Procedural Level Generation for Roguelike Games

Ethan Younger Banks – 16039846

Dr RG Davison

Abstract

The Roguelike genre started from a single game and has grown into one of the most widely used game mechanics in the industry today. Procedural generation has many applications outside of roguelike games, but this is where I first became interested in it, and this situation where the developer has no control to fix any mistakes made by the generation system before the result is given to the player makes for a more interesting algorithm design process. This project aims to produce a system for generating levels for a roguelike game, with a focus on producing a variety of level styles from the same program by the use changeable parameters that the developer can tweak to produce varying, but reliable, results.

Declaration

"I declare that this document represents my own work except where otherwise stated"

Ethan Younger Banks

Acknowledgements

I would like to thank the following people for their advice and guidance which contributed to this project:

Rich Davison, my dissertation supervisor, for his understanding when I had to change topics at a very late stage, and providing extra meeting time to ensure I was keeping on track with the new project.

Rami Ismael and Jan Willem Nijman, for helping me track down Jan's article on incremental diggers, which was a very helpful part of my research and contributed ideas that fed into my own implementation.

Contents

1 Introduction

1.1 Purpose

In “roguelike” games, the goal of massive replayability can only be achieved by the inclusion of a very large number of levels – enough to ensure that every playthrough is unique and offers a different experience to the player. The only way to create such a vast number of unique levels within reasonable development time and cost is through procedural generation. Procedural level generation allows for an effectively infinite number of levels to be available to the player, and given sufficiently well thought out rules and algorithms, the generated levels can approach the quality of bespoke human-designed levels.

1.2 Project Aim and Objectives

The aim of the project is:

> *“To understand and implement techniques for automated procedural generation of dungeon levels for roguelike games”*

This aim can be split into multiple objectives:

- *Research existing techniques used for procedural level generation*
- *Design, implement, and document my own procedural level generation system*
- *Assess the effectiveness of this system in comparison to existing solutions*

1.3 Dissertation Outline

Introduction

A brief introduction to the area of research and the purpose and intentions of the project.

- Purpose
- Project Aim and Objectives
- Dissertation Outline

Background Research

Exploration of the existing domain of the problem and assessment of both commercial and academic solutions.

- Defining Roguelike Games
- Problems with Procedural Generation
- Research of Available Techniques

Design

Outline of requirements of the system and decisions on how best to develop it

- Requirements
- Generation Method
- Ideas from Existing Implementations
- New Ideas

Implementation

What I did, how I did it, and changes that occurred throughout development

- Engine Choice
- Implementation Structure

Testing

How I assessed by implementation and how successful it was

- Testing Method
- Testing Results

Conclusion

Wrap up of the project and discussion of future possibilities

- Fulfilment of Requirements
- Future Work

References

Sources that assisted in my research and development

2 Background and Research

2.1 Defining Roguelike Games

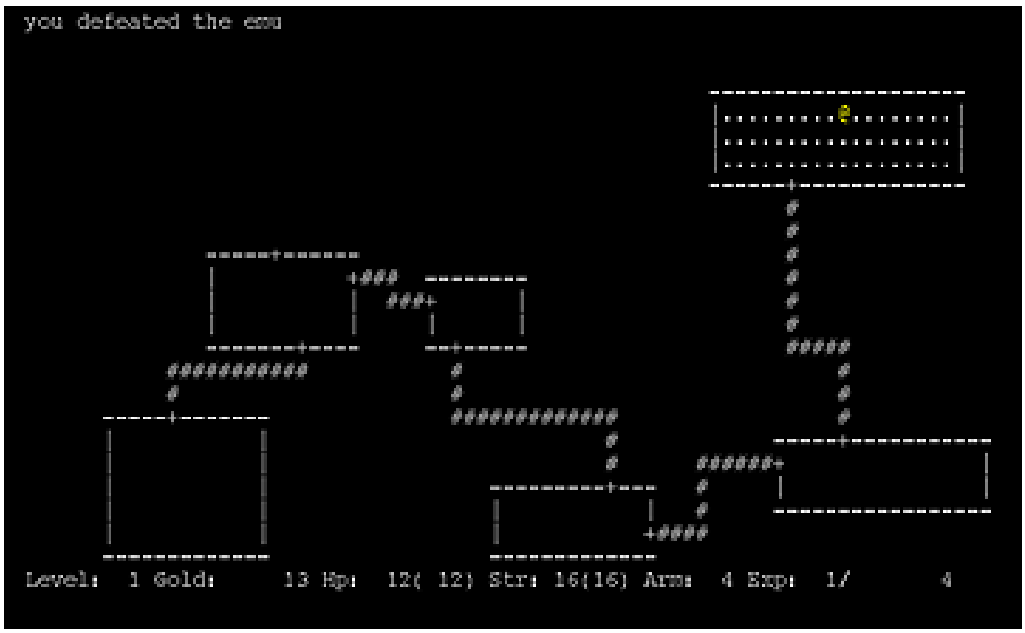

Figure 1: Rogue (2009)

The term "roguelike" originated from the 1980 game *Rogue*, with which we saw one of the first implementations of procedurally generated maps in a video game. *Rogue* took place in a multi-level dungeon filled with monsters that needed to be defeated, and treasures that help the player by boosting attack and defence capabilities. Each time the player starts the game again, the dungeon is generated again, meaning that each playthrough is on a different layout, and the player never plays the same dungeon twice. A defining characteristic of *Rogue* is the concept of *permadeath* – once a player character dies, that character is gone forever, and the player must start over with none of the items or progress that they had on the previous playthrough. *Permadeath* is an important part of any roguelike game, but sometimes in a "soft" form, allowing the player to keep some items and progress between playthroughs, or otherwise having previous runs affect an overall progression system. An example of this can be seen in *Rogue Legacy*, (Cellar Door Games) where the progress through the castle and gold collected on each run is lost upon death, but gold can be spent on upgrades between runs which then persist throughout future attempts.

In conclusion, for the purpose of this project, I will define a roguelike game as:

"*A game characterised by procedurally generated levels, and some element of permadeath between playthroughs*"

2.2 Problems with Procedural Generation

The purpose of procedural generation is, in essence, to reduce design and development time in hand creating levels. Creating enough levels for the scale of replayability that is desired form roguelike games is simply unfeasible, so procedural generation is required to produce the thousands or often millions of possible level configurations that are demanded in the name of effectively infinite replayability. However, avoiding human involvement in level design introduces many problems. For instance, when generating levels from a set of rules, those rules need to be very well thought out and implemented to ensure that all levels are not only possible to complete, but appropriately difficult for a given level's position in the overall progression (later levels should be harder, while the first level usually should be completable by even a new player). The quality of procedurally generated levels is often criticised, with many critics saying that the levels feel "soulless", and can never match the quality of those created by a human designer. Often emergent design elements that come from procedural generation are misunderstood by players who don't have the developer's insight into the process, as Dan Marshall found with his game *Swindle* (Marshall, 2009). Roguelike games are seen by many as a cop out to avoid spending time designing levels "properly", and so any new commercially used procedural generation system needs to ensure that it produces a balance of consistency and unpredictability sufficient to avoid such criticism.

2.4 Research of Available Techniques

2.4.1 Noise Functions

Many different approaches exist for generating *noise* - pseudo-random data, usually visualised as a two dimensional image, which can be used to generate heightmaps, terrain structures, or really any feature which requires a randomised set of values within certain criteria. There are two main forms of noise that we will be looking at here: *value noise*, and *gradient noise*. Value noise simply generates a random grid of values and then uses linear interpolation to smooth out the result, while gradient noise uses wave-like patterns to ensure a smoother, more natural looking output. Perhaps the most famous version of gradient noise is *Perlin* noise, along with its improved counterpart, *simplex* noise (Flafla2, 2014).

Using one noise function is sometimes enough, but multiple noise functions can be combined, by adding, subtracting, multiplying, XOR, or using any other multitude of methods to combine the output of two different noise functions. By combining multiple noise functions, *fractals* can be produced, and that is the basis of cave generation in *Terraria* (Tippets, n.d.).

Terraria was first released in 2011, and features large procedurally generated levels filled with complex cave systems which can be explored and excavated by the player. The levels are generated in stages, first using a gradient and applying *turbulence* to define the ground/sky divide, then applying a vertical offset based on generated noise to create the overall shape of the terrain. This terrain is then *perturbed* by another noise function to create more interesting textures and overhangs to the hills, and then

the cave generation can begin. Caves in *Terraria* are distinct from the cellular automata based caves discussed later in my research, in that there is no need whatsoever for the caves to be accessible to the player in their initial state. It is preferable for some caves to have openings on the surface for the player to stumble upon, but hidden caves only accessible by digging through the terrain are acceptable if not encouraged here. This brings up a red flag for my implementation, as I am generating levels with the assumption that the generated blocks are unmodifiable by the player (beyond doors being opened), however this being a problem for my implementation doesn't invalidate the utility of the method, and some valuable research can still be gained from *Terraria*'s level generation.

Figure 2: Generated Terrain Map (Tippets, n.d.)

Terraria uses noise functions again in the cave generation step, and here the relationship between the visual representation of the output of the noise function, and the cave layout generated by the algorithm, is clearly apparent, even after the fractal noise is applied to rough up the edges and make the caves more natural looking. The cave noise can be *attenuated* across the vertical span of the map, producing narrow tunnels towards the surface and larger open caves deeper in the ground, to create what appears to be a complex, very natural system of caves.

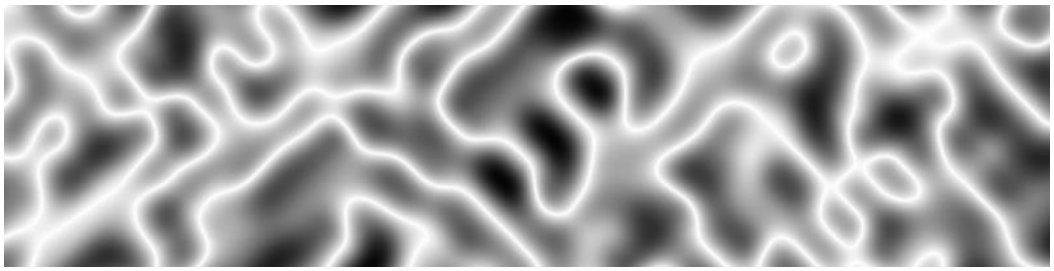

Figure 3: Cave Noise in Terraria (Tippets, n.d.)

The results produced by a noise-based approach can be very convincing in their natural appearance, but are less well suited to a game with clearly defined levels and objectives like most of the roguelike genre, than they are to a freely flowing open world game like Terraria.

2.4.2 Cellular Automata

Cellular Automata are nature inspired systems in which a set of rules is applied to each cell in a grid, step by step, creating natural-looking patterns or movements. Although cellular automata can be more complex than this, our scope extends only to two dimensional cellular automata on a grid of square cells, with only two states - “dead”, or “alive”, or in level generation based applications, “walls” and “floors”. The most famous example of this kind of cellular automata is John Conway’s *Game of Life*, which is designed to simulate populations of simple organisms such as amoebas or bacteria.

For my algorithm, I’m interested in cellular automata that simulate the formation of natural structures, mainly cave systems, to generate a less square and structured level than is possible using the approaches used by *Captive* or *Hauberk*. In this project I will use the “Golly” notation for cellular automata rules, which defines the game of life as B3/S23, that is, a dead cell with 3 live neighbours will be *born* (become a live cell), and a live cell with 2 or 4 neighbours will *survive*, that is, any live cell with 1, 4, 5, 6, 7, or 8 live neighbours will die.

Jeremy Kun proposes the use of rule B678/S345678 to generate 2D caves from a random seed state of live/dead cells (Kun, 2012), where a live cell is a wall and dead is passable terrain. This rule’s high survival rate means that only initially isolated cells die, and the state of the cave system stabilises very quickly (usually within 15 steps). The levels produced look very natural, but very often have blocked off, inaccessible rooms which at best limit the usable play space and at worst cause critical items to be placed out of reach of the player, making the level impossible to complete.

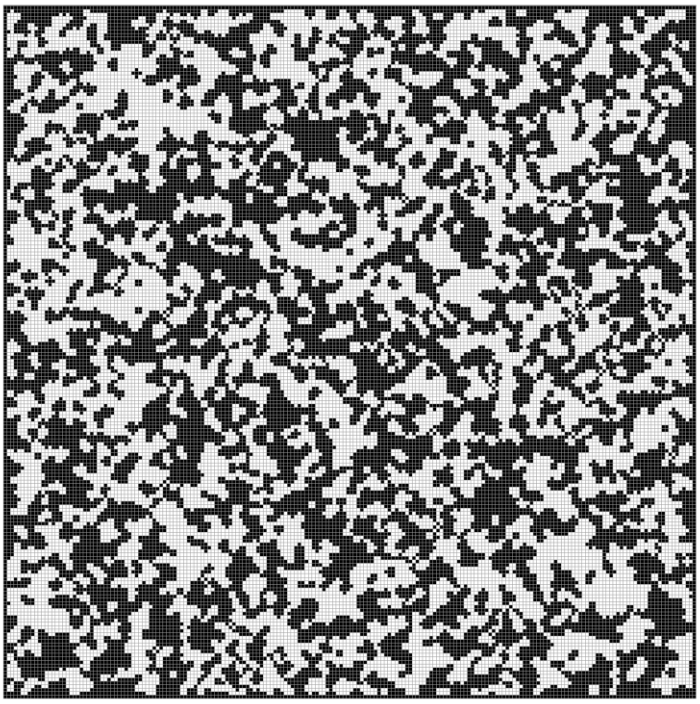

Figure 4: Cave system generated by rule B678/S345678 (Kun, 2012)

Michael Cook instead proposes rule B3/S23, which, despite looking far more concise, produces very similar (but often "rounder") results (Cook, 2013). The issue of blocked off rooms is still a problem here, but there is a slight benefit in that very small isolated spaces become filled with walls, removing these smaller inaccessible areas entirely.

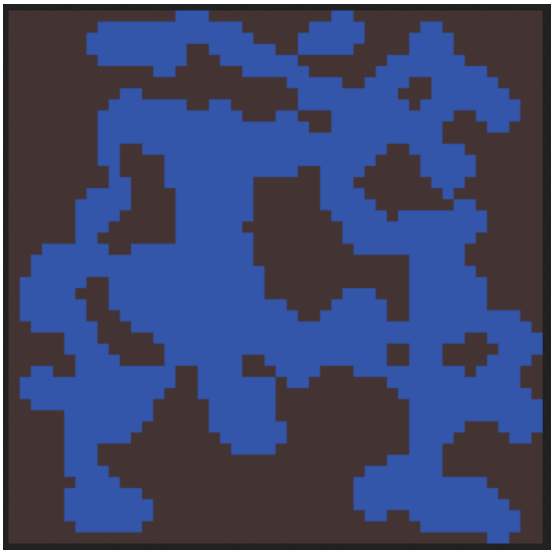

Figure 5: Cave created by rule B3/S23 (Cook, 2013)

Due to the nature of cellular automata only taking into consideration the immediate neighbours of each cell, it is impossible to ensure that all areas of the generated level will be accessible without performing checks on the generated level and retroactively digging paths to connect inaccessible areas. This may seem like reason enough to discard a cellular automata approach entirely, but we must also consider that some games may allow the player to destroy walls. Games like *Terraria* even use digging through terrain as a major gameplay mechanic, and in these situations the isolated caverns created by these cellular automata approaches are in fact desirable, and make great places for hiding valuable treasures or dangerous surprises for the player.

I considered the idea of using a noise function to generate the initial seed and then applying cellular automata to refine the layout. This would be cheaper to process than layers upon layers of noise combining into complex fractals, and would have been interesting to explore, however I decided not to base my method on cellular automata, due to the lack of structure the levels have with regards to discrete rooms and corridors which are useful for defining gameplay features.

2.4.3 Incremental Diggers

Almost as old as *Rogue* itself, incremental diggers were used in *Captive,* a game released by Mindscape in 1990 (Fournier, n.d.). *Captive* itself wasn't in fact a roguelike game - the sequence of levels followed a set sequence of seeds, but procedural generation was used to enable the game to contain a vast number of levels within a very small memory requirement, and without requiring the countless hours of design

time involved with producing that number of levels. My research focuses on *Captive,* as well as *Wasteland Kings*, a game created by *Vlambeer* for the MOJAM game jam in 2013, which would later go on to become the popular indie roguelike game *Nuclear Throne* (Nijman, 2013).

The basic idea behind incremental diggers is that a level is generated by first initialising a grid of non-traversable "wall" blocks, which are then "excavated" - turned into traversable "floor" blocks by one or more "diggers", which move in the same way the player is able to create corridors and rooms. At its core, a digger behaves probabilistically, with a set of possible actions and probabilities assigned to them. Actions include moving forward, changing direction either 90 or 180 degrees, or stopping its current corridor path to create a room or branching path. These last two features are handled differently between *Captive* and *Wasteland Kings*, and I will go into their different approaches, and the pros and cons of each.

Captive uses only a single digger, while *Wasteland Kings* can have multiple. This changes the way branching corridors are handled. While *Captive*'s digger can teleport back to a previously excavated block in order to start digging a new branched corridor, *Wasteland Kings* can have two or more corridors generated simultaneously. Benefits of parallelisation here are somewhat moot, as the game only runs on a single thread and process regardless of the number of diggers, but this potential for performance improvement should be noted. The main difference of the multiple digger approach is in the finished level layout. Examining a *Captive* level it is often possible to identify which corridors were branched late in the generation process, as, with a large number of corridors and rooms already excavated, the potential explorable space is much more limited, and later paths are often short and uninteresting due to having less space left over to excavate. *Wasteland Kings,* however, avoids this problem by giving each path an equal chance to explore the full level area.

If all levels filled the same size grid, it would be easy to tell roughly how much of the level you have explored, as each level contains roughly the same total amount of space. *Wasteland Kings* uses a totally variable size grid, and places a random number of blocks so that the size and shape of the level is always different. *Captive,* however, uses a set size grid, but splits it up into *floors* which can only be travelled between via elevators dotted around the level. This breaks up the continuity of the level but doesn't change the total amount of space it fills, so to further vary the size of the levels, *Captive* includes a random assortment of *undiggable zones*. These are randomly sized rectangles placed around the grid, which the digger cannot pass through. This leaves voids in the level, changing its overall size and making corridors appear to have more of a structure to them when they twist to avoid these undiggable zones.

Incremental diggers allow for some convenience in terms of post-generation checks to ensure a level is completable. When the movement rules for the diggers are the same as for the player, the fact that the level was generated at all proves for free that all areas are accessible by the player, as long as obstacles placed after the digging phase don't break this fact. *Captive* only places its non-traversable fire blocks where there already was a wall, and *Wasteland Kings* simply doesn't place any non-traversable blocks after the digging phase of generation is complete. Diggers can be configured to create corridors as well as larger open rooms, and as seen in *Wasteland Kings,* simply

altering the probabilities of each digger action can vastly change the overall theme of a level to create complex networks of narrow corridors or vast open fields without changing the underlying algorithm.

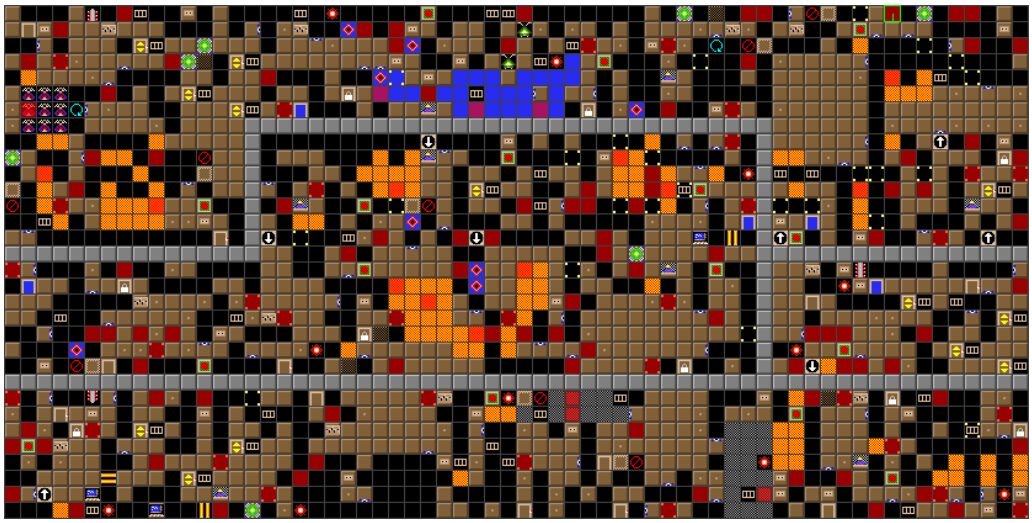

Figure 6: Level Generated by Captive (Fournier, n.d.)

2.4.4 "Rooms and Mazes"

A common approach to procedural level generation is to think a level above just wall and floor blocks, and consider rooms and corridors as discrete objects. Bob Nystrom explores some of the details of this kind of implementation, and how he came to the implementation used in his open-source web-based roguelike game, *Hauberk* (Nystrom, 2014).

Hauberk starts by generating rectangular rooms of random sizes at random positions in the map grid. If a room about to be placed intersects an existing room, it simply is not placed. This means that with a "room attempts" parameter set to 10, 10 rooms or fewer will be placed in the map. Once the rooms are placed, Hauberk generates a perfect maze in the gaps left between rooms. The rooms are then joined to the maze corridors to connect them together, starting at a random room and expanding outwards until the whole map is connected. Some rooms are connected more than once to make the resulting maze imperfect – that is, there is more than one possible route between any two given rooms. This makes exploring the map more interesting, encouraging exploration and causing the player to occasionally revisit previously seen rooms. At this stage, the map is fully dense, and has a lot of dead end corridors, which is not desirable unless the goal is to frustrate the player by making them turn back and spend a long time searching for a route through the level. To remove the dead ends, any corridor tile surrounded by three wall tiles is culled. This is repeated until no more dead ends exist, or some can be left if a small number of dead ends is wanted.

Hauberk's method allows for varying the difficulty of levels based on the structure of the levels themselves rather than relying on later placed key doors and enemies to

increase difficulty. The difficulty can be increased by increasing the number of rooms and therefore the number of places the player must check for important items, but also by choosing to leave some dead end corridors left over after the generation phase, leading the player to get lost down dead end paths. This is useful, but must not be overused or the player will simply get frustrated hitting seemingly endless dead end corridors with no reward.

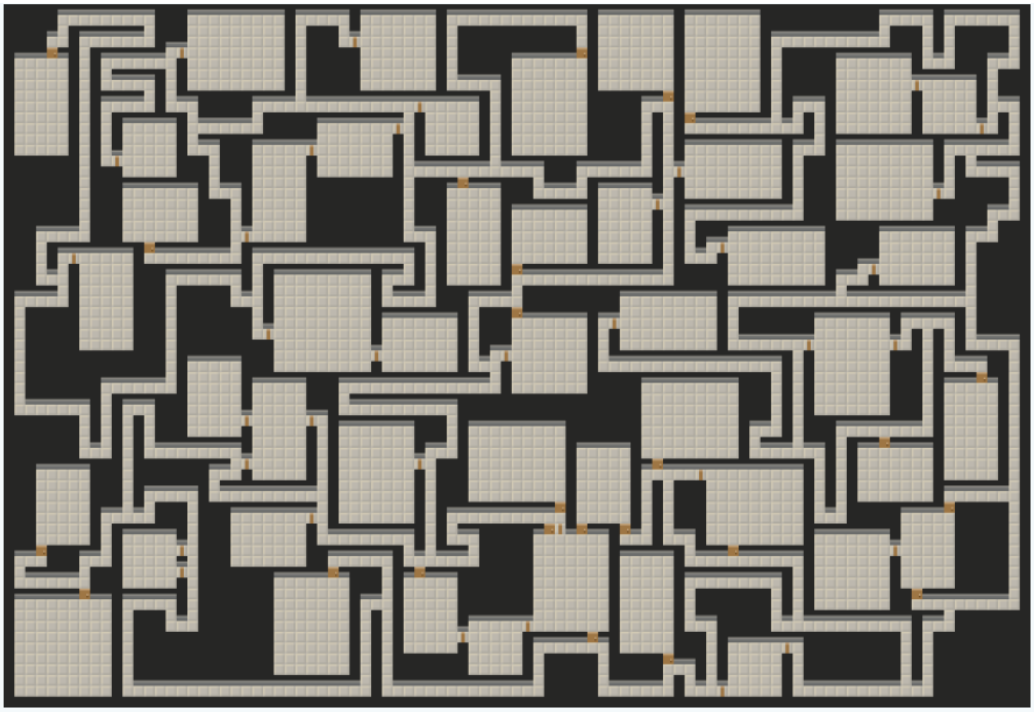

Figure 7: A dungeon map from Hauberk (Nystrom, 2014)

2.4.5 Constructive Generation

My research on constructive generation methods was more limited than that of the other methods I explored, due to the fact that I only became aware of them rather late into the project, and at first didn't realise at first the potential they could lend to my implementation. Given more time I would have liked to use constructive generation methods within the rooms of the map, for example to create city levels of connected buildings. Constructive generation hasn't been used in any production games as far as my research could uncover, but Alexander Dahl and Lars Rinde (Dahl & Rinde, 2008) explore the possibility of constructive methods for application in designing video game environments in their master thesis at Chalmers University of Technology, and I came across another paper exploring this concept from a group of researchers from Delft University of Technology and TNO The Hague (Lopes, et al., n.d.) – clearly this is a popular topic in Scandinavia!

Dahl and Rinde's implementation works by constructing a "skeleton" representing the outer border of the building, then dividing it into rooms and *transition areas*. Transition

areas are, simply put, corridors, and are used to separate rooms in such a way as to minimise the number of rooms with no outside wall – therefore maximising the chance that every room has at least one window. This is an important criteria in achieving realism, as real world architecture very often is developed around maximising natural light, and it is very rare to see a building with many interior, windowless rooms .

While Dahl and Rinde generate a floor plan which then is open to interpretation as to what each building and each room may be, Lopes et al developed an algorithm which generates buildings and rooms with a specific purpose. By generating a hierarchical layout, starting by splitting the floor into a private and public zone, and then dividing these zones into bedrooms, bathrooms, or living spaces and dining rooms based on the privacy of that section. This could be combined with a collection of premade furniture objects to create a fully realized house entirely procedurally. The possibilities of constructive generation for roguelike games are intriguing, but ended up falling outside the scope of my implementation.

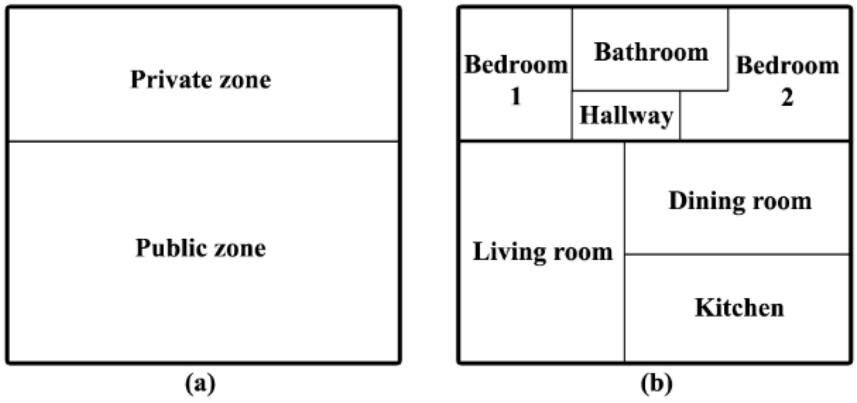

Figure 8: Hierarchical Building Zones (Lopes, et al., n.d.)

3 Design

3.1 Requirements

1: The program must generate dungeon-like levels for a roguelike game

- a: Levels must be of variable size
- b: Levels must contain a variable number of rooms
- c: All generated levels must be *completable*
- d: The *completability* of the levels must be *uncompromisable*
- e: Generated levels must not be *trivial*

2: Levels should be modelled as a two dimensional array of integer values corresponding to block types

3.1.1 Explanation of Requirements

Completable means that the player is able to make their way from the start of the level to the end goal, using the movement rules established by the game.

Uncompromisable means that it is impossible for the player to get themselves and the level into a state from which the level becomes non-completable. An example of this could be pushable blocks which can be moved into positions which trap the player, or a door that locks behind the player after they potentially missed an item that is required later in the level.

Trivial levels are those which have a single linear path from the beginning to the end, with no opportunity for exploration or alternative routes through the level. This can be avoided by increasing the number of connecting corridors between rooms, and by not placing keys in the same rooms as the doors they unlock.

The levels my program generates are played as a top-down view, with graphics and collision boxes defined in axis aligned bounding boxes (AABBs) - imagine a floor plan made up of square tiles or blocks. By representing the level as an array of integers, where for example 1 corresponds to a floor block, 2 is a wall, 3 is water, etc, the levels generated by the program are not limited to being used for a single game, and the same generation logic can be used to produce levels for different games by simply parsing the integer values in whatever manner makes sense for the game being developed.

3.2 Generation Method

I wanted my level generator to be a combination of approaches, taking the best ideas from different implementations and adding my own to the mix. I narrowed down the features I could use and assessed the pros and cons of each.

Noise Functions : This method produces very organic looking results, with seemingly incredibly varied and complex geometries that rival anything that could be created by a human designer aiming for a natural effect. However, the process of generating and combining many different noise functions is very computationally expensive. As level size increases, processing time increases exponentially, and even at smaller scales the processing cost for noise generation far surpasses that of cellular automata or the “rooms and mazes” method. In a non real-time application such as I am developing, this processing cost is not a major issue, as the level is never being generated during gameplay, but reducing load times is always a benefit. Load times aside, noise function based methods are not appropriate for this project, as I need to ensure that levels are completable and all areas accessible without the player having the ability to break through walls. Noise functions are simply too random to ensure this criteria is fulfilled without very complex methods to find paths between caves and connect them appropriately, and so it was ruled out at an early stage of my decision making process.

Cellular Automata : These were effective in creating natural looking cave structures, and could probably be adapted to form other natural environments such as forests with clumps of trees and clearings, swamps with water flowing between small patches of ground, or on a larger scale whole world maps of islands and continents. However, the cellular automata I found in my research stood out to me as producing levels that were more about the overall aesthetic result, and less focused on tuning the gameplay effects of the level structure and layout. Although the generated levels looked convincing and were varied, the natural structure could lead many unobservant players to adopt a "seen one, seem them all" mindset - caves and forests start to look alike when you get lost in them, whereas more man-made looking levels are easier to differentiate between, thus making cellular automata less ideal in terms of player engagement and perhaps retention. These cave like levels seemed to be so far removed from the other approaches that I wasn't sure I'd be able to properly assess the effectiveness in the same way, making cellular automata a poor choice for direct comparison to other methods. I did however consider the idea of working cellular automata into my final implementation for the aesthetic effects they can create, to give certain levels a more realistic organic finish.

Incremental Diggers :

This method better fits the gameplay focused generation that I wanted to implement, with a more structured approach to completability. Where cellular automata create one homogenous map of wall and floor blocks with no inherent structure, incremental diggers produce levels with distinct rooms and corridors, allowing for a much more ordered layout. This ordered structure helps gameplay, allowing us to place doors with keys, level objectives, and other items more critically by searching for specifically hard to reach or more convenient areas than in the more erratic cave systems that force us to place objects almost at random (and often don't allow for doors at all). I liked the convenience of incremental diggers in terms of proving completability by matching the player's movement rules, and decided I did want to include them in my implementation in some way, but I didn't want to follow an entirely digger based approach.

Rooms and Mazes :

Arguably the most structured and orderly of the approaches I explored, this method involves generating corridors and rooms separately, and allows the generator to instantiate room objects, distinctly separating floor blocks which are part of rooms and those which are corridors. This allows for object placement rules such as only allowing doors near the entrances of rooms rather than in any corridor, or making enemies stay within the room they spawned in, which are much more complex to implement when the whole map is one continuous object as it is with the incremental diggers approach. Although this involves many more object initialisations than the other methods mentioned, and therefore is inevitably less performant, the scope of my project does not involve generating levels while they are being played, and so extra processing required during generation isn't a major gameplay issue, and the extra control and structure allowed by this object based approach was worth the tradeoff.

3.3 Ideas from Existing Implementations

I liked *Hauberk*'s implementation when I first came across it, but wondered how I could draw from it while making improvements, without just copying the entire thing and

calling it done. There was no doubt that this was a good algorithm, and fit a lot of my requirements very well, so I began by reading through Nystrom's article and imitating his approach in GMS2. Once this was up and running I was able to tweak some of the values and quickly reveal the shortcomings of this approach when applied to my project. While *Hauberk* (and my implementation of its method) creates very good dense, dungeon-like levels, when tweaking my generator to try to achieve a variety of level styles, it became clear that this algorithm only worked for these standard dungeons and not much else. The perfect maze method used for the corridors works well when the level is dense with rooms, but when the corridors are given a large amount of space to fill, the level starts to become centralised on the point that is the starting point of the maze, creating a very odd aesthetic effect when the map is viewed as a whole, and causing the player to return to this central point far too often, even sometimes getting lost going round in circles. It's important to note that getting lost is not always a bad thing, but here it comes at the cost of level variety.

Another limitation of the Hauberk's method is that it is purely limited to man-made areas, and cannot produce anything organic looking. Although I was avoiding the cellular automata methods due to how vague they are in terms of discrete gameplay elements, I knew it was possible to create more organic levels with a more structured approach, from looking at *Wasteland Kings*' desert and swamp areas, so I started to look at how I could add in some of this variety myself. The immediately apparent difference was that *Wasteland Kings* allowed rooms to overlap, which is what leads to the vast open clearings seen in the desert levels, and the oddly shaped smaller rooms in swamp levels. Creating large open spaces was as simple as removing the rule that prevents rooms from overlapping, but smaller disconnected clusters of rooms required some extra code to increase the probability of a room being placed near to an existing room. I did consider adding different room shapes as possible objects to place in the map, but these preset shapes would quickly become recognisable to the player, making levels seem repetitive, and allowing compound rooms to form randomly like this created a much more natural look to the finished levels.

3.4 New Ideas

In the program's state as described in the previous section, I had a working level generator with adjustable parameters to create some different kinds of level, but before I moved on to gameplay elements such as item placement, I wanted to refine the layout generation further. The corridors generated follow random paths, often taking a lot of unnecessary twists and turns and even briefly doubling back on themselves. This is great for organic looking levels, whether that be a natural formation itself, or tunnels dug in an underground cave system which would twist like this to avoid rock deposits etc. However, I want to be able to differentiate between these organic tunnels, and more regimented, pre-planned structures like a castle or space station. These levels should have less random corridors, and so a new, more structured algorithm was needed for corridor generation. To fill this requirement, I added a toggleable parameter *straight corridors*. When this is unset, the corridors are generated as previously explained, as a perfect maze, but when it is set, the corridors are generated specifically between rooms, by picking one of the four cardinal directions and checking if there is an existing room or

corridor in that direction. If there is, a corridor is built to connect the room with that room or corridor. If not, the next direction anti-clockwise is chosen and the check is performed again. If all four directions have been checked and no valid room or corridor exists to link to in a direct manner, a right-angled corridor is required. The algorithm returns to the initial direction and starts checking one cell at a time in that direction, while checking the row (if the corridor is extending vertically) or column (if the corridor is extending horizontally) of the current cell for a corridor or room to link to. If the algorithm reaches the edge of the level without finding a valid corridor or room, the next direction is checked. Eventually, a room or corridor will be found, as by the time this algorithm terminates for all four cardinal directions, it will have checked every cell on the map grid.

This algorithm results in purely straight or right-angled corridors, creating a much more man-made look, well suited to a building's corridors. This "building" approach ran much faster than the "maze" approach, due to only placing blocks where they are needed, rather than filling the entire screen and then cleaning up what isn't required. The increased variety of level types that this corridor method allows for is beneficial, however the levels generated with the maze-like corridor approach tend to be much more interesting both visually and in terms of playability.

4 Implementation

4.1 Engine Choice

My requirements were very generic in terms of what framework I could use for implementation. Since the desired output needed to be easily represented as a two dimensional array I considered simply creating a command line application in C++ with Visual Studio with no graphical component. This would benefit from being lightweight and easy to plug in to any future game development project with few compatibility issues. I needed to test each level for *completability*. At first I thought this would mean building a game to play the levels in, which would make this raw C++ approach more complex, however I quickly found that this would not be necessary as many mechanics are the same across most games (for example the player can move up, down, left, or right, can move over floor tiles but not through walls, takes damage when moving through fire, etc). This meant that I could simply run a pathfinding algorithm like A* on the generated level array, adding weight to nodes that represent a more difficult path containing enemies or obstacles. If this algorithm finds a route through the level then I know it is completable. This pathfinding approach can also be used to find if a level is *trivial*, by counting the number of rooms visited (and more importantly the number of rooms that don't need to be visited).

However, I did in fact create a simple game to play the levels in after all, consisting of a playable character simply walking through each level from beginning to end. Although this is a technical project, and objective, quantifiable methods of assessing the generated levels are the priority, what I am creating is a game, and for all the memory efficiency, processing time optimisation, and technically "non-boring" layouts, if my program outputs levels that are not fun to play then it has failed. "Fun" may be impossible to

quantify in a scientific manner, but I decided I did need to at least test it in some way, so I would be building a game to go with the level generator I produced.

The level generator is the focus of the project, with the game itself really only existing as a small part of the assessment process. This meant I needed to keep development time of the gameplay elements as small as possible to leave room to focus on the main part of the project. For this reason I decided to use GameMaker Studio 2 (GMS2) as the engine and IDE for the level generator and game. Because of my pre-existing familiarity with the engine, and my access to code and assets from previous personal projects, it would be very easy for me to put together the gameplay elements necessary for testing the levels with minimal extra time invested. GMS2 has considerably higher memory overhead than a raw C++ implementation would, and by using it I would be sacrificing some performance, but I would still be able to measure results for different approaches against each other within the same engine, and the performance drop compared to the Visual Studio option was small enough as not to be an issue.

4.2 Implementation Structure

4.2.1 Data Structures

Although the requirements stated a two dimensional array as the data structure for the level data, GMS2 has a bespoke “ds_grid” data structure, which contains methods that are very useful for working on grid-based data such as the levels I am generating. The data from a ds_grid can be very easily copied to a 2d array if necessary after the level is complete, but during the generation phases, a ds_grid is more useful.

Within the grid itself I made further changes from the initial requirements. The finished level can be represented as purely integer data to be parsed in whatever manner the game in question can best interpret it, but I instead populated the grid with “map_block” objects. These objects store their own position in the grid and a “block_type” enum value which simply maps a human-readable keyword “wall”, “floor”, etc, to the integer values which are parsed by the map renderer. The map blocks also store an array of their immediate neighbouring blocks which improves the efficiency of the maze-like corridor generation and connection methods, and when adding further features the issue of preventing blocks on the edges of the map from looking for neighbours outside the bounds of the grid is taken away from the programmer and handled by the underlying classes. As with the decision to replace the 2D array of the map with a ds_grid, I can still easily produce a 2d array filled with integer values (copied from the *type* value of each map block) once generation has completed.

4.2.2 Generation Sequence

Sticking to the “rooms and mazes” approach, I chose to generate the rooms first, giving them full freedom to be placed anywhere in the space, and then add the corridors to connect them, but once I included overlapping rooms it wasn’t so simple. Although each rectangle of floor blocks is classed as a separate room object, when multiple rooms overlap they for all intents and purposes become one room and the program should treat them as such. For this purpose I created a new object obj_room_group. After the room generation step, the program will iterate through the rooms array, and check the top,

bottom, left, and right bounds against all the other rooms. This is implemented similarly to any AABB collision system, but with no broad phase calculation. A broad phase could have been included to reduce processing time, but the number of rooms in a level is usually relatively low (under 30 in a 100x100 level with no overlap allowed) , and this step would be constant between different level types, so I didn't want to spend unnecessary time optimising it when I could be focused on more important aspects of the program. If a room is overlapping with another, the overlapping room is removed from the list, a new obj_room_group is created, and the blocks they contain are added to the blocks array of the new room group before the overlapping room is deleted. Once a room has been checked against every other room in the map, it too is removed from the list and deleted. If a room does not overlap with any other room, a new room group is created and its blocks array is simply a copy of the room in question. When this process is complete, all room objects have been deleted and replaced with (usually a smaller number of) room groups, which will be the rooms used for the rest of the generation method.

4.2.3 Pathfinding

This section is merely the final step of the generation sequence discussed in the previous subsection, but the process became so complex, and the explanation so long, that it warrants its own subheading. Each level contains a starting room, where the player would begin the level, and an end room, with the player's ultimate goal being to reach this end room, any bonus pickups and treasures along the way being optional. The start room is chosen at random, and then the end room is selected as follows:

A new ds_grid is created with the same dimensions as the map grid, and then populated with obj_AStar node objects, one in each cell in which the map grid has a obj_map_block with type "floor". I then iterate through the newly generated A* grid and set each node's neighbours array to refer to each neighbouring node in the four cardinal directions (neighbours[0] is the right neighbour, neighbours[1] is the up neighbour, and so on anticlockwise), or *noone* if there is no neighbour in that direction, whether that is because the map grid had no floor block in that cell, or because that cell would be out of bounds of the grid.

Before I start running the pathfinding algorithm itself, the program creates a new ds_list (GMS2's dynamic array structure) which will hold references to all of the floor blocks which are contained within a room, rather than a corridor. To find which blocks this should refer to, the program iterates through the array of rooms, and for each of them iterates through all of the blocks it contains (which we know based on the position and dimensions of the room), adding each to the new room_blocks list. Of course, when rooms are allowed to overlap, we should make sure we're not adding duplicate blocks and wasting memory and check time, but this was an optimisation that I ended up not having time to implement.

Now that the A star grid and list of room blocks are set up, I pick the top right cell of the player start room as the starting point for the pathfinding algorithm, and then, in turn, in the order the rooms were generated (there is no need for there to be a specific order, but this is the order the rooms are indexed in the rooms array), run the standard A* pathfinding algorithm to find the shortest route from the starting room to the current room in question. Once the route has been found, the array of

nodes used in the route is stored as a field in an obj_aStar_route object, which, along with the array of nodes in the route, contains an array of the same size as the room array, with each value initialised to false. The program then iterates through the nodes in the route array, and checks if each occurs in the room_blocks array. If it does, then the value in the visited_rooms array with the same index as the room the block belongs to inhabits in the rooms array is set to true. After this check has terminated, the number of true values in the visited_rooms array then corresponds to the number of rooms visited by this route (more on why this is relevant later).

Once we have a route from the starting room to each other room, and we have both route length and the number of rooms each route visits, we can decide which room should be the end goal of the level. Route length may seem relevant here - we want the distance from the start to the end of the level to be as long as possible - but really this length doesn't matter at all if the player is just walking through a very long, empty corridor. What really matters to making a level take advantage of its size to produce the longest playtime possible is that it forces the player to visit as many rooms as possible. For this reason, the end room should be the one with the highest number of visited rooms along the shortest path to it from the start room. Of course it is likely that the player will visit more rooms not along this critical path, but we want to maximise the number of rooms they need to visit as much as possible. If more than one room has the same number of visited rooms, the one with the shorter route length is chosen - this may seem backwards, but we actually want to minimise the length of potentially boring corridors that the player needs to follow - we want them to visit all the rooms, but not necessarily via all of the corridors. If the route length also results in a tie between rooms, one of the candidates is chosen at random as there is no meaningful difference in the quality of the level based on which room is the end.

Each route also needs an *interest* value. This is calculated after the route is found by the A* algorithm. For each node in the route, if it is not a room node (i.e. it is part of a corridor), the program checks how many neighbours the node has. If a node has one neighbour, it is a dead end. If it has two neighbours, it is part of a linear path with only one way through (i.e. not interesting). If, however, a node in the route has more than two neighbours, it is now a point at which corridors diverge, and the player must make a choice on which direction to go. In short, for each node in the route, the route's *interest* score is increased by [number of neighbours - 2] if number of neighbours > 2. *Interest* is a desirable metric and is used in evaluating the overall quality of a level, but considering that the player is unlikely to choose the optimum path from the beginning to the end of the level, I found that the actual time taken for me to complete the level, and, subjectively, how interesting the level felt, was more affected by the *non-triviality* than the *interest* of the critical path, so I chose to only use *non-triviality* to decide which room should be the end.

Now that the end room has been chosen, the *non-triviality* score of the level is simply the number of rooms visited on the critical start-end path divided by the total number of rooms in the level, and the *interest* score is the number of decisions taken by the critical route. In short, non-triviality measures how efficiently the level makes use of all of the rooms it contains, while interest is a measure of how likely

the player is to “go the wrong way”, thus visiting rooms that weren’t strictly necessary to the completion of the level

5 Testing

5.1 Testing Method

In testing the effectiveness of the program I assessed both performance and efficacy.

Performance was assessed by the total amount of memory used by the program, and by processing time required both for each stage and for the level overall, for each generation algorithm for different level sizes and parameters. Of course I don’t have access to the source code for commercially available games, and while the source code for *Hauberk* is publicly available, this single point of comparison proved not particularly helpful, due to the wildly different engine overheads between the two implementations. This means I have no real point of reference to see how my implementation performs compared to other games’ level generators, so my conclusions are based mainly on personal opinion of questions such as “how long is a reasonable load time”.

Efficacy is further broken down into categories - *interest, non-triviality, failure rate,* and *fun*. The main stage of testing each level is to run a pathfinding algorithm on the map grid to simulate a player finding their way through the level. For this application, I chose the A* algorithm, as it is the standard for video game pathfinding, and the shortest possible route it is guaranteed to return represents the worst case scenario for a given level - that is, the player gets the smallest amount of playtime and exploration possible from the level, and it seems fitting that I assess the levels based on this worst case scenario. I already have this shortest route calculated, as well as the *interest* and *triviality* values, as part of the process of choosing the end room, so all I needed to do was collect these values to assess which parameters produced the best levels.

5.2 Testing Results

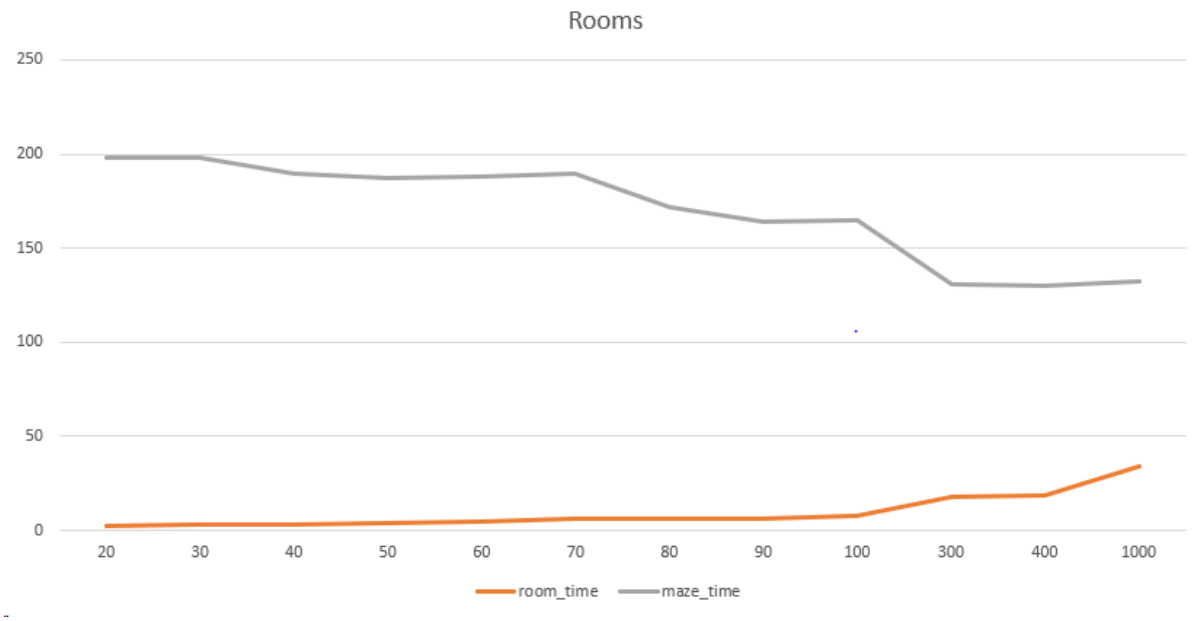

Figure 9: Room and Maze time (ms) vs Room Number

As the number of rooms increases, total generation time actually goes down, as the more space is taken up by rooms, the fewer corridors need to be generated. Eventually the number of actual rooms stops going up, as there is no more space to place rooms, but it still takes time to calculate a position for these unplaced rooms. This “soft room cap” seems to occur at around 0.002 * [total number of tiles in the map], or 20 rooms for a 100x100 map size.

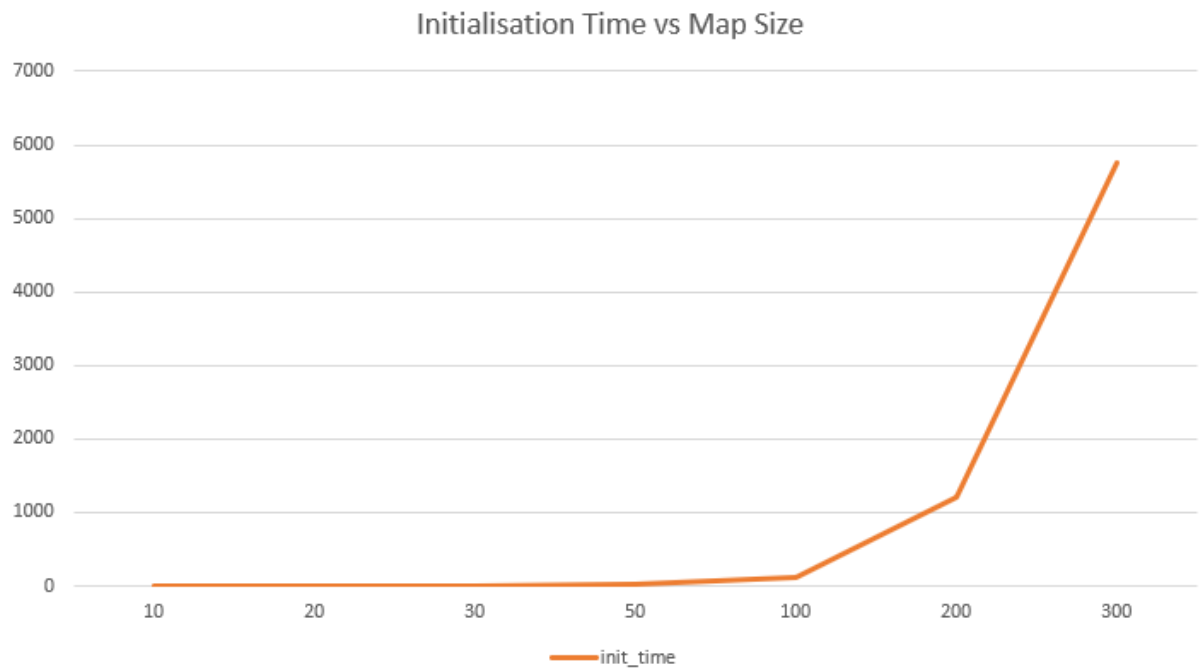

Figure 10: Grid Initialisation Time (ms) vs Map Size (sqrt(tiles))

As map size increases, the total generation time increases with roughly order n^2, where n is the x and y size of a square map. Although corridor generation time increases with map size, most of the time increase is actually in the initialisation of the ds_grid data structure, which quickly becomes unreasonably expensive at level sizes above 100x100. Although this size is more than big enough for most roguelike games, the possibility of using this algorithm for truly massive scale exploration based maps is incredibly limited by this initialisation time.

6 Conclusion

6.1 Fulfilment of Requirements

Overall I am happy with my implementation, but the direction of the project shifted slightly from the beginning to the end. I ended up focusing only on the generation of the main structure and layout of the level and didn't implement any doors, keys, obstacles or rewards. This came with the tradeoff of exploring the variety of levels I could produce – being able to create very different levels, from complex twisting corridors to large open expanses with the same generation algorithm simply by changing parameters, was, I think, a better investment of my time.

Requirements 1a, and 1b were fulfilled by allowing the map grid to be of any size (within processing time constraints), and by placing any number of rooms (although when overlaps are not allowed, there is a soft cap on the number of rooms that can fit in a level of a given size).

Requirement 1c was fulfilled by the implementation of the A* algorithm when deciding the end room, which ensures there is provably a traversable route from the beginning to the end of any given level.

Requirement 1d was fulfilled by technicality – by not including any enemies, obstacles, lockable doors or movable objects, it is impossible to change the state of the completability of the level. This requirement is a remnant of the initial aim to involve object placement in my implementation.

Requirement 1e was fulfilled by the method of connecting rooms to the corridors after the generation of the perfect maze, which makes the maze imperfect, creating loops and alternate paths by moving through rooms, which creates multiple possible route from the beginning to the end, and introduces the possibility of getting lost.

Requirement 2 was fulfilled, as the contents of the generated ds_grid structure can easily be translated into a 2D array which can then be parsed however necessary by whatever game is using the level generator.

6.2 Future Work

This project has set a solid foundation for the ideas I had on procedural level generation, but I haven't achieved everything I (perhaps unrealistically) set out to, and intend to develop it further in the future. The obvious final step of this is to create a roguelike game which uses this level generator, but the undertaking of developing an entire game, especially from the visual, character, and narrative design, is an entirely different undertaking to the more technical project I have started for this dissertation.

While I'm unsure if I will continue to that final finished point, the next step for certain is to flesh out the levels with enemies, objects, and obstacles to turn the output of the program from an empty structure of rooms and corridors to what can really be described as a game level.

I also would like to explore ways of turning the rooms from empty rectangles into more interesting structures themselves. This could be achieved by implementing the constructive generation methods I explored in the research phase to generate a floor plan layout for each room, or by running the algorithm I already have developed on a much reduced size to correspond to the room – although the rooms output by this method are not nearly as interesting as I could achieve through constructive generation techniques.

7 References

Cellar Door Games, n.d. *Rogue Legacy.* [Online]
Available at: cellardoorgames.com/roguelegacy/

Cook, M., 2013. *Gemerate Random Cave Levels Using Cellular Automata.* [Online]
Available at: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664
[Accessed April 2019].

Dahl, A. & Rinde, L., 2008. *Procedural Generation of Indoor Environments.* [Online]
Available at:
http://www.cse.chalmers.se/~uffe/xjobb/Lars%20Rinde%20o%20Alexander%20Dahl-Procedural%20Generation%20of%20Indoor%20Environments.pdf
[Accessed May 2019].

Flafla2, 2014. *Understanding Perlin Noise.* [Online]
Available at: https://flafla2.github.io/2014/08/09/perlinnoise.html
[Accessed April 2019].

Fournier, P., n.d. *The Ultimate Captive Guide.* [Online]
Available at: http://captive.atari.org/Technical/MapGen/Introduction.php
[Accessed March 2019].

Kun, J., 2012. *The Cellular Automaton Method for Cave Generation.* [Online]
Available at: https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/
[Accessed April 2019].

Lopes, R. et al., n.d. *A Constrained Growth Method For Procedural Floor Plan Generation.* [Online]
Available at: https://graphics.tudelft.nl/Publications-new/2010/LTSDB10a/LTSDB10a.pdf
[Accessed May 2019].

Marshall, D., 2009. *Procedural Generation, and the problem of Player Perception.* [Online]
Available at:
https://www.gamasutra.com/blogs/DanMarshall/20161110/285234/Procedural_Generation_and_the_problem_of_Player_Perception.php
[Accessed 05 2019].

Nijman, J. W., 2013. *Random Level Generation in Wasteland Kings.* [Online]
Available at:
https://web.archive.org/web/20130430215452/https://www.vlambeer.com/2013/04/02/random-level-generation-in-wasteland-kings/
[Accessed May 2019].

Nystrom, B., 2014. *Rooms and Mazes: A Procedural Dungeon Generator.* [Online]
Available at: http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/
[Accessed March 2019].

Staff", "., 2009. *The Making Of: Rogue.* [Online]
Available at: https://web.archive.org/web/20120815191124/http://www.edge-online.com/features/making-rogue

Tippets, J., n.d. *3D Cube World Level Generation.* [Online]
Available at: http://accidentalnoise.sourceforge.net/minecraftworlds.html
[Accessed April 2019].

Yoyo Games, 2019. *GameMaker Studio 2 Documentation.* [Online]
Available at: http://docs2.yoyogames.com/
[Accessed April 2019].