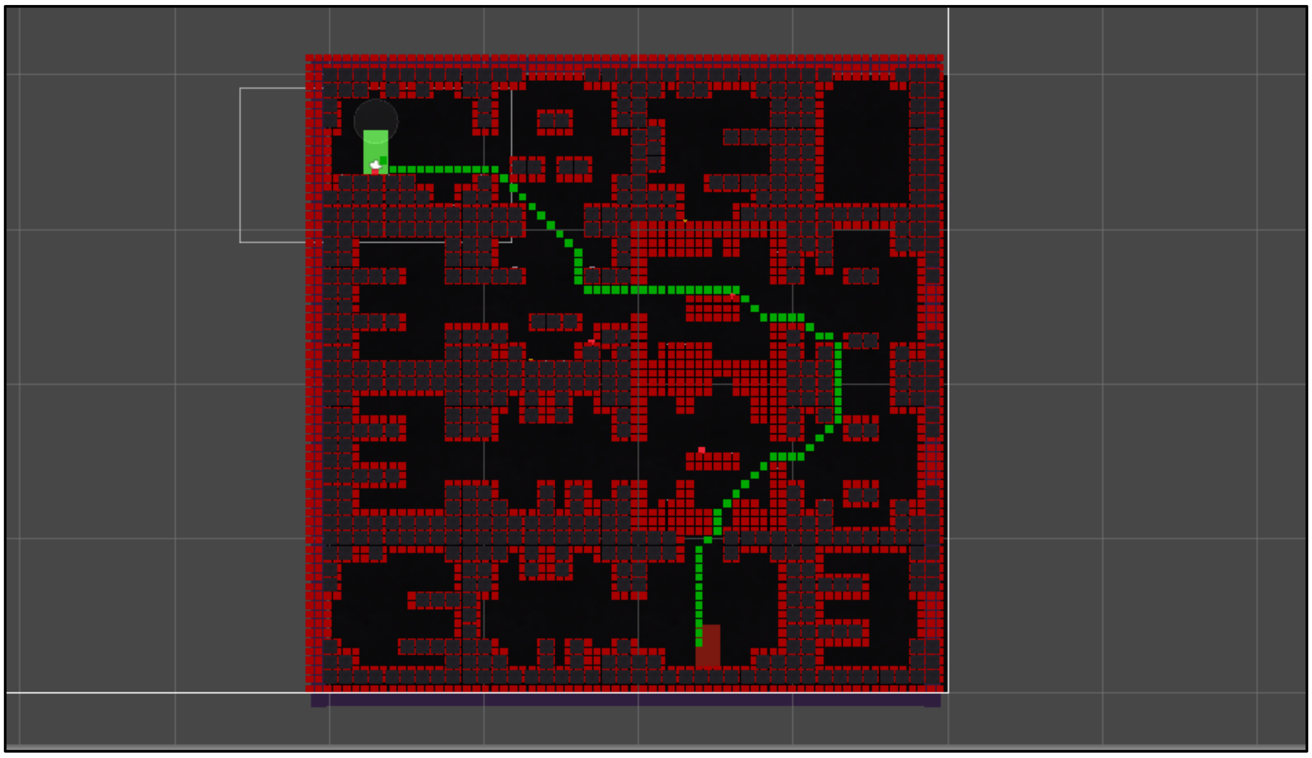
Comprehensive Creative Technologies Project: Project Title (in Tahoma, 24pt)

Kieran Boyce
Kieran2.boyce@live.uwe.ac.uk
Supervisor: James Huxtable

Department of Computer Science and Creative Technology
University of the West of England
Coldharbour Lane
Bristol BS16 1QY



Abstract

The project is a procedural level generator for 2D platformer style games where a user can change around parameters to change how a level is played. The project is built in the unity engine using an algorithm that generates prebuilt rooms in random orders to give a unique level each time. the aim of the project is to make it easier for people to build a 2D platformer.

Keywords: Procedural Generation, Unity, Platformer, Artificial Intelligence

Brief biography

I am Kieran Boyce my aims professionally in the future is to be a gameplay programmer I chose to a project on procedurally generated levels because I thought it would be a intrresting project to try especially due to the difficulty there can be planning gameplay around randomly generated levels. During my time at university I have deloped skills programming in multiple languages as well as learning several design techniques that can use to make this project.

How to access the project

The project can be acceseed here: https://github.com/Boyceinfenwa/CCTPSub

1. Introduction

Procedural generation - the practice of content and asset generation using algorithmic processes a lot of the time this is done using a combination of assets created from assets and randomness of a computer. In recent years procedural generation has found popularity in the games industry and this newfound focus on topic placed emphasis on the quality of the content that is generated. Especially as a lot of early content on procedural generation focused on creating systems that were able to generate large worlds without any consideration on whether or not the player would stay interested in the product.

The problem of having good quality content for a system that generates content is imperative as if left unsupervised some results could end up generating completely irrelevant content. Therefore, some designers put constraints on the generation like in Spelunky (2008) where the content generated is crucial to the main structure of the game which in this case would be the levels. The focus of this project is to be able to generate levels based off of parameters set by a user while taking into account whether or not levels are actually feasible to complete with the hope to make it easier to create 2d platformers of a good quality.

This Project will be executed by building a 2d level generator using the Unity Engine where the user is able to edit parameters such as:

- Difficulty
- Level rooms
- Rewards
- Level size

To ensure that it is possible for all the levels to be compelted there will be an AI using a search algorthm to check whether each level generated can be completed.

The key deliverables of the project are

- A 2D level generator project.
- Unity package for generator.
- AI that can tell the gnerator whether levels are feasble or not.

The project objectives are to:

- Generate playable levels using a procedural generation algorthm.
- Have a method to check wether or not levels are feasable using an AI search algorithm.
- Have a small game that is made of generated levels.
- Generate fun levels for players.

2. Literature review

Level Generators

According to Diaz (2015) there are a couple of different ways to generate levels which consist of:

- Combining premade parts - putting premade parts i.e., rooms into a pattern and repeat using different levels of complexity.
- Rhythm based – uses smaller premade parts and generates levels based on player actions also referred to as the rhythm of the level.
- AI based – Getting an AI to play a random level and provide feedback.

The advantage to using the premade parts method is the fact that it is simple and easy to implement which is why it is the most used method as well as this the user has more control over what the outcomes can be. Rhythm based generators are best used in long levels i.e., Mario levels rather than small rooms. AI based generators are good as they can eliminate the need to test levels and eliminate any repetition however it is least efficient as it is best developed when the full game in mind as well as this it is not a lot of documentation on how it can be implemented.

Flow theory

Flow is the state of engagement and concentration achieved when somebody is undergoing a challenging task according to Cziksentmihalyi (1990). This can be applied to games as well, because whether a game is a success can depend on if the player has achieved the flow state and how long it is maintained. A good way to do this is to scale the difficulty with the skill of the player so as they get better the difficulty increases. Having the game be too difficult at the beginning can stress players causing them to not achieve flow the opposite can be said if the player is skilled and the games too easy, they can get bored and quit as reported by Baron (2012). So being able to keep a player in flow is important.

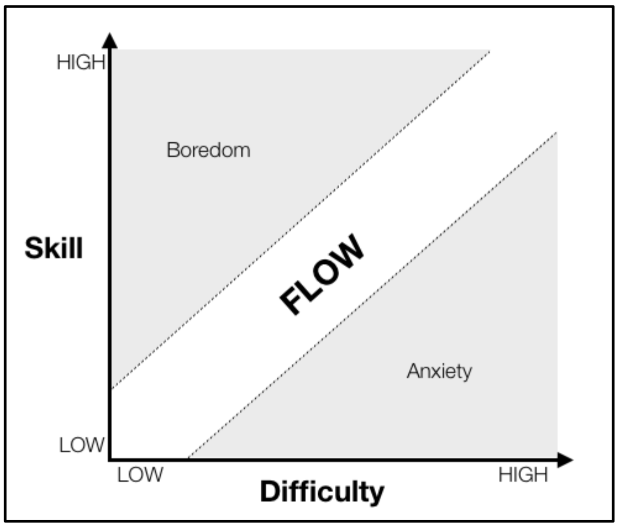
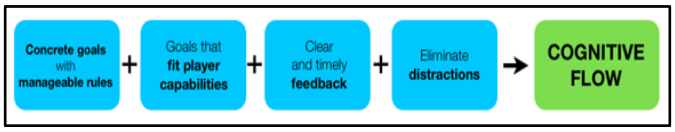


Figure 1 Difficulty and flow (Gamasutra 2012)

According to Baron (2012) there are 4 characteristics that promote flow which are

- Goals with manageable rules
- Goals that that the players current skills – example: giving a beginner a fetch quest in a starter area.
- Feedback for example rewards for completing tasks at decent intervals
- No distractions, for example not overloading the player with too much information like a cluttered HUD.

when taking these design factors all together player engagement will improve ways of implementing these factors.


Concrete goals with manageable rules
+
Goals that fit player capabilities
+
Clear and timely feedback
+
Eliminate distractions
COGNITIVE FLOW
Figure 2 Characteristics to achieve Flow (Gamasutra 2012)

Enter the Gungeon (2016) is a bullet hell dungeon game it is also a good area of research as it also uses procedurally generated maps. However, it is not all automated there is a sort of pre-planned routine to help the dungeons have a decent flow (Boris 2019).

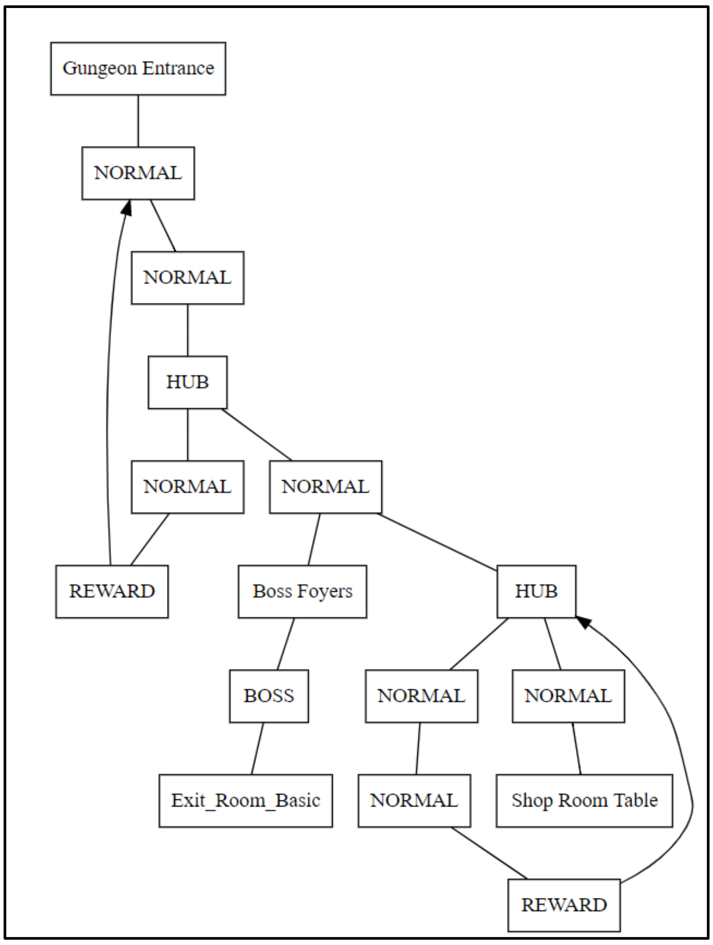


Figure 3 Flow diagram of how enter the Gungeons maps are structured (BorisTheBrave.com)

Normal are randomly generated rooms which spawn enemies, reward rooms have rewards, hub rooms are big rooms with multiple exits for the player and so on. The algorithm also allows for secret rooms to connect to a room when a map is generated.
This works by picking a flow file which is a graph data structure which stores the relationship between the different rooms but not the room positions. Each of the rooms contains metadata which is information about what type of room the room will be and what connections are required. These connections are directed as each flow chart begins with a root node and then creates a tree of child nodes and then adds extra to break the tree structure and create loops.
The flow file is then transformed, and alternate paths and room types are decided as well as these extra nodes are added in each new added node has data consisting of where it needs to be positioned its spawn probability type of room conditions that need to be met for it to spawn etc. see figure 4.
After the flow is complete is split into a composite which are just a single loop of rooms, or a set of connected, loopless rooms. These are laid out by first placing a room in a spot and then adding more one by one by selecting exits in the

new room and the previous one as the exits are defined in the rooms metadata the new room will be aligned so that the previous rooms exit will connect with it this then repeats till the level is done. After this it is just putting the composites into the map and joining any rooms without connections.

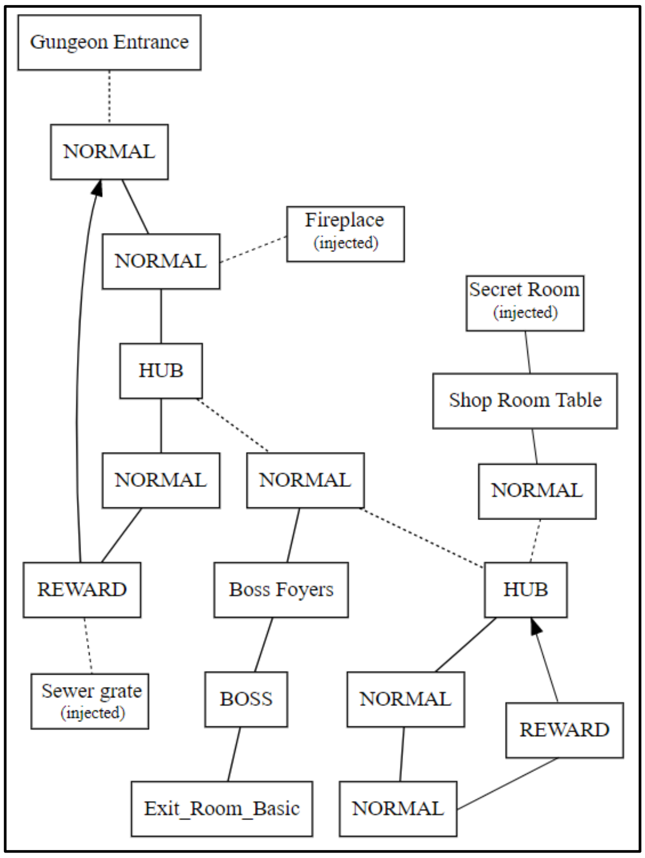


Figure 4 The flow after adding extra nodes and separating into composites (BorisTheBrave.com)

Spelunky (2008) is one of the main inspirations of the project which is the biggest reason for this being an area of research it is a 2D platformer that uses a procedural generation algorithm which uses a 4x4 grid and 16 rooms Kazemi (n.d). The algorithm creates a path which can begin from anywhere on the top and then continue to the left, right or downward directions till it reaches the bottom. Then the areas around the path are filled with filler rooms which are not necessary for completion but good for exploration. There are different room types which consist of:

- A side room which is not required to pass through (random/filler room)
- A room which is guaranteed to have a left and a right exit
- A room which has exits on the left, right and bottom (this room will also contain a top exit if there is the same type of room above)
- A room with exits on the top, left and right.

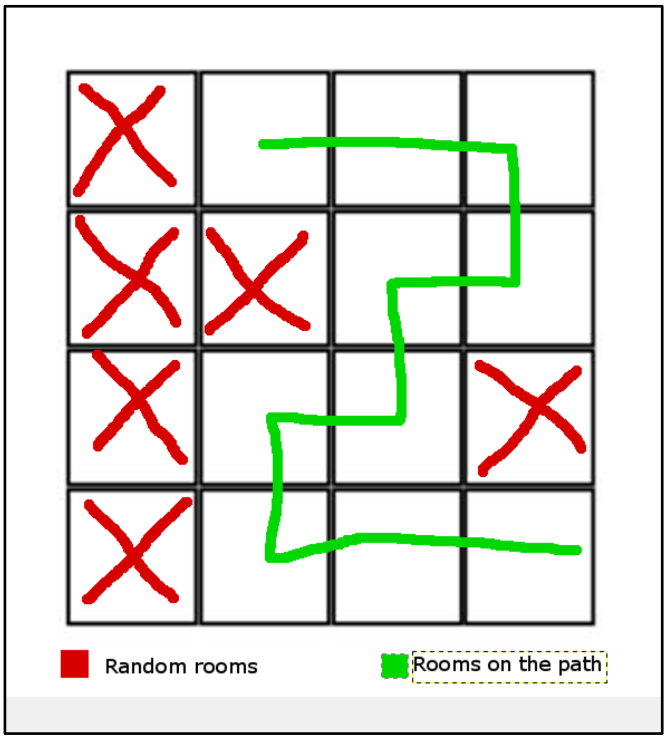


Figure 5 how the Spelunky level generation works

AI

In the game Wijk and the Water Battle, Grendal Games wanted to create an AI that could test out their procedurally generated levels to check if it was possible to complete them. After doing research into different methods that they could do this which consisted of a neural network system, an imitation learning algorithm and graph-based path finding. In the end they decided that graph-based path finding would be the best way to do this considering optimization and applying to other games. The system they created was called CHIMP (Challenge Intensity Measurement Platform).

Pathfinding strategies have the responsibility of finding a path from any coordinate in the game world to another (Graham 2003). According to Graham (2003) pathfinding can be split into 2 different categories directed and undirected. Undirected pathfinding can be compared to running around blindly in a maze this is not ideal for finding the best path quickly however can show demonstrate all different paths. Algorithms that can be used for undirected are breadth first search and depth first. Depth first search utilizes the "go deep, head first" philosophy in its implementation (Khov 2020). It goes down one path till it can no longer go any further and if it has not reached the end, it will backtrack and go down an alternate path. Breadth first is the opposite and it will go to its nearest neighbour first till it reaches its destination.

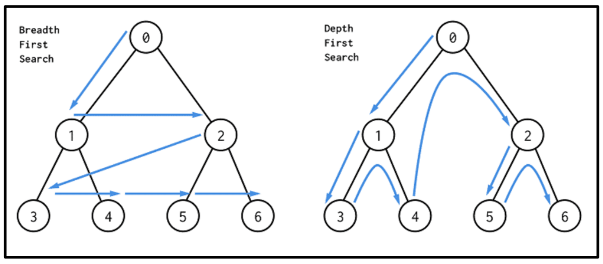


Figure 6 depth first and breadth first(dev.to)

Then there is directed pathfinding these methods do not search blindly for the goal they usually have some sort of method to assess the progress from the nodes around it before picking where to go next. The most used algorithms for directed pathfinding in video games are A* and Dijkstra algorithm. A* aims to find a path to the goal while having the smallest cost (i.e., least distance travelled). Dijkstra's Algorithm seeks to find the shortest path between two nodes in a graph with weighted edges.

3. Research questions

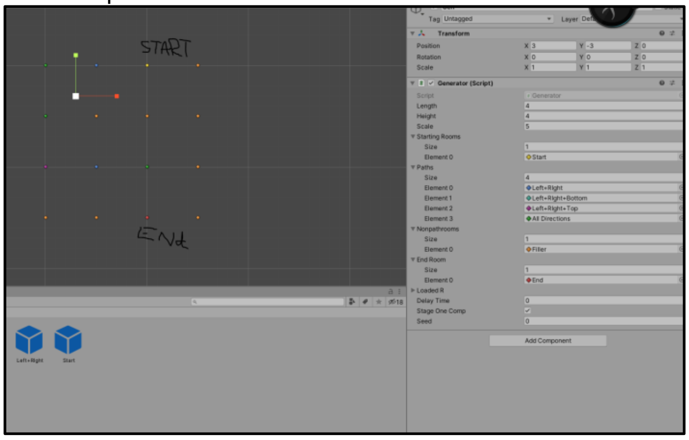
From the research gathered a number of research questions have arisen and these questions are:
What can be done to make Procedurally generated levels engaging for players?
This first question will explore how flow can be successfully be incorporated into levels that have been procedurally generated.

What is the best method for an AI that tests levels?
The second question will explore the best method to check the levels that have been generated using an AI search algorithm.

What are the strengths and weaknesses of level generators?
The third and final question will explore the benefits of using a level generator to create levels as well as what issues arise from using them.

4. Research methods

As there a number of areas that needed to be looked into for this project the methods varied depending on the subject area. When exploring procedural generation both primary and secondary research was necessary to learn how to create an algorithm that can be used for the artifact. For the secondary research different papers by Wiles and Beaupre (2018) where they generated levels for Bomberman, Frogger and The Legend of Zelda using different generators and, a paper by Summerville and Mateas (2016) where they procedurally generated Mario levels. These sources were discovered by looking on Google Scholar and using search terms such as: "Procedural level generation in games" or "Level Generator". For primary research, a prototype generator was constructed based off of the level generation of Spelunky however in this example instead of generating rooms in this case only nodes were generated to for testing purposes however the different nodes represent a different type of room Yellow being the start, red being the end, orange being filler rooms and the remaining colours are regular rooms on the solution path.



```
Tag Untagged          Layer Def
Transform
Position      X 3   Y -3   Z 0
Rotation      X 0   Y 0    Z 0
Scale         X 1   Y 1    Z 1
Generator (Script)
Script        Generator
Length        4
Height        4
Scale         5
Starting Rooms
  Size        1
  Element 0   Start
Paths
  Size        4
  Element 0   Left+Right
  Element 1   Left+Right+Bottom
  Element 2   Left+Right+Top
  Element 3   All Directions
Nonpathrooms
  Size        1
  Element 0   Filler
End Room
  Size        1
  Element 0   End
Loaded R
Delay Time    0
Stage One Comp  ✓
Seed          0
Add Component
```

Figure 7 Screenshot of Procedural Generation prototype

When exploring other areas of research secondary research seemed like the best course of action due to the various sources out there to look at. This consisted of looking at footage on YouTube of games that successfully used procedural generation in successful games i.e., Spelunky and Enter the Gungeon looking into successful examples for the project give better ideas on how levels should be laid out in the final project.

When looking into AI secondary research was used and like Google scholar was used as well as YouTube to look at successful implementation of different search algorithms. Search terms used to find research sources were: "A* Pathfinding in Unity", "Pathfinding in video games", "Best pathfinding algorithm for video games".
When looking for secondary research sources there was a slight bias towards examples that were demonstrated in the Unity engine this was mainly as a result of this project being developed in the engine.

5. Ethical and professional principles

This Project does not raise any ethical concerns, nor will it have any negative impact on communities or individuals because there are no human participants being used for the project. Instead of that the project uses an AI search algorithm to check whether the levels generated by the generator. No sensitive areas are covered by this project either as a result of this no controversies will come of the project.

The majority of the assets used in this project are sprites and although royalty free ones could have been used to be safe original assets were created specifically for this project.

6. Research findings

By first looking into what different kinds of generators existed and what types it made it possible to come up with a good solution to generate decent levels for the project. From the research it was summarised that the easiest and simplest way to procedurally generate levels would be to use the method which combines prebuilt areas or rooms. There were good examples of this being a successful method with it being supported with Sperlunky and Enter the Gungeon being successful games which use this method.

One thing these games had in common was the fact that things such as enemies and rewards like treasure and powerups would be randomly spawned in the levels which is something that can be incorporated into the algorithm for this generator. Adding more interactivity through things like that should in theory keep players in the state of flow and make keep them engaged enough for them to want to keep playing. Another thing to be taken from these games is the fact that they use different room types and to keep the levels interesting because if the levels start to feel repetitive then then players will likely lose interest as they will get the feeling of déjà vu like they have just been in that place before, which could possibly knock them out of a flow state which is not ideal.

When looking into the different AI search algorithms there were quite a few possibilities that could be explored due to the fact that a lot of them would eventually give the required result in the end however they key difference would be the efficiency of the algorithms. Using an undirected search algorithm for this project could be inefficient due to them exploring every possible solution till the goal is found. So, using a directed search to find the goal like A* algorithm could be better for efficiency.

Creating a prototype generator for primary research made for a good base to start off the project. With the way it is currently structured by generating the nodes it should be possible to convert those nodes into room prefabs to make it possible to generate an actual level. One thing that was discovered when making the prototype was that having the seed as a variable that can be changed manually limits the levels so making the seed random every time can ensure purely random levels every time.

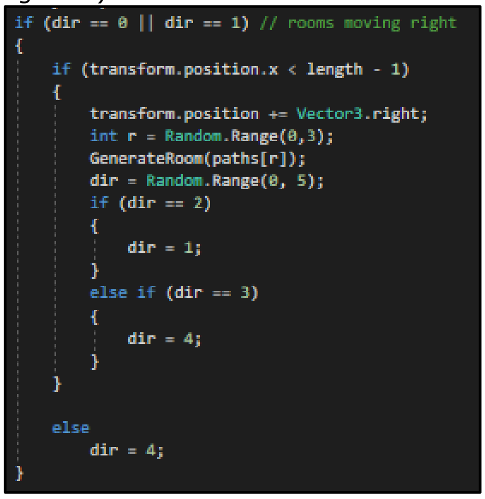
7. Practice

For the main algorithm first of all an array needs to be created to hold game objects which represent the different types of rooms. For this generator there are 4 types of room, start rooms, end rooms, path rooms and filler rooms. Variables for the size of the room need to be created as well these all need to be public so that they can be set in the inspector by the user, the variables for the size are a height and length variable and they are integers. An integer for a seed is also needed so that the levels can be randomized.

After everything is set up the generator needs to be initialised a function is made for this and in here essentially the start and end rooms are generated as well as these other variables are set such as the seed which is set to a random range so that it can produce different results every time. the size of the level is also set here as well by giving the room array game object the length and height values. An example of this is if length and height were both equal to 4 the level will be 4 rooms by 4 rooms. As there are going to be 2 stages of generation Booleans a required to check if they are completed so for now, they are set to false; Once everything is set the new function can be placed in the start function, so it starts as soon as play is pressed.

Once everything is initialized comes generating the rooms. This is done by first doing some checks. There is an integer which represents which direction each room is going to spawn 0&1 represent moving to the right 2&3 represent moving left and 4 represents moving down. The algorithm then first checks if the room is moving right and if it is it then checks if the room is at the furthest point to the right it can go. If it is not, then a path room will be spawned to the right of the previous room and a new direction is set randomly between 0 and 5. If the room is the furthest it can go instead of spawning right it will spawn below starting a new row.

Doing this did originally cause some problems for the generator as the way it was set up if the room moved to the right then the direction changed to the left afterwards then the rooms just overlap each other, and a mess of a level is created so to stop this from happening some extra checks were added in to stop this from occurring. When the random direction is being decided the method checks if the direction is being set to move back in the previous direction, in this current case that would be going left, and if it is will change the direction to either move to the right again if it can or move downwards. (See figure 8)



```
if (dir == 0 || dir == 1) // rooms moving right
{
    if (transform.position.x < length - 1)
    {
        transform.position += Vector3.right;
        int r = Random.Range(0,3);
        GenerateRoom(paths[r]);
        dir = Random.Range(0, 5);
        if (dir == 2)
        {
            dir = 1;
        }
        else if (dir == 3)
        {
            dir = 4;
        }
    }

    else
        dir = 4;
}
```

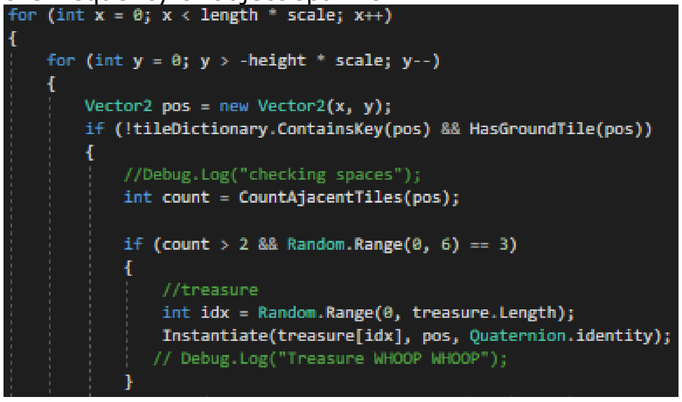
Figure 8 Method of placing rooms

The same checks above apply to spawning rooms to the left as well however just reversed so instead of checking if the position is furthest right it will be furthest left and instead of checking if the direction is going to go back upon itself it will check if it is going right. Spawning rooms that move down is slightly different but similar. Here it checks essentially if the room is in the bottom row or not and then spawns a room.

Another issue arises here when trying to do this because there is a possibility that a room without a bottom can spawn this is an issue because then anyone playing the level could be stuck on one floor unable to progress any further which would be frustrating and make the level pointless. So, to stop this a function to get a room was made all this does is gets a room and its position. This function is then called in the destroy game object function so that it can destroy the previous room. After this that room is replaced with a room which has a bottom exit this then solves the problem. When that is the previous checks are then done again so if the room is on the row before the bottom a room will be created if not then direction is set to 4 and the return is called. Once the bottom row is reached a room which has an exit at the top is always spawned so that the bottom can be reached by a player. Once on the bottom row rooms will spawn like normal however if the direction changes to downwards the current room will be deleted and replaced with an end room. The level is almost fully generated after this all is left is to add in the filler rooms as there will currently be some empty spaces in the level which need to be filled with a room this is done with an extra function which loops through all of the level and checks if there is a room in each space. If there is a room present it will move along if there is not a room then the space will be filled by a filler room which completes the level. This then finishes the first stage of generation.

The second stage of the generation deals with spawning things such as treasure enemies and traps in the level. This is done by checking what tiles are around and spawning them randomly around the level. To do this first a dictionary of tiles is made so that all the tiles can be checked, and this is then initialized in the InitGen function. Tiles are then added to the dictionary from another script made to spawn the tiles for rooms which will be explained later. After a function is created to check tiles, this function essentially loops through the x and y axis to see the surrounding tiles and then returns a counter. Another function is also there to check if there is a floor.

in the second stage generation the functions are called within 2 loops (Figure 9) which loop through the level size and then in the loop checks if the tile dictionary contains a tile and checks if it is a floor tile if it’s not then an integer is made and set to equal the adjacent tiles, and this is used in another check and if that new integer is less than another integer which will determine frequency of the spawn. When this check returns true then an index variable is set to the be a random range from zero and the length of the object array after that the objects are instantiated in the level. There was an issue here with the spawn frequency being a bit too high and levels being covered in objects so to counteract this an extra check was added. So instead of what was mentioned before if the count is less than an integer and a random number is equal to another integer. This way a user can play with the random range and choose the frequency of object spawns.



```
for (int x = 0; x < length * scale; x++)
{
    for (int y = 0; y > -height * scale; y--)
    {
        Vector2 pos = new Vector2(x, y);
        if (!tileDictionary.ContainsKey(pos) && HasGroundTile(pos))
        {
            //Debug.Log("checking spaces");
            int count = CountAjacentTiles(pos);

            if (count > 2 && Random.Range(0, 6) == 3)
            {
                //treasure
                int idx = Random.Range(0, treasure.Length);
                Instantiate(treasure[idx], pos, Quaternion.identity);
               // Debug.Log("Treasure WHOOP WHOOP");
            }
```

Figure 9 Spawning objects

With that the second stage of generation is done. After this a collection of prefabs needed to be created to get a level built a tile spawner, nodes for the rooms and the actual rooms themselves. The nodes are easy enough to make they are essentially empty game objects with an attached script. In those scripts is an array for different variations of a room type and then just a function that spawns the room. The tile spawner works in the same way being an empty object with a script attached this will insatiate a tile as well as adding it to the dictionary needed for the item generation.

The rooms themselves were made up of a game object as the parent and built of tile spawners. The design for the rooms were based of off cave levels like Sperlunky and terraria underground. There were 7 different room types to design for and each one of these types had at least 4 variations (minus start and end rooms) so that each level could look different. (Figure 10) each start and end room had its own unique object in it. The start room would have an entrance object which would have the tag start, (the tag is to help identify the objects when creating the pathfinding algorithm) and the end room would have an exit object with the tag finish.

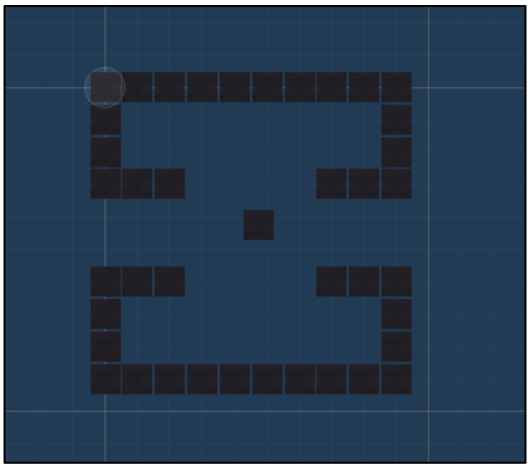


Figure 10 Room with exits on the left and the right

All that is left is to create a generator prefab which is just an empty game object with the generator script that contains the main algorithm and then drag the prefabs created into the correct areas and the result is a level (figure 11).

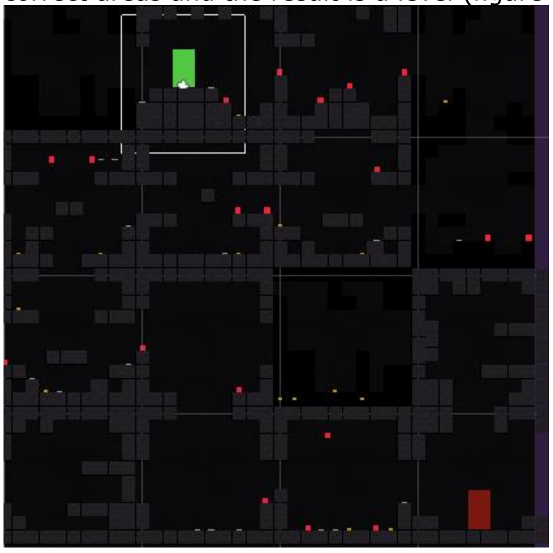
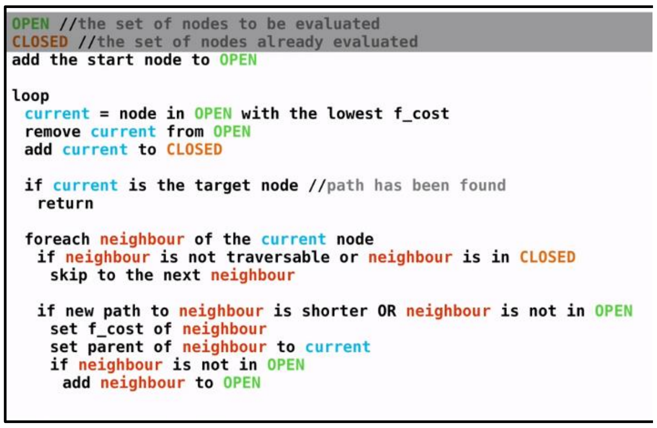


Figure 11 Generated level

The last thing is the pathfinder which checks if there is a successful route from the start to the end. This is done by creating a grid class and a node class for the nodes in the grid. The grid class is simple it creates a grid using the nodes on the node class. In this class a layer mask is made to check whether a surface in the grid is “walkable” if it is not then it will show as red in the grid this is so that the path will not go through tiles to find the path. Walkable tiles are set to clear, and the path is green.

There was an issue with the grid originally as it was unable to detect any of the floor tiles even when they had the correct layer assigned to solve this some changes had to be made to all the objects in the scene. Up to this point they had been using 2D colliders and rigid bodies as well as the grid using a vector 2 to calculate the positions of nodes in the grid. To fix the problem these all had to be changed to the regular 3D versions, so the objects and tiles were using regular colliders and the grid was now using vector 3 instead which fixed the issue and the non-walkable surfaces showed on the grid. Doing some of this did break some of the other scripts that took in a vector 2 so they needed to be fixed before continuing with the pathfinding algorithm.

The algorithm used to find the path is the A* algorithm (Figure 12) which works by finding the best path possible. To find the path it adds the start node to the open list then in a while loop sets the current node to the one with the lowest f cost then removes that form the open list and adds it to the closed list. When the path is found the path will be drawn on the grid and the loop will end. If not, it will continue and check for each adjacent node of the current node and see if it is walkable or not if it is not then the node is added straight to the closed list. If the path to the adjacent node is shorter or the node is not in the open list, then the adjacent node has its f cost set and is made parent to the current node and then it is added to the open list. To make sure the path this works a new layer needs to be created in the inspector to check whether a surface can be walked on and then applied to all the tiles. And tags created for the start and the exit so they can be applied as the targets for the path.



```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

Figure 12 Pseudo code for the A algorithm (Sebastian Laugue 2014)*

Due to an issue with the way the rooms are set up there was an issue with the pathfinding where if a start room was on the edge of the map the path would leave the level and go around the side if that was the fastest route. For now, a border around the level fixes the issue however this is limited to when the size of the level is 4x4 so not a permanent fix (figure 13).

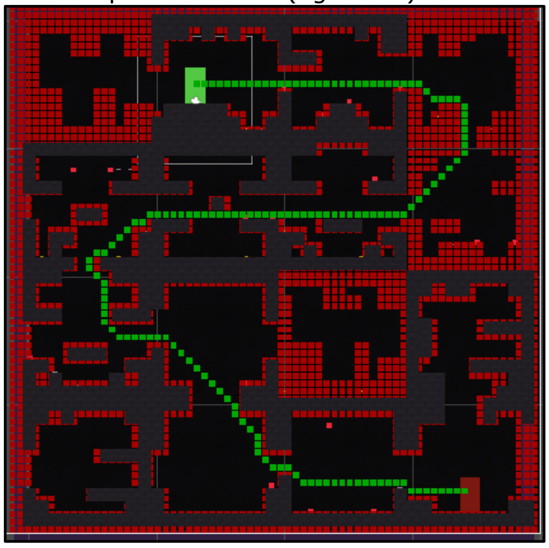


Figure 13 Path being found through the level

To test if the level actually works a player needs to be created so to do that a gameobject with a sptite box collider and rigidbody were added. Then a player script was created for the movement. In this script there are 2 functions one fore player movement and one for jumping after these 2 functions are added to the update function the player is then added to the generator script to be insantiated in the same position as the entrance.

After that all works a basic player was will in to spawn with the level after both stages are done (Figure 14)and levels can be played to the end.

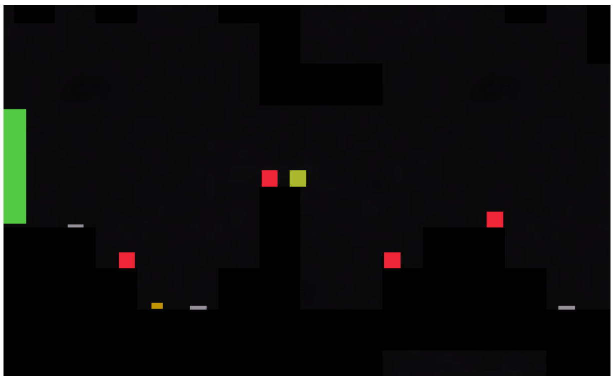


Figure 14 In game view of level

8. Discussion of outcomes

Looking back on the generator as a whole it works to an extent however it is flawed in it is execution. Levels are generated fine however not all the functionality is implemented things such as the difficulty parameter and different level types etc were not executed due to more focus on getting the main level generation working properly and although a level is playable in this current iteration objects in the levels do not fully function the traps and the treasure are just objects, and the enemies move but cannot damage the player. Performance is also an issue especially when large levels are generated anything that was larger than 10x10 was unplayable due to the abysmal framerate even at 4x4 the frame rate dropped to as low as 3 frames per second. The pathfinder though itself worked ok in most situations the implementation was inefficient and could have been vastly improved by using data structures instead of storing everything in lists. The pathfinder was definitely one of the contributions to the poor performance as it needed to be run in update for it to work properly.

The room designs were basic however they worked quite well each of the room types had multiple designs around about 4 each which made for some unique levels however especially in some larger levels there are a lot of repeating rooms so perhaps a few more designs for each level could have made for better randomness. The start and end room as well only had one design as well which meant they were the same each time meaning that if they were at the edge there would be an exit off the edge leading into nothingness. However, this was not just a start and exit room issue there is an oversight in the algorithm which allows for rooms on the edge of the map to have exits that lead off the edge in future iterations of this project this is something that would have to be addressed especially if the same algorithm is used. There is also some collision issues with the player and the ground tiles at times the player is able to go straight through the walls which would also have to be addressed.

When looking at the research first research question for this project which was "What can be done to make Procedurally generated levels engaging for players?" the generator created this project does not give a definitive answer to this as though it can create unique levels using the algorithm and the prebuilt rooms, the AI search algorithm as it is now only finds the best path to the end of the level checking if it is feasible to complete so it is difficult to come to a conclusion here. In hindsight using another method would be best to answer this question in a project by Forsblom and Johansson (2018) a questionnaire was made and based on different things on the level and asked questions such as was the level size, ok? Was Enemy health good? etc. being able to have human feedback would have been a lot more beneficial to the project. An additional idea to this is to have some of the level parameters mutate based on the players feedback and possibly a balanced level could be generated. It would be a lot easier to measure if a player were entering the state of flow using this method as well as it would actually be able to evaluate their progress.

The second question "What is the best method for an AI that tests levels?" the answers again are quite uncertain. Although the A* search algorithm found the quickest and most efficient path to the end of the level it is hard to say whether or not it's the best search algorithm for level testing. Because it only finds the quickest path it could be fair to say it could be the best for checking for speed run routes or for if the game, however if the game were a collectathon where the player needed to explore all routes for collectables to complete the level maybe a breath or depth first search would have been better. a better solution to get a better answer could be to have tested multiple different algorithms and discover the best algorithm depending on the game type and then could have use the best algorithms for each game type in the algorithm giving a user the choice on what they want to use it for. The path found by these search algorithms could be given to a player character and then have an actual player follow the path and check if it is beatable that way. Extra parameters and checks would have to be added to the algorithm to check if the player character needs to take certain actions such as jumping, attacking, and moving while following paths throughout the level.

Finally, the third question "What are the strengths and weaknesses of level generators?" could be answered quite well because although it had its problems the generator still successfully generated levels. As for the strengths the first would be the uniqueness of each level, no level is ever the same allowing for high a replayability value however in this case this can depend on how many room types there are because if there were only one or two then the levels could get quite repetitive after a few iterations which is why multiple designs are used for rooms here. Another one would be the huge levels that can be generated, although the generator has some performance issues it can effortlessly generate levels bigger than 100x100. As for the weaknesses first, there is the performance generated levels especially with multiple objects can slow down performance as well as this the size can cause performance issues as well. The algorithm can improve upon this though. Normal weaknesses that could be found in some level generators such as repetitive worlds and unplayable worlds have been negated using this generator due to the fact that the algorithm makes sure unplayable worlds don't exist due to the way the rooms are spawned. However, one weakness is adding unique pre scripted events because the levels generated are just too unpredictable. This makes procedurally generated levels fairly useless in games more focused on story as it would be a whole lot more trouble. One final weakness found while working on this project is that the more expensive a procedural generation algorithm becomes the more difficult to design well which can lead to the performance weakness mentioned before as once objects like the treasure and enemies etc started being added into the algorithm to be spawned in then the worse the performance got.

The methods used for this project have room for improvement, if the project were to be made again the Unity engine would no longer be used as using it restricted a lot of the project especially research wise with a lot of research being focused on methods and examples made in Unity whereas instead it could have been more beneficial to find other sources. As well as this it is not simple to save levels in runtime in Unity making it difficult to save a generated level at all. The algorithm could be improved on as well due to it is performance being so slow even with smaller levels so some research could go into finding more efficient ways of generating rooms. Doing testing with people would be something else that would be changed for this project it was definitely a mistake not using it this time around as a result some of the project objectives were not achieved it was an oversight in the planning stage and should have been picked up earlier.

9. Conclusion and recommendations 300

The level generator ended up having mixed results though it managed to accomplish some of the project objectives which were to generate a playable level using a procedural generation algorithm and having an AI algorithm to check the feasibility of levels. Having a small game with these levels was not achieved due to the methods chosen for the project and generating fun levels is ambiguous there is no way for the generator to check whether or not the levels are fun or not so there is no answer here.

The next steps for this project if someone else were to pick it up would to first review everything and then make some changes to the project so that the clear answers to the research questions can be found. Looking into some of the alternative methods suggested in the previous sections would be a good route to follow especially adding human testing using the questionnaire and then having the results of that questionnaire effect the level. Another recommendation would be to drop Unity and rebuild the project using something else as this would open up a lot of possibilities and give off opportunities for research. Using multiple AI search algorithms to check paths within the level could get a better and more definitive answer to the second research question.

The project when working could easily benefit professionals and newcomers to game dev as tool to generate levels for their 2d platformer, there’s a lot of work that needs to go into it before it reaches that standard however could be helpful and end up saving a lot of time when developing these types of games.

This procedural level generator has its ups and downs, but the project has shown that procedural generation can be great for generating levels for a 2d platformer.

10. References

A.J. and Mateas, M. (2016) Super Mario as a String: Platformer Level Generation Via Lstms. [online]. [Accessed 04 November 2020].
Baron, S (2012) Cognitive Flow: The Psychology of Great Game Design, Available from: https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php [Accessed 11 November 2020]
Beaupre, S. M., & Wiles, T. G. (2018). General Video Game Level Generation. Available from https://digitalcommons.wpi.edu/mqp-all/2197Summerville, [Accessed 07 January]
Benoit-Koch, F. Gamedev.net (2014), Procedural Generation for a 2D Platformer. Available from: https://www.gamedev.net/tutorials/programming/general-and-gameplay-programming/procedural-level-generation-for-a-2dplatformer-r3794/ [Accessed 08 November 2020].
Boris (2019) Dungeon Generation in enter the Gungeon. Available from: https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/ [Accessed 20 January 2021]
Cziksentmihalyi, M. (1990) Flow – the Psychology of Optimal Experience. 1st ed. : Harper & Row. [Accessed 11 November 2020]
Dıaz, A.C. (2015) Procedural Generation Applied to a Video Game Level Design. [online]. [Accessed 05 January 2021].
Dodge Roll. Enter the Gungeon(2016) [Game] Available from: https://store.steampowered.com/app/311690/Enter_the_Gungeon/ [Accessed 18 January 2021]
Forsblom, J & Johansson, J (2018) Genetic Improvements To Procedural Generation in Games [online], Available from: https://www.diva-portal.org/smash/get/diva2:1190476/FULLTEXT01.pdf [Accessed 22 January]
Graham, Ross; McCabe, Hugh; and Sheridan, Stephen (2003) "Pathfinding in Computer Games," The ITB Journal: Vol. 4: Iss. 2, Article 6.[Accessed 22 January 2021]
Grendel Games (n.d). Automating level testing with AI, Available from https://grendelgames.com/automating-level-testing-with-ai/ [Accessed 07 January 2021]
Kazemi. D. Spelunky (n.d) Generator Lessons[webpage] Available from: http://tinysubversions.com/spelunkyGen/ [Accessed 06 November 2020].
Khov, T. (2020) Algorithms on Graphs: Lets talk about Depth-First Search (DFS) and Breadth-First Search(BFS) [online] Available from: https://trykv.medium.com/algorithms-on-graphs-lets-talk-depth-first-search-dfs-and-breadth-first-search-bfs-5250c31d831a [Accessed 22 February 2021]
Lauge. S,(2014) A* Pathfinding(E03: algorithm and implementation) [video] Available from: https://www.youtube.com/watch?v=mZfyt03LDH4&ab_channel=SebastianLague [Accessed 22 February 2021]
Yu, D. Spelunky(2008)[Game] Available from: https://www.spelunkyworld.com/original.html [Accessed 06 November 2020]
Zaltsman, D. (2020) Difference between Depth First and Search and Breath First Search [online] Available from: https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om [Accessed 22 February 2021]

11. Bibliography

Benoit-Koch, F. Gamedev.net (2014), Procedural Generation for a 2D Platformer. Available from: https://www.gamedev.net/tutorials/programmin

g/general-and-gameplay-programming/procedural-level-generation-for-a-2dplatformer-r3794/ [Accessed 08 November 2020].
Bubriski J. Procedural Generation 101 (for games) () Available from: https://johnnycode.com/2016/11/08/procedural-generation-101-for-games/ [Accessed 18 January 2021]
FBX Software. Jack Benoit(2014) [Game] Available from: https://play.google.com/store/apps/details?id=com.fbksoft.jb&hl=en_GB&gl=US [Accessed 05 January 2021]
Green, D. (2016) Procedural Content Generation For C++ Game Development [online].: Packt Publishing Ltd. [Accessed 10 November 2020].
Grendel Games. Wijk & Water Battle (2015) [Game] [Accessed 07 January 2021]
Lauge. S,(2014) A* Pathfinding(E02: node grid) [video] Available from: https://www.youtube.com/watch?v=nhiFx28e7JY&ab_channel=SebastianLague [Accessed 22 February 2021]
Potocek, T. (2015) Level Generation Techniques For Platformer Games. [online]. [Accessed 05 November 2020].
Rodgers, S. (2014) Level Up! the Guide to Great Video Game Design. 2nd ed. : . [Accessed 09 November 2020].
Rogers,S.A (2015) Cognitive perspective of flow theory and video games. Available from: https://teacherrogers.wordpress.com/2015/09/14/cognitive-perspective-of-flow-theory-and-videogames/ [Accessed 11 November 2020]
Schier S Pros and cons of procedural generation (2015) Available from: https://schier.co/blog/pros-and-cons-of-procedural-level-generation [Accessed 05 January]
Shaker, M. (2016) A Procedural Method For Automatic Generation of Spelunky Levels. [online]. [Accessed 18 January 2021].
Short,T.X Gamasutra (2014) Level Design in Procedural Generation, Available from: https://www.gamasutra.com/blogs/TanyaXShort/20140204/209176/Level_Design_in_Procedural_Generation.php [Accessed 05 January 2021]

Short, T.X. and Adams, T. (2019) *Procedural Storytelling in Game Design*. : CRC Press.[Accessed 28 December 2020]
Taylor,D. (2013) Ten Principles of good level design (Part 1), Available from: https://www.gamasutra.com/blogs/DanTaylor/20130929/196791/Ten_Principles_of_Good_Level_Design_Part_1.php [Accessed 08 November 2020].
Togelius, T., Kastbejerg, E., Schedl, D., Yannakakis, G.N. (2011) What Is Procedural Content Generation? Mario on the Borderline. 2nd International Workshop on Procedural Content Generation in Games, Pcgames 2011 - Co-located with the 6th International Conference on the Foundations of Digital Games [online]., p. 1. [Accessed 04 November 2020].

Appendix A: Project Log (not included in word count)

Name: *Kieran Boyce* ***Student Number:*** *18021179*	***Project Title:*** *Level Generator for 2D platformers using procedural generation*	
Week begins:	Tasks and summary of what was achieved	Tasks to be carried forward/new tasks
19/10/20	Had a meeting with my supervisor to pitch my project idea. The feedback I received was to have a more solid project idea as my current idea was too vague and potentially too easy.	1. Come up with a better proposal idea. 2. Research previous projects and games to get ideas. 3. Start drafting proposal.
26/10/20	Proposal drafts After researching into other possible ideas, I found Spelunky, a game which uses procedural generation to generate its levels and I decided to replicate something similar for my project and build a 2d platformer level generator that	1. Continue writing proposal drafts. 2. Send draft to Supervisor to get feedback

	uses procedural generation where the user can tweak parameters to get different results. After getting my idea I begun work on getting the first draft of my project proposal done as it stands it is a work in progress.	
02/11/20	Proposal drafts This week I finished the first draft of my project proposal and sent it off to my supervisor to get some feedback and was told that I should do some research into flow theory and keeping players engaged. As well as to put a bit more image representation in and to find some more references because they were lacking slightly. So, the next week will be making these changes to my proposal ready to submit on time.	1. Make changes to my proposal based on supervisor feedback. 2. Last proofread of final proposal. 3. Submit project proposal.
09/11/20	Finished changes made to my proposal Submitted my proposal I made changes to my proposal based on my supervisor's feedback form the previous week and wrote out my final proposal. I then submitted on time. Next, I need to begin research for the main project.	1. Begin research into procedural generation and how it works.
16/11/20	Begun looking into procedural generation methods and noting down findings while documenting sources	1. Continue research into different generation methods
23/11/20	Looked into different games that use procedural generation other than Spelunky such as Enter the Gungeon, Terraria etc	1. researchintolevel design to learn to create engaging levels.
30/11/20	Looked into level design and flow theory to get an idea on what I will need to generate engaging levels	1. Start working on some sort of procedural generation prototype
07/12/20	begun on procedural Generation prototype	1. Continue work on the prototype
14/12/20	Have a scene procedurally generating icons which represent different rooms can change	1. Christmas break (do bits of research ready for reports)
21/12/20	Christmas Break	
28/12/20	Christmas Break	1. Begin work on research report
04/01/21	Research report	1. Have a draft finished next week
11/01/21	Finished draft of research report and sent it to my supervisor for feedback	1. Research report hand in finish research report
18/01/21	Made changes to research report based on feedback given from my supervisor. Research report is ready to hand in and work on the demo can start	1. Begin work on demo ready for the first milestone 2.
25/01/21	Begun work ready for the demo. Built rooms of	1. Present WIP Demo

	different types to be spawned in the level generator so now the level generator spawns full levels based on the level seed. Prepared a video for the WIP demo and sent that off to my supervisor ready for the demo on 05/02/21.	2. Create more room types so that levels are not always the same when loaded.
01/02/21	Presented WIP Demo video to supervisor. Created multiple iterations of each type of room to create different level patterns. Changed the level seed to be random every time so the user does not have to keep changing it to get random levels, left the option to have it entered manually.	1. Start expanding algorithm to spawn in enemies and objects.
08/02/21	Added a second stage to the generation and the algorithm can now spawn different objects in random amounts based on the location of tiles surrounding it. This applies to enemy's traps and treasure. Also created prefabs for enemy's traps and treasure to be spawned into the level.	1. Start trying to implement the AI path finding. 2. Add player so that the levels can be played.
15/02/21	Created a basic player character that can be used to run through the levels. Attempting to create a pathfinding algorithm using the a* method using a grid.	1. Do some research on a*algorithm. 2. Create algorithm.
22/01/21	Did some research on creating a a* pathfinding algorithm looking at videos and articles explaining how the algorithm works and how it can be implemented. Created a grid class which is supposed to detect whether a surface is walkable or not however its currently buggy and not working correctly as non-walkable objects are not showing on the grid.	1. Fix bugs with grid class. 2. If the grid class is fixed, then start the pathfinder.
01/03/21	Fixed the issue involving the grid the problem was the colliders being 2d. Everything has been changed to have 3d colliders and that seems to have fixed the issue. Did not manage to start the pathfinder so that rolls on to next week.	1. Create path finder. 2. Fix bugs created because of the collider change.
08/03/21	Created pathfinder class using the a* algorithm. The pathfinding somewhat works but some of the non-walkable areas are being ignored by the algorithm. Fixed bugs created from last week.	1. Figure out the cause of the path not working properly.
15/03/21	Solved the pathfinding problem by having the algorithm create the pathfinder grid after all the rooms spawned. The reason they ignored some rooms is because the grid was being created before some of the rooms causing them to be ignored. A new problem arises though as some edge rooms have exits so the path leaves the map if	1. Fix new problem created by edge rooms.

	it's the quickest path.	
22/03/21	Created a temporary fix by creating a border around the level. Issue is currently it only works if the level is 4x4.	1. See if a solution can be found.
29/03/21	Tried to create a fix by rescaling the border with the level but it was not a success. Seeing if the algorithm can be tweaked to fix the issue	1. Continue finding solution to border problem. 2. Start writing out final report.
05/04/21	Started off report just filling out the first couple of sections ready. Still no solution to border issue.	1. Fix bugs. 2. Continue writing report. 3. Start work on a final version of the project.
12/04/21	Started work on a final demo created some assets for bricks however they could not be used in the end due to the size being too big and trying to scale them down or even recreate them at the size of the tiles seems to be an impossible task. Filled out the research sections of the final report.	1. Continue work on final demo. 2. Continue work on final report. 3. Fix bugs.
19/04/21	Trying to iron out some of the persistent bugs in the generator. Filled out the practice section of the final report.	1. Package final submission 2. Finish up report
26/04/21	Packaging final submission some annoying problems were unable to be fixed before the deadline. Finished off the report filling out the outcomes and conclusion to the report	1. Submit artifact 2. Add references to the report. 3. Submit report.
03/05/21	Hand in! Added references into the report and handed everything in successfully.	1. Create video and start Viva preparation.
10/05/21		
17/05/21		
24/05/21		

Appendix B: Project Timeline (not included in word count)

Month	Task	Days taken(estimate)
November	Proposal Drafts	10
	Final proposal to be submitted by (12/11/20)	2
	Research	7
	Create simple 2D platformer with basic mechanics	2
December	Work on Procedurally generating levels	14
	Work on manually generated levels	14
January	Prepare AI Prepare demo	7 7
	WIP Demo to be submitted by (14/01/21)	1

	Tweaking procedural generation algorithms & manual generator	(Throughout remainder of project)
February	Creating prefabs	3
March	Finishing off left over tasks	7
	Fixing any remaining bugs in project	14
April	Make final demo	7
	Write final report	14
	Package final submission	1
May	Final Hand-in (06/05/21)	1

Appendix C: Assets used in the Project (not included in word count)
This is a list of project assets: all source materials used in the project. Clearly state which were produced by yourself and which were not. If not produced by yourself, include their reference, and status with regard to copyright/ creative commons licensing.

Asset	Created by
Wall (Texture)	Kieran Boyce (myself)
Wall 2 (Texture)	Kieran Boyce (myself)
Gen (Gameobject)	Kieran Boyce (myself)
A_path (Gameobject)	Kieran Boyce (myself)
Left + right (Gameobject)	Kieran Boyce (myself)
Left + right + bottom (Gameobject)	Kieran Boyce (myself)
Left +right + top (Gameobject)	Kieran Boyce (myself)
All directions (Gameobject)	Kieran Boyce (myself)
Start (Gameobject)	Kieran Boyce (myself)
End (Gameobject)	Kieran Boyce (myself)
ALL DIR_1,2 and 3 (Gameobjects)	Kieran Boyce (myself)
LR -Uground and LR_2,3 and 4 (Gameobjects)	Kieran Boyce (myself)
LRD 1,2 and 3 (Gameobjects)	Kieran Boyce (myself)
LRU, 1,2 and 3 (Gameobjects)	Kieran Boyce (myself)
FILLER 1,2,3,4,5 and 6 (Gameobjects)	Kieran Boyce (myself)
Treasure, Enemy, Player and Trap (Gameobjects)	Kieran Boyce (myself)
Tile and Tile 2 (Gameobjects)	Kieran Boyce (myself)
Ground (texture)	Kieran Boyce (myself)
Enemy, Entrance, Exit, Generator, Grid, Node, PathFind, PlayerController SpawnRoom, TileGen, Traps, Treasure (Scripts)	Kieran Boyce (myself)
START (Gameobject)	Kieran Boyce (myself)
END (Gameobject)	Kieran Boyce (myself)