Adaptive Procedural Generation in Minecraft

Blake Patterson, Michael Ward

CIS 5603 Final Project, Dr. Pei Wang
Department of Computer & Information Sciences
Temple University
May 3, 2022

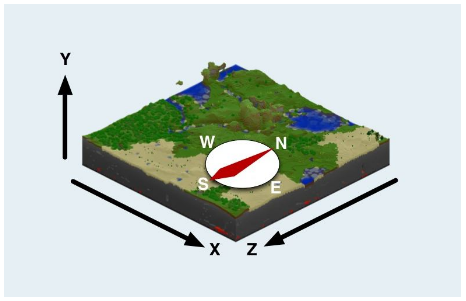
Contents

1 Abstract

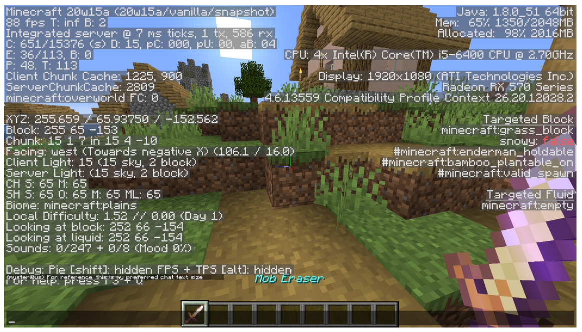
Abstract here

2 Introduction

Minecraft is a sandbox video game, created by Mojang in 2009, where players explore and build in a procedurally generated 3D grid-like world with infinite terrain. The main game-play element of Minecraft consists of collecting various types of materials and using them to build tools and structures. The world is divided into 1x1x1 blocks which can vary in material, spawning location, and usability. Aside from the popularity of the base game, Minecraft has become well known for it's customization possibilities through a variety of open source application program interfaces. These application program interfaces allow players to modify textures and color palettes, add new items, block types and enemies, and more.



(a) Grid



```
Minecraft 20w15a (20w15a/vanilla/snapshot)
88 fps T: inf B: 2
Integrated server @ 7 ms ticks, 1 tx, 586 rx
C: 651/15376 (s) D: 15, pC: 000, pU: 00, aB: 04
E: 36/113, B: 0
P: 48. T: 113
Client Chunk Cache: 1225, 900
ServerChunkCache: 2809
minecraft:overworld FC: 0

XYZ: 255.659 / 65.93750 / -152.562
Block: 255 65 -153
Chunk: 15 1 7 in 15 4 -10
Facing: west (Towards negative X) (106.1 / 16.0)
Client Light: 15 (15 sky, 2 block)
Server Light: (15 sky, 2 block)
CH S: 65 M: 65
SH S: 65 O: 65 M: 65 ML: 65
Biome: minecraft:plains
Local Difficulty: 1.52 // 0.00 (Day 1)
Looking at block: 252 66 -154
Looking at liquid: 252 66 -154
Sounds: 0/247 + 0/8 (Mood 0%)

Debug: Pie [shift]: hidden FPS + TPS [alt]: hidden
For help: press F3 + Q
<mysteri0us> For reference, this is my preferred chat text size
```

```
Java: 1.8.0_51 64bit
Mem:  65% 1350/2048MB
Allocated:  98% 2016MB

CPU: 4x Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz

Display: 1920x1080 (ATI Technologies Inc.)
Radeon RX 570 Series
4.6.13559 Compatibility Profile Context 26.20.12028.2

Targeted Block
minecraft:grass_block
snowy: false
#minecraft:enderman_holdable
#minecraft:bamboo_plantable_on
#minecraft:valid_spawn

Targeted Fluid
minecraft:empty
```

(b) Developer Overlay

Figure 1: Observable Minecraft Environment

The open source nature and in-game environment of Minecraft has also caught the attention of artificial intelligence researchers. The environment of Minecraft is ideal for research in AI because of the endless possibilities, from training an agent on simple tasks like searching for a specific object or material, to building complex structures or navigating obstacle courses. Since the environment is divided into a three-dimensional grid of equal sized cubes, it is also easy to measure and evaluate the performance of AI in Minecraft.

This project is focused on the application of AI for Procedural Content Generation (PCG) within Minecraft. PCG is defined as the algorithmic creation of game content with limited or indirect user input [13]. Content in the context of PCG can be described as most of what can be contained within a game including maps, rules, textures, items, quests, music, characters, and more [13]. Many popular games have made use of PCG including Rogue, Dwarf Fortress, Diablo, Spore, Civilization, Spelunky, as well as Minecraft itself [13]. The usage of PCG varies from game to game and can range from fully autonomous game design, to automating routine or common aspects of game design. One major critique of PCG in game design has to do with repetition and functionality; rule-based agents are likely to create good looking and functional content that looks similar, and search-based agents are likely to create more diverse content, but takes more time and resources to ensure that it is functional for the player [6].

Most instances of PCG in video games operate from a 'clean-state' where the generator does not have to consider interaction with preexisting in-game elements [6]. Exploring PCG within Minecraft opens up a new challenge within AI, in which the goal is to produce a functional and believable village settlement that adapts to different environments within a Minecraft map [11]. Instead of generating a village on a clean slate, this problem restricts the generator with the presence of preexisting game elements and focuses on adaptive generation of artifacts [6]. A map in Minecraft is made up of various biomes which contain different types of terrain, elevation gradients, fauna, and bodies of water. In order for a procedurally generated settlement to be functional and believable, it must be adaptive and able to build on top of and in response to elements that already exist in the Minecraft environment. The Generative Design in Minecraft Competition (GDMC) AI settlement generation competition initially proposed this problem in 2018 [10]. GMDC has ran a yearly open competition for researchers and students to submit their algorithm, which is scored by a panel in terms of the algorithms adaptability and functionality [5].

We propose to develop a Procedural Content Generation AI that is capable of generating a functional and believable Minecraft settlement, which is adaptive to varying environmental factors. Based on our review of literature, it is apparent that developing multiple different algorithms to handle individual pieces of the problem has led to better outcomes in previous research.

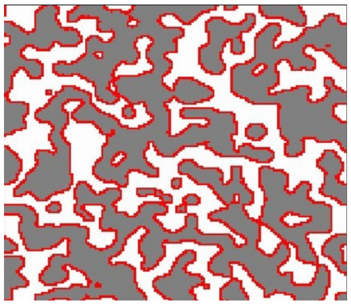
3 Related Work

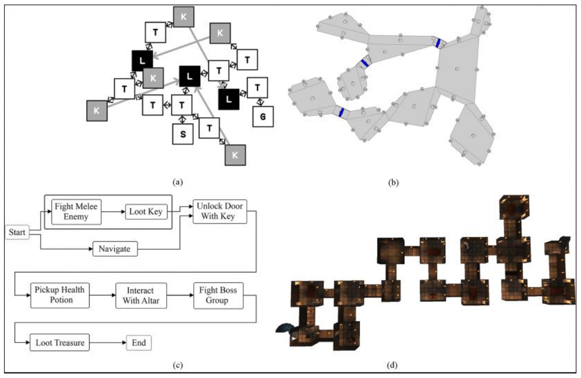
3.1 PCG in Games

3.1.1 Dungeon Generation

The most common use of Procedural Content Generation (PCG) in video games historically has been dungeon generation. Dungeons are labyrinthine-like environments which are made up of intricate pathways leading to rewards, puzzles, or progression points [17]. Surveys on the use of PCG for dungeon generation found that it has been applied mostly to 2D games, rarely being found used in 3D games [19]. These same surveys found that two main approaches have been used for PCG, constructive algorithms and search-based algorithms. Constructive algorithms are usually based on random positioning, cellular automata, or graphs grammars.

Cellular automata applies a set of rules for specifying cell-state transitions based on cells' neighbors to a two dimension grid of cells [2]. This method repeatedly applies the defined set of rules to the grid, which leads to the change of cell states until the grid stabilizes. Cellular automata methods for PCG generates large 2D maze-like patterns, which are generally merged using a region merging algorithm to ensure that the entire area is playable [2]. This approach is generally used in PCG to create a randomized spatial structure / layout for a level or dungeon. See figure 2 (a) for an example of a map generated with cellular automata.


(a) Map generated with ceullar automata. Grey areas represent floor, red represents walls, and white represents rocks [8]



(b) (a) mission with tasks, keys, and locks. (b) mission structure from (a) mapped spatially. (c) gameplay graph (d) dungeon layout generated by (c) [17]

Figure 2: Dungeon Methods

Graph grammar is an extension of generative grammar, a system of grammatical rules used to generate sentences or words [15]. Graph grammar is used in PCG to support the creation of levels that are divided into rooms. Typically, player actions are mapped onto a directed graph, which is then used to generate the dungeon or level layout. Nodes on the graph are the player actions, while graph edges indicate the order of actions. This procedure works well for dungeon or level based games because typically, the geometry and content of a level are dependent on gameplay requirements, not the other way around [18]. See figure 2 (b) for examples of gameplay graphs and their generated layouts.

Overall, few dungeon PCG approaches present solutions for 3D games, and only one in recent times supported adaptive generation [19]. So while this work is relevant and informative to our work, we did not opt to use any of these approaches in the current iteration of our project.

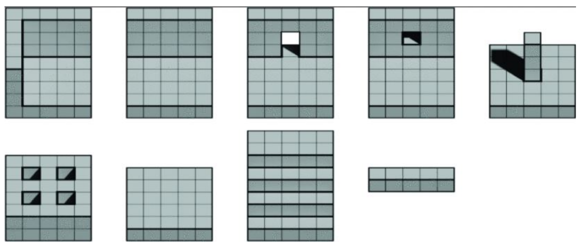
3.1.2 Search based approaches

Search based procedural content generation [16].

3.1.3 Architectural generation

PCG for architecture has been used in video games, as well as general digital media. One recent approach to PCG for architecture uses a declarative (easily steered to intended typology) and comprehensive (capable of creating multiple typologies) tile-based generator [1]. This method makes use of architectural profiles (also called tile profiles), which are a semantic segmentation that characterizes different types of architectural building blocks. These architectural profiles are then combined, using adjacency conditions that are applied to each type of architectural profile. A tile solver translates the adjacency conditions into logic constraints, which then determines tile placement. Examples of tile profiles and their tile adjacency semantics can be seen in figure 3. This method is capable of producing a large range of architectural styles, in terms of density (ratio of interior and exterior space) and repetitiveness (prevalence of patterns in terms of tile placements).

We have not gotten far enough in our current project iteration to tackle building interiors, but this approach looks promising for creating adaptive building interiors & exteriors in Minecraft. The authors of this work comment that while this approach is game agnostic, they believe it has potential for generating urban environments in Minecraft [1].


(a) Tile profiles. From left to right, first row (interior tiles): corner, wall, door, window, interior area; second row (exterior tiles): roof, street, stairs, landing, void tile. [1]

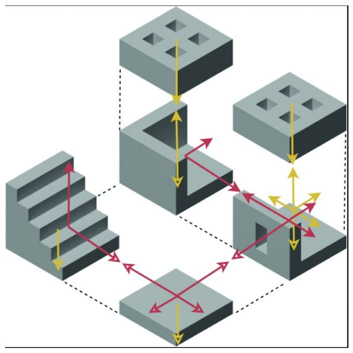

(b) Semantics on tile adjacency conditions, in two types: traversal (in red) and construction (in yellow). [1]

Figure 3: Architecture Methods

3.1.4 Reinforcement Learning in PCG

While reinforcement learning has been used to teach an agent how to play a game many times, using it to teach an agent how to generate a game level has not been explored to the same extent [9]. This particular application of reinforcement learning has not appeared in literature until 2020. Khalifa et al. speculate that the reason for it's absence is that it's unclear how to approach the level generation process as a reinforcement learning problem.

To frame this problem as a RL problem, the authors describe PCG as an iterative task rather than generating all content in one step, which allows them to model PCG as a markov decision process. To represent the level generation problem as a MDP, they used three different representations of the state space, action space, and transition function that are taken from earlier RL approaches. In the first representation which they call 'Narrow', the agent is restricted to a predetermined sequence of build locations. In the second representation which they call 'Turtle', agents have control over their current location, but only with respect to their last location. Lastly, in the third representation which they coined as 'Wide', the agent has full control over location and tile type. They applied these three agents to three different 2D level generation problems. To illustrate how their approach works, we'll describe one of the problems. In the 'Zelda' problem, the level must have exactly one player, one door, and one key, and the player must be able to reach the key and the door in a set number of steps. The level may have enemies, which cannot spawn within a certain radius of the player. The goal of the agent in this scenario is to modify the 2D level [9].

Ultimately the authors found that the agent struggled to design complex levels, but was still able to generate a large number of playable levels. This approach isn't easily mappable to village generation in Minecraft, but is a unique recent approach to PCG in game design. We aren't sure of how easily this methodology could be extended to 3D games but if possible, then player created village could be used as the training material.

3.2 PCG in Minecraft

Multi agent settlement gen in minecraft [4].

World GAN [3].

Growing artefacts and machines with neural cellular automata in minecraft [14].

Green et al. generate floor plans using a constrained growth algorithm and cellular automata [6].

4 Methodology

4.1 Interface Mod and Python Client

We chose to implement our various algorithms & methodologies using Python, but we still needed some way of communicating with Minecraft. To do this we chose to make use of the Generative Design in Minecraft Challenge (GDMC) HTTP Interface Mod and corresponding Generative Design Python Client (GDPC). As mentioned before, GDMC is a yearly competition for researchers and students to submit their procedural generation algorithms for Minecraft. In order to allow competitors to focus more on the algorithms themselves as well as to foster more consistency between how the submissions communicated with Minecraft, some members of GDMC developed these tools & made them open source for all to use.

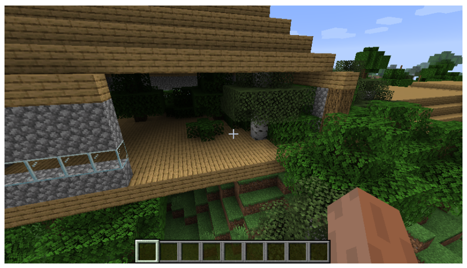
The GDMC HTTP Interface Mod, is, as it sounds, a mod for Minecraft. A Minecraft mod, or modification, is a user made piece of additional software that adds onto the core source code of Minecraft. Countless mods exist for Minecraft, most of which are for the purpose of enhancing the game's performance/visuals or adding new content to the game, but many functional mods such as this interface mod also exist. In order to make additional software work with Minecraft, however, an API needs to be in place that will make Minecraft recognize and properly implement the mod's code (since Minecraft's code itself is closed & not directly modifiable). The most popular of which is Minecraft Forge, so we had to set up Forge for our machines. With Forge set up, we simply installed the GDMC HTTP Interface Mod and that was that.

The mod itself does one simple yet vital thing within Minecraft: whenever a world is opened a corresponding HTTP server is launched on localhost port 9000. Thus, with the mod set up on our machines, as long as we had a Minecraft world open, we could communicate to it through this HTTP server via basic HTTP requests to certain endpoints. For example, if we wanted to know what block was at a certain (x, y, z) coordinate in the world, we could make a GET request to the server at the blocks endpoint (i.e., ”localhost:9000/blocks”) with the coordinates as parameters and it would return information about the block at those coordinates. On the other side of things, if we wanted to place a block, we could perform the same request but as a PUT rather than GET request as well as provide a block ID as a parameter, and the corresponding block would be placed at those coordinates in the world. There are multiple other endpoints which serve multiple other purposes, all of which can be found in the GDMC documentation.

Technically the mod alone is enough for us to communicate with Minecraft, although having to write the HTTP requests ourselves is tedious. Luckily the mod's developers also thought of this and created the GDPC, which we will refer to simply as the Python client, to alleviate this tediousness. The Python client is a framework strictly to be used alongside the GDMC HTTP Interface Mod in order to make sending the necessary HTTP requests much easier. It can essentially be thought of as a wrapper, which allows us to simply call functions to communicate with Minecraft rather than write the HTTP requests ourselves. For example, rather than writing GET & PUT requests to get & place blocks manually, the Python client defines two functions to do exactly that, getBlock & placeBlock. It even goes beyond the basic operations and defines functions such as placeCuboid & placeCenteredCylinder to make building larger more involved structures easier. The Python client also makes obtaining information about the world substantially easier. For example, in one line of code the Python client allows us to obtain a world slice, which, in short, contains all of the information about a certain section of the Minecraft world that we could ever need & puts it in a nicely packaged data structure for us. The rest of the functionality provided by the Python client can again be found in the GDMC documentation, however, the functionality described here is largely all that was needed for this project.

4.2 Terraforming

When testing our program, we quickly encountered an issue specific to Minecraft for this type of procedural village generation, trees. Trees in Minecraft can appear in almost any biome, & can be densely populated or sparsely populated. Trying to place a building on an patch of land that already contains trees leads to various undesirable results, as seen in figure 4. Building on top of preexisting trees can lead to houses being filled with trees, trees poking out the tops of houses, & sometimes even houses being placed on top of trees. Since our goal is to build a settlement in any randomized location within Minecraft, handling the presence of trees is a key factor of an adaptable village generator.


(a) Trees in house

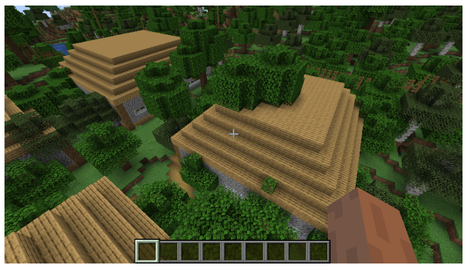
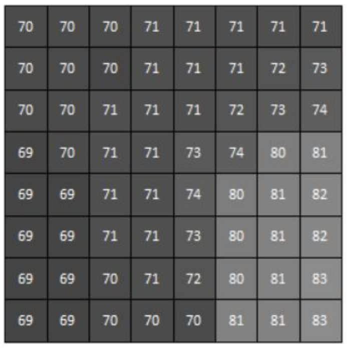

(b) Trees on house

Figure 4: Building on trees examples

This problem of adapting to existing structures is one unique to our project. As discussed previously, procedural content generation most typically operates on a clean slate. Since this problem is somewhat unique to our project, there is not an accepted methodology for handling preexisting tree structures. While this is a problem that any teams who submit or have submitted work to the GDMC competition would have to address, few teams have produced written work detailing their methodology.

To solve this problem, we used our knowledge of trees & the data structures that are accessible via the GDMC Python client to devise an algorithm that can efficiently locate & remove trees that are present in a randomly selected build area. Trees in Minecraft have the same attributes as real-life trees; they grow straight up from the ground, are made of wood, & typically have leaves surrounding the canopy. Trees in Minecraft are made specifically from wood blocks, which are solely found in the form of trees. With this knowledge, we could create an algorithm that searches the 3-dimensional space until it finds a wood block, then searches the y-coordinate space up & down from that block until wood isn't found any more in either direction. This was our first approach, but it was computationally expensive. The typical build area size we tested on is 128 x 128 x 128 blocks, so at worst this algorithm would explore roughly 2 million blocks. To come up with a more efficient algorithm, we needed to find a better way to search the 3D space - this was quite difficult as there is no maximum number or minimum number of trees in any given x,y,z boundaries within Minecraft.



70	70	70	71	71	71	71	71
70	70	70	71	71	71	72	73
70	70	71	71	71	72	73	74
69	70	71	71	73	74	80	81
69	69	71	71	74	80	81	82
69	69	71	71	73	80	81	82
69	69	70	71	72	80	81	83
69	69	70	70	70	81	81	83

Figure 5: Example of an 8 x 8 heightmap [4]

In order to restrict the search space, we extracted a height map of the given build area using the GDMC Python client - which is a 2D array the size of the the x,z coordinate space that contains the highest y coordinate value (not containing air or leaves) at each x,z coordinate combination. See figure 5 for an example of a height map for an 8 x 8 area. Instead of searching potentially the entire 3D space, we could search the 2D height map for the presence of wood blocks instead, which would give us a starting node for each tree - the top block of each tree. Since the height map is the highest y coordinate in each x,z coordinate combination, we can assume that any given coordinate in a height map is the top of a tree if the coordinate contains a wood block.

Once the height map has been searched for the presence of wood blocks, the 3D space can be searched, using the tree top coordinates obtained from the height map search as starting nodes. Each node is expanded downwards until wood blocks are not found, giving us the full coordinate map of trees in the build area. An inverted conical buffer is applied to each tree's coordinate space to ensure the removal of leaves, and then the tree coordinates and buffer space are replaced with blank space. This search algorithm is much more efficient than the first search algorithm used, & at worst searches roughly 16,000 blocks in a 128 x 128 x 128 build area.

4.3 Plot Analysis

After clearing the designated build area of trees, it was necessary to compute how buildings and structures should be distributed across the build area. The distribution of buildings is a key piece of the problem that we want to solve, which is determining how a procedurally generated village should adapt to various randomly selected terrain. For this problem we also applied a search-based approach, using a fitness/evaluation function which grades the suitability of different locations in the build area for the purpose of house building.

Our plot analysis algorithm divides the randomly selected build area into subdivisions that are based on the size of the total build area. For each subdivision, it generates a height map & calculates the standard deviation of height within each respective subdivision. From here, the fitness function uses the standard deviations of height to evaluate which subdivisions are suitable for building houses on, with a preference for locations with low standard deviation of heights.

This algorithm could be further improved by adding additional vectors to the fitness function. Some vectors which we think could be useful to include are the percentage of water present in subdivision & distance to the nearest body of water from subdivision centroid. These vectors could help the fitness function to avoid building in areas with small patches of water & also be set to build either near or far from water sources such as rivers or ponds. Another problem that our current algorithm runs into is that sometimes it will unnaturally place one building far away from the main grouping of buildings - this can happen if there's a hill that has a small flat area on top of it, while the rest of the area is relatively flat. This could be solved by calculating the height gradient of the entire build area & tuning the fitness function to prefer placing buildings in the same gradient of the build area.

4.4 House Building

With land cleared & plot locations determined, the next step was to build a structure at each plot. To start off we chose to implement arguably the most common structure in Minecraft, something that is a must for any settlement, a house. There are different methods out there that other researchers have developed which make use of various AI techniques to build a house which changes based on different factors, but as our focus was more on the settlement as a whole and making the layout adapt to the land around it, we chose to implement a basic house building procedure. This allowed us to put more time into plot analysis & path building, which were both more important to our overall goal than each individual house.

In the end, we developed one main function which takes in six parameters: a set of starting x, y, & z coordinates, and a set of ending x, y, & z coordinates. The starting x & z coordinates determine the bottom left most point of the house & the ending x & z coordinates determine the top right most point of the house. The starting y coordinate determines floor level & the ending y coordinate determines ceiling level (this does not account for the roof, which goes about five blocks above ceiling level). With these parameters set up & some predetermined materials in place (i.e., oak planks for the floor, cobblestone for the wall, etc.), the function iterates through the different dimensions, placing the floor & some support pillars, followed by the walls & windows, followed by the roof, and finishes off with a door.

Finally. as one last step, we added one supplementary function on top of the main house building function. This function takes in an array of coordinates to build houses at (i.e., the result of our plot analysis) and calls the build house function with a slight random adjustment to each of the parameters. This not only made sense from a code organization perspective, but allowed us to add some random naturalness to our houses rather than have them be all the same size.

4.5 Path Building

With the all of the houses placed, our settlement is nearly completely built, save for one final piece: the paths. Every settlement has paths connecting the different structures, even in real life, so we had to come up with a methodology to generate paths between each of our structures in a natural looking yet still functional way. The immediate choice that comes to mind, which also turned out to be the best choice for our situation, was A*.

This is not the place to be reviewing the exact details of what A* is, but in short it is a state space searching technique popular in the field of AI. The reason it is popular within AI, and considered an AI technique itself, is because it takes into account the cost of the path (i.e., $cost(p)$) as well as a heuristic evaluation of the path (i.e., $h(p)$), rather than simply calculating the exact optimal path outright. This means we had to determine how to evaluate the cost of our path so far as well as how to heuristically evaluate it in such a way that led to a natural looking yet functional path within Minecraft. As it turns out, the most typical cost & heuristic functions used in standard grid search problems worked incredibly well in Minecraft. In short, the cost function was simply equal to how many blocks the path had stretched so far, and the heuristic function was equal to the Manhattan distance from the end of the path to the goal destination. More specifically, we calculated the Manhattan distance as $|x' - x| + |z' - z|$, where (x, z) are the coordinates for the end of the current path & (x', z') are the coordinates for the goal destination. One might understandably ask why the vertical y coordinate is excluded, and it is simply because the resulting paths were better without it. This is most likely because we did not ultimately want to traverse vertically all that often, and all that really mattered was the horizontal traversal of the path.

With the cost & heuristic functions set up, there is only one thing left to determine: the boundaries of our path. More specifically, where can A* look when determining where to place the path, or even more accurately, where can it not look? This is where our implementation of A* had to become slightly modified from most. See, most implementations of A* are concerned with two dimensional grids/planes of some sort, and they do not have to worry about a third, vertical dimension. Simply ignoring the third dimension would not work though, as if we ran it normally it would lead to massive, immediate vertical jumps in the path which not only do not look natural, are non-functional. Also, simply restricting the path to only work in two dimensions is not only does not make sense theoretically since the player is entirely capable of moving vertically, but almost never works in practice since the land in Minecraft is rarely perfectly flat. Thus, this led us to determine that the path should be able to move vertically, but only by one block at a time because any more and the player would not be able to traverse it. This, unsurprisingly, worked very well and led to natural looking & functional paths.

There was still one problem, however, and that was the horizontal boundaries at which A* was able to place paths. If A* is unable to reach its destination in a relatively direct way, it will start trying increasingly indirect ways to reach its destination. In fact, by default, it will exhaustively search the entire possible set of paths within the grid it is searching to determine if there is a possible path. This does not work in Minecraft, however, because the world of Minecraft is, for all intensive purposes, infinitely large. Thus, if there was no immediate direct path, or even a relatively indirect one, A* would continue for as long as possible before terminating. Not only did this lead to ridiculously long run times, but it also led to extremely inefficient paths. To solve this, we simply put in a restriction that said A* cannot go outside a certain border that was predefined along with the settlement itself. If it was not possible to reach the destination within these boundaries it would simply place the best path it could before terminating.

With all of these parameters & rules in place, we simply had A* run and then place a chunk of blocks at each point along its determined path (as one block did not look natural enough). Then, we set up a simple function which took in the coordinates of the front of each house & built paths between each of the houses such that everything was interconnected. All said & done, this implementation of A* led to relatively natural looking & functional paths.

5 Discussion

5.1 Results

Results here.

5.2 Limitations

Limitations here.

6 Acknowledgments

We'd like to thank the contributors and maintainers of both the Minecraft HTTP Interface Mod and Generative Design Python Client: Nils Gawlik, Claus Aranha, Mayank Jain, and Flashing-Blinkenlights. This project would not be possible without the tools they built that allow for interacting with Minecraft via Python scripts.

7 References

[1] Levi van Aanholt and Rafael Bidarra. "Declarative procedural generation of architecture with semantic architectural profiles". In: *2020 IEEE Conference on Games (CoG)*. IEEE. 2020, pp. 351–358.

[2] Chad Adams and Sushil Louis. "Procedural maze level generation with evolutionary cellular automata". In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 2017, pp. 1–8.

[3] Maren Awiszus, Frederik Schubert, and Bodo Rosenhahn. "World-GAN: a Generative Model for Minecraft Worlds". In: *2021 IEEE Conference on Games (CoG)*. IEEE. 2021, pp. 1–8.

[4] Albin Esko and Johan Fritiofsson. *Multi-Agent Based Settlement Generation In Minecraft*. 2021.

[5] Marcus Fridh and Fredrik Sy. "Settlement Generation in Minecraft". In: (), p. 55.

[6] Michael Cerny Green, Christoph Salge, and Julian Togelius. "Organic Building Generation in Minecraft". In: *arXiv:1906.05094 [cs]* (June 11, 2019). arXiv: `1906.05094`. URL: `http://arxiv.org/abs/1906.05094` (visited on 02/24/2022).

[7] Jean-Baptiste Hervé and Christoph Salge. "Comparing PCG metrics with Human Evaluation in Minecraft Settlement Generation". In: *arXiv:2107.02457 [cs]* (July 6, 2021). arXiv: `2107.02457`. URL: `http://arxiv.org/abs/2107.02457` (visited on 02/24/2022).

[8] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. "Cellular automata for real-time generation of infinite cave levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. 2010, pp. 1–4.

[9] Ahmed Khalifa et al. "Pcgrl: Procedural content generation via reinforcement learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 16. 1. 2020, pp. 95–101.

[10] Christoph Salge et al. "Generative Design in Minecraft (GDMC), Settlement Generation Competition". In: *Proceedings of the 13th International Conference on the Foundations of Digital Games* (Aug. 7, 2018), pp. 1–10. DOI: `10.1145/3235765.3235814`. arXiv: `1803.09853`. URL: `http://arxiv.org/abs/1803.09853` (visited on 02/24/2022).

[11] Christoph Salge et al. "Generative Design in Minecraft: Chronicle Challenge". In: *arXiv:1905.05888 [cs]* (May 14, 2019). arXiv: `1905.05888`. URL: `http://arxiv.org/abs/1905.05888` (visited on 02/24/2022).

[12] Christoph Salge et al. "The AI Settlement Generation Challenge in Minecraft: First Year Report". In: *KI - Künstliche Intelligenz* 34.1 (Mar. 2020), pp. 19–31. ISSN: 0933-1875, 1610-1987. DOI: `10.1007/s13218-020-00635-0`. URL: `http://link.springer.com/10.1007/s13218-020-00635-0` (visited on 02/24/2022).

[13] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[14] Shyam Sudhakaran et al. "Growing 3d artefacts and functional machines with neural cellular automata". In: *arXiv preprint arXiv:2103.08737* (2021).

[15] Tommy Thompson and Becky Lavender. "A generative grammar approach for action-adventure map generation in the legend of zelda". In: (2017).

[16] Julian Togelius et al. "Search-based procedural content generation: A taxonomy and survey". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186.

[17] Roland Van Der Linden, Ricardo Lopes, and Rafael Bidarra. "Procedural generation of dungeons". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (2013), pp. 78–89.

[18] Roland Van der Linden, Ricardo Lopes, and Rafael Bidarra. "Designing procedurally generated levels". In: *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2013.

[19] Breno MF Viana and Selan R dos Santos. "A survey of procedural dungeon generation". In: *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE. 2019, pp. 29–38.