



# 3D Level Generation Algorithm

Samuel Pols

Games Design & Development

K00251426

## Table of Contents

Introduction & Problem Definition	1
Literature Review	5
System Design & Example of Use	9
Implementation	15
Testing & User Manual	21
Conclusions & Recommendations	25
Appendices	30

## Introduction & Problem Definition

The aim of this project is to use procedural generation for random level generation consisting of a series of 3D rooms forming an interior using a C# algorithm inside Unity.

To demonstrate the achievement of this project, this project will identify 3D room generation algorithms currently in use and their underpinning algorithms. This project will then define performance metrics, i.e., rooms not overlapping/adequate generation times, and introduce several constraints, i.e., it's not a level editor, it's an algorithm to make the level. This project will be packaged in a simple package with adjustments being made inside the Unity inspector. Lastly, the concept of procedural generation, and it's use as a gameplay mechanic will be demonstrated by a simple game which uses these random levels as a core concept.

The intended audience for this project are other game developers who seek to use someone else's algorithm for level generation, whether to save time or because of lack of programming skills. The games this algorithm could be used for are fairly limited, it can't be used for genres like strategy, racing etc. however this algorithm could be used for a lot of action/shooter/RPG games, either first or third person, that have an overwhelming focus on interiors, such as dungeon crawlers or roguelikes.

The problem I've identified is that a lot of random level generation algorithms in places like the Unity Asset Store are using solely prefabs, which is good for developers who already have assets or have an asset artist lined up to do that for them, however, this might not be an option for every developer, hence it making more sense for them to use my level generator which generates code from scratch with the algorithm creating some rudimentary assets, i.e., solid colour textures, if the developer doesn't have their own. Also, a lot of these are made specifically for dungeons, which again, are interiors of connected rooms, however, they will not suit everyone's game who's looking for an algorithm to generate a level for them.

The stakeholders within this problem area are game developers who are looking to make a game with random level generation as one of its core themes, but are not satisfied with the current solutions to that problem, e.g., using prefabs, not liking the dungeon layout and looking for rooms only, that will find a use for my algorithm.

The scope of the project is to produce an algorithm, that will make a level either by pure code, i.e., making meshes using points and meshes between the points, along with random number generation to determine the layout of the level being generated. This algorithm will be configurable, with parts of it being able to be replaced by the developer's their own assets if they wish to do so. Lastly, to show the algorithm off, and to demonstrate that it is truly random and working as intended, a simple game will be made where the objective is to explore the randomly generated level and find the exit to progress, and the rooms will have enemies and/or useful items to challenge the player or make them stronger in the case of the latter.

This project will follow an agile methodology, where the focus will be placed on getting the level generation algorithm to work and work properly and once the basics are done, developing additional features or refining the code further.

The project plan will largely reflect the delivery dates for the submissions of project components, i.e., 17<sup>th</sup> of October for this segment, 21<sup>st</sup> of November for literature review etc. There will be additional segments added on top of the submission list on moodle, and dependencies will have to be factored in, i.e., I can't do much on the game without the algorithm properly generating levels. I will be looking to have the very basics done by the end of December, with the algorithm at least generating the starting room and starting to generate rooms next to it.

Deliverable	Submission	Date
-------------	------------	------

Introduction and Problem Definition	PDF	17 <sup>th</sup> October 2022
Literature Review	PDF	21 <sup>st</sup> November 2022
Set up online repository	Link	~End of November 2022
Basic prototype – room being generated	C#	~End of December 2022
System Design and Example of Use	PDF	23 <sup>rd</sup> of January 2023
Initial Prototype	C#	~End of January 2023
Implementation	PDF	27 <sup>th</sup> February 2023
Alpha version of the project	C#	~ End of February 2023
Testing and User Manual	PDF	6 <sup>th</sup> March 2023
Beta version of the project	C#	~ End of March 2023
Conclusions/Recommendations	PDF	20 <sup>th</sup> March 2023
Screencasts	MP4	27 <sup>th</sup> March 2023
Thesis	PDF	12 <sup>th</sup> April 2023

The primary assumption I'm going to be working under is that my algorithm won't be too dissimilar from similar 2D room generating algorithms, as the generation of X and Z coordinates should be the same as if the game was two dimensional. The biggest difference of course will be having to factor in the Y value, which shouldn't be too hard to do as if the room is two meters tall, the Y values will likely be 0 and 2 for a room, however, this will get more complicated if multiple floors will start to get placed on top of each other, as things such as floors colliding will cause textures to glitch out and that would be very detrimental for the game.

Another assumption is that hardware will remain consistent, i.e., the project will be able to run on school computers, and that I will keep the same Unity version to prevent issues with changing editor versions.

The most important outcome of this project is to produce an algorithm that generates a level consisting of rooms, and for that level to be usable in a game, i.e., not just a collection of empty rooms, which should hopefully be demonstrated by the game that will be developed alongside this project to show the algorithms usability. If the algorithm functions properly in

my game, then it will be very likely that other developers should be able to use this algorithm without many core modifications to it to suit their needs. Another important outcome is for this algorithm to function properly, i.e., no overlapping textures, rooms not spawning inside each other etc.

# Literature Review

## Introduction

Procedural Content Generation “is the algorithmic creation of game content with limited or indirect user input” (Shaker, Togelius & Nelson, 2017). Procedurally generated content is a very large and diverse topic, as this encompasses a lot of game aspects, such as textures, items, quests, music, characters etc. The topic I am focusing on, and the one that is the most well known of these and has been in gaming for decades, is procedural level generation.

Procedural level generation is present in a lot of game genres, some of which feature it slightly, such as strategy games, which have overall map layouts, but random level generation adds a lot of randomisation to it, while sticking to the overall intended map, while others revolve around it. Roguelike games are known for this, and almost all of them are based around procedurally generated levels. In roguelike games, it is an essential mechanic, as the player is expected to either restart often, or replay parts of the game over and over again until they get past it. Procedural level generation is a great way of either presenting content in a way that is made to be replayable, or to make sure that the player doesn't get bored of having to replay certain sections. A level, for example, can still fulfil the same gameplay and narrative needs, i.e., the same characters are still present on a level, the objective is still the same, but the level and the details of the level change, meaning it doesn't get predictable, and most importantly, repetitive. Procedurally generated levels are also a great addition even for games that have a set amount of levels that aren't procedurally generated, as randomly generated levels can still offer the same experience that the regular game does, but adds potentially infinite replayability, as developing a method to procedurally generate levels can give players countless new levels to play.

## History

Dungeon crawlers in the late 1970's and early 1980's started to feature procedurally generated levels. According to the paper published by Brewer in 2016, Beneath Apple Manor was an early example of procedural generation in games. It was released in 1978, and

featured procedurally generated dungeon levels, which had the player face a different dungeon every time they played. The more well known example from this era was the game *Rogue*, which was released in 1980.

*Rogue* also featured procedurally generated levels, but it also had other procedurally generated elements. An example given by the paper was that of a tin wand, which had different functionality on each playthrough. These early examples were made for early, low resolution computers, and as such, featured simplistic ASCII graphics. However, especially with *Rogue*, the levels that were being generated aren't too different when compared to some more modern examples, as *Rogue* features a series of rooms, connected by tunnels. The rooms in *Rogue* varied in size, and had either items or enemies in them.

Procedural generation of levels continues to evolve, and some modern games choose to feature procedurally generated open worlds rather than underground dungeons. *Minecraft* is the game everyone thinks of when it comes to procedurally generated open world levels, but there are other examples of it too. This goes to show how versatile procedural level generation is, and that it is applicable to a lot of different games, spanning across different genres, and across a lot of settings, from dungeons to simulating whole worlds.

## Approaches

When it comes to different approaches to implementing procedural level generation, there are a lot of viable approaches, but not all of them are universal, and sometimes, depending on the complexity of levels, or lack thereof, different approaches might be better suited for different applications. In this section, I will look at three different approaches that are very different from the approach I want to use, and these will be Markov Chains, Cellular Automata and Minimum Spanning Trees.

According to the paper published by Zafar, Ifran and Sabir in 2019, markov chains “is a method that models probabilistic transitions between different states” (Zafar, Ifran & Sabir, 2019). Markov chains are made up of three main factors, the set of states that the chain is composed by, the transition scheme between these states, and the emissions of outputs. Naive



Markov chains are restricted to one previous state, but higher order markov chains can have more than one state.

Markov chains are a good way of representing a 3D level similar to what mine will produce, albeit, markov chains could be used to create multiple rooms, with the state transition being used to create paths between the rooms, and this further benefits from higher markov chains which can contain more than one transition, i.e., a room being connected to more than one room.

Markov chains are most often used for creating 2D grid-based games, with several examples being given by the aforementioned paper by Zafar et al. In this paper, several different games are created to show markov chains being used to generate levels, and the way it was done was by creating a 2 dimensional array, and populating it with sprite data, including the avatar, which was controlled by the player, and other objects such as harmful and goal sprites. Then constraints were applied to it, to ensure such things as there being only one player sprite. After that, they calculated probabilities by counting how many times objects appeared and then finally generated the level based on those calculations.

Cellular Automata, on the other hand, according to the paper by Yahya, Fabroyir, Herumurti, Kuswardayan and Arifiani, published in 2021, is “an abstract object with two intrinsically bound components” (Yahya, Fabroyir, Herumurti, Kuswardayan & Arifiani, 2021). The first is the spatial structure that underlies cellular automata, and the second is a finite automaton, which is a copy that takes place at each node of the net.

Cellular Automata works best when paired with traversing algorithm, such as Breadth First Search, or in the case of the authors of the paper, Depth First Search. This combined with Cellular Automata can be used to create various types of levels, such as caves, or outdoor areas. The way the authors implemented their level generation algorithm is using Depth First Search by extending its root child until a destination is found.

Lastly, minimum spanning trees are graphs of nodes, that are connected using edges. Minimum spanning trees work along the principle that the nodes in the graph are connected

by edges, and the edges that are used are determined by their cost, with the lowest cost edge being used. The cost of an edge can be defined as many things, such as the distance between nodes, or even an arbitrary number given to an edge to make a minimum spanning tree.

In the paper by Lipinski, Seibt, Roth and Abé, published in 2019, a level generation algorithm is shown, where “vertices correspond to positions of junctions and rooms” and “edges to connecting hallways” (Lipinski, Seibt, Roth & Abé, 2019). This method produced pretty rectangular levels, however, due to all nodes not being used, the shape was less rectangular, therefore making their levels more interesting.

## Conclusion

Using this literature review, I’ve learned of different techniques that can be used for procedural level generation, and I learned that there is not a method that is the best to use in every situation, and it largely depends on the type of environment that is to be created. For example, with my particular algorithm, I see no reason for using any of the techniques discussed above, as due to the specification I set on the algorithm of it being box rooms connected to each other, a simple dice roll is enough to generate a level. Some of these techniques, mainly Minimum Spanning Trees, do have their uses and can create interesting levels and might not be that hard to implement if I ever make another level generation algorithm.

## System Design & Example of Use

For my algorithm, I want it to be robust but simple. For this reason, I made the level generator one big class, with the peripheral scripts, i.e., the room prefab, door, and wall scripts, being separate scripts that get called by the main class, but aren't necessary for the algorithm to function. For example, if the developer wishes to replace the room object with their own, rather than rely on the one being generated by the algorithm, they can do so, and the main algorithm will still function properly.

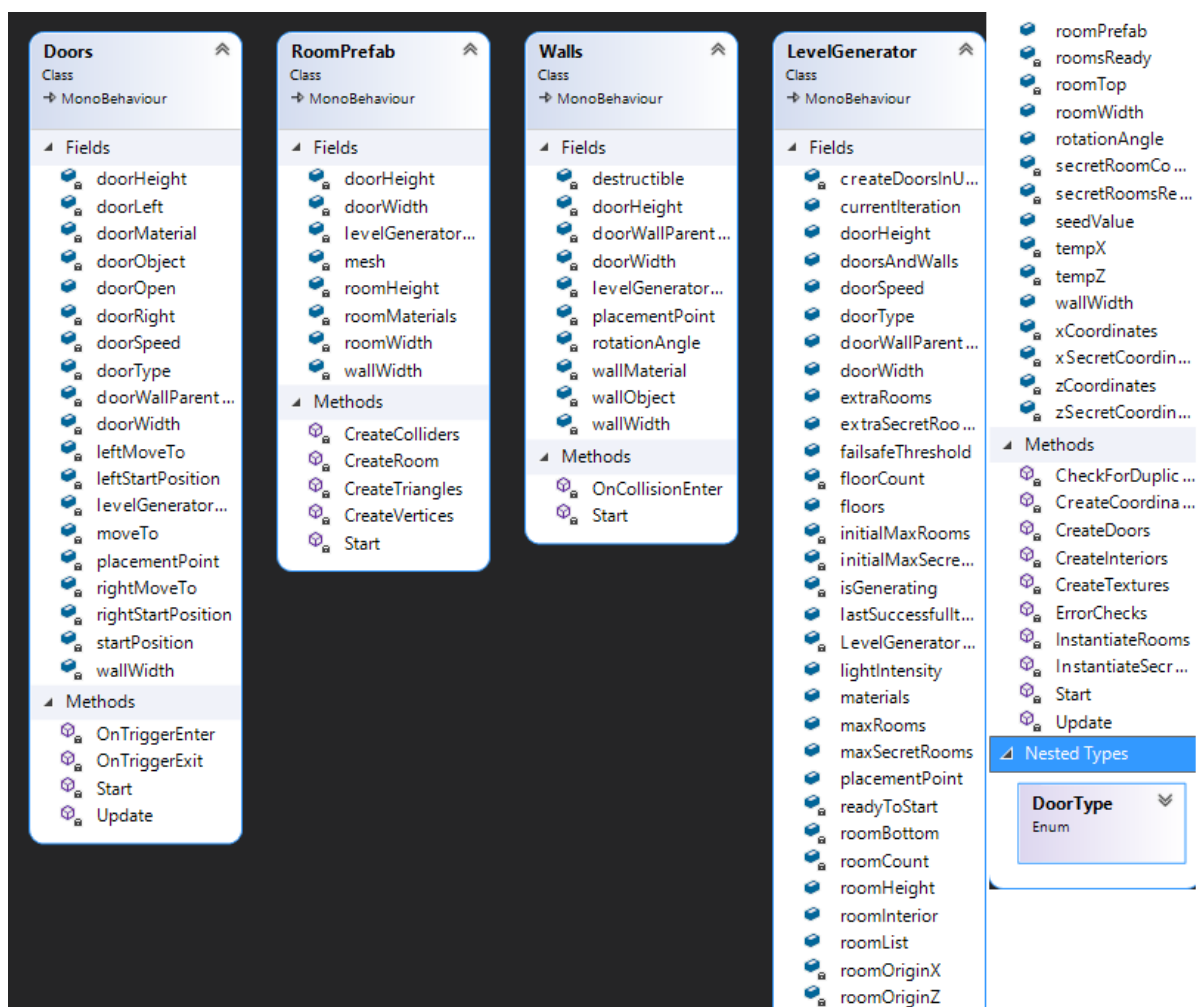
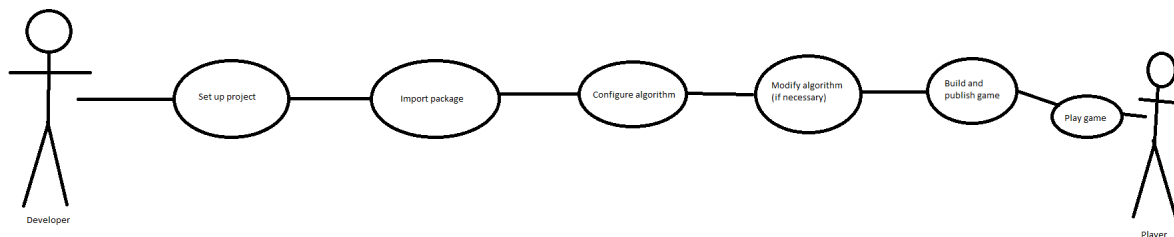


Figure 1: Class diagrams, in order, Doors, RoomPrefab, Walls and the Level Generator itself. The level generator is split into two parts because of its length.

Initial class diagrams had a lot less detail, with doors and walls being a later addition. This was done because I wanted to use raycasting to create the doors and walls, and this had to be done after the rooms were generated, rather than during room generation, as raycasting doesn't work if the objects aren't made yet. For this reason, I decided to create the walls and doors scripts.



*Figure 2: Use Case Diagram*

The scripts have two main purposes, the purpose of the main algorithm is to create the entire level, by generating coordinates, instantiating objects, performing raycasting to place doors and walls and so on, while the purpose of the other 3 scripts is to create a specific object, and to give that object the required amount of functionality, i.e., rooms must have meshes and colliders, doors must have a trigger that allows them to be opened, etc.

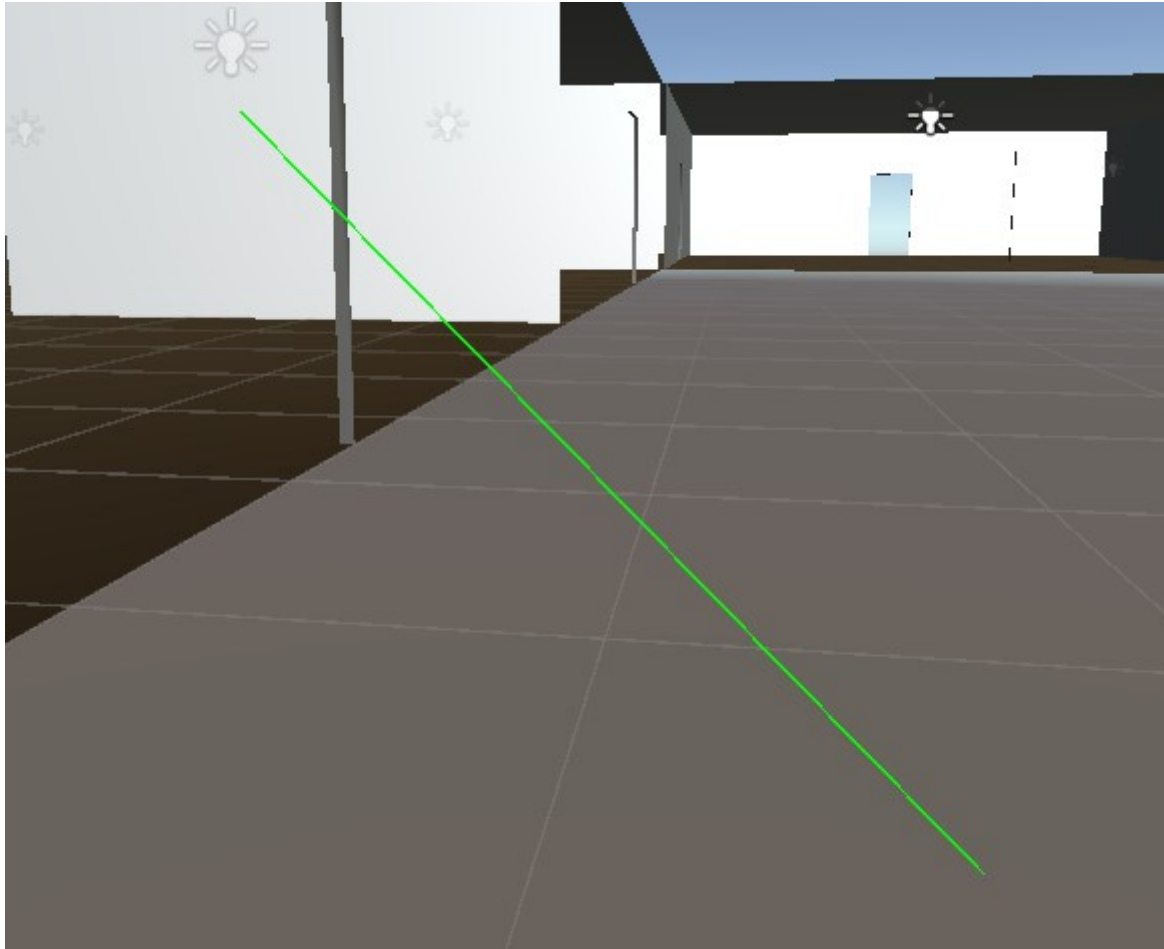
The level generation algorithm will contain two main algorithms that allow it to function properly. The first one is coordinate generation.

Coordinate generation will be done in a fairly simple manor, but will still provide an interesting level thanks to random number generation. How it will work is that it will be given a starting room, at (0, 0), and then it will branch out, going up, down, left and right. This will be done using two randomly generated numbers, either 1 or 2, and these numbers will make the algorithm go up or down, and left or right respectively. Another factor in making this work will be that only one of the two coordinates, X or Z, as Y isn't necessary

because the elevation will remain the same, will move at a time, meaning that the rooms will always be connected.

For example, starting at (0, 0), if the next two rooms are (0, 10) and (0, 20), this means that they are still on the same X axis and are therefore connected. If the rooms are (0, 0), (0, 10), (-10, 10), then the first two rooms are connected on the X axis, while the latter two rooms are connected on the Z axis, meaning that all 3 rooms are still connected and can be traversed. The two criteria that must be met with coordinates are that they must be unique, as to not make rooms inside rooms, and that they all must be connected to make sure there are no separate rooms that are inaccessible. Thanks to changing one axis at a time, and going in 4 directions only, with no diagonal movement, this means that the rooms will be connected no matter what.

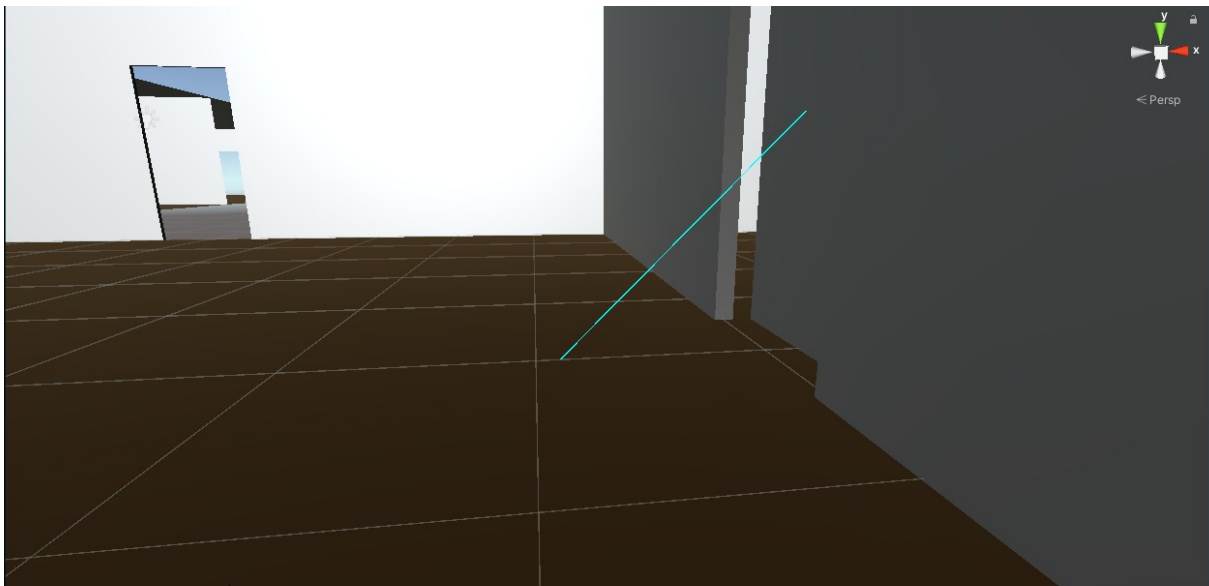
The other core algorithm will be raycasting, for determining whether a door or a wall needs to be placed, or checking if there is an object already there to avoid collisions. This is necessary for every room, as every room is a box with 4 doorways. How this will work is fairly simple, a raycast will be done 4 times for every room, for each of the 4 doorways. The raycast will be done from inside the room, slightly behind the doorway, so that it has room to check if there is already a door or a wall there, and the raycast will have 3 outcomes.



*Figure 3: How raycasting works, demonstrated by the Debug.DrawRay function in Unity. DrawRay was chosen over DrawLine, as the code to show it is the same as the raycast, with one additional parameter being added to it, the colour of the ray.*

The first one is, that if nothing is hit at all, it will create a wall, because if nothing is on the other side, that means there is no room there, therefore the area is not playable, meaning a wall is placed there to prevent the player from going outside of the level. The second outcome is that it hits an existing door or a wall, meaning that it doesn't have to do anything, because there was already a wall or a door placed there from a different room, and an additional object is not to be placed as it would cause collisions and duplicate objects. The third outcome of a raycast will be that it hits a room, and if a different room is hit, it needs a check to see if the Y position of that room matches the Y position given to the raycast. Due to the scalability of this algorithm, the exact distance needed for a raycast will depend on room

sizes, therefore, the raycast will overshoot the necessary distance to check the floor of the adjacent room, and can hit the roof of the room on the level below it if there is one present. This is the reason for the Y check, as if the Y doesn't match the expected Y, it will create a wall, because the room that was hit isn't on the same level, therefore, it isn't reachable. If the Y does match, however, this means that the room that was hit is adjacent to it, and is on the same level, therefore a door is created as the room is meant to be able to be reached by the player.



*Figure 4: Raycasting being done using adjacent rooms. This was taken as the raycasting method was being developed, therefore, there are visible issues, such as a wall instead of a door, and the positioning of the wall is also incorrect.*

The main reason behind the use of raycasting is twofold. The first one is that it is a lot less resource intensive, and faster, than comparing the room to every single coordinate, and checking whether a door or a wall is to be placed. If this method was used, it would take a lot longer, and there would have to be additional code added to it to make sure duplicates aren't placed in every room that has a room, or additional rooms, adjacent to it.

The second reason is that I already code very similar to this, inside the duplicate checking method, and as this is a project that is coding focused, I felt the need to use a different method to develop this functionality, as using the same basic code twice would be neither

impressive or good for marks. Also, using raycasting with the raycast origin being outside of the placement point, means that I don't have to write additional code to check and to remove duplicates, as if it didn't match one of the two if statements, then it would simply do nothing, and wouldn't waste resources on instantiating the object and having to remove it later. An additional benefit to using raycasting is that it will help me learn about the method itself, as I didn't get to use raycasting in a project, and raycasting is a very powerful technique, that is light on performance, and has a lot of different uses, as shown in its prevalence ever since the 1990's in games.

	I want to generate a level	Using an algorithm to generate the level	This algorithm will be very configurable	And make a game using this algorithm
Make it work	I want the algorithm to generate the level using two arrays for co-ordinates	Using one script to generate the level will make the level using these co-ordinates by iterating through the co-ordinates and making rooms from scratch	The algorithm will be very versatile, and will allow the developer to configure almost all aspects of it	The game will be a rogue-like FPS where the player must explore the level to progress
	V	V	V	V
	Using these random co-ordinates, the algorithm will make rooms at these points	The rooms will be created using an array of materials, and several sub-meshes created by vertices and triangles, and these rooms will have box colliders so that they will be playable	The algorithm will also enable the developer to replace parts of it with their own assets, namely, the room objects and interiors	The level will have rooms with enemies and hazards as well as regular furniture
		V		V
Make it better		The rooms will be generated, with their origin being in the bottom left, and using raycasting, door and walls will be added		The game will have several weapon types and several enemy types
		V		V
		These rooms will have interiors, comprising of lights as well as random props		The player will be encouraged to explore by getting upgrades, more ammo, health etc.
		V		
		If the user wants more floors, they will be generated after the first floor, with extra rooms if so specified		
Make it best		V		
		The level might have secret rooms, and these rooms will be accessible through secret doors		

*Figure 5: User story map. Green represents the feature being 100% complete, yellow represents it being at least partially complete, albeit with issues, and red represents it being absent from the project*

The user story map has changed over the course of the project, mainly due to scrapping the alternate algorithm as was initially planned, although this version was never the focus, and was included as a backup if I couldn't get the original version of the algorithm of it working. Its purpose, however, still remains, as the developer can still replace the rooms being generated by the algorithm with their own, but it will take a bit of messing around with the configuration of the algorithm to get it to work correctly.



## Implementation

Using class diagrams and user story maps created at the start of this project, I started by creating the basics of the algorithm, which was split between creating the rooms themselves, and getting the basics of the level generation algorithm working at the same time. While doing this, I stuck to my user story map and class diagram as much as possible, but I added new things if I felt that it was appropriate, or a better way of doing it than how I originally intended.

In the beginning, most of the work was put into getting a room to be generated by code, specifically its meshes and corresponding materials, as well as adding colliders to it so that it could be navigable. To get a mesh to generate was relatively straight forward, as it required to give it the vertices at which to generate, then drawing the appropriate triangles using those vertices, as two triangles were required to create one plane, and this had to be repeated for all the surfaces of the room. Once these were supplied, the appropriate functions were called from Unity to make these surfaces function properly, i.e., calculate normals, recalculate bounds etc.

During this section, I was having a hard time understanding how sub-meshes work in Unity, but once I figured it out, I split the triangles required for each surface into separate arrays so that I could add them one at a time for each of the 4 sub meshes, rather than add them all at once. At first I thought using sub-meshes would be better for performance than generating a mesh for each surface, but after doing some research on Unity forums, this turned out to not be the case and the performance is the same in both cases. However, using sub-meshes made the project a lot more organised, as the room that's generated is one single object, with all the appropriate components attached to it to make it work, and thanks to the sub-meshes, each sub mesh has the appropriate material attached to it, which can be changed by the developer themselves if they want a different one, or the code can be changed for it to better match their desired colours.

Once I figured out how sub-meshes worked, creating the rooms went from confusing to tedious, as I now knew how to get them to work, but I had to go through and create each

vertex and each triangle in a manner that allows it to be scalable no matter what the room size is. This was fine at first, but once I got to the walls this slowed down a lot, because of the walls needing two parts and head room above the door frame, meaning that it was triple the work compared to making the floor or the ceiling, as well as making sure that these vertices and triangles were always in the correct spot. This tedium further applied when having to work with box colliders, and both these processes involved repetitive trial and error work. In the end, however, I got it to create a room that would always act as intended, no matter what size it was, what size the door was, or how wide the wall was, and all of these always had the appropriate colliders.

During this time, I also started working on the level generator itself. My idea was that if I can get a room to generate perfectly, then it will just be a matter of creating more copies of it and placing them appropriately, and this will form the level. Also, to clarify the algorithm, level generation takes place in the main script called `LevelGenerator.cs`, while the room creation is handled in a periphery script called `RoomPrefab.cs`, and this also applies to doors and walls who have their associated scripts. Overall, this algorithm has 4 scripts all together, but aside from the main script, the others can be replaced by other objects and other scripts.

Before starting the algorithm itself, I begun by adding a system to seed the generation, as a random level generator is hard to test without being able to reproduce the same level over and over again, and a seed functionality is really useful for the developer who this algorithm is made for as well, for both testing and the gameplay of the game developed using this algorithm. The seed itself is always randomised by default, i.e., when the seed is left on 0, but the seed gets set if a number is provided for the algorithm. To test this, I created a simple for loop that generated 2 sets of 10 random numbers, and ran the test on multiple seeds to make sure the numbers are being reproduced on specific seeds, and randomly generated when left random.

Once I had this working, I started writing the function to generate the random coordinates. The hardest part of this was to make sure that each coordinate was unique, and that the coordinates are always navigable, meaning they're all connected, and that every coordinate is reachable. Thankfully, because of my algorithm creating square rooms, and going up, down,

left and right, this meant that any coordinate will be reachable, as long as the rooms are connected on one axis.

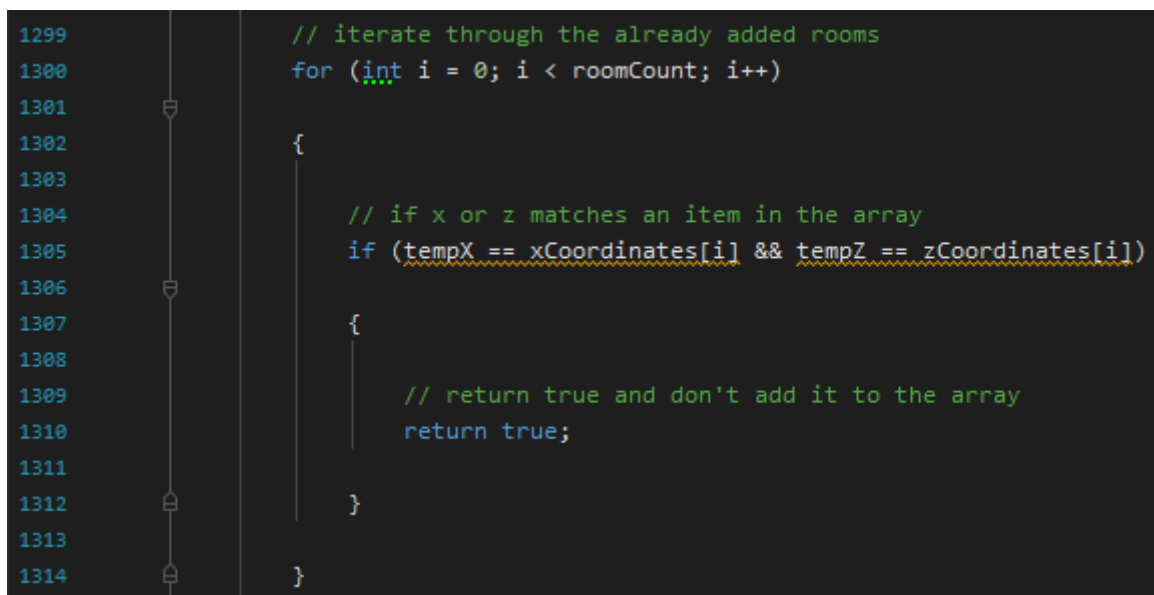
How the algorithm works is that it branches out from the previous coordinate, and goes in one of the 4 directions, chosen by random number generation, and this keeps going until the desired number of rooms is met. The function for doing this was largely unchanged for the duration of the project, with the exception of having more features added to it. The short version of it is that it generates the X and Z coordinates, with Y being constant for each floor. The first step is creating the first room where the algorithm will branch out from at (0, 0).

After that, two random numbers are generated, `verticalOrHorizontal`, and `increaseAndDecrease`. These both have a range of 2, and can be either 1 or 2. Vertical or horizontal determines whether the coordinate being manipulated will be either X or Z. If the number is 1, X will be manipulated, and if its 2, Z will be manipulated. Due to the fact that only one of the two coordinates of a room is being manipulated at a time, this means that the previous room will always have a connection to the previous, for example, if the X changes, the Z will still be the same, so that means the player can still move to the old room along the Z axis. Once X or Z is chosen, the second variable comes into play, and this either increases or decreases the value of the chosen axis.

```
381 // if verticalOrHorizontal is 1, it will alter the x axis, which moves the room left or right
382 if (verticalOrHorizontal == 1)
383 {
384     // if leftOrRight is one, increase x axis by the roomWidth
385     if (increaseOrDecrease == 1)
386     {
387         // ...
388     }
389     // else if leftOrRight is two, decrease x axis by the roomWidth
390     else if (increaseOrDecrease == 2)
391     {
392         // ...
393     }
394 }
395
396 // if verticalOrHorizontal is 2, it will alter the z axis, which moves the room forward and backwards
397 else if (verticalOrHorizontal == 2)
398 {
399     // ...
400 }
```

*Figure 6: Code responsible for choosing wheter to increase or decrease X or Z, inside the if statements it simply adds or subtracts the number and calls the CheckForDuplicates function.*

Once the new coordinate is determined, it gets checked against the existing coordinates. Existing coordinates are stored in two arrays, one array that holds X values and the other that holds Z values. This is checked using the CheckForDuplicates() function, which sorts through the array of existing, unique coordinates, and if the coordinate being checked isn't in the array yet, it gets added. This happens when the method returns false, if it returns true, a new coordinate is generated from the previous room. This is done inside a while loop, and the loop only gets broken out of on a successful new coordinate. This process is repeated until the desired number of coordinates is generated.



```
1299 // iterate through the already added rooms
1300 for (int i = 0; i < roomCount; i++)
1301 {
1302     // if x or z matches an item in the array
1303     if (tempX == xCoordinates[i] && tempZ == zCoordinates[i])
1304     {
1305         // return true and don't add it to the array
1306         return true;
1307     }
1308 }
1309
1310 // return false if no duplicates found
1311 return false;
1312
1313 }
1314 }
```

*Figure 7: The function for checking duplicates, if none of the statements are met, it returns false as the coordinate being generated isn't a duplicate.*

With this functionality, creating the rooms themselves was easy, as at each coordinate, an empty object was made, and by attaching the RoomPrefab script to it, which holds all the code for creating a room, a room is constructed at the point, and this is repeated for as many coordinates as there are. The origin of these rooms is in the bottom left corner, but that only comes into play with raycasting, as specific coordinates are required for that, and with interiors.

While testing room generation, I found out that rooms having nowhere to branch out from, due to being surrounded on all 4 sides by other rooms, caused an infinite while loop. To fix this, I made a fail safe, and how it works is that each time the loop is iterated through, the iteration count increases. When a coordinate is successfully added, the last successful iteration is recorded. If the current iteration is far enough from the last successful iteration, it will jump to a random coordinate that has already been added to the unique coordinates, and it tries to branch out from there. I used the same approach for secret rooms, and how they work is that once the regular rooms are created, a random normal room is chosen, and the secret room tries to branch out from it, and like regular rooms that are stuck, will jump to a random coordinate until it finds one where it can be placed. The fail safe number is set in the inspector, for my testing I used 10, however, this still doesn't mean all options are exhausted, but that really doesn't matter. In fact, setting this to lower numbers might even produce a more interesting pattern because it will have to branch out more often, the only downside this will have is that the algorithm might take longer to produce the level.

After this I then implemented the other peripheral scripts to help support this level generator, the Doors.cs and Walls.cs scripts, but I also created a basic first person character controller with mouse movement. While it didn't get to be used in a game, it was still helpful for testing how navigable the level is, and testing the functionality of colliders and doors.

The walls and doors are very similar to each other, the main difference between the two is that they both use different textures, the doors use the 4<sup>th</sup> material in the Materials array, which is used for door frames, while the walls use the 3<sup>rd</sup> material, which is used for the walls themselves. Besides that, the doors have an additional functions added so that the doors open when the player is touching the trigger attached to them and close when the player isn't touching the trigger. This was done using the same box colliders as for the walls, except the boolean IsTrigger is also used. The additional advantage of having a player controller to fully test this, is that triggers require one of the two colliders to have a rigidbody to work, which in this case, the player does. The event itself is very simple, the door gets set to open when the player enters the trigger and closed when they exit the trigger. Then inside the update method, there are additional if statements which open the door if the door is opened by the

trigger, otherwise they are set to close if they aren't in their starting position. Most of the code is simply creating a primitive cube, setting it up to fit its role, and inheriting the values to make it work from the level generator script itself.

When it comes to the implementation of raycasting, there is an empty object, which serves as a point of reference for the raycast, as this is where it originates from. After that, the ray gets cast downwards, and in the direction of which the door is facing, i.e., north door will be downwards and forwards, east door will be downwards and left etc. What it's meant to do is to cast a ray, and if nothing is hit, then place a door because there is no room there, if a door or a wall gets hit, nothing happens because there is something already there, and if it hits another room, if the Y of the room hit doesn't match the Y of that room, i.e., its a floor below, place a wall, and if the Y matches, then place a door because there's a room there and it's on the same floor, therefore the player must be able to walk there. At the moment, it doesn't work properly and only places points for walls, and I don't know the cause of this, it worked with only one secret room being checked and then it stopped working once multiple secret rooms and regular rooms were being checked also.

With all the elements present, and with peripheral scripts ready, the only thing the algorithm has to do is call methods now. First, the seed is checked to determine if it has to be seeded using the specified number or not. Next, it checks for errors, which causes the play mode in Unity to stop if semantic errors are present, i.e., the height of door frames is higher than the height of the room itself etc. Once it is established that there are no errors present, it first sets up the textures and coordinates. Textures are either made from code, or using the developer specified textures, and coordinates are generated using the aforementioned function, then rooms and secret rooms are instantiated. After this is done, the door creation process begins which involves the raycasting. Once this is done, finally, the interiors are made, and this process is repeated inside the for loop until the required number of floors and rooms are created. The number of floors, rooms, secret rooms, as well as how many extra rooms and secret rooms appear in each subsequent floor, are all determined by the configuration of the level generator script in the inspector.

# Testing & User Manual

## Testing

In the initial stages of developing the algorithm, a lot of testing was done to ensure that the algorithm worked properly with seeding. A lot of this testing was done while writing the coordinate generation function. The first wave of testing was checking if random number generation is seeded properly, this was done by running the same seed multiple times to make sure the results were the same, and running multiple seeds to see if they work properly, as well as leaving the seed value at 0 to force it to be random, and making sure that the results were random each time.

Once seeds were implemented, the next feature I tested a lot was coordinate generation once I added the requirement for the coordinate to be unique. At first, I made sure that these coordinates are the same when using the same seed, but more importantly, I tested a bunch of seeds multiple times, and then went through the coordinates to make sure that all of the coordinates are linked in a linear sequence, meaning that every coordinate is connected to the previous one, meaning that the level is one linked series of rooms.

Later, after rooms were being created at the coordinates provided by the aforementioned function, I did some stress testing by creating several hundreds of rooms at the same time. This testing produced an error that I should have seen coming, but I haven't at the time. After narrowing down what room is causing the issue, I realised that the issue arose from a room having nowhere to go because all 4 sides of it were already used by a different room. This, combined with being inside a while loop, created an infinite loop, which causes Unity to freeze forever. Once this issue was resolved, I tried creating thousands of rooms, which were made without problems, seeing the fail safe get triggered several times. This gave me the confidence that the algorithm will work flawlessly on any seed until other features, namely, secret rooms and multiple floors, are added. The only downside with the number of rooms increasing would be that it would take longer and longer to generate, but once the level generates, the level would function.

The final wave of testing came when secret rooms and additional floors were implemented. The main focus was mainly to figure out if the levels were still generating properly with multiple floors, and whether the additional room feature worked properly or not, i.e., the second floor had X rooms more than the first, and the third had X more than the second, both for regular and secret rooms, depending on the number that was set. Doing some stress testing to see if it would work, I tried generating thousands of rooms, but due to having to deal with multiple floors, the load time was significantly longer than before, as with one floor only, the difference between thousands and tens of rooms was barely noticeable when generating, but now it took close to a minute to generate thousands of rooms per floor with 3 floors. However, the level still generated, albeit I didn't check if the proper number of rooms was generated as I didn't have time to go through the estimated  $3006 + 6$  secret rooms per level.

The areas I wanted to test more but I couldn't due to time constraints were mainly raycasts. When writing this function, I started with secret rooms, and I made the code to raycast for that room only. When I saw that it returned the appropriate results, in the case of that specific room, 3 misses and the last one hit, I didn't test it from then on and added more code to the method to repeat that for every room, starting with secret rooms and then doing regular rooms. After testing this later, I noticed that the method no longer worked, as it always returned a miss on the raycast.

Overall, besides the raycasting function, the testing I've done across multiple seeds with random configurations of the level generator, makes me pretty confident that the algorithm will produce a playable level no matter what the developer will throw at it, the biggest concern here would be generation time, but once done the level should be playable and function properly.

## Manual

The first step in using this algorithm is to install Unity and open a new project. I developed this algorithm in Unity 2020.3.36f1, but this script should be compatible with almost all version of Unity.



Once inside Unity, with the algorithm package downloaded, the next thing that's required is to add the package to the script. This is done by going to the top right and clicking:

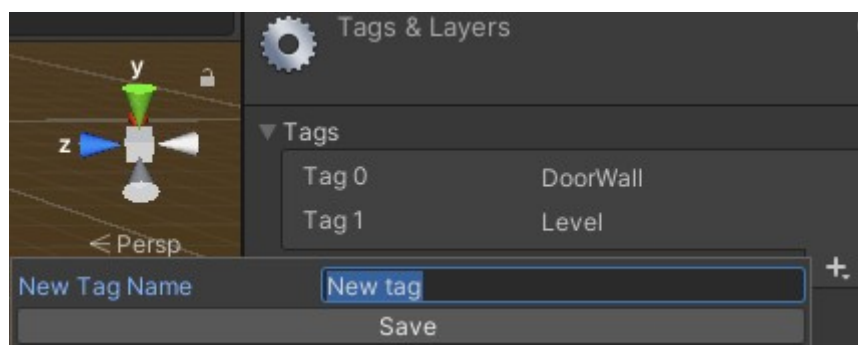
Assets > Import Package > Custom Package...

and pointing to the download location of the package.

Once the package is imported, we will set up the tags necessary for this script to work properly, as this step is easy to forget. Sadly, Unity doesn't allow tag creation at run time, so this step has to be done manually. Inside the hierarchy, select any object (the object doesn't matter), and in the inspector, select the Tag field. By default, the object will be Untagged, however, some objects will have a tag that's different, for example, the Main Camera will have the MainCamera tag. Regardless, expand this and select Add Tag...

Once you press Add Tag... you will see a new menu in the inspector, here you want to head over to the Tags drop down, and click the plus. Here you will add two separate tags:

- DoorWall
- Level



*Figure 8: The tags menu, and how the tags should look like after adding the two necessary tags.*

These can be in any order, on top of your own tags for later. Once added, click on a different object in the hierarchy to go out of that menu. You don't have to worry about these, as the script will add them to appropriate objects. You might however use those tags again, for example, the Level tag might be used if you are to add features such as jumping, which will check if the player is touching the ground, i.e., an object with the Level tag.

Once the tags are set up, the level generator itself can be added to the scene in several ways:

Option 1:

You can simply open the Game scene I included in the package. This is an empty scene I used to develop this algorithm, and contains a player and an empty game object containing the script.

Option 2:

Inside the scene where you want the level to generate, create an empty game object in the hierarchy by right clicking it and selecting Create Empty. Once this empty object is made, attach the LevelGenerator.cs script to it.

Option 3:

Inside the scene where you want the level to generate, drag in Level Generator prefab (blue cube) from the assets folder into the hierarchy.

Once the level generator object is inside the scene, with the script attached, the next part is to configure it. Simply go through each of the fields in the inspector, hover over it to read the tooltip, and set the number you want. If the field in the inspector doesn't have a label, it can generally be ignored and left as is.

Once the generator is configured, if you run the project and it closes instantly, check the console for error messages and address them. If there are no errors and the level generates properly, feel free to mess around with it further, and if you want, include your own assets, namely, custom textures in the Materials array, or modify it to add more functionality to it.

## Conclusions & Recommendations

In conclusion, I'm happy with with project, as besides developing a game made using this algorithm, I added all the functionality I aimed to add, even if some of it is only partially implemented. The features I worked on towards the end are rushed, and not 100% complete, but they're still there, and aren't too far off being polished and made to function properly.

The glaring issue with this algorithm is that the doors and walls aren't being spawned properly, as the points for them to spawn are in place, but they all spawn on top of each other, and the second problem with it is that the raycasting, at least the order in which Unity does it, doesn't spawn doors in their appropriate places. If these issues, along with fixing the code that makes every interior spawn twice in every floor except the first, are fixed, then this algorithm will be 100% ready to use for any developer wanting to use it.

There are other minor issues, such as there being no way to move to upper floors at the moment, and the doors always moving in one direction regardless of rotation, however, these wouldn't be hard to implement and fix respectively, as the navigation between floors isn't hard to add, the easiest way, and the way I was going to implement it, would be to have an object in one of the rooms, most likely, the last one that's generated, that when interacted with, will set the player's position to the starting room and increase their Y position so that they are on the floor above the one they just finished.

The reason I didn't add this, is because I felt that this code wasn't very impressive, and didn't add much to the project, so I instead focused on getting the other aspects of it more polished. The more elegant solution would be to create a separate room, that is attached to one door in a randomly selected room, that acts like an elevator, and when the player enters it, they get transported to a different floor. A simpler version of this could also be achieved by spawning a special door which functions in the exact same way, by being attached to a random room, and once interacted with, the player gets moved to the next floor. Also, with the doors, the problem can be easily solved by making the door move right on a local position, rather than

global, so that the door will always go to the right, no matter the rotation. Again, I instead chose to focus on getting other features polished.

The one feature I didn't really get to work with was the interior system. The reason behind this was pretty simple, everything in my algorithm has scalability built into it, no matter what size the rooms are, the algorithm takes it into account and builds the level the developer wants. The problem with interiors is that for it to not feel random, and feel like a proper level, is that it has to be built for the specific room. The problem when taking that into account with my algorithm is, that if an interior is built for a 10x10 meter room, how will it look in a 9x9 meter room, and a 11x11 meter room? In the 9x9 meter room, it will be too big for the room and start clipping through walls, in the 11x11 meter room, it will be too small, and will either be out of place, or will look weird, as there will be space around it where the walls should be in the smaller room.

What's the solution for this? There isn't one, the interior would have to be custom built for one specific room dimension. Even things such as object scaling wouldn't help, as no matter what size the room is, if it's not the same as the room it's built for, it will feel out of place, and it will be distracting as it will be noticeable in every room. In every room, the interior will feel out of scale, either by a bit or drastically, and this will be very off-putting for most players. The only solution for scaleable rooms like this is to have random props, such as barrels or boxes, that are randomly placed inside the room. However, this wouldn't make for a very interesting level either, therefore, I decided to leave the until the end and focus on more important things, and I didn't get far enough to implement this.

However, an interior is still created, albeit, it only contains a light. The code can still take a game object instead of being forced to build the blank room with a light that I currently have for the interior, meaning that with some fiddling, developers can still add their own interiors, but it will be the same for every room, meaning that the function will have to be extended. This can be straight forward to do, as instead of taking one game object reference for an interior, take several in the form of an array, and randomly select one for each room, maybe hardcoding that the first room will always have a specified one, and the final room will always have a specified one.

Besides these points, the algorithm achieved everything I wanted. It can still build a level properly, it still has all the scalability, and additional settings to help the developer customise it for themselves, its still written in a relatively straight forward manner, meaning that modifying it shouldn't be too hard. While I haven't put much testing into this feature, it still lets the developer replace peripheral scripts, such as the room prefab script, and replace it with their own objects. While it would take some tampering to make work properly, as long as the room dimensions in the script are set to match the ones of the developer's objects, I don't see how this couldn't work, as any room from the outside is just a cube or a rectangle, the inside is the only thing that changes.

Performance, especially when it comes to generation times, aren't a concern for small levels, although, higher floor and room counts will certainly start to cause slow generation times at one point or another. There is most likely a way to optimise this, but the delays that were added, e.g., the rooms having to be generated before raycasting can start, certainly don't help this problem.

This project helped me learn a lot about Unity, this is excluding the simpler stuff such as new function calls to achieve new functionality, although there was a lot of this too, with me learning the basics of raycasting being certainly beneficial to any other future game development project, even outside of Unity. The main things I learned were how Unity isn't a perfect engine, and this issue kept coming up, but was especially prevalent during the final stages of this project. One of the big downsides I saw to Unity is the ambiguity when it comes to order of code execution, and the lack of options for it. In Unity, awake is called first, then start, and after start every other function is called, as well as update and all of its variations being started once start is finished. One big example of seeing this was when I added the error checking method, and seeing how half the level was already generated by the time it stopped, meaning that it was both wasting resources generating a level that will be cancelled, but it also showed that the start method isn't working exactly as I expected it to. This also resulted in having to add additional delays, such as the raycasting starting once two methods were finished, and having to use booleans to let it know its ready.

For the future work that has to be done for this project, the obvious thing to point out is fixing the issues that are currently present. This ranges from smaller things, such as doors always moving in one direction, to more important ones, such as fixing raycasting, as well as the door and wall spawning issue, as fixing these two issues will solve almost all the problems with this algorithm. For raycasting, the raycasting seemed to have worked when it was only testing one room, as when using debug statements, it logged it hitting one room and missing the other 3, the exact result I expected and the one that was correct in that case. My advice would be that there is code that messes with this for regular rooms, or that the issue is being caused by the start method, and trying to move the code somewhere else.

For the walls spawning in one spot, the code has to be rewritten, as currently, the wall is being spawned at a point supplied by the level generator script inside the wall and door scripts, however, this doesn't get updated in the start function in the level generator script, meaning they all spawn in one place. When moving this to the awake function to see if it would make a difference, Unity instead was picking up the places to spawn them one at a time, the problem was that the rest of the code was still in start, and awake takes place before start, meaning it did nothing. Perhaps moving this code to awake could help fix this, but I'm sure it would cause other issues as well, the better method might be to create the rooms after start, once update starts, and prevent the player from moving until everything is in place.

After the algorithm is fixed, where to go next? Next would be a way to deal with the interiors, but as I mentioned before, this is highly game specific, and might be better instead to create an array of game objects and spawn them randomly for one room dimension rather than trying to be modular.

Another improvement that could be done to the algorithm is rewrite some of it, so that it takes room width and room breadth, meaning that the rooms could be rectangles as well and not just cubes, this could add some nice gameplay variety, and would make for more interesting level design too, as the whole level wouldn't just be a series of square shaped rooms.

The last point for future work, would be to make a game using this algorithm, whether it would be the developer using this algorithm, assuming they can fix it, or if I'm using it, if I

decide to come back later and fix it. If I were to make a game with this, I would still stick to my original idea, which would be a Wolfenstein 3D type game with a bunch of weapons and enemies, and some roguelike elements in it, such as the player levelling up, and keeping levels in between restarts.

## Appendices

### Bibliography

- Brewer, N. (2016). Going Rogue: A Brief History of Computerized Dungeon Crawl.  
<https://web.archive.org/web/20160919020229/http://insight.ieeeusa.org/insight/content/views/371703>.
- Lipinski, B. v., Seibt, S., Roth, J., & Abé, D. (2019). Level Graph – Incremental Procedural Generation of Indoor Levels using Minimum Spanning Trees.  
<https://ieeexplore.ieee.org/document/8847956>.
- Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural Content Generation in Games*. Springer.
- Yahya, N. M., Fabroyir, H., Herumurti, D., Kuswardayan, I., & Arifiani, S. (2021). Dungeon's Room Generation Using Cellular Automata and Poisson Disk Sampling in Roguelike Game. <https://ieeexplore.ieee.org/document/9608037>.
- Zafar, A., Irfan, A., & Sabir, M. Z. (2019). Generating General Levels using Markov Chains.  
<https://ieeexplore.ieee.org/document/8974310>.