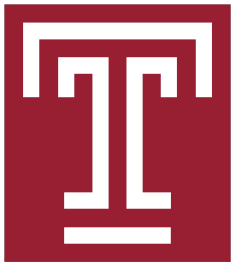
Adaptive Procedural Generation in Minecraft

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 change textures and color palettes, add new items, block types and enemies, and more.

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 [7]. 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 [7]. Many popular games have made use of PCG including Rogue, Dwarf Fortress, Diablo, Spore, Civilization, Spelunky, as well as Minecraft [7]. 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 [2].

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 [2]. 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 [5]. 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 [2]. 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 [4]. 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 [1].

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 first 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. For example, iterations of the A* algorithm have been successful in creating road networks between houses within the settlements. For other parts of the problem such as terrain analysis and building generation, different approaches will need to be employed that require more research. In the tasks and timetable sections, we lay out our current expectations for what steps will be needed and the order of steps.

3 Related Work

Related work here

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

Talk about terraforming

4.3 Plot Analysis

Talk about plot analysis

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

Acknowledgements here.

7 References

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

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

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