Modern Game AI Algorithms Assignment 3 - Free Project
Minecraft Settlement Generation

Tom Stein [1] **Bram van Kooten** [1] **Isaac Braam** [1] **Sam Bonello** [1] **Juri Morisse** [1]

1. Introduction

Procedural Content Generation (PCG) has become a popular technique for automatically creating content in video games and providing players with a distinctive experience. This approach lessens the workload on human game designers by creating content that adapts to the game's surrounding environment while also typically containing some random generation such that the content is unique and dynamic. PCG has transformed game development by drastically cutting the time needed for game developers to produce a wide variety of immersive experiences, from simple buildings to whole game worlds. The classic Rogue dungeon crawler and the enormous, randomly generated cosmos of No Man's Sky are notable instances of PCG implementation. In this paper, we harness the power of PCG in Minecraft, a popular sandbox video game where players explore a vast and open world filled with a variety of biomes, terrains, resources, and creatures. More specifically, we use PCG to create a complete settlement that seamlessly blends in with Minecraft's vast universe. This undertaking is in preparation for the upcoming Generative Design in Minecraft (GDMC)[1] competition.

To accomplish this feat, we employ several different algorithms; namely the wave function collapse (WFC) algorithm for building generation, ant colony optimization for path creation, and a large language model for story generation. The WFC algorithm facilitates generation of believable structures by simulating collapse of particle states, ensuring consistent placement and controlled randomness throughout the city. Ant colony optimization makes use of the collective behavior of artificial ants to determine the most effective routes between buildings. It does this by leaving pheromones and following trails, simulating the foraging activity of actual ants. Finally, we made use of GPT2 to add an in-game narrator that will provide a short story about every building that a player enters.

[1]Faculty of Science, Leiden University, Leiden, The Netherlands. Correspondence to: Tom Stein <tom.stein@tu-dortmund.de>.

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

In section 2, we introduce the reader to the specifics of the GDMC competition which we plan to enter. Furthermore, we provide a brief explanation as to how the `GDPC` package in Python allows us to interact with the world generated. Next, in section 3, we describe the methods used to generate a believable settlement in any environment. In section 4, we display our results in various different environments. Finally, in section 5, we discuss limitations to our current algorithms and propose solutions for future work.

2. Preliminaries

The goal of this paper is to create an algorithm that generates a settlement for any given, unknown Minecraft map. The end goal is to enroll in the GDMC; a yearly competition that evaluates its submissions based on believable narrative, adaptability, visual aesthetics, and functionality (Salge et al., 2018). Furthermore, the GDMC competition provide an HTTP interface mod together with a corresponding Generative Design Python Client (GDPC). When used in combination with Minecraft Forge, an API that allows proper implementation of this mod, we are able to use the GDPC to interact with an open Minecraft world and retrieve and place blocks with ease.

3. Methods

In this section, we describe the algorithms that are used to generate a settlement in Minecraft. The following sections are ordered by the order of execution in the actual generator. First, the positions to place buildings are located. Second, the actual buildings are generated. Third, the buildings are connected to each other via paths. Fourth, a background story is added to each building.

3.1. Building Location Finding

To find adequate building locations, we used a 2-dimensional slope array, of which the values indicate the roughness of the terrain nearby. Within this array, the regions that have the lowest slope represent the areas that are the flattest within the given region. An example of these slope calculations can be found in Figure 1. In this figure, we can see that the green areas are the flattest areas, while

the red areas are the steepest areas. This array was calculated by taking the sum of the difference from every squares height to all its neighbours. Than this value is propagated to neighbouring states, to get a local slope score. In the last assignment we used to propagate/smoother these values out to nearby states with Q-value update steps, but we found that a Gaussian kernel does as good as the same thing, only in a lot more efficient manner.

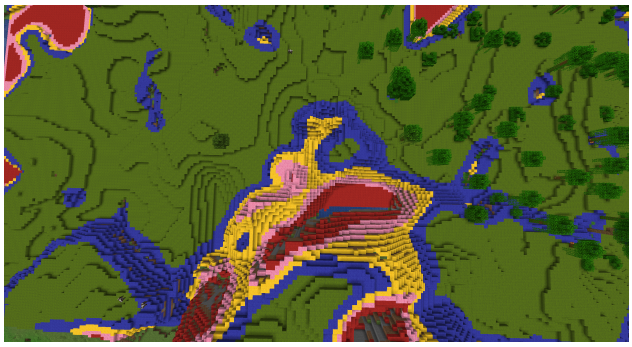

Figure 1. Example of the slope calculations in a given area. Green area is regarded as the most flat, while the red area is the least flat area.

The next step for finding a proper building location is to find a spot where there is enough space to place a building. Because of the way our buildings are generated, the location of the building is claimed in the form of multiple equally sized adjacent squares. More information about this generation can be found in subsection 3.2. An example of how building spots are claimed can be seen in Figure 2. The biggest advantage of this method is that after calculating the initial values, we can find new building spots in O(1) time complexity as we simply take the argmin of the array as the most suited building spot. To make sure buildings are not put on top of each other, once we select an argmin spot, we update the squares used by the building to the argmax. Thi is so that these indexes will not come up again when we select the argmin for the next building.

3.2. Building Generation

This section describes the generation of buildings. This is done using the wave function collapse (WFC) algorithm, which will be described in detail. However, before we introduce the WFC algorithm, we show the necessary work around the actual algorithm, that is required to generate buildings, in the following chapter.

3.2.1. Building Tiles

In order to use the WFC algorithm a set of small structures, sometimes called tiles or prefabs, is required to combine them to a larger structure, i.e. a building. All the small

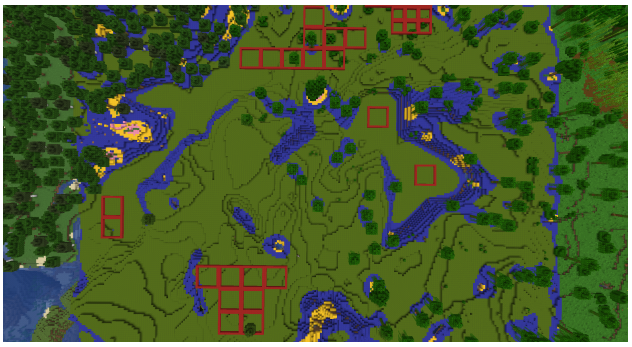

Figure 2. Example of claiming locations for building houses. Chosen locations are outlined by the red squares. Adjacent squares are part of the same building.

structures represent a part of the larger structure, i.e. walls, corners, corridors, etc. In order to find a trade-off between variation and believability, the decision was made to use relatively large tiles. The smallest tile that is available is 7 x 10 x 7, while the largest tile is 11 x 25 x 11. These sizes differ for different types of buildings. We will give an overview of the different building types used.

The brickhouse is the original building type that the wave function collapse system was used with (Stein, 2023). One notable thing about this building type is that it has two floors, where both floors have their own distinct set of tiles it can use. In total there are 14 tiles, of which an overview can be found in Figure 5. Examples of what a generated farm building might look like can be found in Figure 3 and Figure 4.

Figure 3. An example of what the brickhouse building type could look like from the outside.

The farm building type consists of two connected parts, namely an outside animal pen and an inside area that functions more like a shop. In total the farm consists of 16 tiles, of which an overview can be found in Figure 10. Examples

Figure 4. An example of what the brickhouse building type could look like from the inside.

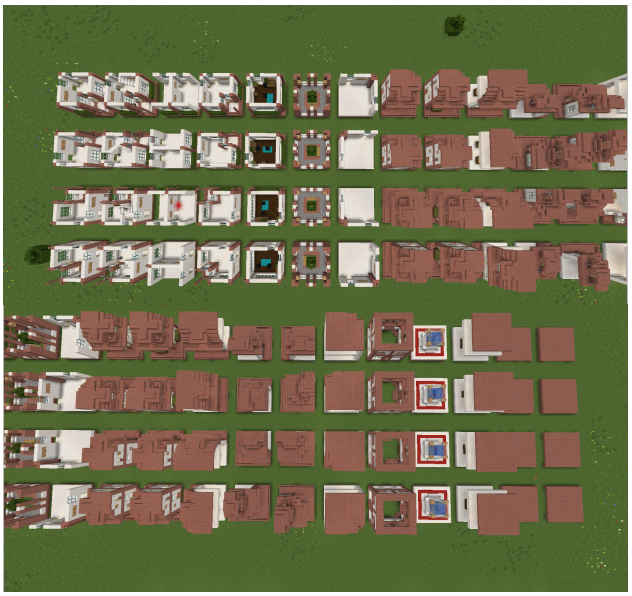

Figure 5. An overview of the tiles used for the brickhouse building type. To save space rows are continued on new line. Each column represents a single structure and each of the four rows represent a rotation of that structure. The lowermost row is the baseline rotation 0. The tiles for the ground floor have size 11 x 6 x 11, while the tiles for the roof pieces have size 11 x 9 x 11

of what a generated farm building might look like can be found in Figure 6 and Figure 7.

The bakery building type is unique due to its combination of a more open space together with single tile wide corridors. In order to properly build buildings with this, certain tiles are needed to transition from a corridor to an open space and vice versa. This has to be done in a number of different directions. Besides this, the bakery wall are not placed at the edge of the tile. This provided the opportunity to differ between two placements of walls within the tile, namely near the edge and in the middle of the tile. This difference means that the buildings end up with more interesting shapes, mostly that the buildings are less rectangular. In total the bakery consists of 21 tiles, of which an overview can be found in Figure 10. Examples of what a generated bakery building might look like can be found in Figure 8 and Figure 9.

The school building type creates its buildings by mostly stacking layers of the same set of tiles on top of each other (for example, a classroom in the second floor will be placed on top of a classroom in the first floor). Some exceptions to this are for the stair tiles and the cafeteria tiles, where specific tiles were created for the second floor. This allows for variance in the height of the building. Finally, there is a simple roof tile that gets placed on top of the top layer, which finishes the building. In total the school consists of 10 tiles, of which an overview can be found in Figure 13. Examples of what a generated bakery building might look like can be found in Figure 11 and Figure 12.

The church building type really is a statement building that stands out a lot withing a city. It is the largest type of building, having a spire towering above all the other building types that are available. However, although there are some corner tiles, it does not create buildings that are multiple tiles thick, which hinders the overall variability of the building type. Variability in the church mainly consists of the length of the building. In total the church consists of 7 tiles, of which an overview can be found in Figure 16. Examples of what a generated church building might look like can be found in Figure 14 and Figure 15.

3.2.2. Wave Function Collapse

The wave function collapse (WFC) algorithm is used to create a random version of a building that consists of the tiles described in subsubsection 3.2.1. It does this by, initially, creating a state space where all the the cells are in a superposition of all the available building tiles. From this point, the state space is updated to exclude combinations of tiles that are not possible to exist (a roof tile on the bottom floor for instance). In order for the system to know what tiles can exist next to each other, a set of rules has to be created. These rules state which tiles can exist next to each other in the x, y, and z directions. It also states which rotation of the tiles are allowed to be adjacent. In order for the system to work, all the rules have to be symmetric, meaning that if tile A is denoted to be a neighbour of tile B, tile B should also be denoted to be a neighbour of tile A.

From this point, a cell is selected to be collapsed to one of the available tiles based on the entropy of the cell, with lower entropies (meaning fewer options are available for that tile) are chosen first. In the case of multiple cells having

Figure 6. An example of what the farm building type could look like from the outside.

Figure 7. An example of what the farm building type could look like from the inside.

the same entropy, a random choice is made between these cells. After collapsing the single cell, the other cells get updated to reflect the adjacency rules of this new tile. This process continues until all the cells have collapsed to a state where they represent a single tile. It is possible however, that through the process of randomly collapsing the cells a situation occurs where one or more of the cells do not have any tiles available to collapse to. At this point, we completely restart the entire WFC process.

In order for the WFC algorithm to always build closed structures, each building type also introduces a tile set that is just air. Using this structure and the appropriate rules that define which sides of an actual building structure need to be placed next to air, one can enforce closed buildings by collapsing the outermost rectangle of cells in the state-space to air. However, this causes the entropy of the outside tiles to drop, meaning that these tiles are selected more frequently than the tiles that are not adjacent to this outer rectangle. To circumvent this issue, a number of random cells are collapsed beforehand, which in turn creates some more variation in the buildings that get generated. An example of

Figure 8. An example of what the bakery building type could look like from the outside.

Figure 9. An example of what the bakery building type could look like from the inside.

what the generation process looks like can be observed in a video[2]. (Stein, 2023)

3.3. Adaptive natural pathing

For the pathing in our settlement we have constructed the following Desiderata:

1. Pathing must adjust to the terrain well.
2. Pathing must incorporate different paths into a network of paths in a natural and optimal manner [3].
3. Pathing must not alter the buildings we built.
4. There has to exist at least one path to every building.
5. Pathing between houses must be moderately optimal.

[2] https://cloud.fachschaften.org/s/iHzxSNXsZpsr46Z

[3] Optimal meaning both minimizing travel time, and building material used for the paths

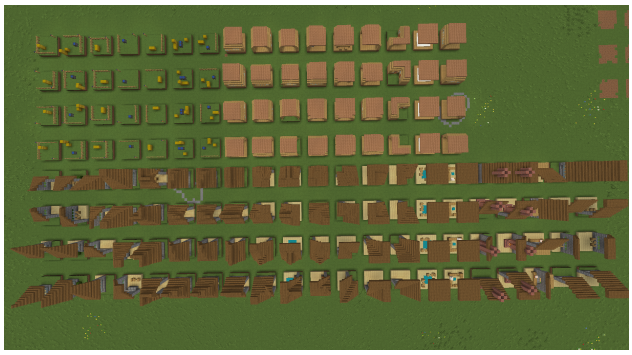

Figure 10. An overview of the tiles used for both the farm (top 4 rows) and bakery (bottom 4 rows) building types used by the wave function collapse algorithm. All tiles are 7 x 10 x 7.

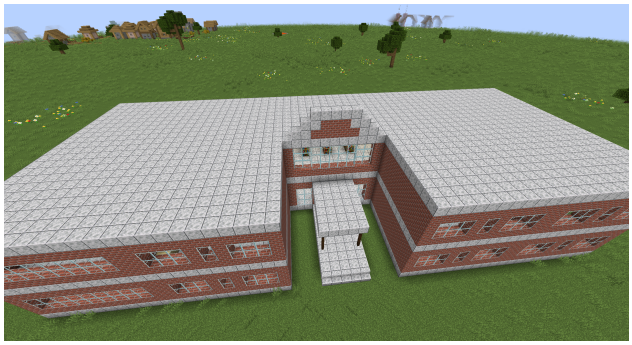

Figure 11. An example of what the school building type could look like from the outside.

Figure 12. An example of what the school building type could look like from the inside.

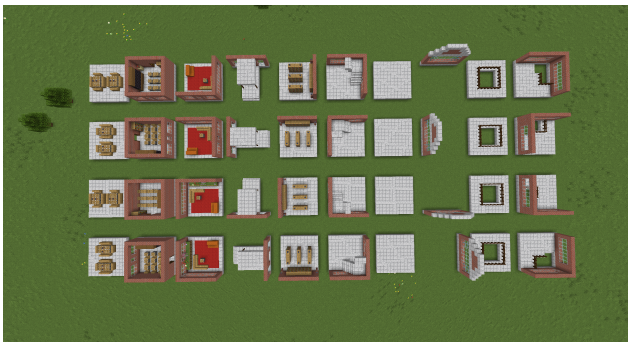

Figure 13. An overview of the tiles used for the school building type used by the wave function collapse algorithm. All tiles are 11 x 5 x 11.

3.3.1. A* ALGORITHM

A lot of pathing in video games is done with A*. That is why we tried this in the previous building assignment. What we found was that while it did produce optimal paths between houses (Desideratum 5), it failed to incorporate these paths into a network (Desideratum 2). We found that when using A* there were a lot of parallel paths running close to each other, which is sub optimal when laying a network of roads, as this requires the hypothetical builders to use a lot of pathing material. A* has no mechanism of modifying previously constructed paths, so paths that are now being constructed. Even if we gave the algorithm a reduced cost while building over previously laid paths, we could only adjust the latter paths to the former, which is an issue because for an optimal network, we would need to adapt all paths to each other.

3.3.2. ANT COLONY SIMULATION

Slime mold has been used to create pathing between multiple nodes (Tero et al., 2010). The paths created by this mold can bee seen in Figure 17. This however wasn't a computer simulation, but real slime mold. Slime mold simulations, when simplified come close to Ant colony simulation, of which the pseudo code is found in Algorithm 1. We theorised that by creating an ant colony simulation, we could give rise to natural looking paths. Empirically, variations of these simulations already inherently meet all required Desiderata, as we can also see in the real life counterpart in Figure 17.) The only Desiderata not inherently met would be Desiderata 1, the adjustment to the terrain. To incorporate this, we could make the ants evaporate less pheromones when on rough terrain, and more on smooth terrain, so we would naturally get paths that traverse through smoother terrain rather than rougher. For this we can re-use the smooth terrain evaluation array that we have constructed in subsection 3.1.

RESULTS

We constructed our own ant colony simulation and quickly discovered a major drawback. The drawback is that it makes

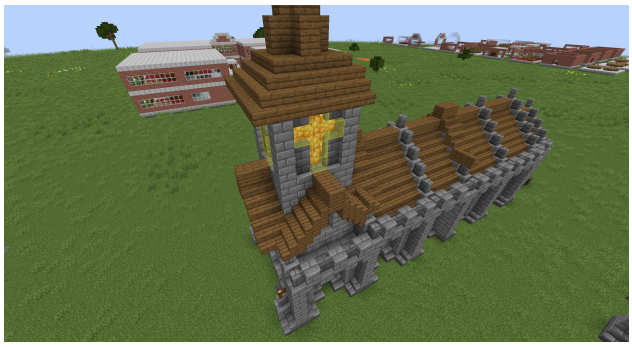

Figure 14. An example of what the church building type could look like from the outside.

Figure 15. An example of what the church building type could look like from the inside.

extensive use of Hyper-parameters. Our algorithm uses the following hyper parameters:

1. NUM_ANTS: The number of ants in the simulation.
2. FOOD_AMOUNT: The amount of times that ants can eat from each food source until it is depleted.
3. PHEROMONE_EVAPORATION: The rate at which pheromones diminish (Half-life of pheromones)
4. PHEROMONE_DECAY: The rate at which the pheromone production of an ant diminishes after finding food.
5. RANDOMNESS: the randomness of the ant (exploration versus exploitation)
6. SAME_DIR_MULTIPLIER: How much the ant prefers to walk in the direction he has been walking already (movement pattern adjuster.)
7. SLOPE_MODIFIER: How much the pheromones decay faster on rougher terrain.

Figure 16. An overview of the tiles used for the church building type used by the wave function collapse algorithm. All tiles are 11 x 25 x 11.

8. ITERATIONS: the number of iterations are algorithm simulates

All these hyper parameters change the outcome of the pathing behaviour drastically. We have experimented extensively with these hyper parameters, and have tried to tune them so that the simulation would result in natural looking paths. The bottleneck we face here is that there is no automated way to tune these hyper-parameters, as there exists no program that can confidently identify if all our desiderata are met. This results in having to tune the parameters solely by hand. Although we have found Interesting constellations, we have found none that consistently make natural paths. The problem in most cases was that we needed to keep the exploration/randomness parameter high to find all the food sources/buildings (Desiderata 4), but this did this result in paths that were to wide and sparse, like shown in Figure 18. No set of hyper parameter values could be found to produce paths that adhered to all desiderata, making the ant colony simulation intriguing, but unfit for our purpose.

Algorithm 1 Ant Colony Simulation

Initialize colony with N ants and M places
Initialize pheromone levels on each place with initial value zero
while simulation not terminated **do**
    for each ant i **do**
        Choose next step based on pheromone findings and movement patterns.
        if ant i finds food at place j **then**
            Go back home, and update pheromone levels with the quantity of pheromones released by ant i.
    Evaporate pheromone trails on each place
Output: Pheromone trails indicating high-traffic paths

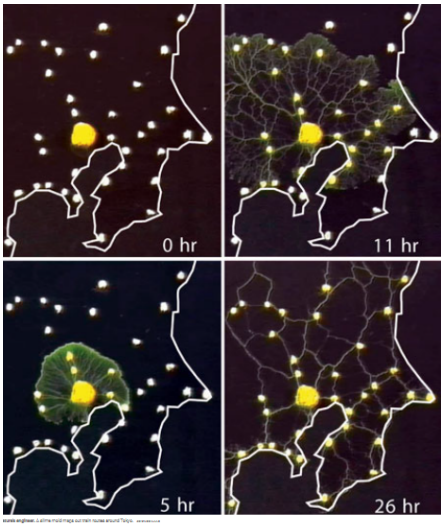

Figure 17. Here we can see a mold, that is placed on a pallet with multiple food sources. After 26 hours we can see natural and moderately optimal pathing.

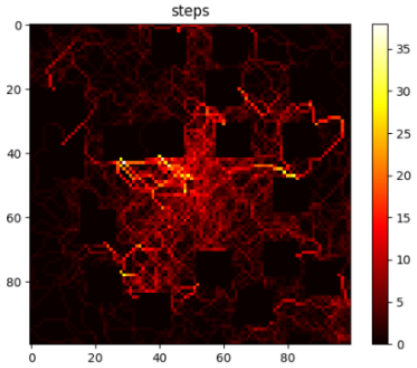

Figure 18. In this figure we have the pathing network made by our ants, even though it looks intriguing, and maybe even natural for ants, humans prefer more defined roads. The figure shows the total number of steps on each square. The black boxes (no steps) are the houses in our settlement.

3.3.3. Genetic algorithm.

We also looked at genetic algorithms (GA) as a solution for Adaptive natural pathing. Where individuals would be constellations of paths from house to house. By initialising an individual with straight paths, as can be seen in Figure 19, we already satisfied Desiderata 4 and 5, we could than evolve the paths to satisfy the other Desiderata. This would mean we needed a fitness function that punished building over houses (Desiderata 3), punished laying a path in rough terrain, and rewarded overlapping paths. To finish the GA we would need a mutation function that altered the paths slightly.

RESULTS

Our GA works moderately well when we run the algorithm on a couple of paths as seen in Figure 20. The paths do have sharp edges when changing directions, this is an artifact of the mutation algorithm combined with the fact that we also minimize for path length in our fitness function. We would need to add an extra smoothing reward. Smooth edges are however hard to define for an algorithm. We would prefer an algorithm that inherently creates smooth paths with rounded corners.

Smooth edges could be fixable with different path mutation functions, however the algorithm also suffers from the curse of dimensionality. GAs inherently do not suffer from this, as they have a set population of candidate solutions, and with enough time can search the multiple dimension spaces effectively. The reason why we face the curse here is that the chance for increasing a path becomes astronomically small when we mutate a lot of paths in a constellation. This is because a constellation is partly valued by the value of the individual paths. When mutating a path, the chance of worsening the score is greater than improving the score, thus when exponentially adding paths, we will find that our chance for finding a better constellation becomes shrinks exponentially. When running our GA with 18 buildings for 10.000 iterations, there was no single constellation in any iteration that scored better than the initial constellation. To avoid this unlikeliness to mutate to a better constellation , we would need to alter paths individually, and see these as individuals. This wouldn't work since it would make evaluating the pathing system as a connected whole impossible because there would be many versions of paths in the GA, as a GA normally only works with multiple individuals. Now we could adjust the fitness function, so we would evaluate each path, based on all other paths and their variants, but again this leaves us faced with the curse of dimensionality.

3.3.4. Water-like simulation.

As we could find no algorithm that satisfied all our Desideratum,so we came up with our own algorithm. The pseudo code for this algorithm is found in Algorithm 2. The algorithm is inspired by the dipole-dipole interaction found in water. When the dipole parts in water are close enough, they will attract the water molecules to form a whole. To keep the molecules/cells that are attracting each other still resembling a path, we add molecules in between the molecules, that drift too far apart in each path. Just like water the algorithm is also inclined to move through the path of least resistance (smooth terrain.)

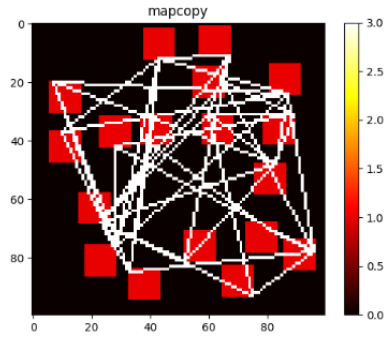

Figure 19. Here we have a starting constellation for our GA (and water-like simulation, each door draws a straight line to an n number of other randomly selected doors (In this figure n is set to 3).

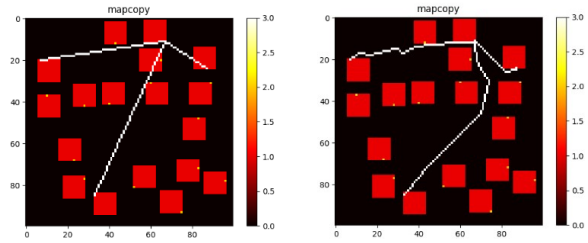

Figure 20. In sub-figure a we have a starting constellation for our GA with only 3 straight paths, after a thousand iterations of our GA we get an alright looking path in sub-figure b, which does avoids all houses.

Using this algorithm, we input information about where all our buildings are located, as well as the locations of where the doors are located in the buildings. The algorithm will then attempt to create paths that connect all the buildings within the region.

RESULTS

A sample result for a settlement can be found in Figure 21. Next, the information from the graph gets translated to the Minecraft world, where path blocks are placed at the positions of the white paths in the graph. The ingame result can be seen in Figure 22. As we can see in Figure 22, we found an algorithm that satisfies all our pre set desiderata, making it ideal for our purpose of building adaptive and natural looking paths.

3.4. Large Language Model

In our opinion, the fascination of video games is reliant to large parts on their ability to immerse the player in a fantasy world. While a believable game environment is crucial

Algorithm 2 Water Simulation for Natural Pathing

```
Initialize straight paths from building to building
while Not converged do
    for each path do
        for each cell in path do
            Calculate attraction to other paths and repulsion to rough terrains.
            Update to position of most relative attraction
            if path is not a continuous string anymore then
                Repair path by filling in gaps with new path cells
```

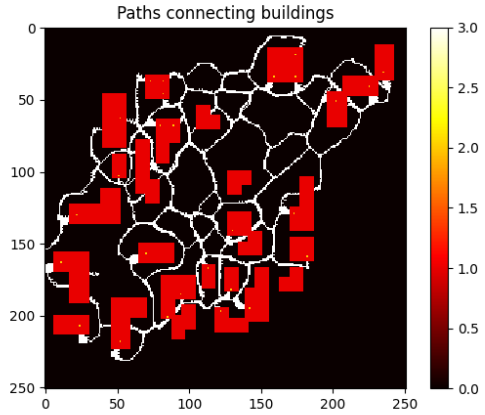

Figure 21. Graph of the results from the water simulation path algorithm on a random settlement.

Figure 22. Ingame representation of the paths generated by our water simulation algorithm.

for that, we think communication between the player and the game world has the potential to lift the immersion to a whole new level. In most games, communication between the player and the game world is achieved through manually written dialogues or narrations. Given the recent advancements in language modelling, we see a large potential to automate the generation of communicative interaction. To showcase that potential, we decided to incorporate the use of a large language model (LLM) in our settlement generation. We use this LLM to generate messages sent to a player upon entering a building. These messages are specific to the entered building to increase the immersion effect. In the following, we provide details on this functionality. We start by outlining which and how the LLM was used to generate narrations. Following that, we describe how we incorporate these narrations into our settlement.

3.4.1. Using a LLM to Generate Narrations

Following the rise in popularity of OpenAI's ChatGPT towards the end of 2022, a large number of LLMs have been published. The complexity and quality of these models varies significantly. For this project, we defined three criteria to ensure accessibility and replicability of our project:

- The LLM needs to be freely available.
- The LLM needs to be available for download and local execution.
- Using the LLM does not require any registration or key.

In addition to these specific criteria, we also want the LLM to not require special hardware. However, this criteria is not clearly defined. To ensure it nonetheless, we only considered LLM that successfully run on a Lenovo ThinkPad E470 without the use of a GPU. Following these criteria, we decided to use the GPT2-Large model available through HuggingFace's transformers package.

The chosen LLM is able to complete given prompts. Thus, to obtain narrations, we need to provide narration starts that are then completed by the LLM. Each of these starts should act as a template that can be changed according to the type of building for which a narration is to be generated. We manually define three such templates:

- *Welcome to the towns* {*building_type*}*! Here, you can*
- *You entered a* {*building_type*}*! This is a place where*
- *This is a* {*building_type*}*! In this* {*building_type*}*,*

Not the *building_type* variable which is substituted with the respective type of building (e.g. school or church) for which a narration is to be generated. To allow for more variation and to demonstrate that also the formulation of these narration templates can be automated, we decided to generate seven more templates with the use of OpenAI's free ChatGPT. We decided to use ChatGPT because of its few-shot learning abilities which allow it to generate similar templates given only a few examples. Further, the quality of the templates is crucial for the quality of the narrations and ChatGPT is one of the current state-of-the-art models for language generation tasks. For specifics on ChatGPT's use for template generation, see Appendix B. The seven narration templates produced by ChatGPT are:

- *Welcome to our* {*building_type*}*! Here, you'll find*
- *Step inside the* {*building_type*} *and discover*
- *Explore the wonders of the* {*building_type*}*! From*
- *Have you ever been to a* {*building_type*} *like this? It offers*
- *Behold the magnificent* {*building_type*}*! It's a haven for*
- *Immerse yourself in the enchantment of the* {*building_type*} *and experience*
- *Prepare to be amazed by the* {*building_type*}*! With its*

Given these templates, we use the GPT2 model to complete each template 3 times for each building block. We set GPT2's temperature to 0.95 and allow it to sample in order to be able to produce different narrations given the same template. Following this procedure, we obtain up to 30 narrations per building type, the exact number can differ if a complete narration is removed during post-processing.

Inspecting initial results, we observed two major problems with the generated narrations. The first problem is narrations were often incomplete and stopped in the middle of a sentence. This is likely due to the fact that we set GPT2's max length parameter to 50. However, simply extending the limit would not solve the problem but only result in longer narrations that end with an incomplete sentence. The second problem is that narrations tended to be repetitive in that from a certain point onward, the same sentence was repeated over and over again until the end. To improve the narration quality, we added a post-processing procedure that aims to solve these problems. To do so, it splits up narrations into their individual sentences, removes sentences that are not finished or that already occurred earlier in the narration, and then joins the sentences back together to form the final narration. While simple, we found this post-processing to vastly improve the narration quality. Using the generation and post-processing procedure, we produce the generations ahead of the settlement generation to shorten the time of the actual settlement generation.

3.4.2. Incorporating Narrations in the Settlement

The narration generation is implemented in Python. However, it is not possible to execute a Python script from within Minecraft and, therefore, not possible to generate and display narrations whenever a player enters a building. We overcome this issue by following a specific workflow for the narration generation and incorporation. We start by generating all narrations ahead of the settlement generation and storing them in a JSON file. During settlement generation, every time a building is generated a fitting narration is sampled. The narration is then put integrated into a command of the form "msg @p narration". This command is then inserted into a command block which is placed next to the buildings entry and a pressure plate is placed on top. Now, when a player enters a building and walks over this pressure plate, the narration is sent as a message to the player. An example of this can be seen in

Figure 23. Example of a narration block at the entrance of a villager house. As soon as the player enters the building, stepping over the pressure plate, a short information message is sent to him.

4. Results

After combining all the parts, we managed to generate a number of settlements that fit well within the world. First we will take a look at a settlement that is located in a grass-like biome. A top down view of this settlement can be found in Figure 24. Here we can clearly see that the building location selector manages to select a diverse set of locations for placing down the houses. All the locations selected are also on flat ground, meaning that the terrain does not have to be altered much in order to place the buildings. This can be seen more clearly in Figure 25 and Figure 26, which show some close-ups of the settlement.

Figure 24. Top-down view of the settlement generated in a grass-like biome.

Figure 25. Alternate angle of the settlement generated in a grass-like biome.

Next, we built a settlement in a desert-like biome. This area offers a much more challenging environment, because there are a lot less flat areas and there is a lot of water within the area. The resulting settlement can be found in Figure 27. Here we can clearly see the effect that the rough terrain has on the settlement generation. The algorithm manages to find less places to put the homes, so overall the settlement will be a lot sparser. We also see that, while the paths run through the water, this is not a problem. When the part of our system that places paths finds it is placing above water, it replaces the gravel normally used for a path with a wooden slab that symbolizes a bridge. Some additional close-ups of this settlement can be seen in Figure 28 and Figure 29.

5. Future Work

In this paper, we demonstrated the application of PCG for generating complete settlements within Minecraft. Our research culminated in the development of an algorithm capable of producing diverse structures such as brickhouses, bakeries, farms, churches, and schools, all connected by a pathway. Additionally, we integrated a LLM to provide narratives about significant buildings whenever a player

Figure 26. Another angle of the settlement generated in a grass-like biome.

Figure 27. Top-down view of the settlement generated in a desert-like biome.

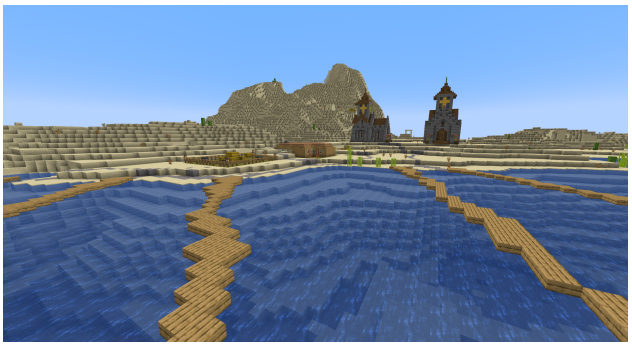

Figure 28. Alternate angle of the settlement generated in a desert-like biome.

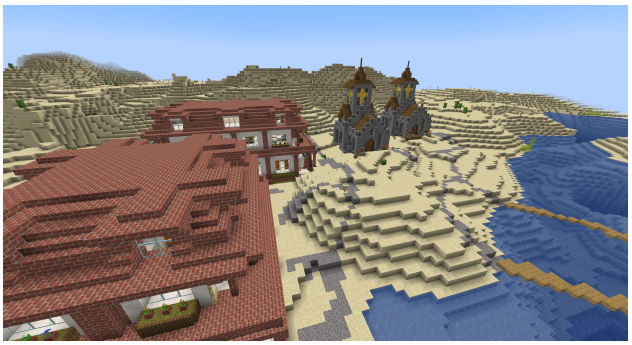

Figure 29. Another angle of the settlement generated in a desert-like biome.

explores the settlement. Despite this, there are some limitations to the settlement being generated.

To begin with, during the implementation of the WFC algorithm, blocks are collapsed in a manner that can result in tiles being placed with empty spaces in between. While the individual houses remain appropriately enclosed, there is a possibility of encountering scenarios where a single building location contains multiple buildings of the same type (e.g. church) instead of a single larger building. Furthermore, there exists a minor concern within the algorithm employed to guarantee specific tiles. Presently, if certain buildings necessitate particular tiles, such as an altar in a church or an entrance in other buildings, we employ a forget and retry technique until these requirements are satisfied. However, certain buildings currently have a condition that permits only one instance of a specific tile within the structure. Consequently, if multiple disjointed buildings are constructed within a single building area, as mentioned previously, the second building may lack this mandatory tile.

One final addition to the settlement that we did not get around to implementing was the spawning of different entities at the buildings. This could for instance be villagers who occupy the houses, or animal that populate the farm. This would add an additional level of believability to the settlement, and would therefore be a great addition in the future.

Our use of a LLM for making the settlement more immersive is merely a proof of concept. There are many limitations to our approach, e.g., the fact that the narration for a certain building always remains the same and that the narrations are not influenced by the generated settlement but only dependent on the building type. Further, the generated narrations still show issues regarding repetitions within sentences that are not accounted for by our post-processing. Nonetheless, we see our approach as a powerful demonstration of the capabilities that LLMs have in regards to providing an immersive gaming experience. Considering that we are using GPT2, a very small and limited LLM, and are still able to obtain reasonably good results indicates what might be possible using state-of-the-art systems. We are confident that switching to better LLMs will solve issues regarding the quality of narrations. However, we see the biggest potential improvement in developing a plug-in that allows to use a Python script for language generation from within Minecraft.

This would enable dynamic text generation, which would allow for interactive communication with non-playable characters (NPC) and text generation that is influenced by player behaviour or changes in the game world. Without such a plugin, using LLMs for settlement generation is limited to static generations. However, we also see further potential in the use of static text generation. It could be used to generate texts for books or stories about the settlement that the player could explore. Further, it could be used to name streets, houses, NPCs and the settlement itself. For all these tasks, it would be beneficial for the text quality if the LLM could be conditioned on the generated settlement such that it takes the settlement into account when generating these texts. While out of scope for this project, we see big potential improvements in such a conditioning of LLM generations. Paired together with a state-of-the-art model, we believe future works could use this approach to achieve impressing results even without dynamic generation. For now, we have shown the potential even simple LLMs have for PCG that is aimed at making gaming more immersive through language.

6. Conclusion

This paper presents our approach to using PCG to generate settlements in Minecraft. Our methodology involved combining various techniques to achieve the desired outcome. Initially, we utilized a 2-dimensional slope array to identify optimal building locations within the Minecraft world. Subsequently, we employed the WFC algorithm to generate diverse building types on these identified spots. By utilizing this algorithm, we were able to adapt the buildings to the terrain and ensure randomization, resulting in unique structures for each building. Additionally, we experimented with different algorithms and ultimately incorporated a water-like simulation algorithm to create a realistic path connecting all the generated houses. To enhance the immersive experience, we introduced a narrative element through the implementation of an LLM. When users enter a building, a pressure-plate is activate to trigger the display of an interesting story related to the building in their text box. This addition aimed to satisfy the evaluation criteria of the GDMC competition, which assesses submissions based on criteria such as believable narrative, adaptability, visual aesthetics, and functionality.

Despite our achievements, there are still some issues within the settlement that require attention before the competition deadline. Specifically, adjustments need to be made to the WFC algorithm to prevent disjointed building placement in the same location. Furthermore, to enhance believability, we intend to introduce villagers and animals into the generated world.

In summary, this project successfully amalgamated a variety of techniques to produce impressive settlements in Minecraft. The combined efforts resulted in a final product that meets the requirements of the competition while offering a visually appealing and functional experience. We publish all code open-source on GitHub[4].

References

Salge, C., Green, M., Canaan, R., and Togelius, J. Generative design in minecraft (GDMC): settlement generation competition. pp. 1–10, August 2018. doi: 10.1145/3235765.3235814.

Stein, T. Mgaia-minecraft. https://github.com/ScholliYT/MGAIA-Minecraft, 2023.

Tero, A., Takagi, S., Saigusa, T., Ito, K., Bebber, D., Fricker, M., Yumiki, K., Kobayashi, R., and Nakagaki, T. Rules for biologically inspired adaptive network design. *Science (New York, N.Y.)*, 327:439–42, 01 2010. doi: 10.1126/science.1177894.

A. Connection to the MGAIA Lecture

The first connection to the lecture we were interested in implementing had to do with optimizing the paths created between the houses. Our initial idea was to use Ant Colony optimization, which was mentioned in lecture 2. This however, did not turn out to work for our case, so we adapted it into a water-like pathfinding algorithm as described in subsection 3.3.

We took inspiration from the third lecture covering Procedural Content Generation (PCG) to use the Wave Function Collapse (WFC) algorithm that was mentioned in the lecture. This was used to generate the buildings that populate the settlement.

We wanted to go beyond an empty settlement and allow for interaction between the player and its environment. However, we did not want to manually code interaction but instead use generative AI to do that for us. This inspiration mainly stemmed from the last lecture where we discussed the potential of generative AI in the context of game development, specifically mentioning the usage and testing of language models in games. We saw the recent developments in LLMs as a great motivation to explore this use of generative language models to create an immersive and interactive settlement. One of the main challenges was to make the generations believable which is a topic we discussed in lecture 9 and which we addressed through prompting generation with hand-crafted prompt templates and a post-processing procedure.

[4] https://github.com/ScholliYT/MGAIA-Minecraft-GDMC

B. Use of ChatGPT for Narration Template Generation

As described, we used OpenAI's ChatGPT to produce more narration templates. Specifically, we used the May 24 2023 free research preview of ChatGPT. We started a new chat and obtained the new narration templates by providing the following prompt: