Procedural World Building in Roguelike Games

Student Name: Benjamin Jones

Supervisor Name: Dr Tom Friedetzky

Submitted as part of the degree of MSci Natural Sciences to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

Abstract —

Context/Background

Procedural content generation in videogames has seen a surge in popularity in recent years. However as game development progress, users expect more and more intelligent and engaging game level design and as a result, investigating new, interesting methods of procedural content generation is vital to keeping end users happy.

Aims

This project aims to draw comparisons between current methods of procedural content generation in a roguelike game environment and build upon them to create different and exciting methods of level design. The implementation will focus on creating a number of novel algorithms alongside an intelligent method of evaluating their suitability.

Method

A variety of algorithms will be developed and implemented by building on and combining various existing methods found in existing roguelikes and more sophisticated modern games. Perlin and Simplex noise algorithms will be used to generate the initial random seed, which will then be modified by a variety of methods including the *Cellular Automata* approach and the use of geometric space-filling curves such as the Hilbert curve and Sierpinski curve. The effectiveness of these will then be evaluated by a custom heuristic, ensuring the quality of the resulting world.

Results

Conclusion

Keywords — Procedural content generation, roguelike, world building, noise algorithms, artificial intelligence, heuristic, space-filling algorithms.

I INTRODUCTION

A Roguelike Games

The term Roguelike [CAPITALISE OR NOT? emph?] stems from the original 1980s UNIX dungeon crawler *Rogue* created by Michael Toy and Glenn Wichman, giving birth to hundreds of games such as *Nethack* and *Dwarf Fortress* that follow the same style and structure and revolutionising the gaming industry as we know it today (Craddock 2015). There are many different styles of roguelikes these days, but most share the same core features: turn-based gameplay, procedurally generated tile-based dungeons or maps, and randomly created monsters and items (Shaker et al. 2015).

The basic premise of the game has the player controlling a single character through numerous ASCII represented areas filled with monsters, items and traps. However what made it so revolutionary at the time was the fact that the game was different (and the gameplay unique), every time you played it thanks to its procedurally generated nature, and its ability to create an infinite set of playable dungeons was unparalleled(K. Compton 2006). However these days players expect more from games and this work aims to advance on the relatively primitive types of procedural generation seen in these early fan favourites.

This work aims to investigate existing methods of procedural generation within videogames and create a new approach that is both visually appealing and interesting to play to meet the increasing demands of the modern gamer. The genre of roguelike games provides an excellent [appropriate?] framework for this aim, combining sufficient complexity with an appropriate method of visualisation of the algorithms. [not sure about this?]

B Procedural Content Generation

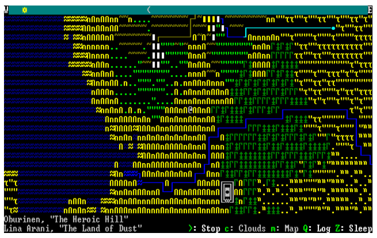
Procedural content generation (PCG) is the process of creating content through the use of random numbers as seeds to generate objects using algorithms and mathematical functions, as opposed to the typical method of being manually created by a developer (Shaker et al. 2015). This has many possible applications, from running real-world simulations on procedurally generated objects and environments (Adams 2007), to generating interesting and unique textures and graphics (Perlin 1988), and of course for use in video games. In short, this is typically achieved by modifying the results of mathematical noise algorithms (Togelius et al. 2015), which generate heightmaps or intensity maps that can be adapted to create procedurally generated content, such as terrains or graphics (Perlin 2002).

Procedural generation has many advantages unrivalled by other generation methods: it is typically much faster to evaluate, it has the ability to generate arbitrarily complex and intricate designs on-the-fly and has a particularly low memory footprint (A. Lagae 2010). Additionally, the variation of input parameters can easily generate an incredible number of uniquely styled designs. The result is that procedurally generated game content has the capability to offer every game user a unique gaming experience every time they play.

A perfect example of the power and potential of PCG is *Minecraft*. At the time of writing, *Minecraft* sales have hit more than 70 million world-wide, making it one of the most popular games of all time (UKIE 2016). However at it's base, *Minecraft* draws many parallels with the methods of generation seen in even some of the earliest Roguelike games, for example it uses at its core a 3D adoption of the 2D Perlin fractal noise algorithm adopted by many popular Roguelikes (Persson 2012), including the 2006 variant, *Dwarf Fortress* (Craddock 2015).

Also recently hitting the spotlights recently is indie game *No Man's Sky*, a futuristic sci-fi role-playing game (RPG) that procedurally generates an entire universe populated with up to 18 quintillion planets for the player to explore at their leisure (Games 2015) [IS THIS HOW TO CITE A CONFERENCE?]. Developed by a team of only 10 people, *No Man's Sky* demonstrates the power and potential that procedural generation has to offer to the gaming market.

A final game worth mentioning is award-winning 2004 German 3D shooter *.kkrieger*. The game won its award not because of its content however, but because of its extensive use of procedural generation techniques mean everything from the textures to the in-game sounds (as well as the traditional


(a) A sample of the playable overworld in *Dwarf Fortress*. Image sourced from: http://www.bay12games.com

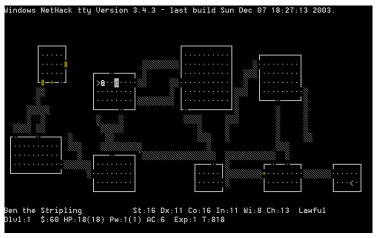

Windows NetHack tty Version 3.4.3 - last build Sun Dec 07 18:27:13 2003.
(b) Typical generated dungeon structure in *Nethack*.

Figure 1: A figure with two subfigures

monsters and levels) are generated completely procedurally. This allows the entire game to be coded in just 96 kilobytes of memory, meaning that even screenshots of the game take up more room than the game itself- allowing *.kkrieger* to take up 3000x less storage than an equivalent conventionally designed game (.kkrieger 2004). This game exemplifies what can be achieved using procedural techniques.

As games get more complex and take longer to build, coming up with new and exciting ways to programatically generate game content could be the key to a revolution a game industry expected to reach $103bn in 2017 (Newzoo 2014). Furthermore, as the industry continues to get more and more competitive, procedural generation is likely to play a key role in allowing indie game developers to keep up with their big-budget counterparts.

C Contribution Direction

While roguelikes and their derivatives remain popular more than 40 years after their introduction, current favourites such as *Nethack* and *Dwarf Fortress* struggle to bridge the gap between being visually appealing and functionally engaging. While *Nethack* offers intricate and complex gameplay mechanics, its level design is basic and while more recent editions have offered a graphical tile-based experience, it fails to keep up with the demands of the modern market. On the other hand, *Dwarf Fortress* offers incredibly complex algorithmic techniques (owing to its 10 years of open source development), but again falls short in level aesthetics.

This work aims to bridge the gap filled by a generation of roguelikes, intending to create worlds in a roguelike setting that are both visually pleasing to the modern gamer whilst maintaining the complexity and gameplay aspects that have enticed thousands of players over the years. We predicted that successful implementations of procedural content generation, both in roguelikes and in a wider context, have the ability to satisfy gamers while reducing the cost of content creation significantly, as well as increasing entertainment and replayablility. The latter not only has important implications in the well established single-player market, but also in the expansive domain of massively multiplayer online games (MMOGs), most of which gain revenue from the continued and repeated involvement of players month to month, and estimated to lead the global games market in 2018 with a predicted revenue of approximately $34.3bn dollars (Newzoo 2014).

D Deliverables

The purpose of this work in summary was to investigate the various ways in which procedural content generation is used to generate game environments, and then design and build one or more variants based on the best of these approaches with the intent to create exciting and visually interesting in-game environments in a roguelike design space. The algorithm also intended to analyse created worlds to evaluate a measure of how interesting a world is, and only produce worlds that pass a set threshold to the player. Please note unfamiliar concepts in this section will be discussed in greater detail in section III.

In order to achieve this, the project was split into basic, intermediate and advanced deliverables. The basic objective intended to produce a Java executable capable of handling user input, and output an ASCII grid output, providing a platform to display 2D ASCII environments for a player to explore indicative of the roguelike genre. The world generation algorithm was to be deterministic and include a 2D noise generator, providing the fundamental basis for procedural environment generation. The algorithm was then to provide a method of 2D noise smoothing, with the smoothing function able to produce explorable areas and structures for a player to explore.

We achieved the basic objective with an implementation in Java using the AsciiPanel library to produce a graphical user interface (GUI) to provide input and output to the player. A basic noise function of binary random output was created, and a smoothing function based on the *Cellular Automata* ruleset was applied to form arbitrarily sized 'caves' from the noise, explorable by a player.

The intermediate objectives were focussed on enhancing the in-game environment by implementing a more complex heightmap noise algorithm capable of producing more realistic in-game environments that exhibit contour behaviour, whereby the noise in one location is dependant on the surrounding elevation, mimicking the gradient nature of real-world environments. A further criteria was the that the generation algorithm was capable of operating efficiently, such that worlds could be evaluated based on a 'variability' heuristic and then re-drawn instantaneously or even multiple times should a map not reach the thresholds set.

These intermediate objectives were achieved after researching various different noise algorithms. In the end, the fractal *diamond-square* algorithm was adopted due its capability to produce contour based continuous heightmaps with low memory and CPU overheads, making it perfect for a roguelike game. This was then combined with a custom-made variant of the *Cellular Automata* algorithm to smooth the noise, producing worlds that match the criteria to create more realistic worlds. Furthermore the efficient method of this noise generation and smoothing approach means that the variability coefficient-calculated by statistical analysis of the generated heightmap- can be used to evaluate worlds and regenerate worlds on-the-fly that do not match up to the variability criteria set in order to make a world 'interesting'.

The advanced criteria for this work was that worlds created should have 'distinctive' features, and that worlds would be distinguishable on each playthrough to create a unique gaming experience that worked so well in *Rogue* and subsequent derivatives. An extension to this includes the use of underlying mathematical functions, space-filling curves, that should exist within the structure of the heightmap to decrease the likelihood of encountering ‘dead ends’ within game structures and increase player enjoyment. Another aspect of the advanced criteria was that algorithm should also be able to identify connected regions of the map, with an understanding of how connected the produced environment is.

One of the advantages of procedural generation over traditional content production is the ability to create vastly different environments with minimal effort. We demonstrated this capability with the introduction of an overworld with a customisable probability function, allowing the developer to create a variety of different *biomes* by adapting the heightmap transformation probabilities. Furthermore, we introduced the ability to feed in distinctive world features such as the *Hilbert* space-filling curve to the noise generation algorithm, adding an underlying structure to the caves that are prevalent throughout the overworld. Finally, a breadth first *flood-fill* search algorithm was used to identify connected regions of the map from the players position.

[This may well be too long]

II RELATED WORK

—- provide a narrative, not a list, so aim to structure this —- look at the lit survey? - CRITICAL ANALYSIS

- In this section it is important to note that work in this field comes from both academic sources but also work by amateur enthusiasts should not be ignored, but instead critically evaluated for its value. - draw from the very large active community - equally important - there exists many examples of different evaluation techniques - here we will discuss just some of the relevant examples from across big name games in the industry,

- Talk about image generation of TerraGen - using midpoint displacement

- Talk about other relevant examples of the algorithms i use

- Talk about perlin noise research and why i didnt use that- allude to the negatives of this approach/ scalability. talk about simplex noise also – but implementation difficulty and runtime didn’t match criteria for advanced objectives of tradeoff for worlds capabale of being produced quickly

- Talk about the extended usage of these in minecraft and other things

- Procedural generation as a subsection of other games eg something like borderlands but not -skin generation ? -population identification - Hilbert curve research?

- Noise generation in amateur games - many references on this in my diss folder

- Research into cellular automata?

- Could talk about dwarf fortress/ other roguelikes

- Canyon 3d work - work by M carli et al on canyon creation using mean shift algorithm - shows how you can bridge the gap between 2d and 3d, how 2d algorithms are relevant - explain that an implementation represented in 2d is still a 3-dimensional map just projected into 2d - k means

III SOLUTION

A Specification of Software

The implementation of this work was developed and produced in Java (SE 1.8). There are a number of reasons why Java was well suited to this work. Java is a well established, high-level language preferred by thousands of amateur and professional game developers alike, and as such there are many previous implementations and libraries specific to roguelikes and procedural content generation from which this work was able to benefit from. By utilising copyright-free libraries and examples from previous roguelike enthusiasts, a quick prototype framework for the game was implemented and subsequently developed to form the foundation of the back-end of the game. This enabled the focus of the project to be on more important and interesting aspects of the generation algorithms. To this extent, many suitable libraries exist, and this work made particular use of the *JFrame* and *AsciiPanel* libraries, in addition to the *Apache Commons Math* packages exising within the Java framework. *JFrame* is a well known and established library enabling the use of a windowed frame for which the program is able to handle input and output, whereas *AsciiPanel* is described as a 'Java Console System Interface', popular among roguelikes and allowing for multi-colour ASCII text character output to the *JFrame* environment. In addition to *AsciiPanel*, the implementation made use of an open source framework developed by the creator of *AsciiPanel*, providing access to the various functions that the *AsciiPanel* library provides while allowing the project to focus on more complex aspects of the procedural generation algorithmic design (Spangler 2015). The capabilities of the resulting game engine allowed for an interactive window of a similar quality to that of estabilished roguelikes such as *Nethack* and even *Dwarf Fortress*

Java is also particularly useful due to its highly object-orientated nature. Since the design aspect of the game can be separated into many parts, it is helpful to be able to abstract and separate each process into different classes, allowing the main focus of the algorithmic side of the project to be apart from the engine handling input and output to the console. This makes development much easier, as well as improving bug fixing and code readability.

In addition, Java is known as a platform-neutral language (Lemay 1996), meaning the final implementation will be easily run on almost any platform (except mobile) without the need to recompile the source. This is very powerful and allowed for continued development and evaluation from a variety of computers during the project.

Finally, Java was chosen because of its familiarity. While the project may have benefited from features of other languages (such as the rapid prototyping and clear code style of Python), previous familiarity with Java enables a reduced implementation period in the system development life cycle, allowing more time to be spent on the design, planning and evaluation aspects of the cycle.

B Methodology

The *Agile* software development methodology was used in the project in order to meet its aims. This was selected because we felt that the project would benefit from achieving each functional target in individual rapid sprint cycles, which would then be able to be thoroughly tested for quality between cycles before the next implementation aspect was introduced. We also felt that this enabled a higher degree of freedom than other considered software development approaches (such as the *Waterfall* approach), enabling the constant evaluation of the project at regular intervals.

C Noise and Smoothing Algorithms

Noise algorithms form the foundation to most procedurally generated content and feature heavily in this work. The noise environment can then be modified by a variety of different approaches to form the basis of the content. There are a multitude of different noise algorithms [AS DEMONSTRATED IN THE RELATED WORK], however the basic objectives were achieved using discrete binary white noise, that is a 2-dimensional array of randomly assigned bits that are independent and theoretically identically distributed from each other (Clauset 2011).

C.1 Cellular Automata

The binary noise algorithm is simple and easy to implement, paving the foundation for a smoothing technique known as *Cellular Automata*. This technique came about in the 1970s when British mathematician John Horton Conway devised his evolutionary *Game of Life*, a game that transforms an initial state of 'cells' into intricate and interesting patterns and designs by the use of 3 rules of state (Gardner 1970):

- Survivals: Every cell with two or three neighbouring cells of the same type survives for the next generation.
- Deaths: Each cell with four or more similar neighbours dies (is removed) from overpopulation. Every cell with one neighbour or none dies from isolation.
- Births: Each empty cell adjacent to exactly three neighbours–no more, no fewer–is a birth cell. A cell is placed on it at the next move.

This process is used upon the result of the binary white noise to 'smooth' the landscape by turning areas that are mostly neighbouring walls into walls and areas neighbouring floors into floors, however we have adapted the ruleset in order for the algorithm to converge after a finite number of iterations- namely that cells do not 'die' if they are surrounded by similar neighbours, and births are suppressed. The process is then repeated a number of times until the noise is sufficiently smoothed and the landscape converges towards a stable environment.

It's clear to see from Figure 2 that this process very quickly converges to produce interesting, procedurally generated cave-like structures for a roguelike game setting, in addition to being computationally inexpensive (the generation is instantaneous even for very high levels of iterations (1000+) on a moderately powerful machine, and has time complexity O(wh) where w is the width and h is the height of a map (Wolfram 1984)).

It is, however, limited in its usefulness by itself, as the results of the algorithm are inconsistent, occasionally producing uninteresting or unnatural worlds. Worse still, its particularly prone to what

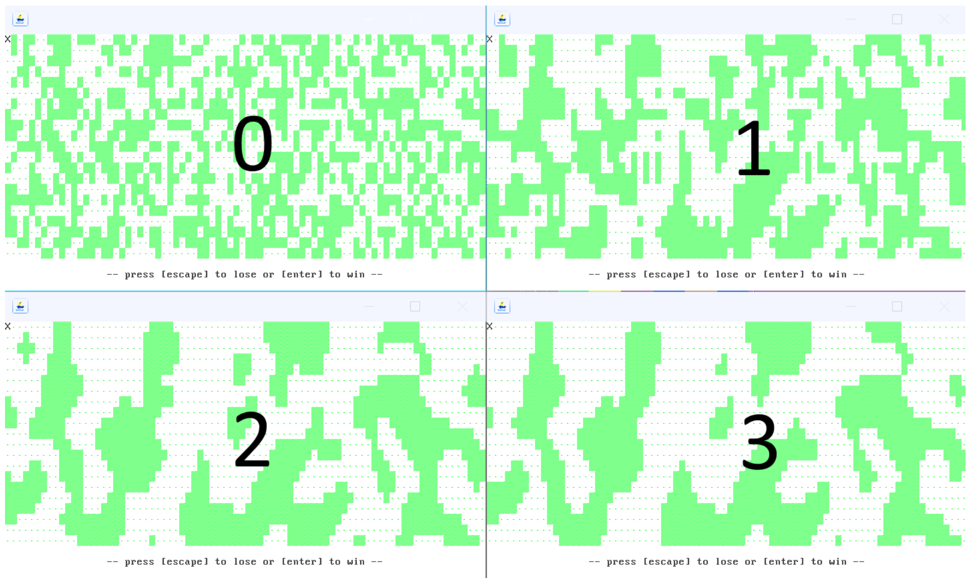

0
1
2
3
-- press [escape] to lose or [enter] to win --
-- press [escape] to lose or [enter] to win --
-- press [escape] to lose or [enter] to win --
-- press [escape] to lose or [enter] to win --

Figure 2: Cellular Automata cave smoothing. The numbers indicate the iterations of the algorithm, where the top-left image is before the smoothing process. Image self generated.

is known as ‘the isolated cave problem’, whereby produced maps are often particularly ‘disconnected’- that is the player is often unable to advance to large regions of the cave.

This nature (seen in Figure 2) is undesirable in order to fulfil the intermediate aims of the project, and a number of alternative approaches were considered to solve this problem.

One approach for example would be to generate many worlds and test for completeness, discarding the worlds that do not fit our criteria. Because of the instantaneous nature of this method of world population, we are able to create and evaluate many worlds very quickly, however the problem arises when we wish to expand our world size. A larger world boundary results in an increased statistical probability that the created world will be highly disjoint almost every time. An alternative approach would be to identify disjoint sections and post-process connect them up by removing wall segments. This does guarantee connectivity, but is also prone to making the world look unnatural, defeating the point of using the algorithm in the first place. One of the biggest advantages with *Cellular Automata* however is that rules can be tweaked easily to modify the output to the developers preference, and this is exactly what amateur developers at *RougeBasin* have done (RogueBasin 2016). By experimenting with the probabilities of birth and death rates, they are able to create connected caves that still maintain their natural look and feel.

This solution works well for smaller game environments, however even with the finest tweaking of parameters, its output is still unable to cope with larger maps of the nature this work aims to create. In this work, we come up with a different solution to the problem, owing to the use of both a completeness check using a breadth-first search and the introduction of space-filling curves, discussed in detail in section X and section Y respectively. [REFERENCE]

C.2 Diamond-Square Algorithm

The above approach is a good starting point and has good potential for certain use cases, but even if it were possible to solve all of the aforementioned issues, it is fundamentally unable to create continuous heightmaps and struggles to create defining features; two aims of this project. With this in mind, we looked to more complex noise generation methods and while there are many available options, we selected the *Diamond-Square* algorithm for this work.

THIS IS ALL BETTER OFF IN THE RELATED WORK SECTION—- The *Diamond-Square* algorithm was initially proposed by Fournier, Fussell and Carpenter at the SIGGRAPH conference in 1982 and is an efficient way of generating fractal heightmaps in computer graphics (A. Fournier 1982). It is particularly effective at modelling the contours of natural landscapes and is used by Planetside Software in their professional scenery generator program *Terragen*(Claghorn 2014), whose outputs can be seen in popular movies such as *Star Trek Nemesis* and *The Golden Compass* (Fairclough n.d.).

It is important to point out however that the algorithm is not without its flaws and Gavin S. Miller notes in his 1986 paper *The Definition and Rendering of Terrain Maps* that the algorithm is subject to 'the creasing problem', whereby creases and slope discontinuities occur along the boundaries between fractals (Miller 1986). Having evaluated his argument however, we feel these negatives are outweighed by its fast evaluation time and impressive results, and are unlikely to become a problem in a roguelike setting.

—

The recursive algorithm starts with a square grid of dimension n^2+1 and giving random heights to the four corner values. The algorithm then recursively iterates over two steps, as shown in Fig 3 and summarised in brief below:

> *The Diamond Step:* Calculate the mean average of the 4 corners of a square and add a random perturbation to this value. The result is given to the value in the centre of the square, and produces 4 sided diamonds.
>
> *The Square Step:* Calculate the mean average of the 4 points of each diamond and add a random perturbation to generate the value of the midpoint, producing more squares.

The process is repeated recursively until every point in the grid is filled. However while this forms the foundation for the algorithm, we make a couple of modifications to the standard implementation to improve the results. The first is a measure to counteract the implementation issues that occur at the heightmap boundaries where there is no edge-side diamond value to use. While many applications simply ignore this condition at the edge and introduce edge artefacts, this work makes use of 'edge wrapping', the concept of using the values at the opposing edge to complete the boundary cases. This is key to the prevention of edge artefacts, and even allows the algorithm to be used in an infinite basis, although that will not be explored in this work.

The second is the variation parameter, h, which dictates how the world varies with respect to each iteration. Without a variable h, each iteration is free to vary from the last by whatever maximum height limit is set, however by reducing h with each iteration, the smaller the square/diamond, the less the central midpoint value is allowed to deviate from its surroundings. Therefore, the variation function dictating how h changes is key to how the algorithm reacts and behaves. These modifications are best seen in the psuedocode in Algorithm 1.

Algorithm 1: The Diamond-Square Algorithm

```
   Data: arrayLength, seed, startingMaximum
   Result: heightMap[][]
1  Generate 2D array of type double = heightMap[][];
2  Initialise random number generator with seed = randGen();
3  Set corner values of heightmap with random perturbations;
4  Initialise $h$ as startingMaximum/2;
5  while heightMap is incomplete do
       // Diamond step
6      Calculate square corner average = $avg$;
7      Set midpoint = $avg$*h*randGen();
       // Square step
8      if boundary==true then
9          Get value from opposite side
10     end
11     Calculate diamond corner average = $avg$;
12     Set midpoint = $avg$*h*randGen();
13     $h$ = f($h$);
14 end
```

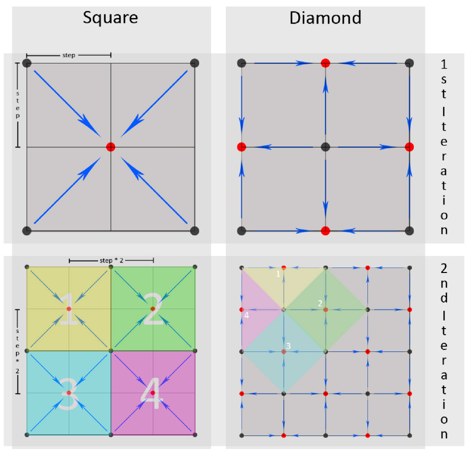


Figure 3: The process of the Diamond Square algorithm can be seen clearly here. Image taken from: `https://cmcbain.com/2014/07/31/procedural-terrain-generation-diamond-square-algorithm/`

The resulting output of this produces a heightmap that can be used in conjunction with a set of tile transformation probabilities, allowing for the introduction of alternative tile types such as hills, forests, plains and mountains that occur at varying heights. Such depth adds considerable complexity to the game, expanding the number of ways the game can be played.

+ multiple biomes

D Space-Filling Curves

+ ASCII feature read in

E Testing

Unit testing?

Qualatitive testing

Bug testing- robust boundary stress testing by forcing boundary conditions

F Solution Pipeline

(or something similar)
describes how each feature comes together

—
Mark Scheme
-Adequacy of the solution
-Specification and design
-Outline of implementation issues
-Description of tools used
- and why
-Verification and validation
-Discussion of testing

IV RESULTS

—Mark Scheme
-Evaluation method adopted?
-Clarity of the results - use python graphs?

Timing data

Use of parameters h, v for variablility coefficient, o, the openness parameter

Parameters to be decided based on a range of values to be decided by user feedback -¿ which produces the most interesting results?

V EVALUATION

How suitable was my approach
- Agile caused problems
Appraisal of project organisation- what?

Strengths:
Produces excellent visual worlds most of the time
Ability to introduce arbitrary features inside caves from ASCII graphics
Very fast to generate worlds

Limitations:
Slow breadth first search
Only capable of doing square maps currently
Doesnt generate space filling curves by itself
Could report more statistical information to describe and shape worlds
Ability to add underlying ASCII only in caves

——————-

VI QUESTIONS:

When to use *Cellular Automata* or 'Cellular Automata' Psuedocode improvements?
References:
-too many urls?
-unnecessarily takes up lots of space currently
Linebreaks between paragraphs?
Image URL sources, shorten to goo.gl/asdfgh ?
Introduction- Shift some content to related work?
Related work:
-Roguelikes or wider context?
-Introduce perlin noise here? then talk about in evaluation why not used?

Concerned about lack of formal evaluative measures -¿ low marks for large sections

References

A. Fournier, D. Fussell, L. C. (1982), 'Computer rendering of stochastic models', *Communications of the ACM* **25**(6), 371–384.

A. Lagae, e. a. (2010), 'A survey of procedural noise functions', *Computer Graphics Forum* **0 (1981)**(0).

Adams, S. Goel, A. (2007), 'Making sense of vast data', *Intelligence and Security Informatics, 2007 IEEE* pp. 270–273.

Claghorn, J. (2014), 'Fractal terrain generator example 9.3 — generative landscapes', `https://generativelandscapes.wordpress.com/2014/09/15/fractal-terrain-generator-example-9-3/`. (Accessed on 04/30/2016).

Clauset, A. (2011), ‘A brief primer of probability distributions’.

Craddock, D. L. (2015), *Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Changed the Course of Video Games*, Authors Republic.

Fairclough, M. (n.d.), ‘Planetside software (terragen)’, `http://planetside.co.uk/`. (Accessed on 04/30/2016).

Games, H. (2015), ‘No man’s sky gameplay demo’.

Gardner, M. (1970), ‘Mathematical games - the fantastic combinations of John Conway’s new solitaire game ‘life”, *Scientific American* **223**, 120–123.

K. Compton, M. M. (2006), ‘Procedural level design for platform games’.

.kkrieger (2004), `http://web.archive.org/web/20110717024227/http://www.theprodukkt.com/kkrieger`. (Accessed on 02/05/2016).

Lemay, L. Perkins, C. (1996), *Teach Yourself Java in 21 Days*, Sams.net Publishing, Indiana.

Miller, G. (1986), ‘The definition and rendering of terrain maps’, *Computer Graphics* **20**(4).

Newzoo (2014), ‘Global games market will reach $102.9 billion in 2017’, `http://newzoo.com/globalgamesdata`. (Accessed on 02/05/2016).

Perlin, K. (1988), ‘An image synthesizer’, *Computer Graphics* **19**(3).

Perlin, K. (2002), ‘Improving noise’, *Computer Graphics* **35**(3).

Persson, M. A. (2012), ‘The word of notch terrain generation, part 1’, `http://notch.tumblr.com/post/3746989361/terrain-generation-part-1`. (Visited on 02/12/2016).

RogueBasin (2016), ‘Cellular automata method for generating random cave-like levels’, `http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels`. (Visited on 02/07/2016).

Shaker, N., Togelius, J. & Nelson, M. J. (2015), *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, Springer.

Spangler, T. (2015), ‘Asciipanel’, `https://github.com/trystan/AsciiPanel`. (Accessed on 02/05/2016).

Togelius, J., Shaker, N. & Nelson, M. J. (2015), Fractals, noise and agents with applications to landscapes, *in* N. Shaker, J. Togelius & M. J. Nelson, eds, ‘Procedural Content Generation in Games: A Textbook and an Overview of Current Research’, Springer.

UKIE (2016), ‘The games industry in numbers — ukie’, `http://ukie.org.uk/research`. (Visited on 02/12/2016).

Wolfram, S. (1984), ‘Cellular automata as models of complexity’, *Nature* **311**(5985), 419–424.