Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Agent Based Modelling and Generative Design For Settlement Generation (in Minecraft)

Charlie Haslam

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Monday 17th May, 2021

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Charlie Haslam, Monday 17th May, 2021

Contents

List of Figures

Executive Summary

Procedurally generating various objects (such as settlements) using various existing methods (e.g. random noise) is a fairly well documented process, with uses in media such as video games and CGI for films. However, these traditional methods have their limitations and simulationist approaches through the use of methods such as agent based modelling can help to create more interesting and realistic generated artefacts. This project aims to investigate the suitability of such methods by creating a generator for a specific example problem, generating settlements in the game Minecraft. This means that the project has two main aims. The first, to create a generator that can demonstrate some of these agent-based methods and the second, to use this generator to evaluate the value and suitability of such methods in a wider context.

- I started the project with a lot of time devoted to research and collecting materials relating to the project.
- I wrote a nearly 2000 lines of source code (Python), creating a highly capable settlement generator.
- This represents more than 100 hours of writing and debugging code.
- Results were then gathered and evaluated by making use of the generator to create a variety of settlements

Supporting Technologies

- The `GDMC-HTTP-Interface` was used for my project to provide an interface to the minecraft world: https://github.com/nilsgawlik/gdmc_http_interface
- Various features of `MatPlotLib` were used to visualise results and outputs of my project: https://matplotlib.org
- Used a 'Gist' for the 'largest area in a histogram' algorithm found here: https://gist.github.com/zed/776423
- Where running times are used as data, the program was run on my 2020 MacBook Pro with a 2 GHz Quad-Core Intel Core i5 and 16GB of RAM

Acknowledgements

I would like to thank my supervisor, Seth Bullock, who has been extremely helpful throughout the project and offered valuable insight. I would also like to thank my friends and family who supported me during my studies.

Chapter 1

Introduction

This first chapter is intended as an introduction to the project and the motivations behind it, without going into too much technical detail (discussed further in chapter 2). To begin, an introduction into agent based modelling, generative design and generated content in video games. Following on from that, a more detailed description of project objectives and motivations, including details of the competition that first inspired the project.

1.1 Core Concepts

1.1.1 Generative Design

Figure 1.1: Different designs for a part created using generative design in CAD/CAM software, credit: autodesk.com

Designing complex objects with a variety of constraints and features can be a challenging task. Sometimes, human designers can struggle to create a suitable solution. Designing such complex elements can be assisted by computers and software in a variety of ways, generative design being a key way of doing this. Generative design makes use of software to aid in the iterative design of some desired object, an *output*. A designer uses a series of constraints to fine tune a *feasible region*, a set of valid solutions for the given constraints. Often, the designer is not a human but a computer program which iteratively tweaks constraints and values to improve past solutions, a form of evolutionary design [4].

There are many uses of generative design in a variety of areas. A very typical example consists of generating a variety of design models for a part [17], like those seen in Figure 1.1. This enables the design of complex and novel parts that would otherwise be difficult (or impossible) for a human to design whilst still conforming to design constraints [19]. Another example is the use of generative design for planning architectural designs such as office spaces [20].

1.1.2 Agent Based Modelling

Agent based models are a type of simulation which show the interaction and behaviours of individual agents [6]. These agents can be thought of as autonomous decision making entities. They individually assess situations and act according to their own set of rules. Properties of agents describing these behaviours and interactions are known as *elementary properties*. Properties that become apparent at a higher, overall level are called *emergent properties*. This means that the predictable (or pre-determined) actions and interactions of individual agents result in more complex and nuanced results at a higher level.

The complicated nature of these simulations and the intertwining interactions of agents can make these simulations hard to predict. Sometimes, these simulations result in emergent properties that were never considered or do not agree with some previously defined assumptions. Once results are obtained (even if a desirable result is achieved), the complexity and opacity of certain computer models can make it hard to gain an understanding of the outcome [21].

Agent based modelling has been used for a variety of problems. Notably, areas such as simulating the evacuation of buildings [15] or the impact and spread of a disease [14], a problem very relevant during the current Covid-19 pandemic.

For this project, the agents can be thought of as 'settlers'. They will be responsible for exploring the land, building structures and harvesting materials themselves. The full description of these agents is detailed in later chapters, but for now it is enough to consider them as *player-like* individuals.

1.1.3 Generated Content in Games

Video games often use generated (or random) content to help them be more interesting and engaging for players whilst saving production time and costs. For many games, this is done during the production stages of a game [13]. *Procedurally generated content* is used to help make the vast amounts of 3D models, textures and other assets used to create games. This reduces the amount of time required to make all the content for a particular title, lowering production costs and making games quicker to develop. An example of this would be generating a series of dungeon layouts to be used in a game [28].

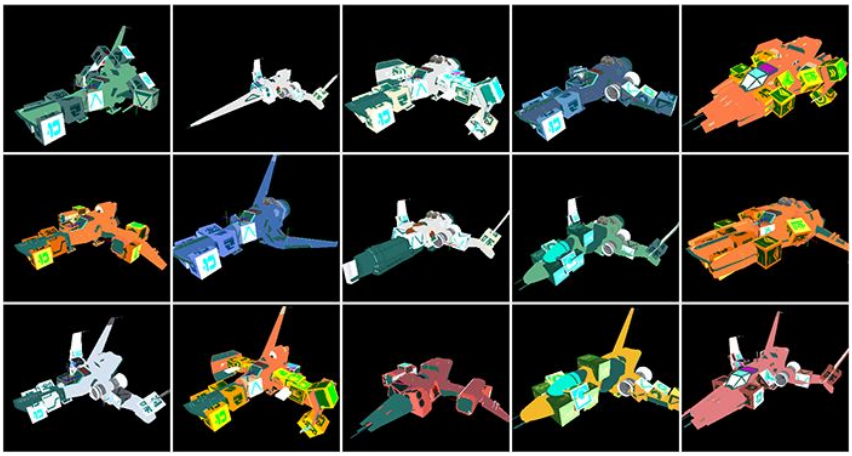


Figure 1.2: A collection of ships procedurally generated to demonstrate the methods used in No Man's Sky, credit: [12]

Many games also use generated content as a way of varying various elements during gameplay. For example, generating terrain layouts for game worlds [3]. Some games such as *No Man's Sky* are even completely procedurally generated and result in nearly infinite planets, plants and creatures to be discovered [27]. Games such as Minecraft don't have a series of terrain assets that were generated during production. Instead, the game includes a terrain generator that runs as part of the game, greatly varying the experience for the player every time they wish to start over [2]. This generator includes the generation of basic villages or *settlements*.

1.2 Minecraft

Minecraft is an open-world game, first fully released in 2011. It is the most sold video game of all time, selling over 200 million units on various platforms [26]. In Minecraft, the world is made of *blocks*, each

representing a unit cube made of a particular material (wood, stone, dirt, etc.). A player can collect these blocks, use them to *craft* new materials and then use them to build their own structures (e.g., houses, castles, statues). There is a story based objective in Minecraft but many players choose to play Minecraft for its creativity and freedom, ignoring the overarching objective and setting their own challenges and goals (giving Minecraft its *sandbox* nature).

While playing, it is common for players to build structures such as houses and farms. These offer game-play advantages like safety and food but they are also a way of expressing creativity. Especially in a multiplayer world, many players may build several structures near to each other to form a developed area resembling a settlement. Minecraft comes with a built-in *Village* generator. Occasionally, small villages with various buildings and a few non-player characters are added to the world as it is first generated.

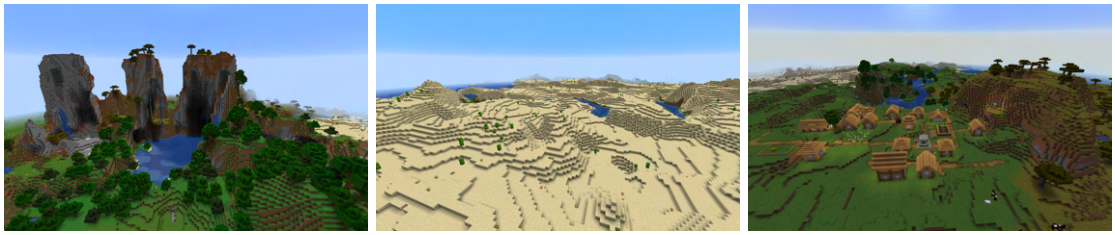


Figure 1.3: Examples areas of Minecraft worlds

1.2.1 Generation in Minecraft

Terrain and Seeds

The worlds in Minecraft are generated using a *seed*. A text based input which results in the deterministic procedural generation of a world [2]. If left empty, the seed is randomly generated for each world. This system allows for generating unique and interesting worlds but also allows for reproducing them if players wish to reuse or share a layout. The seed is then used to generate 3D Perlin noise, a way of computer generating natural looking textures. This Perlin noise represents the density of terrain [22]. More complex generators are then used as extra layers to add things such as caverns, rivers and ravines.

Villages

Occasionally, villages are generated and added to the world. Figure 1.4 shows an example of a generated village. These are often characterised by a series of small, often identical houses connected by paths. Villages often contain extra buildings such as farms or churches. However, this results in relatively small settlements which recycle the same building designs and are not all that interesting.

Figure 1.4: An example of a village generated by Minecraft

1.3 Motivations

The main motivations behind this project is to create a settlement generator and to use this generator to explore the usefulness of methods such as agent based modelling. This generator will use agent-based modelling to create interesting and believable settlements in Minecraft worlds. One of the original inspirations for this was the Generative Design in Minecraft competition.

1.3.1 Generative Design in Minecraft competition

The Generative Design in Minecraft competition (GDMC)[1] is an AI based settlement generator competition set-up by a group of academics [23]. The competition is run yearly and entries are scored by a panel of judges based on several categories:

- Adaptability - How well a settlement adapts to its surroundings but how it also influences its surroundings.
- Functionality - Settlements should provide some functionality such as safety, accessibility and food.
- Narrative - The story or history behind a settlement.
- Aesthetics - How good a settlement looks.

The deadline for the 2021 competition entries is the 31st of May, which is after this thesis is due to be completed but I fully intend to enter the results of this project into this year's competition.

1.4 Objectives and Challenges

The high-level objective of this project is to explore the suitability of newer methods such as agent based modelling for content generation. This will be explored by creating a settlement generator capable of creating settlements for a given area in any Minecraft world. Evaluating how well this generator works and assessing the flexibility of this generator will help draw conclusions about the suitability of such methods.

To achieve this, the task has been split into the following objectives:

1. Create a representation of agents in the Minecraft world. they should be able to move around following similar rules to a player. They should be able to navigate to a specified location using *path-finding*.
2. Agents collaboratively explore and survey the land in the area a settlement is to be built in.
3. Use this data to construct buildings in suitable locations. Where needed, surrounding terrain should be modified. Buildings should be designed with the surrounding area in mind (use of available materials for example).
4. Agent data is then used to create paths between buildings and areas (creating bridges over rivers for example).
5. The resulting settlement should have a sensible layout, be clearly influenced by surrounding terrain and materials whilst also hinting to the different stages of construction and influences of the agents that built it.
6. Use the resulting generator to help evaluate the usefulness of such methods in a broader context. Exploring elements such as the flexibility of the generator and the quality of the generated elements.

[1] https://gendesignmc.engineering.nyu.edu

Chapter 2

Technical Background - write some kind of heading here afer

The aim of this next chapter is to introduce the more technical aspects of the project and relevant work. The first section will cover the GDMC in more detail, exploring the original intent behind the competition, as well as the results of the first year the competition ran. The following sections will aim to cover relevant work and topics which helped influence the final outcome of the project, such as previous competition entries, A* path-finding and collaborative exploration.

2.1 The Competition

The academics who created the competition discussed some of the details behind the competition before running it for the first time in 2018 [23]. The aim of the competition is to create a more open-ended AI competition. Not all games have clear goals and not all AI should be simply about winning the game, the competition aims to focus on areas relating to procedural content generation and generative design.

2.1.1 Judging Categories

The competition is judged on four categories, briefly described in the previous chapter. While success in the competition isn't the only way of measuring the success of this project, it proves to be quite a good benchmark and the previous entries provide insight into what should be possible. These categories are not only useful for the competition but also help to describe what makes a 'good' settlement.

Adaptability

In real life, settlements adapt to their surroundings. The immediate surroundings and shape of the terrain will influence the layout of a settlement and the kinds of building used. For example, a more open and flat area would allow for larger buildings and for each house to have a lot of land. On the other hand, a dense mountainous region may only allow for smaller buildings which are more tightly packed together.

Settlements also adapt to their surrounding resources. It doesn't make sense for houses in a desert area to be built out of lumber, it would be very hard to gather the materials to build those buildings. The availability and abundance of local resources will influence any buildings which need to be built.

While settlements need to adapt to their surroundings, sometimes they alter the surroundings. Clearing away and flattening areas to have enough space to construct a building is essential to being able to create a settlement. The gathering of resources to build a settlement will also alter the landscape.

Functionality

Settlements in Minecraft serve a purpose, they provide the player with several vital necessities. In Minecraft, players have *hunger*, a value which decreases as a player moves around and decreases quicker the more active they are. Hunger allows a player to regenerate *health* (upon reaching zero, the player dies) that they may have lost (from falling or being attacked for example) and if hunger falls to zero, they begin to slowly lose health. Food allows players to regain hunger and one of the best ways of getting

food is to create purpose built farms. An ideal settlement would have a number of farms to be able to provide enough food for all the residents.

Settlements also protect the player from hostile *mobs*, enemy creatures which attack the player, most often spawning during the night time. Mobs cannot spawn in a well lit area so it is essential that settlements are well lit. Enclosed buildings also protect the player from any mobs which may wander in from outside the settlement. Although, a more protected settlement may have some kind of defence for this, walls for example.

(a) Skeleton (b) Spider (c) Creeper

Figure 2.1: A selection of hostile mobs from Minecraft

Functionality also describes how well a settlement is built. Road networks need to be logical and allow a player to reach different areas of the settlement and individual buildings should be accessible. If a river runs through a settlement, perhaps a bridge is required to enable access to other areas. A feature linked to the adaptability of the settlement.

Narrative

Settlements often tell a story of how they were built and why they were built, evoking a sense of history and narrative. Sometimes, this can come about naturally through a long construction and development process. Other times, this can be clearly seen if a settlement is built with a clear purpose in mind. A coastal trading port, for example.

In Minecraft, it can be a bit harder to achieve this as human-built settlements (in Minecraft) are designed with the final form in mind, not through a lengthy iterative process, like real-world settlements. However, iterative procedural generation, a bit like generative design, would yield settlements created as part of an adaptive process.

Aesthetics

Another important feature of a Minecraft settlement is the way it looks. Although subjective, things such as a consistent theme and style contribute to the overall aesthetic. Variety may also play a part in aesthetics. A settlement where every building looked exactly the same wouldn't be very interesting and may even be hard to navigate without distinguishing features.

2.1.2 Review of competition and results

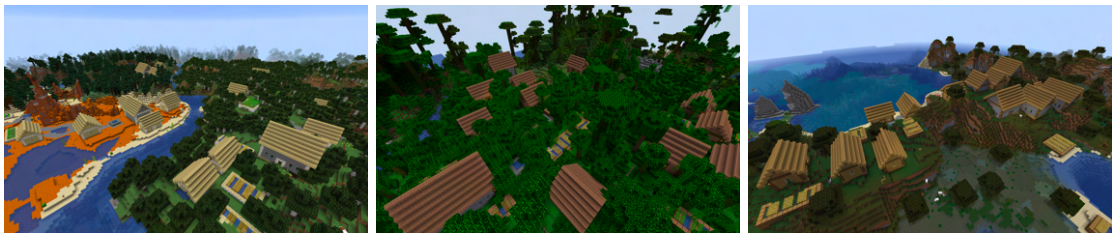
The results of the the first competition were published in a report [24]. There were four entries in 2018, each with their own unique method and style of settlement. Each generator was used on 3 Minecraft worlds. These worlds were specifically chosen for the competition and entrants weren't shown what they looked like beforehand. Judges then gave a score between 0 and 10, 10 representing a 'superhuman performance'. The highest entry that year scored an average of 4.38, with all others scoring 3.08 or lower. The organisers did acknowledge that the lack of quantitative analysis made the competition hard to judge but 7 out of the 8 judges independently agreed on the winner. The top three entries use an interesting array of methods and provide some helpful insight into possible generation methods. The fourth entry worked by simply levelling an area of terrain and placing some pre-described buildings. While this method works, it does not really align with the project aims of creating an interesting and adaptive generator.

1st place: Filip Skwarski

This algorithm continuously fills an area with structures until it can no longer find viable locations to place buildings. Possible buildings are chosen from two types of houses, plazas and farms. This is done by ranking a set of possible structures using three criteria:

- Elevation: terrain needs to be flat in order to place a building.
- Layout: Making building types fit with surrounding buildings.
- Distance: Ensuring buildings aren't position too far away.

Buildings are all symmetrical and always designed in the same way as others of the same type but size can vary. Buildings are then linked to their nearest neighbour by a road, generated with the A* path-finding algorithm [18], used to find the shortest route between two locations. Building materials are chosen based on local availability. While this generator shows some very good adaptability, it has issues with connectivity and variability. Some buildings are positioned in a way which results in them not being connected to other buildings and the way in which buildings are created results in little variation, only in size.

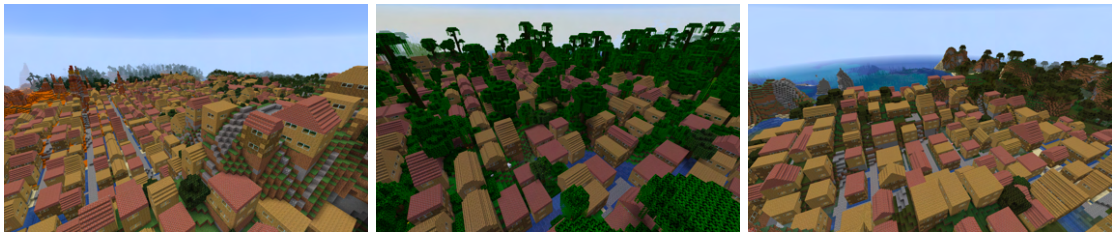


(a) (b) (c)

Figure 2.2: Results of the 1st place generator on the three competition maps

2nd place: Rafael Fritsch

Adapting to the environment was the main focus of this algorithm. To begin, the algorithm analyses the terrain to see which blocks are available as well as analysing the elevation of the terrain. The area is then divided into several subplots, each plot is then flattened to a surface level. Structures are then placed on this plot based on a fitness measure. Possible structures include houses, roads and farms. This entry results in dense settlements with slight variation in building style. However, this can lead to buildings being positioned in awkward places (the very edge of a cliff or over water, for example).



(a) (b) (c)

Figure 2.3: Results of the 2nd place generator on the three competition maps

3rd place: Adrian Brightmoore

This entry uses a 2-step process, operating mostly on a 2D representation of the world. The generator first collects the heightmap of the world, a grid which stores the height of the world's surface at those co-ordinates. A map of edges where two blocks have a height difference of 2 or more (a height which cannot be jumped by a player) is then created. Building locations are then chosen such that the area a building is placed in does not contain any edges (it is placed on even ground). Buildings were generated by a seperate building generator, previously made by the author [8].

This generator demonstrates a slightly more data-driven approach than previous entries. It also demonstrates slightly more interesting buildings but the way they are positioned can often result in awkward looking settlements that are disconnected.

(a) (b) (c)

Figure 2.4: Results of the 3rd place generator on the three competition maps

Open Challenges

After the review of the first year of entries, the organisers of the competition specified a few areas that they wished to see improvement on as the competition progressed. This offered some clear insight into further directions to take this project and possible new ideas to experiment with. They offer some key ideas to do with building variety, adaptation to water (bridges over rivers for instance), data-driven and simulationist approaches.

2.2 Generating Settlements

2.2.1 Hierarchical Generation

One of the main focuses of the competition is the idea of *holistic procedurally generated content*. The idea that while it can be quite easy to generate individual elements such as trees, buildings or even whole areas of terrain, it can be a lot harder to generate more complicated structures and combine several generated elements into a coherent result. Generating a whole settlement for instance, requires generating some kind of layout, buildings and roads that connect these buildings. All tasks that can be done separately but need to be brought together to create a settlement which makes sense.

A simple approach to this may just be to do each type of generation individually. One naive example might first generate a series of buildings, then generate a layout for these buildings and just position them accordingly. But as a result, this example has no *emergent properties*. The buildings are generated irrespective of surroundings and the layout has been designed irrespective of the buildings being used in the settlement.

Instead, one suitable approach would be to use some kind of influenced or hierarchical structure with distinct stages [25]. For example, the location of a settlement (with some parameters or features) may be influenced by the terrain. Those parameters then influence the type of building to be built and where to put them. In turn, these buildings then influence the layout of the roads around the settlement. In reality, this could be thought of as a slightly more circular process. The settlement is shaped by the terrain but may also alter the terrain, buildings influence where roads go but certain buildings may be positioned to be connected to an already paved road and so on.

This problem of coherently generating several interlinked elements could be addressed by "simulating the process that actually generates an artifact, rather than just trying to generate the artifact outright" [23]. A very similar concept to agent-based modelling, allowing the agents to go through the process of actually building a settlement for themselves.

However, generated elements sometimes suffer from the *Kaleidoscope effect* [5]. Despite generated elements being distinctly unique, there is an overall style or similarity that makes these elements less interesting, despite the number of possible outcomes, making it obvious they were generated using the same method/generator. This may prove beneficial when generating any particular settlement and ensuring that buildings follow a theme or pattern but when generating a variety of settlements in various settings and environments, may show the limitations of a particular generator.

2.2.2 Generating Buildings

Generating buildings for a settlement is a challenging task. Many of the previous entries to the competition make use of a template system. They define a small selection of pre-fabricated buildings and position them in suitable locations (as seen in Figures 2.2 and 2.3). It is much less common to see buildings which are the result of their own generator system (as seen in Figue 2.4). Using these templates can sometimes

result in more *aesthetic* settlements but they very quickly become less interesting and don't represent a fully generated system.

In cases where fully generated buildings have been used (rather than a pre-determined template), the way in which they are positioned can sometimes be a little awkward. Figure 2.4c shows this quite clearly, with a tall building standing on a pillar of stone raised from the ocean. While the building itself is interesting, the location and way in which it is positioned shows a lack of adaptability and awareness of the surroundings.

Organic Building Generation

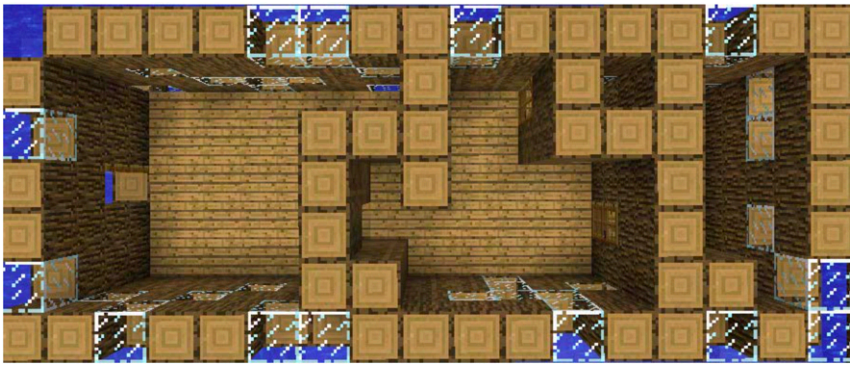


Figure 2.5: An example of an organic building layout (as seen from above), credit: [11]

One proposed method for generating buildings makes use of an organic growth algorithm to first define a floor-plan and then Cellular Automata to generate external walls [11].

To start, the rectangular footprint of the building to be generated is defined. This algorithm uses a regular grid which is easy to visualise in Minecraft when comparing a cell on the grid to one *block*. The number of rooms to be generated is defined as $\sqrt[3]{x}$ where x is the total area of the footprint. This is so that the size of a building directly affects the number of rooms inside it, whilst placing a reasonable number of rooms. Each room is given a random starting location and initially occupies a 2x2 area. Each iteration, rooms take it in turns to grow by one cell. Empty cells adjacent to other rooms cannot be considered as they are reserved for wall spaces. The room order is shuffled each iteration to prevent one room from consistently growing larger than others. Doors are then positioned so that adjacent rooms are connected and one door is placed somewhere along the external walls of the house.

Once the floor-plan is generated, the exterior walls need to be added. These walls are represented by a matrix of the same dimensions as the wall. Solid wall blocks are represented with a 0 and windows represented by a 1. To start, the wall is randomly allocated such that only 25% of the blocks are glass. For each generation, cells sum the surrounding cells and their own value to obtain a value between 0 and 5. If the the sum equals 2 or 3, the block becomes (or remains) a glass block, otherwise the block is made to be solid. This runs for 10 generations and the resulting layout is used to create the wall of the building. Figure 2.5 shows an example of a generated layout.

This results in natural feeling buildings with interesting layouts. However, the way in which rooms are generated and grown means that much larger buildings often end up becoming overly complex and disorientating. The authors only use rectangular buildings in their demonstrations but explain that it should be possible to use an alternative shape with a bit of extra work.

Generating Non-Rectangular Buildings

Generating buildings which aren't symmetrical results in slightly more complex and interesting designs but is also a more difficult task. A project inspired by the GDMC describes a process of creating layouts for buildings using two separate rectangles [10].

To start, a rectangular building plot is defined as the largest area that the building can occupy. A main square is randomly created within the plot based on some parameters such as minimum building dimensions. In some instances, only the main square is used and a rectangular building is generated. However, a secondary square can also be added to the layout. Depending on the way in which the second rectangle is positioned, the two rectangles are adjusted so that the area they describe can be defined by two rectangles without any overlapping areas. This results in either L-shaped or T-shaped houses.

Once the shape of the foundation has been defined, the number of floors and height of each floor is randomly selected. A 2D array is created and filled with values representing the shape of the house as seen from above. Various values are used to describe outer edges and corners of the building.

The interior of each building is then divided into a series of rooms and interior furniture positioned around the edges of each room. Roofs are built over each rectangle separately, using a function based on the distance from the closest wall for how high each block of the roof should be placed.

2.3 Simulating Agents

A very important and novel aspect of this project is the use of agent simulations. Previous competition entries have yet to use an agent based approach to simulating a group of settlers constructing a settlement. To do this, a variety of techniques need to be used in order to create a believable simulation which creates interesting results. In particular, looking into methods of path-finding and collaborative exploration environments.

2.3.1 Path-Finding and the A* Algorithm

Path-finding is used to find the shortest path between two points. This is essential when using agent-based simulations as agents need to be able to travel from point to point and do so in an effective manner. In the case of Minecraft this is a path between two blocks, represented by 3D co-ordinates. One of the most common algorithms used for this is the A* algorithm [18]. Many shortest-path and search algorithms such as depth/breadth first search and Dijkstra's algorithm [9] have existed for a long time but A* has quickly become one of the most commonly used in video games [7].

A* Algorithm Overview

A* first begins by representing some search area as a grid where a start and end location are both on the grid. Empty cells represent possible locations for an agent to move into and some cells may contain obstacles or walls preventing movement. The world of Minecraft is easy to represent in a 2D Matrix due to the way it is already divided into blocks. In some cases, the search area does not have to be an even grid, this could be some kind of connected graph with nodes and edges rather than a uniformly divided area (a notion which proves useful when making optimisations). An example grid can be seen in Figure 2.6. The search begins at the starting cell (the green square in the example), checking adjacent cells and moving outwards towards the target cell (the red square in the example).

To begin, the starting cell is added to the *open list*. This list contains all the 'visited' cells which may form part of the final path. All surrounding cells are observed in turn. They are each added to the open list if they do not contain an obstacle. In this instance, all 8 (including diagonal neighbours) are compared. Some versions of the algorithm opt to only allow for movement in the 4 cardinal directions. Each cell added to the open list also has a *parent*, the cell which was 'travelled from'. In this case, the starting cell. Once all neighbouring cells have been considered, the starting cell is dropped from the open list and set to *closed*.

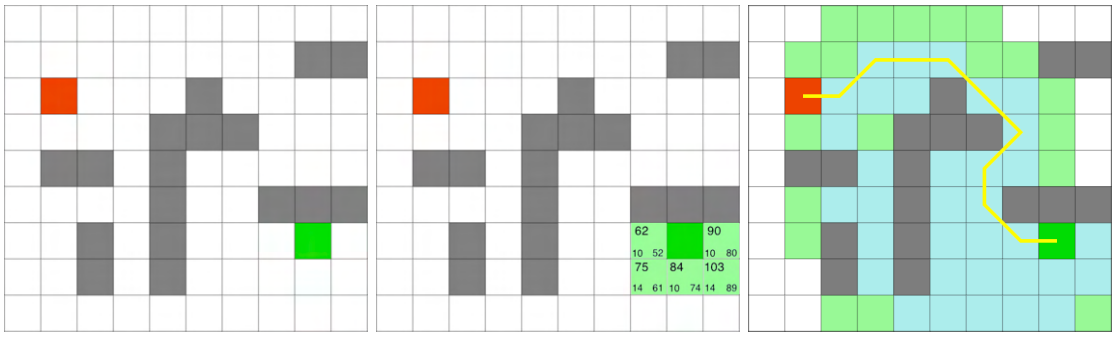
This process continues to repeat for each cell on the open list until the target is reached. However, doing this exhaustively for all cells on the open list would result in a very inefficient algorithm (in fact, one very similar to Dijkstra's algorithm). Instead, cells are given an *F-cost*, the lower the cost, the better the path. Cells with the lowest F-cost are removed from the open list first.

F-Cost

The F-cost of a cell is calculated using the equation:

$$F = G + H$$

- G represents the cost of moving from the starting cell to the current cell, having followed the generated path to get there. In this example, a G-cost of 10 is added for a horizontal or vertical move and a cost of 14 is added for a diagonal move. This roughly represents the ratio of moving diagonally but some simulations use the same cost for both.
- H is the cost of travelling from the current square to the target square. Also known as the *heuristic*, it is a best guess as to how far is needed to travel before getting to the target. H can be calculated



(a) Starting grid layout (b) First iteration and cost values (c) Finished path

Figure 2.6: A* Algorithm example - The green cell is the starting location, red cell is the target location and grey cells represent walls/obstacles

in a variety of ways but in this example it is just the Euclidean distance from the current cell to the target cell. The example shown in figure 2.6b shows values which have not been square rooted, this is an optimisation to avoid expensive square root operations but the result of the algorithm is the same.

As the search continues, adjacent cells to the cell currently being examined may already be on the open list. In this case, a new F-cost is calculated, if it is lower than the one currently assigned to the adjacent cell, it is updated and the parent is changed to the current cell. Figure 2.6b shows the first iterations of the algorithm. The F-cost of each cell can be seen in the top left, G and H costs are in the bottom left and right respectively.

Once the target cell is reached, a path can be traced in reverse by using the parent cells assigned to each cell along the path. Figure 2.6c shows the results of the algorithm. The path is shown as a yellow line with blue cells showing closed cells and green cells representing those still left on the open list.

2.3.2 Area Exploration and Mapping

For agents to be able to make decisions about where to place buildings, they need to gather information about the world through exploration. This is a problem often found in swarm robotics used in applications such as warehouse organisation like the systems found in Amazon warehouses or exploring a cave system [29]. Swarm robotics is a way of using several connected robots or agents to achieve a common goal, almost like a real-world example of agent based modelling. In this case, the exploration of some unknown area or terrain.

Atlas is a method of exploration which makes use of a sparse swarm [1]. Atlas aims to provide an efficient and fast method of exploring an unknown area with a minimal number of agents. Atlas also aims to ensure that all accessible areas are explored fully, unlike other methods like random walk which aren't exhaustive [16]. Figure 2.7 shows a heat-map of agent visits, with the starting position shown in red. Random walk clearly shows that the far end of the layout is not explored.

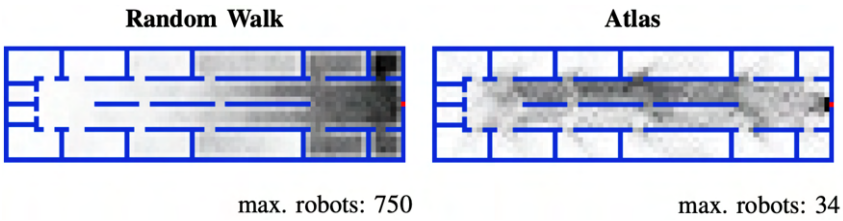


Figure 2.7: Heat-map of agent visits to each cell during exploration on an example layout, credit: [1]

Atlas

In Atlas, robots/agents are organised by a central controller which oversees the exploration and holds all of the data required. The algorithm focuses on exploration through the use of *frontier cells*, cells which

are at the edge of the explored area. Atlas also shares similarities to the A* algorithm, it makes use of an open list to make decisions about where to explore next. Atlas assumes that agents can observe adjacent cells and can communicate with the central controller at all times.

At the start of the simulation, the cells around some starting location are added to the open list. A cell is described as 'open' if it has an unobserved cell with its neighbourhood of 8 adjoining cells. For every available agent, the controller selects the cell on the open list which is closest to the starting position. Atlas uses topological distance (number of steps along a shortest path) to measure how far away a cell is. The controller then finds the agent closest to the selected frontier cell and sends it to explore. Upon arriving, the cell is marked as closed and the agent observes the surrounding neighbourhood. Subsequently adding any newly observed cells with unobserved neighbours to the open list.

This results in a systematic exploration method which slowly expands the explored area in all directions. This even works in the extreme case of a single agent, which just circles the original starting point, drawing larger and larger circles.

Chapter 3

Implementation

Most of the work for this project went into the development of a generator capable of generating settlements on any given Minecraft world. The generator was written in Python and is the result of several weeks spent using *iterative* development. This chapter will explore some of the decisions that went into creating the generator as well discussing the methods used in the generator.

3.1 Design and Process

3.1.1 Iterative Development

Iterative development was used throughout the project to ensure a good structure and carefully planned approach, this style of development follows a cyclical nature. Each stage of a project is planned, implemented, tested and then reviewed. With each subsequent stage aiming to build upon and improve the previous development cycle. Figure 3.1 shows an example of a cycle.

During the process, Git was used for version control. This made it easy to divide tasks into discrete 'issues', used to represent features of the generator, which could each be handled on a separate branch. This helped to make the development process more organised and manageable. Each issue/feature became its own iterative cycle, an initial planning stage, implementation of the feature and then testing before moving onto the next feature.

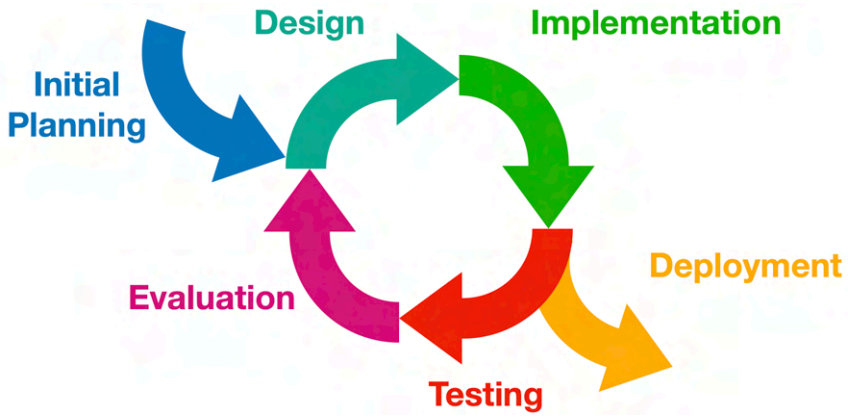


Figure 3.1: An iterative development cycle

3.1.2 Hierarchical Generator Design

The generator is mostly split into two stages, a data gathering stage and a construction stage. Each stage has its own kind of agent. The first data gathering stage is handled by a group of 'explorer' agents, tasked with surveying the land and gathering information as they travel. A second set of 'builder' agents are then responsible for constructing the settlement. Figure 3.2 shows a brief overview of the generator, which will be discussed in more detail throughout this chapter. This shows a hierarchical approach as discussed in the previous chapter, enabling more complex and believable structures to be generated.

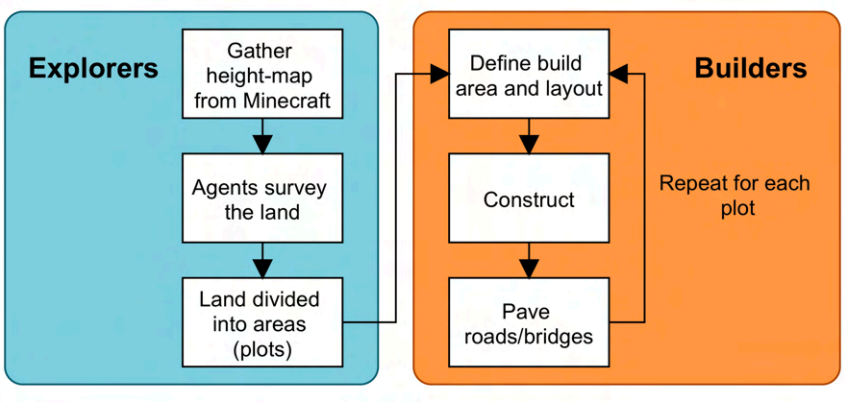


Figure 3.2: A simple overview of the hierarchical generator

3.2 Exploration and Data gathering

3.2.1 Height data

The first stage of generation consists of a series of data gathering steps. The very first step is to collect a height-map of the area a settlement is to be generated in (defined by two corner coordinates). This data is required to inform the movement of the agents as they explore the world. It would be possible to gather it in real-time but Minecraft already has a data-structure which can be easily accessed which contains height data for the world. This saves a lot of computation time and the way in which this data is gathered does not affect the outcome of the generator. The competition has a soft 10 minute limit for how long a generator can be run for. While the competition is not the only aim of this project, it seems sensible to take computation time and complexity into account.

Figure 3.3 shows an example area and their associated height-maps. These are simply stored in a 2D matrix, with integer values representing the y coordinates (height) of the surface level (solid blocks only, ignoring things such as leaves) blocks at each location.

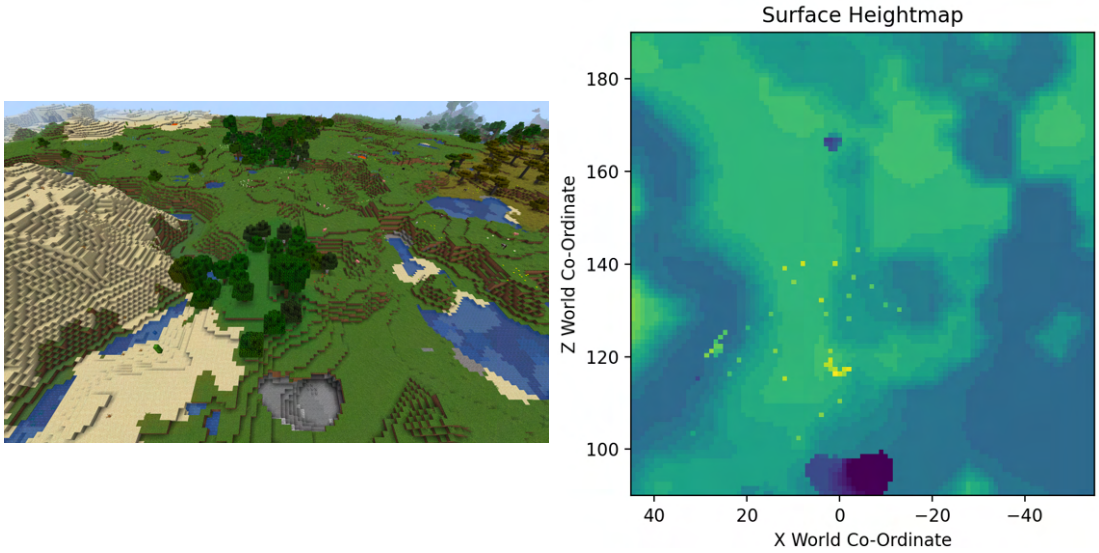


Figure 3.3: An example area of a Minecraft world and the associated height-map

3.2.2 Agent Exploration

Once the initial height data has been gathered, agents can begin to explore the surrounding areas to gather even more data. This is done using adaptations of several algorithms previously discussed in chapter 2.

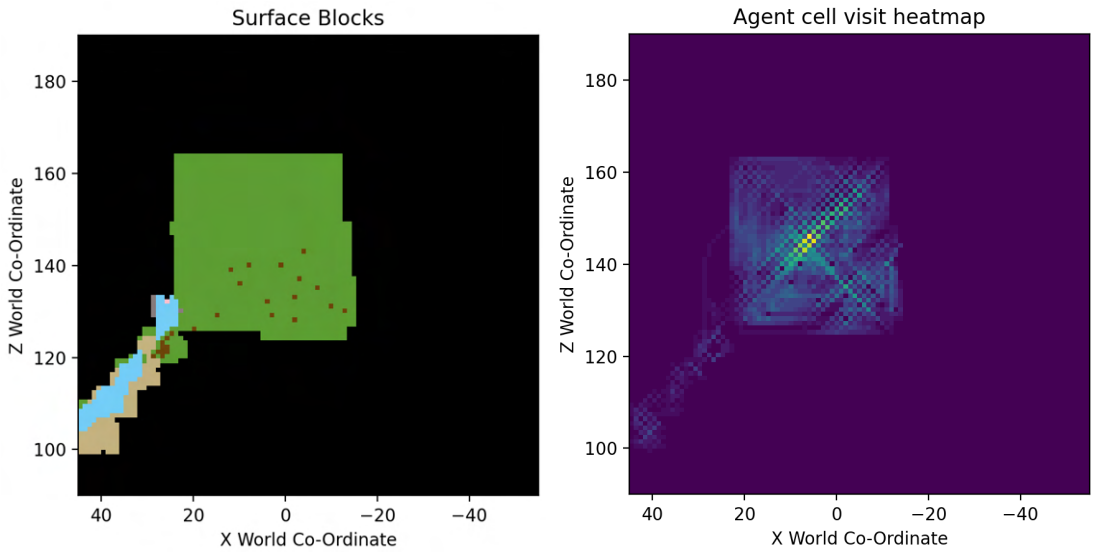
Path-Finding

Before being able to explore, agents need to be able to move around. In Minecraft, a player (and computer controlled *villagers*) can move following certain rules. They can move backwards/forwards and sideways as much as they like (or diagonally), can jump (up to the height of one block) and can swim. They can also descend differences in block height but if they fall too far they take damage.

These explorer agents are simulated in a similar fashion to a player, although their position is expressed as occupying a certain cell on the map (represented as whole block in Minecraft), rather than a player who's position is described as a finite point in 3D space (not restricted to moving distances of a whole block at a time). Agents can travel a whole block to any of their 8 neighbouring blocks, so long as the difference in height is not greater than one. Explorer agents also have the ability to 'swim' over bodies of water.

To move around, agents use a slightly altered version of the A* algorithm. In this adaptation, a move in any of the 8 directions simply carries a G-cost of 1 (meaning travelling diagonally carries the same cost moving forwards/backwards or sideways). Obstacles are also handled in a slightly different way. Instead of checking if the cell the agent wishes to move to contains an obstacle, the validity of the move is verified by comparing the difference in height between the two locations. An agent can be given any location within the defined settlement area and (if possible) will plot a path towards that target location.

Exploration



(a) (b)

Figure 3.4: Exploring an example area in Minecraft. The colours of the map 3.4a approximate the types of blocks discovered in an area. Heat-map 3.4b shows where agents travelled during the simulation (results were gathered after 2000 iterations).

Agent exploration is based on Atlas, a method used in sparse swarm robotics [1]. A central controller is responsible for orchestrating a group of agents and begins by selecting a starting location. This is usually chosen as the centre of the defined settlement area [1] but may be moved if this proves to be an unsuitable location (no valid moves from the starting location).

The default generator uses 4 agents, each with a starting position surrounding the central starting point of exploration. Agents then explore outwards following similar principles as Atlas, placing newly observed cells on an open list and then being told by the controller which cell to travel to next. In Atlas, this would simply be the closest cell to the starting position currently found in the open list. While this produces a working solution which systematically explores an area, the exploration does not adapt to the already explored region.

Instead, the method used in the generator employs a similar system to the cost system used in the A* algorithm. A cell c has a base cost, G which is calculated in the same way as in A*, the number

[1] The settlement area is a rectangular area provided to the simulation for a settlement to be built in.

of moves required to travel from the starting cell (the cell chosen by the controller at the beginning of the simulation) to c. However, where this method differs is the use of a 'modifier', a value which alters the cost for a newly added cell. When an agent arrives at a cell, the modifier is increased for every 'interesting' feature they observe in the surroundings. These include things such as trees, water and valuable resources. When these newly observed cells (the 8 neighbouring cells) are added to the open list, the modifier is subtracted from their cost G, resulting in a new cost, F. This results in cells which have their cost decreased the more interesting their surroundings are, resulting in areas of interest or rich in valuable resources to be prioritised during exploration. Instead of recalculating the G cost every time an agent arrives at a cell, they use the F cost of the cell they just travelled from and add 1. This means that the modifiers of previous cells in the sequence propagate through as an agent explores, promoting the exploration of interesting areas first before returning to other areas. Figure 3.4 shows a map of the explored area from Figure 3.3, clearly showing how agents started to move towards wooded areas and areas near water.

Observing surroundings

Upon arriving at a frontier cell (at the edge of the currently explored area), the agent will observe its surroundings and gather information. To begin, they survey any of the unobserved 8 neighbouring cells and record the block type at that location. The central controller then updates a map of the world accordingly. Any blocks of particular interest such as wood or water are also made note of, this will not only influence where agents explore but also influences where buildings are built and what they are built from. These newly observed cells are then added to the open list if they have their own unobserved neighbours (and are accessible following a valid move). Agents are also responsible for adding cells to a *plot*.

Plots

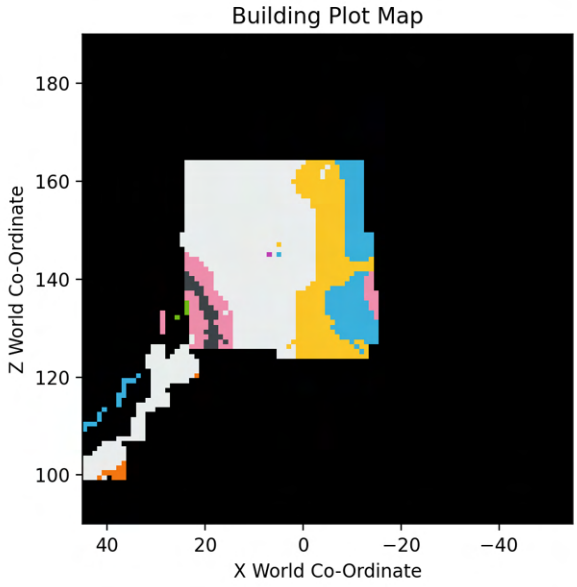


Figure 3.5: An example world where observed cells have been divided into plots. The black area is the unexplored area and each coloured area represents a new plot. Colours cycle through a possible list of 6 (colours of wool in Minecraft) so some plots may be shown as the same colour even although they are different plots.

Plots represent a collection of cells at a similar height level, these are later used to inform building placement and type. A plot stores a list of associated cells and a height value. A cell may be added to a plot if the height of that cell is close enough to the height of the plot (by default this is one more or one less than the height of the plot).

When an agent gets to a target cell and does any observations, they also look at the plots that any neighbouring cells have been assigned to. If any of the cells have been assigned to an existing plot, that

plot is added to a list. Once all neighbouring cells have been observed, the list is checked for a suitable plot to add the current cell to. If one or more is found, the cell is added to the largest plot. Otherwise, a new plot is created at the height of the cell.

Nearby plots can also be merged. Due to the order in which cells are explored, it is possible for two neighbouring cells of the same height to be added to different plots. In this case, it makes sense for both plots to be merged. After the observation step, neighbouring plots are compared and merged if they can form one larger valid plot. This results in the world being divided into areas of land, based on height, as can be seen in Figure 3.5. The resources found in one particular plot influence the *palette* used in that area to construct buildings in the plot.

Palettes

Palettes describe what kind of block should be used to construct buildings in a given area. This is done by defining what kind of block should be used for certain parts of a building (foundation, walls and trim, for example). Palettes are mainly influenced by the surface-level blocks and what kind of trees can be found. For example, in a desert area, the ground is mostly covered in sand, meaning houses can easily be built out of materials such as sandstone. The type and number of trees in an area also greatly influences the materials used. The most common type of tree is used to influence what kind of wood should be used but how many trees are available also influences how much wood should be used during construction. If there are very few trees then houses should use wood sparingly, maybe only as a decorative element. Whereas, houses in dense wooded areas can be built completely out of wood due to the abundance of the material.

3.3 Building the Settlement

Once the settlement area has been explored, buildings can be constructed. These buildings should be constructed in a way which is influenced by the data gathered but can also have an effect on the local area. To begin, plots are given a score based on size, available materials and agent traffic. This ensures that the largest plots with the best materials (wood, water, ores, etc.) and a high frequency of agent visits are built on first, these represent the most important building locations for a settlement. A build area is defined as the largest rectangular area completely contained within a plot (although not necessarily the shape of the building to be generated). Before a building can be placed, this area is flattened to a consistent height and any obstacles such as trees are removed. Builder agents then continue to build in a single plot until no more reasonably sized (based on a minimum building size constraint) build areas can be placed. They then move onto the next plot and continue construction.

3.3.1 Generating buildings

Layouts

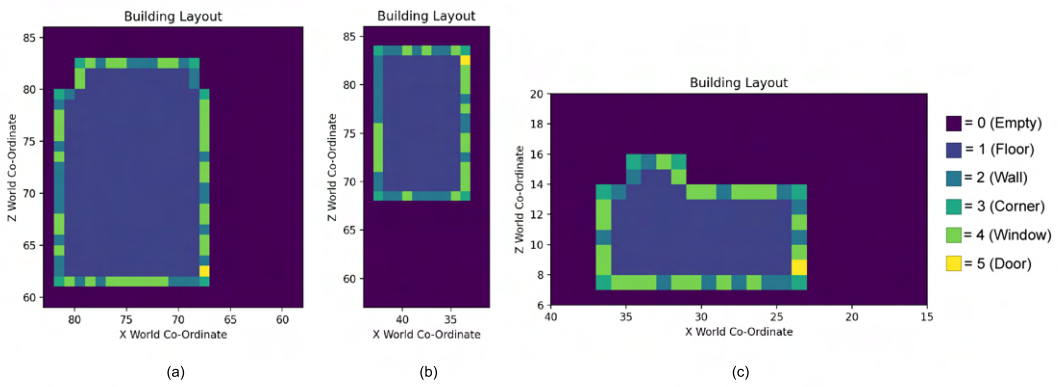


Figure 3.6: Examples of generated building layouts. The colours represent how the different have been labelled (walls, corners, windows and doors).

The first step to generating a building is to generate a layout/floor-plan. This is done by using a similar method to the non-rectangular layout described in chapter 2. Two random rectangles (defined by a corner, width and length) are created so that they fit in the rectangular build area assigned to the building being generated. If the rectangles do not overlap, the largest of the two is labelled as a building and the smaller rectangle is labelled as a farmland area. If the rectangles overlap, they are combined to form an L or T-shaped layout. This is because both of these shapes can be described by two rectangles which don't overlap (regardless of the two rectangles used to generate the layout initially). The largest rectangle becomes the main portion of the building and the smaller area becomes an 'extension'.

The layout for the building is then formalised in a 2D matrix. To begin, all cells within the construction area are set to 1. The matrix is then iterated over again, this time, each cell counts how many neighbouring cells (in the 4 cardinal directions) have a value greater than 0. If the number of neighbours is 4, this cell is within the walls of the building and can be left as a 1. Cells with 2 or 3 neighbours are considered to be exterior walls, these are all assigned a value of 2 (or 3 for corners of a building). Walls (but not corners) then have a random chance of becoming windows, being assigned a new value of 4. A door is then randomly assigned to a section of outer wall and given a value of 5. Figure 3.6 shows some examples of generated building layouts.

Constructing Buildings

Once the layout has been generated, it is relatively simple to construct the generated building. The builder agent randomly selects the number of floors and how high each floor should floor should be. Once the main building has been built, the roof is built on top. The roof is generated separately for each rectangle (the main and extension rectangles). The roof is generated so that the shorter sides of the rectangle form the gable ends. Figure 3.7 shows some examples of generated buildings, with the layouts pictured in 3.6.

(a) (b) (c)

Figure 3.7: Examples of generated building types

Farms

Farms are an essential part of any settlement in Minecraft. During the random generation of the two rectangles used to create the layout for a building, it is possible for two rectangles that don't overlap to be generated. In this case, the largest of the two is used as usual to generate a building and the other rectangle is used to make a farm. Farms consist of farmland type blocks, with a crop planted on them and occasional water blocks, to prevent the drying out of the farmland.

3.3.2 Paving villages

Paths around a village help to make it easier to navigate, providing an easy surface to travel on and directing the player to other buildings. The generator aims to connect all buildings together using an even further adapted version of the A* algorithm. This algorithm works like the traditional A* algorithm used to handle the path-finding of agents but the way in which costs are calculated is slightly different.

During exploration, all moves had an added G-cost of 1. While agents are creating paths through the village, a move that would take an agent over a body of water has a cost of 2 instead. This means that while it is possible for a path to cross a body of water (resulting in a bridge), there is an added cost to this (namely, building the bridge). Therefore, paths over water are only considered if they result in a considerably shorter path from one building to another. Agents also promote the reuse of paved paths by reducing the cost of a move if it is over a pathway. They also prioritise the use of well established

pathways through a settlement, reducing the cost even further if a pathway is more built-up, even if it results in a slightly longer path.

Upgrading pathways

In order to create a sense of narrative and influence from the agent-based approach of the generator, paths are not simply paved right away. Once a path is established from one building to another, an agent travels along that path, constructing the path as it travels. By default, paths are assumed to be 3 blocks wide, although in some cases the shape of the terrain can result in narrower paths (walking through a gap between two buildings, for example). At every stage along the path, the agent has a random chance of 'upgrading' each block. The material used to pave the path reflects how travelled that particular section of path is. Paths begin as patchy gravel trails and are slowly upgraded to properly developed paths before being converted to paved stone roads (if there is enough traffic). This is a good example of an emergent property where the agent-based focus of the generator has had a clear influence on the generated settlement. Figure 3.9 shows a series of paths and bridges throughout a generated settlement and the heat-map of agent movements resulting in this layout.

(a) An example of a less developed, patchy gravel path

(b) A more developed path with only some gravel

(c) Bridge connecting both sides of a river

Figure 3.8: Generated paths throughout a settlement, with varying levels of paving

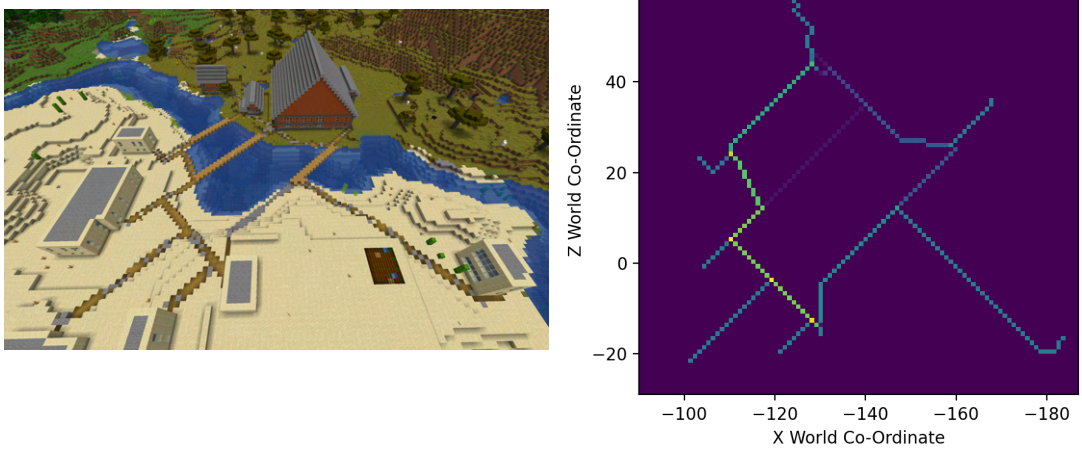


Figure 3.9: A generated settlement with paths viewed from above and the associated path heat-map

Chapter 4

Results and Evaluation

This chapter aims to present some of the results (settlements made by the generator) as well as evaluating the effectiveness and value of such a generator. This can be done by not only comparing the generator to previous entries to the competition (which use different techniques), but also evaluating the various parameters of the generator to achieve different results. This provides some key insight into the *emergent* properties of the generator and how they are influenced by the *elementary* properties.

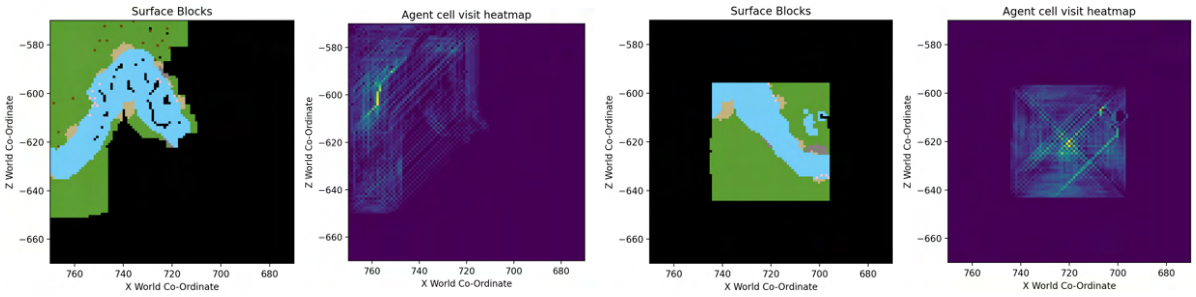
4.1 Experimentation and Viability of the Approach

While one of the aims of the project was to produce a generator which could be entered into the competition, the main goal of the project was to explore the suitability of the methods used (such as agent-based modelling) for not only settlement generation but as a way of generating various types of content. This section aims to explore how certain elementary properties influence settlement generation and what could be possible with a similar generator for other content.

4.1.1 Agent travel methods

It is possible to influence how the agents explore the world by changing the way cells are scored during exploration. For this, two methods will be compared. The first being the method presented in the previous chapter, scoring cells based on local resources to promote exploration of valuable areas. The second method will be to use the more basic scoring method proposed by Atlas, simply using distance to the central starting point. Testing was run on the same area of terrain, pictured in Figure 4.1. Figure 4.2 shows the area explored and associated heat-map for the two methods. In both cases, a maximum of 4000 iterations was used.

Figure 4.1: The area used to demonstrate different methods of exploration



(a) (b) (c) (d)

Figure 4.2: Exploring the area shown in Figure4.1 using two different methods. (a) and (b) show the area explored when agents are influenced by score, (c) and (d) show the area explored when agents are only influenced by distance.

(a) (b)

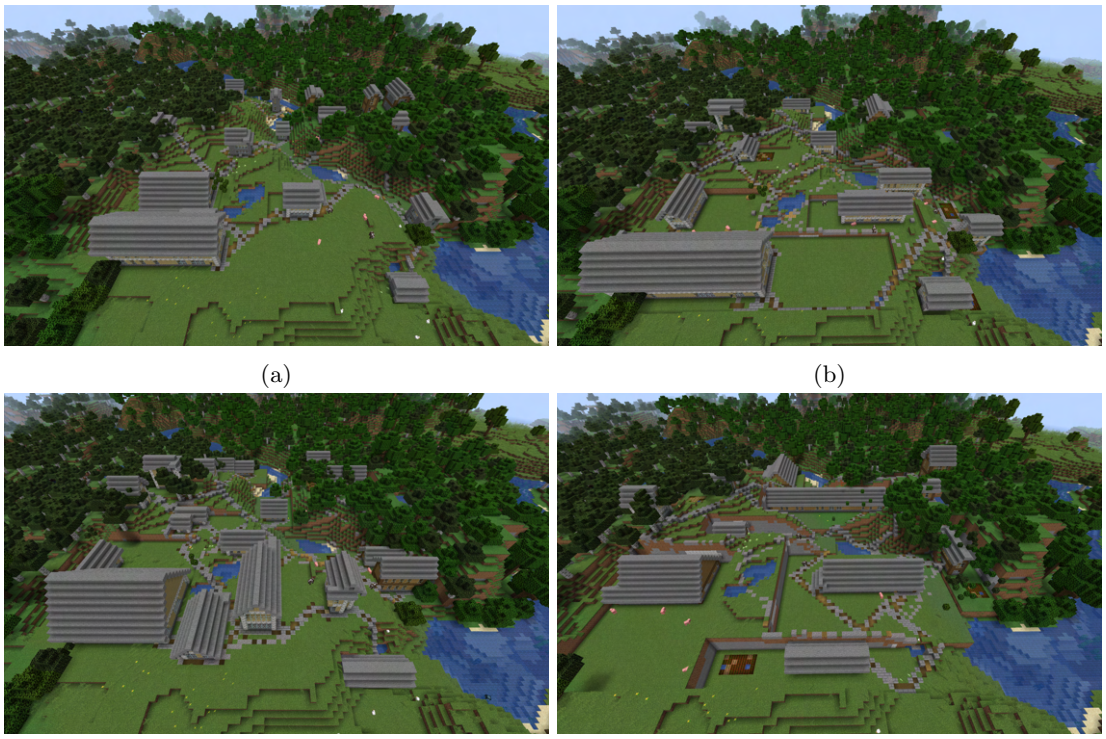
Figure 4.3: Resulting generated villages using two different exploration methods. (a) was generated using the scoring approach and (b) was generated using only distance to influence exploration

Both generated settlements can be seen in Figure 4.3 with both showing some similarities but also some key differences. The two explored areas are very different in shape, with agents in 4.2b having explored more of the area around the river and forest whereas agents in 4.2d have only explored a uniform square at the centre of the area. The first settlement (4.3a) shows a more spaced out settlement, with more buildings positioned on the side of the river where trees can be found. This is because the agents have prioritised this area during exploration as it contained resources deemed to be valuable. The second village (4.3b) results in a more tightly packed group of buildings but none of these have been constructed with anything other than stone (easily found just bellow the surface) as no valuable materials were found during exploration.

4.1.2 Plot height difference

As agents explore, they create *plots*, dividing the terrain into areas of land which all share a similar height. The difference in height permitted in a plot is governed by a value which can be changed, 'height difference'. Altering this value allows the generator to generate a greater variety of settlements, demonstrating that the methods used throughout this project result in generators that work well, producing interesting results. Figure 4.4 shows a collection of settlements, constructed with different values of 'height difference'. Figure 4.5 how the plots of the settlement area were divided during simulation.

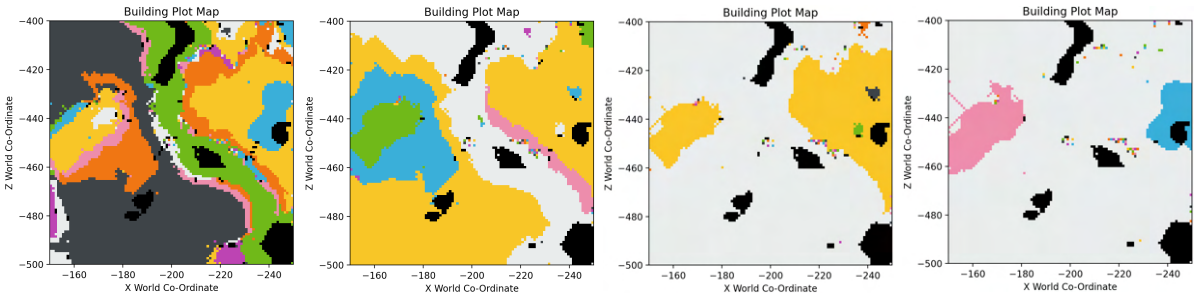
These settlements show that as the height difference of a plot increases, plot size tends to increase, with one plot engulfing the large majority of the settlement area. All 4 generated settlements result in valid settlements but have different emergent properties. In particular, the higher the height difference, the more the land around the settlement is altered. When using height difference values of 5 and 10, the plots contain such differences in height that are lost when areas are flattened to place buildings. This manifests itself as large, flattened rectangular areas which are unsightly and show that settlements are progressively less influenced by the surrounding terrain, as the height difference increases.



(a) (b)

(c) (d)

Figure 4.4: Four different settlements created with different 'height difference' values, higher values mean plots can contain cells at greater height differences. Whereas, a value of 0 means plots can only contain cells at the same height. Settlements were generated with height difference values of 1, 3, 5, and 10, respectively.



(a) (b) (c) (d)

Figure 4.5: Maps showing how the plots divide the settlement area for the settlements shown in Figure 4.4. Note how plots tend to become larger and have been merged together due to a larger tolerance in height difference, clearly seen in 4.5d where the white plot engulfs most of the area.

4.1.3 Agent Based Modelling for Generating Content

The main aim of this project was to explore alternative methods (namely, agent based modelling) for creating generated content. The idea being that, some elements are hard to generate using traditional methods since they rely on being built up over time or are a product of many individual interactions, settlements being used as a key example. This project has clearly shown that these methods do indeed work for generating settlements in Minecraft. The settlements produced are interesting, unique and are influenced by the nuanced agent based interactions that produced them with emergent properties, which are not as easily produced by traditional methods. Since these concepts can be applied to generating settlements in Minecraft, it isn't too much of a stretch to imagine these methods could be used in other areas such as generating content for other games or for CGI backdrops in films.

There are however, some drawbacks. This method is computationally intensive due to the concurrent simulation of several agents. This results in a long processing time for simulations. While the efficiency

could be improved in places, this technique may not prove suitable for real-time construction of elements. In particular, the agent based exploration, which takes up approximately 75% of the computation time. Watching the construction happen in real time [1] is actually quite entertaining but may not offer much benefit as larger settlements can take over 5 minutes to construct.

4.2 Generator Evaluation

This section aims to evaluate and compare the effectiveness of the generator using the 'default' parameters. These parameters are influenced by the exploration done in the previous section and these results represent settlements which are believed to be good examples for entry into the Generative Design in Minecraft Competition. Since these results aim to represent a good competition entry, it makes sense to evaluate the generator against the four original criteria of the competition, discussed previously in chapter 2.

4.2.1 Adaptability

Building Position and Size

The settlements created by the generator clearly show adaptation to the surrounding terrain through a variety of features. Without even looking at the design of the buildings themselves, it is clear that they have been positioned in a reasonable fashion and that their positioning and size is a direct result of observing the terrain. Buildings are positioned in open areas of land, usually where important resources such as trees and water can be found. The size of buildings is also heavily impacted by surrounding terrain, with larger flat areas resulting in much larger buildings whilst uneven areas tend to have smaller and more frequent buildings. Buildings are given adequate space such that settlements feel neither overcrowded or empty. Building positions also influence the landscape around them, reasonably flattening areas of terrain to facilitate construction without simply levelling off a whole area for a settlement to be built on.

Figure 4.6: A generated settlement showing buildings of various shape and size.

Use of Local Materials

Local materials also heavily influence the way in which buildings are made by altering their appearance. During exploration, agents make a note of local materials for an area of land, these influence the blocks which are used in a *palette* for a building. In examples of previous generators, the type of trees in an area would influence the colour of the wooden blocks used in the building templates. However, this influence is not simply a binary decision. A single tree being present on a building plot should not necessarily result in a wooden house. It makes sense that wood be used to build some portions of the house but it is not enough to make a fully wooden house. This project achieves this by gradually making more elements

[1] Time-lapse of settlement being constructed on an island: https://www.youtube.com/watch?v=zrfcVHVUqxc

of a house wooden as more and more wood is gathered to construct it. This results in a spectrum (of sorts) of possible building styles, as can be seen in Figure 4.7.

(a) (b) (c)

Figure 4.7: Various generated buildings with varying amounts of wood used during construction.

4.2.2 Functionality

Road Networks

Road networks are one of the main functionalities present in the generated settlements. The roads allow for easy navigation around a settlement, in some cases allowing access to otherwise unreachable or difficult to reach areas (over a river for example). Generated road networks are created in a way such that all buildings are connected to each other, with a network that aims to balance accessibility and over-paving (generating a road network which is too complex or results in paving a whole area).

The complexity of these networks and bridges generated as a part of these networks also demonstrate elements of adaptability. Bridges are constructed due to the necessity of crossing a body of water to reach other buildings, allowing settlements to adapt even further to the surrounding terrain.

Safety and Buildings

Settlements in Minecraft are also designed to provide safety. The buildings generated are completely enclosed and would protect a player from hostile mobs. However, the settlements generated don't currently have any lighting (such as torches) so they do not prevent these creatures from appearing [2].

While the buildings are fully enclosed and have different floors, the interiors are very basic. Generated buildings have no interior planning, such as different rooms or furniture. Something which can be seen in some of the previous entries to the competition [10].

Food

Settlements also need to provide the players with food. Generated settlements do contain farms and these can successfully grow crops. However, farms are perhaps less frequent than the size of a generated settlement might suggest. Due to the way in which farms are generated (converting the smallest of the layout rectangles), the farms added to the settlement tend to be smaller than required to feed the inferred population of a settlement. Figure 4.7a shows a house and accompanying farm.

4.2.3 Narrative

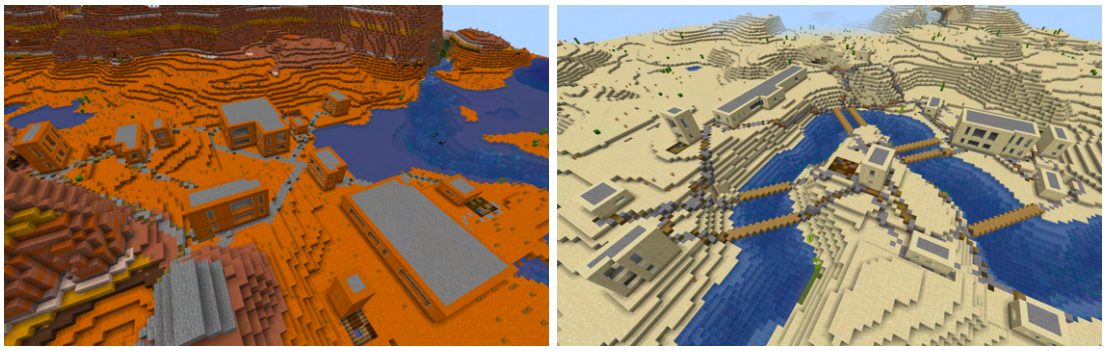
Real settlements often have a story behind them, maybe about how they were built up over time or why they were created in the first place. Objects generated using traditional techniques (such as random noise) tend to make it hard to evoke a sense of narrative or history, due to the way in which they are generated completely as an end result (instead of using a some kind of simulation based approach). The style of generation proposed in this project is a way of avoiding this. The settlements generated using these methods are the results of interactions and behaviours of agents over time. This means that the resulting settlement has a sense of history in the way it was created. Buildings are placed in an order such that larger, more important buildings tend to be given priority and are placed first. The road networks show a clear history of agents travelling around the settlement, progressively improving the road networks which are most used, with less frequently travelled paths on the outskirts not being fully

[2] Hostile creatures can only appear if it is dark enough, usually at night or in areas such as caves.

developed. These features are made possible by the agent based modelling approach of the generator and would not be possible with more traditional entries to the competition.

4.2.4 Aesthetics

Aesthetics is a difficult category to judge, it is highly subjective and cannot be as simply evaluated as other categories. However, there are ways of measuring things that contribute to overall aesthetics, such as an overall theme. These generated settlements show an overall theme to them, making buildings feel as though they are part of a group rather than disjointed. Although, this could be because of the *Kaleidoscope Effect*, meaning that all elements generated by this generator end up looking similar. While this is possible, settlements are clearly influenced by their surroundings, which helps them maintain a cohesive theme but still feel different to other settlements generated in different conditions, such as the desert settlements shown in Figure 4.8.



(a) (b)

Figure 4.8: Settlements built in desert environments, note the lack of pitched roofs due to the lack of rain in this drier climate.

4.3 Summary

These results show that there is merit to using agent based methods such as these for not only generating settlements but for content generation in general. This experimentation has shown that it is possible to create a variety of elements with several layers of complexity, made possible through the use of a hierarchical generator structure. While there are improvements to be made such as performance and improving aesthetics of generated settlements, this initial exploration shows a lot of potential in this kind of generator.

Figure 4.9: An island with a generated settlement built on it.

Chapter 5

Conclusion

This project aimed to explore the possibilities of using agent based modelling for generating content. The Generative Design in Minecraft Competition offers a concrete way of demonstrating this, generating settlements for Minecraft worlds. The generator that was made as a result of this project was designed with the competition in mind but is also intended to demonstrate the validity of such methods for use in content generation.

5.1 Project Status

5.1.1 Generator Produced

The generator produced as part of this project works as intended and forms what could be described as a good entry into the competition. The generator can build settlements in a variety of environments with interesting structures and reasonable layouts. However, there are a few improvements to be made before potentially entering it into the competition but it represents a good exploration into agent based modelling.

5.1.2 Exploration

This generator serves as a good first exploration into new methods for content generation. Some clear implications about the benefits of such methods can be drawn from the results and the experimentation throughout the project further shows some of these benefits whilst also highlighting some of the drawbacks.

5.2 Future Work

5.2.1 Improving the generator for the competition

While the generator represents a good effort at a competition entry, there are some additional features that could be added to improve the solution. These features would help improve the generated results but were not necessarily essential for the main aim of this project, simply exploring use of these methods.

Lighting

The generated settlements contain no lighting. The addition of elements such as torches along paths and interior lighting of buildings would help the functionality of the settlements from a game-play perspective.

Interiors

Currently, the generated buildings do not have any interior layouts or furniture. A few of the previous entries have interesting solutions for interior layouts, such as the use of organically grown rooms [11]. Interior furniture is a slightly harder topic, especially as Minecraft doesn't have a large variety of blocks specifically designed to be furniture. One solution might be to use predetermined furniture styles and position them around the edges of rooms [10].

Efficiency

The generator created for this project is not as efficient as it could be. The competition imposes a soft limit of around 10 minutes to generate a settlement, suggesting to entrants that they aim for settlements to be finished in that time. Due to the complex nature of agent based simulations and needing to simulate individual agents, run-times for this generator can be quite long (approximately 3 minutes for a 100 by 100 block area). This is a big problem for the competition as some of the maps the generator will be tested on can be as big as 1000 by 1000 blocks.

5.2.2 Further Exploration

While this project shows agent based modelling to be suitable for content generation, it would be beneficial to see further exploration of different methods and generating different kinds of content. For example, the generation of things such as terrain or whole worlds or the use of cellular automata. While procedural generation has been used to generate worlds for games such as No Man's Sky, generating worlds with agent based modelling would help give the worlds a sense of history and narrative.

5.3 Original Aims and Motivations

Overall, this project has achieved the goals set out in chapter 1. A valid settlement generator was built which demonstrates the suitability of agent based modelling for modelling settlements in Minecraft, representing a good entry into the Generative Design in Minecraft Competition. This generator also helped to evaluate and assess the suitability of such methods in a wider context, outlining the benefits and drawbacks of agent-based modelling.

When compared to traditional techniques (such as random noise), these methods have drawbacks when considering the time it takes to simulate and overall complexity. However, agent based modelling has been demonstrated to be a valuable and flexible way of creating generated content which results in interesting content that shows how it has clearly been influenced by the nuanced interactions which produced it.

Bibliography

[1] Razanne Abu-Aisheh, Francesco Bronzino, Myriana Rifai, Brian Kilberg, Kris Pister, and Thomas Watteyne. Atlas: Exploration and mapping with a sparse swarm of networked iot robots. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 338–342, 2020.

[2] M. C. Angelides and H. Agius. *Procedural Content Generation*, pages 62–91. John Wiley & Sons, Ltd, 2014.

[3] Travis Archer. Procedurally generating terrain. In *44th annual midwest instruction and computing symposium, Duluth*, pages 378–393. Institute of Electrical and Electronics Engineers, 2011.

[4] Peter Bentley. An introduction to evolutionary design by computers. *Evolutionary design by computers*, pages 1–73, 1999.

[5] Rogelio Enrique Cardona-Rivera. Cognitively-grounded procedural content generation. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[6] Andrew Crooks and A.J. Heppenstall. *Introduction to Agent-Based Modelling*, pages 85–105. 01 2012.

[7] Xiao Cui and Hao Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[8] Marsh Davies. Coding ye olde london. https://www.minecraft.net/en-us/article/coding-ye-olde-london, 2016.

[9] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[10] Marcus Fridh and Fredrik Sy. Settlement generation in minecraft. 2020.

[11] Michael Cerny Green, Christoph Salge, and Julian Togelius. Organic building generation in minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–7, 2019.

[12] Gregkwaste. No man's sky – procedural content. http://3dgamedevblog.com/?p=836, 2016.

[13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1), February 2013.

[14] Nicolas Hoertel, Martin Blachier, Carlos Blanco, Mark Olfson, Marc Massetti, Marina Rico, Frédéric Limosin, and Henri Leleu. A stochastic agent-based model of the sars-cov-2 epidemic in france. *Nature Medicine*, 26:1–5, 09 2020.

[15] Selain Kasereka, Nathanaël Kasoro, Kyandoghere Kyamakya, Emile-Franc Doungmo Goufo, Abiola P Chokki, and Maurice V Yengo. Agent-based modelling and simulation for evacuation of people from a building in case of fire. *Procedia Computer Science*, 130:10–17, 2018.

[16] Miquel Kegeleirs, David Garzón Ramos, and Mauro Birattari. Random walk exploration for swarm mapping. In *Annual Conference Towards Autonomous Robotic Systems*, pages 211–222. Springer, 2019.

[17] Shahroz Khan and Muhammad Junaid Awan. A generative design technique for exploring shape variations. *Advanced Engineering Informatics*, 38:712–724, 2018.

[18] Patrick Lester. A* pathfinding for beginners. *online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009)*, 2005.

[19] Jon McCormack, Alan Dorin, and Troy Innocent. Generative design: A paradigm for design research. 2004.

[20] Danil Nagy, Damon Lau, John Locke, Jim Stoddart, Lorenzo Villaggi, Ray Wang, Dale Zhao, and David Benjamin. Project discover: An application of generative design for architectural space planning. In *Proceedings of the Symposium on Simulation for Architecture and Urban Design*, pages 1–8, 2017.

[21] Ezequiel A. Di Paolo, Jason Noble, and Seth Bullock. Simulation models as opaque thought experiments. In Mark A. Bedau, John S. McCaskill, Norman Packard, and Steen Rasmussen, editors, *Seventh International Conference on Artificial Life*, pages 497–506. MIT Press, Cambridge, MA, 2000.

[22] Markus "Notch" Persson. Terrain generation, part 1. https://web.archive.org/web/20110312082752/http://notch.tum generation-part-1, 2011.

[23] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[24] Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. The ai settlement generation challenge in minecraft. *KI-Künstliche Intelligenz*, 34(1):19–31, 2020.

[25] Anthony J Smith and Joanna J Bryson. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*, 2014.

[26] Statista. Cumulative number of copies of minecraft sold worldwide as of may 2020. https://www.statista.com/statistics/680124/minecraft-unit-sales-worldwide/, 2020.

[27] Emma R Tait and Ingrid L Nelson. Nonscalability and generating digital outer space natures in no man's sky. *Environment and Planning E: Nature and Space*, 0(0):25148486211000746, 0.

[28] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[29] Jinming Wen, Li He, and Fumin Zhu. Swarm robotics control and communications: Imminent challenges for next generation smart logistics. *IEEE Communications Magazine*, 56(7):102–107, 2018.