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

February 11, 2016

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.

Proposed Solution

The proposed game will be built and played in Java using the JFrame and asciipanel libraries as a framework. The combination of existing algorithms and novel techniques will produce multiple potential procedurally generated worlds, which will be evaluated by the program and the best fit will form the playable environment in a roguelike game.

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

I INTRODUCTION

A Roguelike Games

The term Roguelike 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 Angband 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 and randomly created monsters and items [cite PCGBOOK?].

The basic premise of the game has the player controlling a single character through numerous ASCII represented dungeons filled with items, monsters and traps[cite roguebasin?]. However what made it so unique and revolutionary at the time was the fact that the game was different every time you played it thanks to its procedurally generated nature.

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 code and mathematical functions, as opposed to the typical method of being manually created by a developer[possible citation of a paper]. This has many possible applications, from running real-world simulations on procedurally generated objects and environments (Adams 2007), to generating interesting and unique textures[cite] texture generation, and of course for use in video games. This is typically achieved by modifying the results of mathematical noise algorithms (Togelius et al. 2015), which generate heightmaps and intensity maps that can be transformed into random terrains.

Procedural generation has many advantages unrivalled by other generation methods: it is typically much faster to evaluate, it can generate arbitrarily complex and intricate designs on the fly and has 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 [reference- help?]. 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, Minecraft uses at it's core a 3D adaptation of the 2D Perlin noise algorithm adopted by many popular roguelikes[cite notch], including Nethack and Dwarf Fortress[cite twice].

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 galaxy populated with up to 18 quintillion planets for the player to explore at their leisure. 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 its extensive use of procedural generation techniques mean everything from the textures to the in-game sounds (as well as the traditional 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 and allowing .kkrieger to take up to 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 in the gaming industry (expected to reach $103bn in 2017 (Newzoo 2014)).

C Project Aims

The purpose of the project in summary is to investigate the various ways procedural 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 aim to create exciting, interesting and challenging environments for a roguelike game setting. The project will also aim to evaluate the output of the algorithms with a fitness heuristic function to identify whether or not a generated output meets these aims.

The deliverable should be an executable program running in the Java programming language that is capable of producing two dimensional ASCII worlds for a player to explore. The algorithm should be deterministic, and ideally will be efficient such that it is capable of producing and evaluating multiple worlds quickly, identifying the best and putting that forward to the playable game environment. The game world should have the additional constraint that all parts of the map should be traversable from any other part of the map, explicitly checked by the algorithm.

D Deliverables

Please note: unfamiliar concepts introduced here will be discussed later in the design section.
Minimum objectives:

- Produce a basic game engine in Java to handle user input and ASCII grid output, to enable world creation and character movement, including side scrolling.
- Choose appropriate data structures.
- Implement a deterministic 2D noise generator to produce fundamental basis for procedural dungeon creation.
- Implement a smoothing function based on the Cellular Automata approach to produce a basis for a cave-like structure.
- Implement a probability density field for the placement of walls and floor based upon a mathematical space filling function and a noise generator.

Intermediate objectives:

- Develop probability density algorithm to produce interesting and varied worlds based upon various mathematical functions.
- Produce a metadata structure for the world that has extra information about each grid space, including items and other features.
- Look at potential effects of using more complex noise algorithms (eg Perlin & Simplex) in world creation.

- Be able to check that a world is connected.

Advanced objectives:

- Develop world evaluation heuristic capable of rating worlds on a number of criteria.
- Look into maze generation algorithms and underlay them in the world.
- Investigate possibility of procedurally generated dungeon 'puzzle' rooms.
- Optimisation of procedural generation techniques such that generation is fast enough to quickly generate multiple worlds and select the most suitable using the world evaluation heuristic.

II DESIGN

A Requirements

ID	Requirement	Priority
FR1	Implementation of fundamental game engine in Java handling IO	High
FR2	Implementation of rudimentary noise algorithm	High
FR3	Create Cellular Automata style algorithm for noise smoothing	High
FR4	Create probability density algorithm to determine block identity	High
FR5	Created worlds must have the connected property	Medium
FR6	Produce metadata structure for grid space	Medium
FR7	Produce world fitness heuristic	Medium
NFR1	Optimisation of space filling function in probability density algorithm	Medium
NFR2	Improvement of noise algorithm using Perlin approach	High
NFR3	Ability to produce interesting worlds with definable features	High
NFR4	Ability to test for world connectivity	Medium
NFR5	Additional game areas involving maze/puzzle generation	Low

Table 1: Functional & Non-Functional Requirements

B Specification of Software

The implementation of the project will be achieved in Java (SE 1.8). There are a number of reasons why Java is well suited to this project. Java is a well established, high-level language used by millions of people all over the world and as such there are many previous implementations and libraries specific to Roguelikes in Java from which the project can benefit. By utilising copyright free libraries and examples from previous roguelike enthusiasts, a quick framework for the game engine can be easily created and ammended to the project's aims, while focussing development on more important and interesting aspects of the implementation. To this extent, many suitable libraries exist. In particular, we will make extensive use of the 'JFrame' and 'asciipanel' libraries. The former is a well known library enabling the use of a windowed frame for which the program is able to handle input and output, and the latter a 'Java Console System Interface', which is popular among roguelikes and allows for ASCII text character output in various colours.

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 makes it easy to continue development and evaluation from a variety of computers during the project, especially in the event that additional processing power is preferred later on to run and evaluate multiple simulations of the design algorithms. [POTENTIALLY REMOVE PARAGRAPH]

Finally, Java has been chosen because of its familiarity. While the project may benefit from features of other languages (such as the rapid prototyping and clear code of Python), 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.

The ‘Agile’ software development approach will be used in the project in order to meet its aims. This is because the project will benefit from achieving each functional target in individual rapid sprint cycles, which can then be thoroughly tested for quality between cycles before the next implementation aspect is introduced. This also enables a higher degree of freedom than other software development approaches (such as the ‘Waterfall’ approach), enabling the constant evaluation and possible change of separate requirements should the fulfillment of previous ones require so, due to the unpredictable nature of certain algorithms effectiveness and ease of implementation.

C System Design

The game engine that houses the game world and handles input and output as mentioned will primarily use the asciipanel library, but also follow from an open source implementation created by an online roguelike programmer known as ‘Trystan’ (RoguelikeTutorial 2016). This provides a framework that is able to be modified for the purposes of this project, in order to focus on more complex aspects of the procedural generation algorithmic design. The capabilities of the resulting game engine allow for an interactive window of a similar quality to that of estabilished roguelikes such as Nethack and even Dwarf Fortress.

D Noise and Smoothing Algorithms

Noise algorithms are crucial to most procedurally generated content and will feature heavily in the project as they form the foundation for the world. The noise environment can then be moulded by a number of different approaches to form the basis of the map.

There are a few kinds of noise algorithm, from very basic noise algorithms where each ‘pixel’

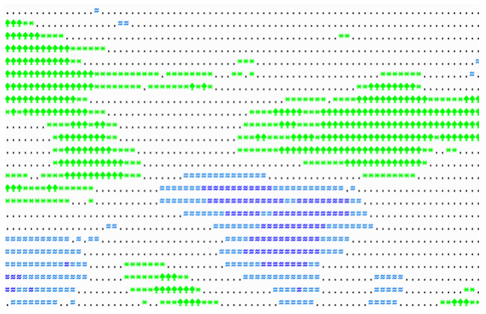
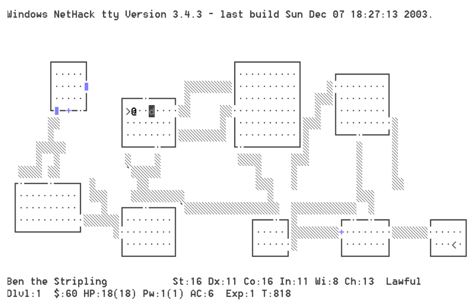


Figure 1: Asciipanel GUI example
Image sourced from:
`http://www.headchant.com/2012/02/15/asciipanel-as/`



```
Windows NetHack tty Version 3.4.3 - last build Sun Dec 07 18:27:13 2003.
```

```
Ben the Stripling          St:16 Dx:11 Co:16 In:11 Wi:8 Ch:13  Lawful
Dlvl:1  $:60 HP:18(18) Pw:1(1) AC:6  Exp:1 T:818
```

Figure 2: Nethack GUI example [probably copyright]

has an equal chance to be filled or unfilled, to much more advanced versions such as Perlin noise and Simplex noise (Ebert 1994)(Perlin 2002). The type of noise algorithm selected depends on the project requirements, and the sort of algorithm that will aim to transform the noise environment into a playable, enjoyable world.

The psuedocode for a basic binary noise algorithm described in FR2 (Table:1) is shown in algorithm 1 below:

Algorithm 1: Basic binary noise production

```
for $x$ <$mapwidth$ do
    for $y$ <$mapheight$ do
        if $tile[x][y]$ = psuedo_random_number() <0.5 then
            $tile[x][y] = floor$
        else
            $tile[x][y] = wall$
```

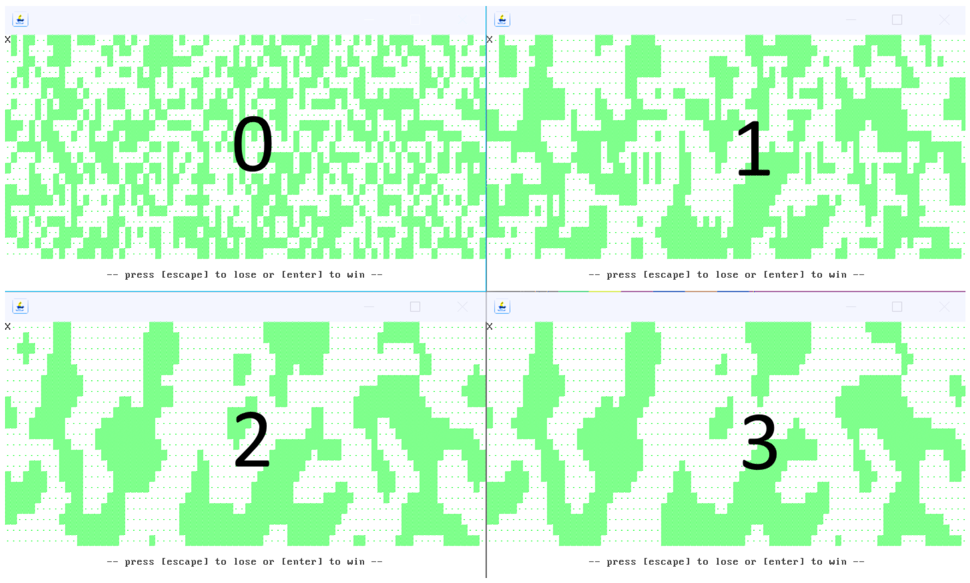
E Cellular Automata

The binary noise algorithm is simple and easy to implement, paving the foundation for a smoothing technique known as 'Cellular Automata'. This technique became famous in the 1970's 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 4 rules (Gardner 1970):

- Survivals: Every cell with two or three neighbouring cells survives for the next generation.
- Deaths. Each cell with four or more neighbours dies (is removed) from overpopulation. Every cell with one neighbor 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 noise algorithm to ‘smooth’ out the landscape by turning areas that are mostly neighbouring walls into walls and areas mostly neighbouring floors into floors. The process is then repeated a number of times until the desired combination of walls and floors is achieved.

An example of the process run on the basic noise algorithm seen in algorithm 1 is shown below:



```
-- 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 3: Cellular Automata cave smoothing. The numbers indicate the iterations of the algorithm, where the top-left image is before the smoothing process

It’s clear to see from figure 3 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).

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 is known as ‘the isolated cave problem’, whereby produced maps are often ‘disconnected’- that is the player is unable to advance from any location to any other location.

This nature (seen in fig 3) breaks functional requirement 5 of the project, and therefore alternative methods or amendments must be implemented to achieve this goal. There are a variety of ways that we may achieve this. One approach for example would be to generate many worlds and test for completeness, discarding the worlds that don’t 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 disjoint almost every time. An alternative approach would be to identify disjoint sections and 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 enhance the output. By experimenting with these rules, previous adopters of this approach have been able to produce vastly different environments, some of which are much less likely than others to form disjointed caves (RogueBasin 2016).

F Perlin Noise Algorithm

The approach above is a good starting point for the project but even with the finest tweaking of the algorithms rules, the output is limited by its ability to only produce binary maps (only walls and floors) and its inability to create specific features or structures procedurally, falling short of our non-functional requirement 3. With this aim in mind, we look to a different noise producing algorithm, Perlin noise, to advance upon this requirement.

Perlin noise was developed by Ken Perlin in 1985 and has been widely used in applications ranging from procedural world generations to computer graphics (Perlin 1988) (A. Lagae 2010). It is particularly good for creating natural looking textures in games and simulated environments, and could be adapted for use in a roguelike setting.

It should be noted at this point that Perlin noise is not without flaws and Perlin himself noted a number of improvements on his original design in his 2002 paper, 'Improving Noise' (Perlin 2002). The result of his paper was the invention of 'Simplex Noise', a superior noise algorithm with better efficiency, however Perlin noise is still useful because it is both simpler to implement and the effect of its inefficiencies are questionable in the scope of this project. The project aims to quantify these at a later date.

Perlin noise generation can be split into two components, a noise generating function, and an interpolation function. The noise generating function operates in a similar fashion to the binary noise generator described in section D, except instead of taking on either 0 or 1, the function generates floating point numbers between 0 and a maximum value (known as the amplitude). The interpolation function then smoothly averages across these values, as seen in figures 4 & 5, and we take the distance between points on the x-axis to be the wavelength of the noise wave (Tulleken 2008).

To generate Perlin noise, we generate many of these waves (in 1 dimension) or images (in 2 dimensions) at varying wavelengths and frequencies (the combination of which is known as an octave), and sum them up to produce our final image. An example of the process may be seen in figure 6.

The applications of this are many-fold, but in a roguelike game this can be used to create the idea of depth into a 2-dimensional world, which was not possible using our previous Cellular Automata approach. The Perlin noise image may be used to form a height-map, where density is proportional to height. This allows the introduction of alternative tile types, such as hills, forests, trees and plains with different movement cost to the character, or even impassable objects such as canyons, crevasses or mountains. Such depth in the game adds considerable complexity, expanding on the many ways the game can already be played. An example can be seen in figure

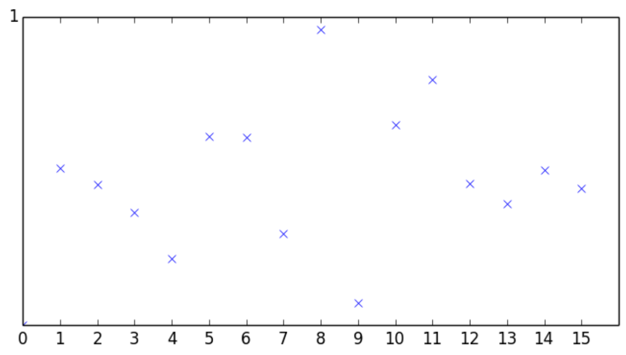


Figure 4: Example 1-dimensional noise function. A random value with a maximum amplitude of 1 is assigned to each point on the X axis.

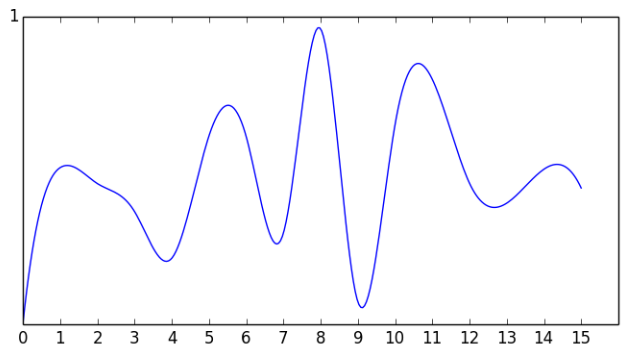


Figure 5: By smoothly interpolating between the values, we obtain a continuous function, with wavelength 1.

7 below:

G Space-Filling Curves

The results of Perlin noise look promising, with interesting structures and added complexity introduced into the produced worlds. However it is clear that a structure to the world is still lacking, with no easily defined start or end point, specific features, rooms or obvious places to put monsters, traps and treasure that make a Roguelike game so special to play. This project aims to solve these problems by hybridising well-known noise algorithms such as Perlin noise with mathematical functions known as ‘space-filling curves’, in a novel approach not yet seen to the best of our knowledge in previous Roguelikes.

Space-filling curves are fractal in nature, and given a starting seed, produce deterministic and characteristic curves, shapes and paths that will be used to form a basis for the world generation algorithm.

This will be achieved by taking random segments of a generated curve and overlaying them on top of a Perlin noise image. The result will become a ‘probability density heightmap’ for the

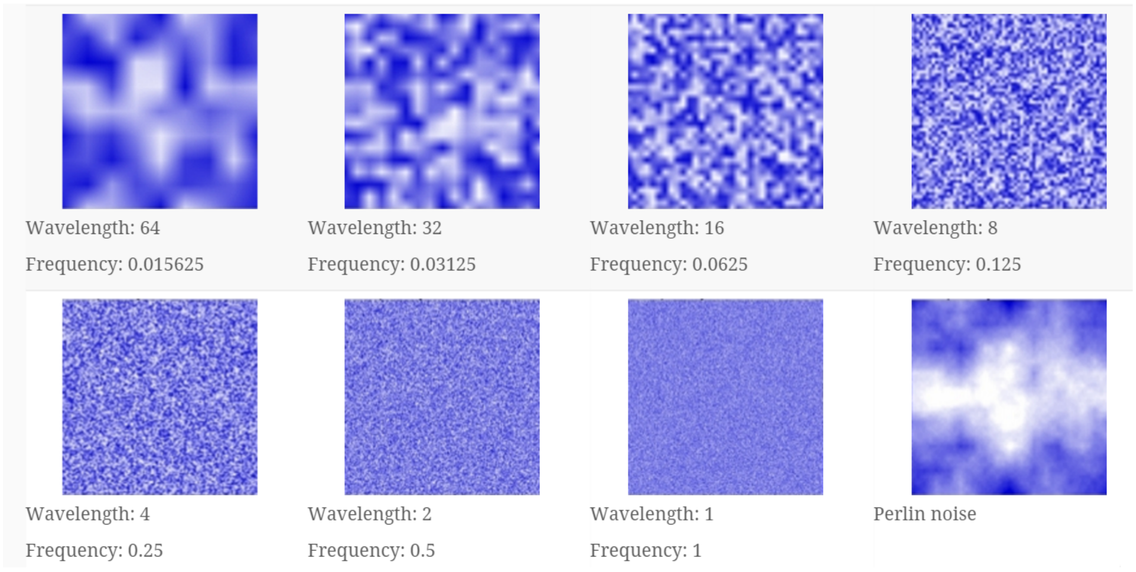


Figure 6: The Perlin noise generation process. The final image is the result of the previous 7 images summed together. Image taken from Tulleken. H, (2009), *'How to Use Perlin Noise in Your Games'*, Dev.Mag, (20), sourced from: `http://devmag.org.za/2009/04/25/perlin-noise/` on 09/02/16

world, where densely populated regions will have a greater chance to become traversable tiles. The advantage of this is that the generated world will be much more likely to have a structure, with defining curves, edges and paths from one edge to another (potentially linking to other generated segments), while maintaining the natural look and feel of a Perlin generated environment. Additionally, there are a wealth of different types of space-filling curves; some examples are displayed below in [ref figure]. The project will experiment with a number of these to determine the resulting output, and potentially use multiple ones to reflect upon different in-game environments or 'biomes', with their own look and feel arising from the different structures provided by the various curves.

As an extension, it may be possible to fulfil non-functional requirement 5 by experimenting with a different set of mathematical algorithms: procedural maze generators. Depending on initial parameters, maze generators are able to procedurally produce a wide variety of shapes in a similar fashion to space filling curves, except they have a specified entrance and exit(s). However the purpose of a roguelike is not necessarily to solve a maze-like puzzle, and the usefulness over a regular space filling curve is questionable.

References

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.

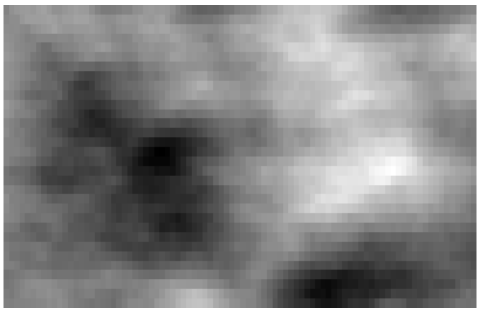


Figure 7: Normalised grayscale fractal Perlin noise, sourced from `http://www.nolithius.com/game-development/world-generation-breakdown`

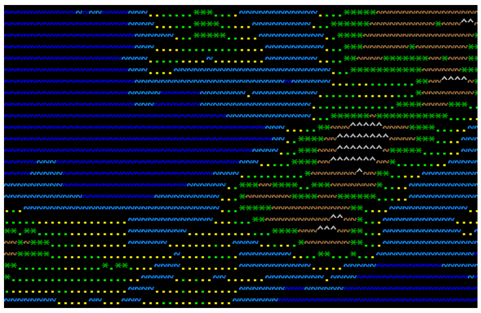


Figure 8: Terrain applied to Perlin noise from figure 7. Sourced from `http://www.nolithius.com/game-development/world-generation-breakdown`

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

Ebert, D. (1994), *Texturing and Modeling, A Procedural Approach*, AP Professional, Cambridge.

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

.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.

Newzoo (2014), 'Global games market will reach $102.9 billion in 2017', `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).

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).

RoguelikeTutorial (2016), 'Trystan's blog', `http://trystans.blogspot.co.uk/`. (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.

Tulleken, H. (2008), 'How to use perlin noise in your games', *Dev. Mag* **20**.

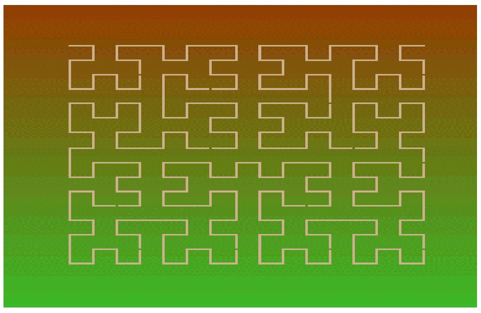


Figure 9: The Hilbert curve provides large, easy paths but is limited to just horizontal and vertical degrees of freedom.

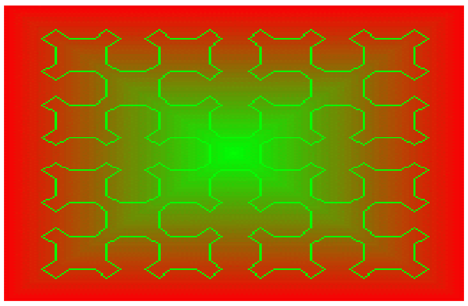


Figure 10: The Sierpinski curve doesn't produce easily followable channels, but could form the basis of several rooms for various gameplay features.

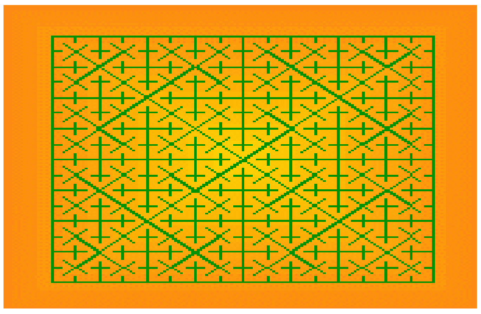


Figure 11: The Cesaro curve looks less natural but might provide interesting results for separate in-game biomes.

Figure 12: All images sourced from TGMDev, available online at: `http://www.tgmdev.be/applications/acheron/acheron.php`