# Minecraft Terrain Generation for GDMC

Gavin Lynch, Kamran Bastani, Beck Schemenauer, Max Schemenauer and Arjun Ranade
Cal Poly,
San Luis Obispo, United States
{galynch, bschemen, ayranade, kmbastan, mschemen}@calpoly.edu

***Abstract*—This project developed for CSC-480: Artificial Intelligence, applies AI and generative design principles within Minecraft, adhering to the Generative Design in Minecraft AI Settlement Generation Competition guidelines. This project employs the Generative Design Python Client (GDPC) and the GDMC-HTTP framework to create a model capable of analyzing the surrounding environment, pinpointing ideal construction sites, efficiently removing unneeded obstacles, and constructing adaptable structures tailored to the environment. Building upon past implementations, this project seeks to further advance the presence of AI in Minecraft by integrating new features and enhancing existing functionalities. This paper offers a comprehensive breakdown of our project's implementation, providing insights into the development process, challenges encountered, and solutions devised.**

## I. INTRODUCTION

This paper presents our AI Generative Design Model, a Minecraft settlement generator that uses agent-based generation to create a realistic settlement that adapts to its surrounding environment. It was created as the final project for CSC-480: Artificial Intelligence at Cal Poly San Luis Obispo. The main motivations behind this project stemmed from Minecraft's resemblance to real-world environments, where players must navigate through varied terrains, adapt to different environmental constraints, and respond to dynamic scenarios. This mimics real-world situations in which adaptability is highly valued. Using AI, we wanted to construct a model capable of adapting to varying scenarios and accurately representing real-world challenges. Our implementation revolves around several key components. We have created algorithms to efficiently clear terrain and ensure optimal build plots through world analysis, schematic generation, schematic use, realistic road infrastructure construction using A* search, and dynamic foundation adaptation to varying terrain heights. These components help create a model for adaptive construction within Minecraft. Using this model, we made a fully functioning settlement within Minecraft, which conformed to varying adaptive constraints. Overall, our results matched closely with those of past GDMC entries, reflecting the success of our model.

## II. SPECIFICATION

For the GDMC competition, entries are evaluated based on various factors, including narrative, adaptability, aesthetics, and functionality. Agents are given several different maps (unknown before submission) and must generate a settlement specific to the environment. The agent should adapt to different factors, such as the terrain and biome, i.e., it should not level every map and create the same buildings every time. While the settlement is built in creative mode, it should also be functional in survival (it should provide protection, food sources, etc). The given inputs are all of the blocks currently in the world. Using the GDPC library, a height map of the world can be obtained. This is a 2D array that contains the y coordinate of the block of air above the highest block given a certain x and z value.

## III. RELATED WORK

Although there are several implementations of GDMC, the one that stood out to us was AgentCraft [5] , led by Ari Iramanesh and Max Kreminski from the University of California: Santa Cruz. AgentCraft utilizes a blend of predefined schematics and algorithms such as straight-line pathfinding/A* search, iterative settlement construction, and feedback loops. The AgentCraft model utilized a multifaceted approach by designing multiple agents, each tasked with distinct functions within Minecraft. For example, some agents were dedicated to developing the paths between the structures, building the structures, and deforestation/removing trees and obstacles. These agents were used in parallel to create iterative settlement construction and progressively see the dynamic building and evolving settlement as the model progressed through various stages.

Some of our other references include the GDPC tutorials on GitHub [1], the GDMC GitHub page [3], and GDMC wiki [4]. We used all of these tools extensively when developing our agent, and they provided indispensable support when we had issues involving both GDPC and the game itself. Without these sources, our job would have been much more difficult.

## IV. METHODS

### A. Environment Removal

To start building our structures within Minecraft, we needed to ensure that the build area was free of obstacles and unnecessary objects. To this end, we created an algorithm that systematically removes specific blocks within the designated build area.

Initially, we began with an algorithm that simply iterated through all the blocks within the build area's three dimensions (x,y,z). From here, the program would inspect each block within the area to determine if it matches any of the specific types defined in the blocks to remove the dictionary. If a block were identified as one to be removed, the script would remove

the block. This process ensured that only the designated obstructive elements, such as leaves, logs, etc, would be removed while keeping the terrain intact. However, one of the significant issues with this program was that it was highly inefficient and expensive. Since we were iterating through every block in the build area, many blocks were being checked. For example, in a 64 x 64 x 64 build area, this program must check 262,144 blocks every time it runs. Additionally, this script continuously generated HTTP requests to the Minecraft world to retrieve specific data about each block, resulting in additional inefficiencies. This frequent checking and iteration through each block introduced notable overhead and resulted in a relatively slow algorithm.

To combat these inefficiencies, we decided to utilize the native Minecraft "/fill" command, which enables users to simultaneously fill a large region of space with blocks. Additionally, to minimize the number of blocks being checked, we utilized the GDPC module's height maps to extract the highest and lowest surface elevations within the designated build area. This approach allowed us to precisely outline the terrain's topography and limit our block-checking process to the surface elevation range from the highest to the lowest points within the surface of the build area. From here, we employed a chunk-based iteration approach where the algorithm traversed 16 x 16 block segments. During each iteration, targeted blocks were systematically replaced with air (using the /fill method) to clear the area of obstacles. This implementation method provided an efficient terrain traversal and a streamlined process of removing blocks within the Minecraft environment.

### *B. Building Site Selection*

To find an optimal location for the settlement, we used the standard deviation of the surface to determine the flattest areas. Additionally, we did not build on areas greater than 30 percent water. To calculate this, we iterated over the height map, which is a 2D array representing the elevation of each block at the world's surface, checking for water blocks. We constructed a 2D array to represent the water coverage, with 1's representing water and 0's representing non-water surface blocks. However, due to the time-consuming nature of the getBlock() command, we decided to check every 3 rows of our height map.

Next, to identify the flattest area, we searched through every 100x100 subplot within the build area (we used a 200x200 area, but it can vary). By using a step length of 1 in both the x and the z direction, we are guaranteed to find the subplot with the lowest standard deviation. For each subplot with a new lowest standard deviation, we examined its corresponding area in the water coverage array to determine the percentage of water coverage. We multiplied this value by 3 to account for only checking every 3 rows for water.

After identifying a suitable subplot for the settlement, meeting the criteria of being flat and having less than 30 percent water coverage, we proceeded to locate optimal positions for our buildings, which are currently 9x9 in size. To do so, we looped through the area previously selected and created a list of potential building locations, which included each building plot's standard deviation. We once again cross-referenced these plots with the water coverage array, throwing anything out that had water in it. It's important to note that because we searched every 3 rows for water, there's a possibility that a house may partially overlap with water. This being said it is still impossible for a house to be entirely on the water as the size of our buildings spans across several rows of the water coverage map. After compiling the list of potential building plots, we sorted them by standard deviation and removed overlapping plots, providing each plot with an additional padding of 3 blocks in all directions. We were left with a sorted list of the top plots within the location we decided to settle on.

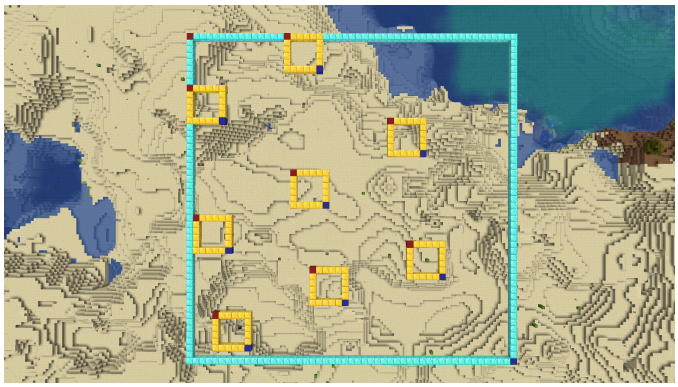

Fig. 1. Example outlines for settlement and building locations

There are several future improvements to note. The first is related to water detection. This can likely be sped up to be almost instant and have 100% water detection accuracy. We would compare the "OCEAN_FLOOR" height map with the "MOTION_BLOCKING_NO_LEAVES" height map from GDPC. Any differences in these height maps would indicate the presence of water. Another potential improvement would be to vary the shapes and sizes of building plots and the settlement plots based on the texture of the terrain.

### *C. Roads*

The base algorithm used for road generation was A star pathing, which guarantees optimality and is complete. The implementation uses a priority queue exploring the node with the lowest f value, where f is calculated by adding the total path cost to reach a node (g) with the heuristic to the goal (h). Additionally, nodes keep track of their parent node, so once the goal is reached, the path can be found by backtracking to the beginning. In our version, the heuristic used was Manhattan distance with the x and z coordinates. Since the building plot for the houses is rarely completely flat, the general algorithm must be modified to account for three dimensions. To do this, an additional variable e is added to the f value of each node, which represents the total change in elevation for the entire path up to this node. For example, if the path goes up 2 units in the y direction and then down 1 unit, the e value for that node would be 3. Since e is being added to f for each node, the algorithm will also optimize the path for elevation. We wanted to discourage the algorithm from generating paths with

a significant amount of elevation change, so each time a new node updates the e value, it multiplies the elevation change by 2 before adding it. By double counting the elevation change, the algorithm prefers a longer path with less elevation over a shorter path, which is quite steep. The choice to multiply by 2, in particular, was arbitrary (it could also be 2.5 or 3), but we found that this created a good balance of somewhat flat paths that also did not go too far out of the way. Figure 1 shows how the roads avoid unnecessary elevation and efficiently connect houses.

Fig. 2. Road network between houses

Instead of connecting all of the houses to every other house, we decided to create a "highway" through the middle of the settlement, which each house could have its own path to. To generate the highway, we calculated the Least Squares Regression Line (LSRL) with the x and z coordinates of the front doors of each house. This ensures the main road is fairly close to each house and roughly runs through the center of the settlement. Since the LSRL is typically a straight line, this would not look very natural and also has the potential of going under certain houses. To resolve this, we selected two points from the LSRL close to the edge of the settlement and used the A star pathing algorithm to connect them. This ensures the path looks more organic, does not go through any obstacles and minimizes elevation. After the main road is built, a method calculates the nearest highway block to each house's front door, and then a star is used to generate a path between them. In Figure 2, the blue line represents the main road, and the red lines are the connections from each house.

The roads could be improved in several ways in the future. For example, stairs could be placed whenever a path changes elevation to make it less abrupt. Also, if a path steps up by more than one block at a time, some blocks could be removed to make it more level. Additionally, if the path ever needs to cross a river or a canyon, a method could be added to build a bridge or some other structure to cross the gap.

### *D. Schematics*

Our team completely rebuilt our 16 biome-specific structures. We initially experimented with structure segment generation. This approach would use prebuilt segments, like walls and roofs, and then assemble them based on specific criteria. However, we decided to go with the entirely prebuilt approach as they proved to be much nicer and gave our villages a polished look. Both of these strategies would require the use of Schematics. Schematics is a specific file format that stores parts of Minecraft worlds using different software and editors, like MCEdit. Schematics allow you to copy, save, and paste specific structures anywhere in any world. The goal from the start of the project was to use Pymclevel, a package as part of the MCEdit game editor. MCEdit features include brush tools for block placement, chest, and mob spawner editing, schematic exporting/importing, and rotation tools. Additionally, it offers clone tools, Python plugin support, and maintenance commands for world repair. Unfortunately, Pymclevel and MCEdit were incompatible with the GDMC-HTTP framework on our local machines. To still be able to utilize schematics, we had to create our version of an in-game editor.

Fig. 3. Highway and side roads

The first step in creating our in-game editor was finding a way to read the blocks from a specified 3D area of a world. The write_schematic_to_file function was a critical component in creating an in-game editor. Our first challenge was to create a method to read blocks within a defined 3D space of the game world. We discovered that GDPC offered a function named 'loop3D', which generated a sequence of 3D points represented by 'ivec3' objects. This function conveniently traversed all points between specified beginning and ending coordinates, excluding the endpoint itself. Recognizing its similarity to our requirements, we adapted this function to lay the groundwork for our function, write_schematic_to_file.

This function takes two coordinates representing opposite corners of the desired 3D space and a filename as input. It automatically calculates a new set of coordinates to represent the same space. It uses the bottom southeast corner as the start corner and its opposite, the top northwest, as its end corner. It then proceeds to iterate through this space, using the logic derived from 'loop3D'; however, our implementation includes the end coordinate. The iteration begins from the 'front' or easternmost face of the structure, and progresses layer by layer towards the 'back' or westernmost face. Within each layer,

blocks are read row by row from bottom to top, with rows being read from south to north, corresponding to left to right when facing the front of the structure.

Utilizing this specific order of traversal, the function captures the blocks along with all associated information and writes them to the specified filename in the format of a 3D array, effectively creating a schematic file. Figure 3 illustrates how this file format visually represents a schematic layer. Each entry comprises the block's name followed by its block states, which are additional data-defining aspects such as appearance or behavior. Despite there being over 60 block states across Minecraft blocks, individual blocks may possess significantly fewer states. For instance, some blocks like 'air' have no states, while others such as fences may have up to five. The generation of the schematic file takes 0.5 seconds on average.

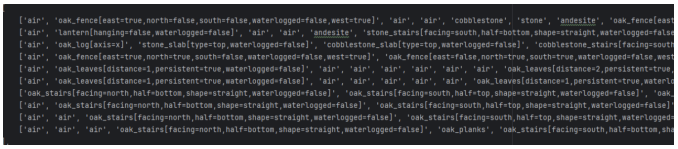

Fig. 4. A layer of a schematic

Now that the schematic file is generated, it can be built in any world using a function we wrote called build_structure. This function takes in the name of a schematic file, a building plot object, and the desired direction in which the structure should face. The one limitation to the direction of the schematic is that the initial build must be built facing east before creating the schematic file from it. If the building is not facing east, it cannot be rotated to face another direction. However, this constraint doesn't impact the quality or functionality of our project since we manually construct all buildings, ensuring they adhere to this rule. Consequently, the generation process proceeds seamlessly at runtime without any issues or drawbacks.

The building process involves several key things. The first is the conversion from the file to a NumPy array. The read_schematic_from_file function converts a schematic file into a 3D NumPy array representation of the blocks. The process is straightforward: it reads the content of the file as a string, then uses the 'ast.literal_eval' function to safely evaluate the string representation of the array, ensuring that it is treated as a literal. The resulting nested lists are then converted into a NumPy array, preserving the structure and data types of the original schematic.

The next step in the building process is dealing with the directionality of the structure. We used a simple NumPy rotate function in order to rotate the 3D array to face the direction specified by the parameters. Next, depending on this direction, we devised mappings to rotate the blocks with a directional state. This would ensure that a block facing any direction will have its direction state updated correctly in order to maintain the integrity and composition of the build. Now that the array is updated to face the correct direction, we must determine where to begin building the structure. This was done by parsing the 'plot' parameter. A plot contains a set of coordinates, a length, and a width. These coordinates are the northwest coordinates, which, as mentioned earlier, are the 'end' coordinates of a build. We simply subtract the length and width, leaving us with the southeast corner, which we have established as the standard start coordinate for schematic reading and construction.

Now that we have a start coordinate corresponding to the same relative position in the schematic as the file representation, we can simply use the same iterative logic to reconstruct the building, considering rotation for direction if necessary. For every block, with one exception, The block name is separated from its states, if it has any. The exception is 'air' blocks. These are skipped and not placed in order to save time, which corresponds to roughly a 50 percent increase in efficiency, varying depending on how much air the build contains. Once the states are acquired, they are then parsed, and if one of them has to do with which direction, then it is updated based on the mappings as described earlier. We did have a minor issue: storage blocks were not being placed because the code did not see these blocks as having a valid format. This is because they contain a list of items which they hold. The code interpreted this as a formatting error and ignored these blocks. We were able to fix this by correcting the parsing tactic to include the entirety of the block states and information. The block was ready to be placed once the states were fully parsed and the directions were updated. We then place the block and move on to the next block and relative position in the schematic.

Once the array is fully traversed, the building is finished, and the method returns a structure object containing the start and end coordinates, the file path to the schematic file used, the direction the structure faces, and the location of the door. The location of the door is needed for the road algorithm to correctly path. This location is found by simply checking a block's name during the building process; if the block is a door, its coordinates are saved to the structure object. This entire building process also takes 0.5 seconds on average.

Future implementations of this custom schematic system include a more segmented and less structured approach. We would like to have schematics with key blocks, like colored concrete or wool, to correspond to certain types of blocks, like logs or planks. This way, during construction, we can replace a certain colored block with a block of our choice. This block will be biome-specific and correspond to the specific part of the structure in which it is located. Essentially, the schematic will be made out of simple segments of colored blocks, each block indicating the type of block that must go there for the structure to make sense. For segments like walls, we can employ texturing, where we randomly select a block from a predefined list of options. This allows for more variability and character to be incorporated into the builds.

## V. Evaluation / Results

Performing an evaluation of our system is difficult, as we did not have a specific metric of success, such as win rate or accuracy. Still, we did set up some criteria before we began

Fig. 5. Cherry blossom houses

Fig. 6. Example of biome specific houses

on what we would like to accomplish. We most importantly wanted our program to be able to build, adapt to terrain, and create cohesive villages and structures in any location. Further down the line, we also set ourselves the goal of being able to automatically construct roads to connect buildings in a way that made sense. We also wanted our roads to be able to adapt to the terrain and connect buildings across the flattest ground possible. We also compared our success to some preexisting solutions, such as Williamcwi's submission [2]. While we didn't quite meet their level of success, we feel our implementation was comparable given the time limit. We achieved all these goals and more, making our project an overall success.

Given a general map area, our program is able to scan the area for the best possible location to build and create level areas to place our different buildings. Our pathing algorithm then creates roads to give our settlement a cohesive feeling and tie everything together. The program is efficient too, taking less than 15 minutes to clear and place all blocks needed. While our program isn't yet on the same level as some of the previous contest entrants, with the very limited time that we had to finish the project, we are satisfied with the result.

## VI. CONCLUSIONS

Manipulating 3D spaces is no easy task, and doing so within Minecraft while adapting to the randomly generated terrain proved to be quite the challenge. The agent we constructed manages to create realistic structures, maintain conformity in its settlement, and do so efficiently. We believe our system matches or exceeds many previous submissions to GDMC, and considering our limited time criteria, we are happy with that result. Our biggest advantage over the competition is our ability to adapt so well to the terrain and choose locations that are ideal to build on, given a limited time frame. We can create structures and roads on almost any plot of land quickly. The one big drawback of our solution is the need to remove trees for the algorithm to be able to select good places to build. Given more time, we could have sped up this process dramatically, but as of now, the time it takes to remove all trees is our weak point. Nonetheless, this gives us a good point to work off in the future, and hopefully come competition time, we can improve our system enough to get on the leaderboard. We all thoroughly enjoyed working on this algorithm and plan to take our system further outside of the classroom.

## REFERENCES


[1] van der Staaij, Arthur. "GDPC Tutorials." GitHub, github.com/avdstaaij/gdpc/tree/master/examples/tutorials, Accessed 18 Mar. 2024.

[2] Williamcwi, "Williamcwi/GDMC: The Generative Design in Minecraft (GDMC) Is an AI Settlement Generation Competition in Minecraft. as a Team, We Used Different Procedural Content Generation Algorithms to Create a Settlement on an Unknown Minecraft Map. Our Aim Is to Produce an Algorithm That Is Adaptive towards the Provided Map, Creates a Settlement That Satisfies a Range of Functional Requirements," GitHub, https://github.com/williamcwi/GDMC, Accessed 2 Feb. 2024.

[3] McGreentn, "GDMC: This Is the Framework for the Generative Design in Minecraft Competition," GitHub, https://github.com/mcgreentn/GDMC?tab=readme-ov-file, Accessed 2 Feb. 2024.

[4] "Generative Design in Minecraft." Settlement Generation Challenge 2022 - Generative Design in Minecraft, 17 Mar. 2023, gendesignmc.wikidot.com/wiki:2022-settlement-generation-competition.

[5] "Iramanesh, Ari, and Max Kreminski. "Agentcraft: An Agent-Based Minecraft Settlement Generator." eScholarship, University of California, 29 Nov. 2021, escholarship.org/uc/item/83p272pq.