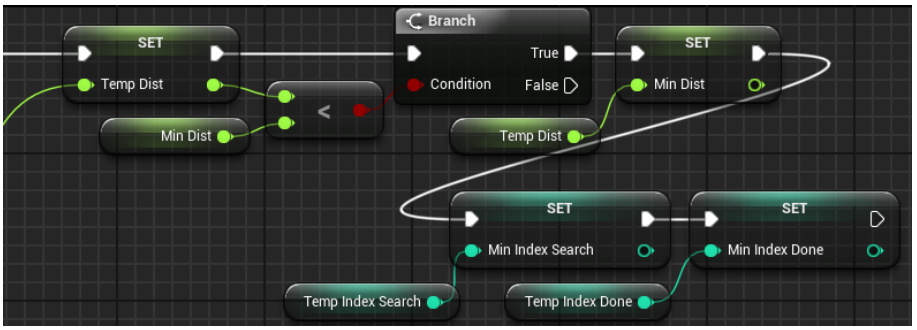
Now, we just have to return the “Edges” we created (at the while node).

III - 2 - 3 °) Find Minimal Distance Between Arrays

In the previous section, we used a bloc called “Find Minimal Distance for Arrays”. I won’t explain this in too many details. Basically, it is the same as searching the minimum in a array. So, for each element of the array “Done”, you look for each element of the array “Search” (so, two imbricated each loop) and determine the distance.



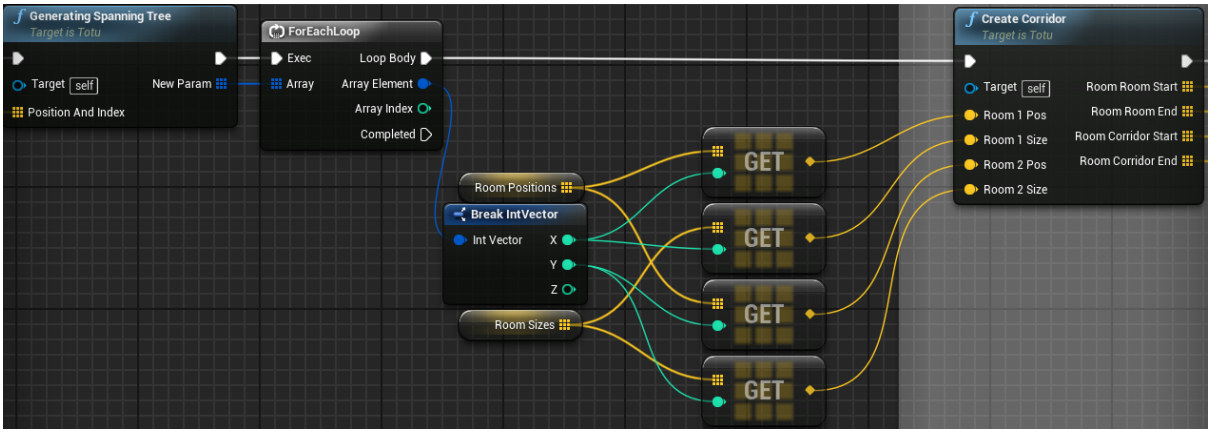
If it is smaller than what you already have since the start, store the two indexes in local variables.



At the end of the bigger for loop, just return the 2 indexes.

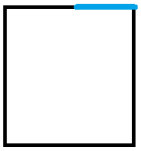
III - 3 °) Constructing a corridor between two rooms

Now that we now how to get the edges, what remains is how to define them. We will consider only horizontal or vertical corridors. There will be a starting and an end point to each corridor. So, two sets of coordinates: we will use two vectors. One for the start point, one for the ending point (it shall be noted that the order is not important, we only consider corridors that can be traveled in both directions).
This means that for each edge of “Edge”, we shall create a corridor

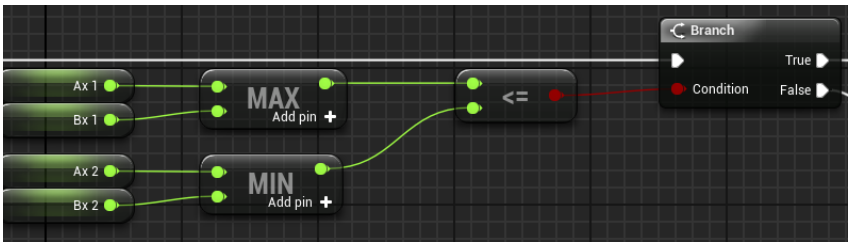


Creating Corridor

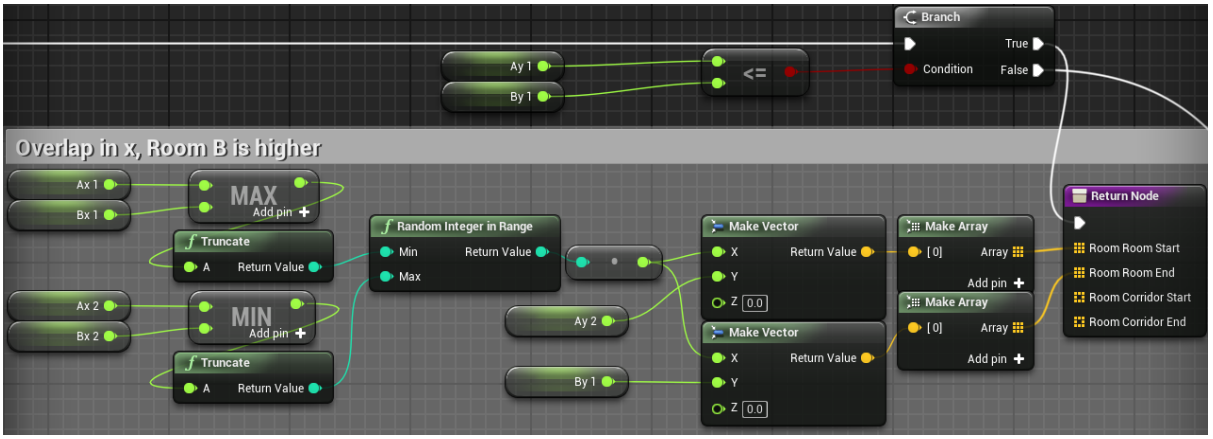
For this function just introduced in the previous image, we will need to consider several case. The easiest one is when there is an overlap in x or in y:



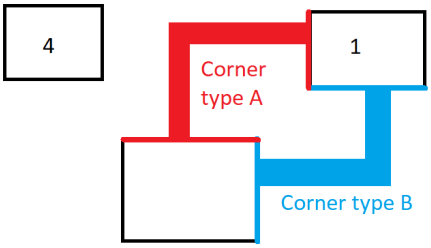
In the case of this image, there is an overlapping in the x direction. We will therefore create a vertical corridor between the blue sections. The process in case of overlap in the y direction is the same and therefore not described.
In order to detect if there is an overlap in x, we just look at the corners of the rooms. Using local variables for the corners similarly to the section “Try placing a room”:



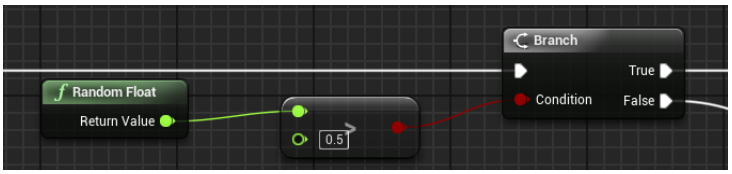
In the case of an x overlap, we need to find which room is above the other one. Based on a that, we pick a random value in the blue area and create a Room to Room corridor. In the image below, the test to see which room is above:



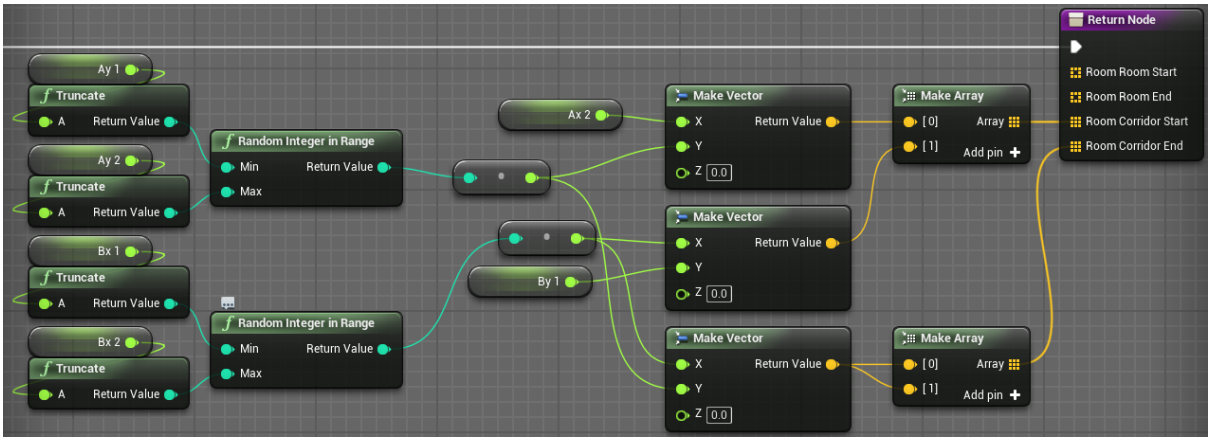
A more general case involve two rooms that don’t share a coordinate overlap. Depending on the positions, we have 4 different possibilities:



For each case, there are two ways to create a corridor. Based on the previous image, a type A or a type B corridor. Here, we present a random choice between both:



Let suppose that we are in the case of a type B corner. We will need to random values along the blue line of the previous illustration: The x and y coordinates of the corner. Here is what the blueprint would look like:



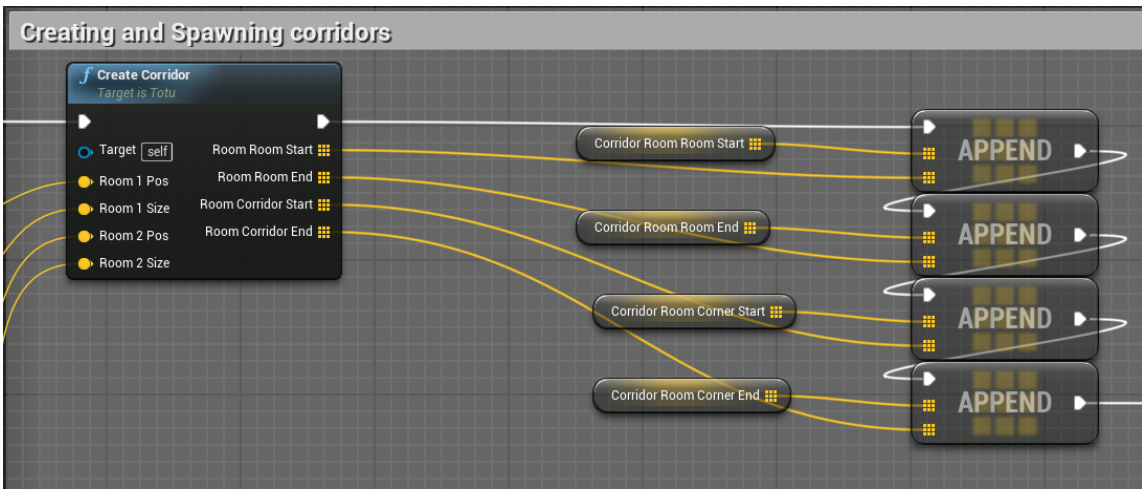
As there are two segment for a corner corridor, we return arrays with 2 elements.

It can be noticed that the return node could have simplified with only two output vector arrays: “Corridor start” and “Corridor end”. The fact that it is here differentiated is because with this structure, it becomes easy to get the positions of doors. As I expect to extend this generator and spawn doors at the entrance of some room, making this separation (and consequently not spawning doors in a corridor corner) can be useful. However, one can perfectly get rid of this complication.

The reason of why we use arrays as return values is because at the entrance of this function, we don’t know yet how many segments we will have. It can be 1 room to room segment or 2 room to corner segments. An array can have any length.

The rest of this function remains quite similar to what we did here for one case. You should now be able to complete all the cases by yourself.

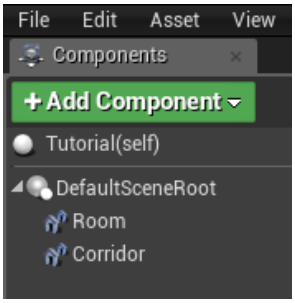
Now that we generated a corridor for an edge, we add it to the global variable defining all the corridors:



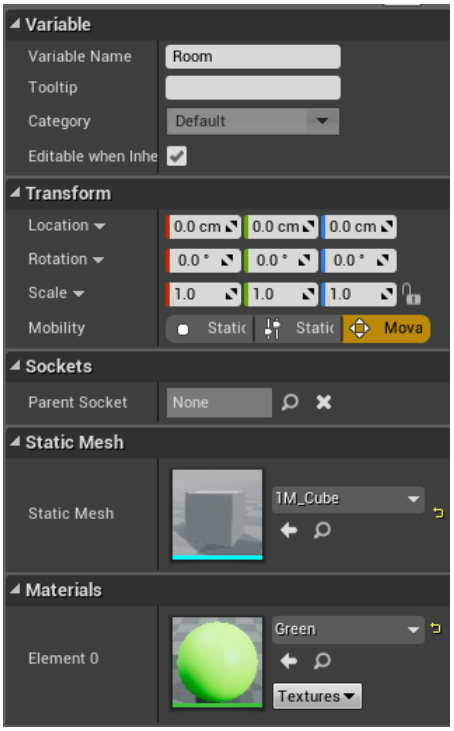
IV °) Spawning the dungeon elements

Now that we created everything we need, this last section is dedicated to make these elements appear on the screen. I used the same way that was described in this Live training: https://www.youtube.com/watch?v=mI7eYXMJ5eI.

We will spawn two different types of static mesh: the rooms and the corridors. To our actor, we need to add 2 instanced static meshes:

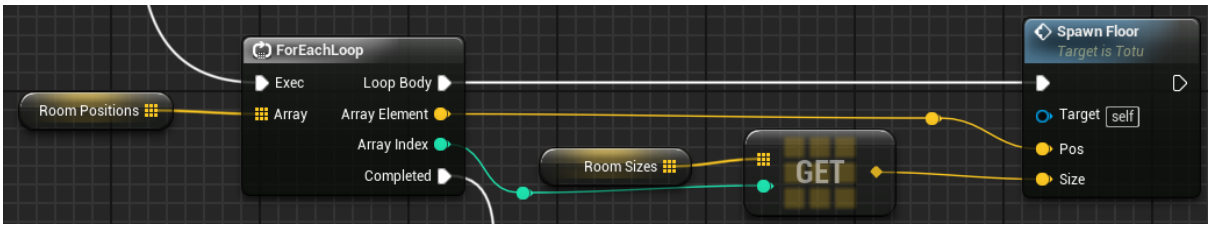


Next step is describing which static mesh will be spawned by these instances. By clicking on each, we can define a cube and a material (here green, it will be grass covered room):

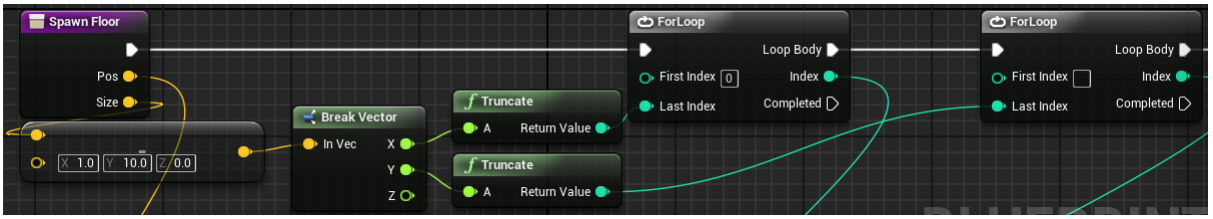


It shall be noted that the cube has a 100x100x100 size. As we are working with rounded numbers with position (remember we are working with tiles), we shall create a global variable that will be set up to 100 by default (the size of the cube):

Now, we will write our spawn room function that will be called for each room:

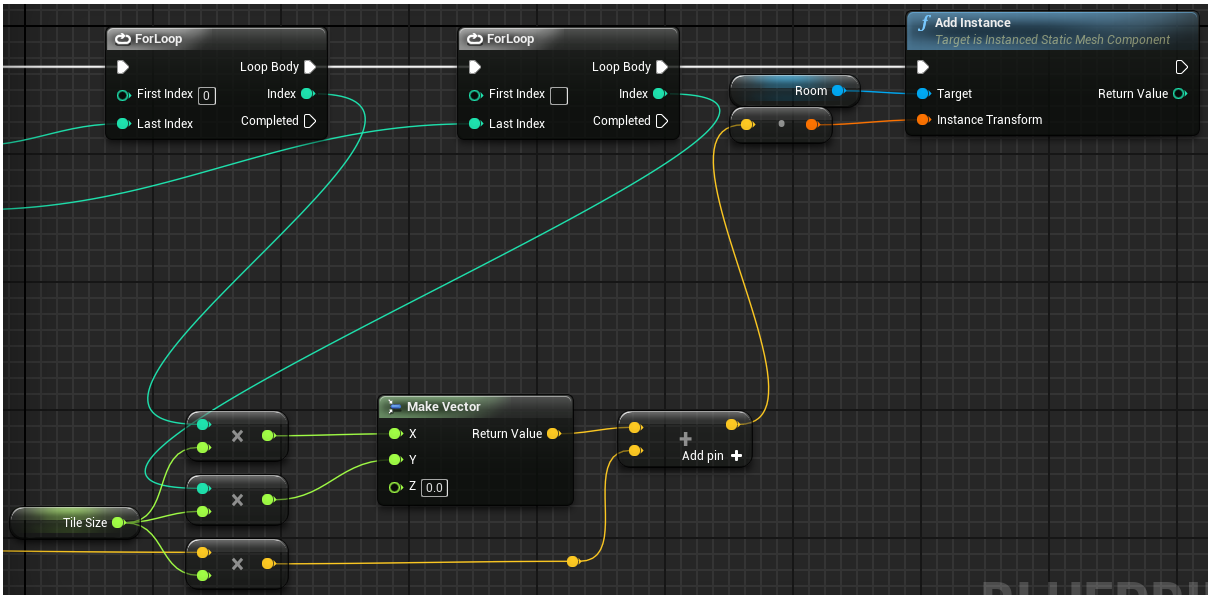


In the “Spawn floor” function, we will basically make 2 for loop imbricated on the size. This will give us coordinates inside the room:

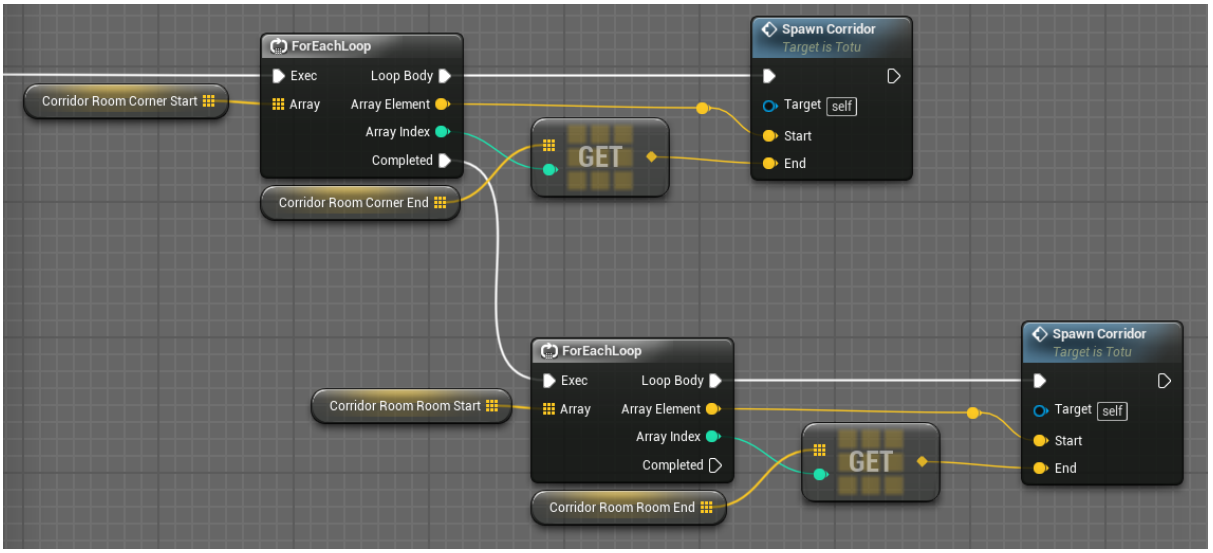


It shall be noted that the size is decreased by 1. This is due to the fact that if for example the room has a size of 1, we want to pass in the for loop only once (basically because here, loop starts at 0).

Using these inside-room coordinates, we just add the room position (yellow wire on the left side) and add this transformation to the instance we created just before:

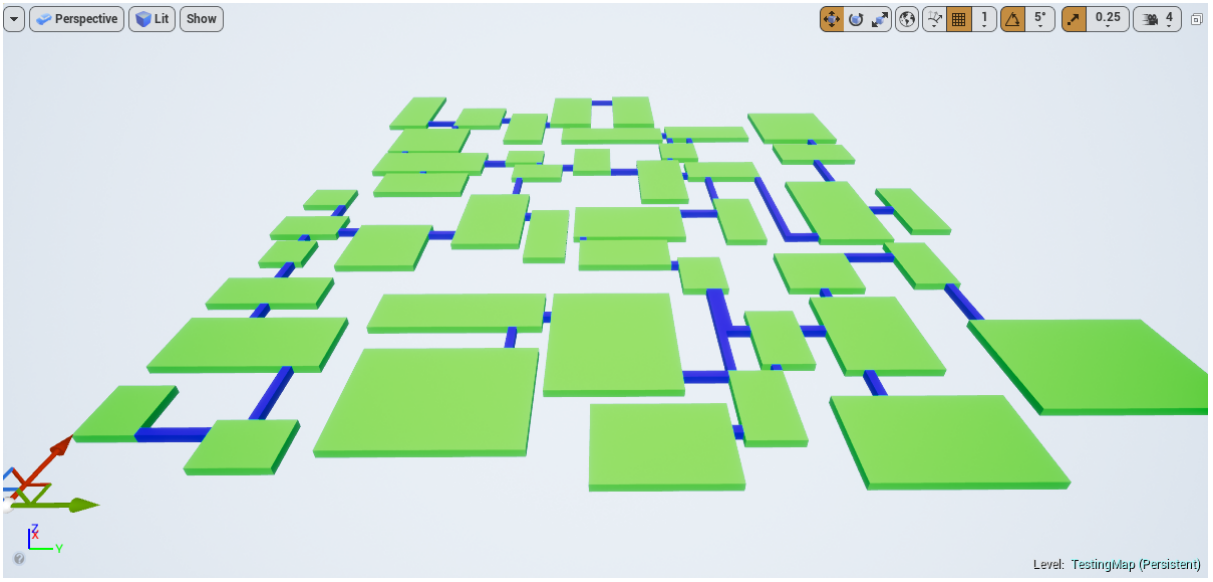


For the corridor, the method is rather similar, with the difference that it will be a segment. Therefore, only one for loop in the spawn corridor function that will be used this way:



Don’t forget to define the “Corridor” type of static mesh that will be spawned!

You shall get something like that when placing your actor on the map:



In this case, the values used were: