The use of Poisson Disc Distribution and A* Pathfinding for Procedural Content Generation in Minecraft

Cristopher Yates

Department of Computer Science
Memorial University

Supervised by Dr. David Churchill

A dissertation submitted to the Department of Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science (Honours) in Computer Science

April 2021

Abstract

With very few strict guidelines on how the game must be played and worlds that are easily manipulated, Minecraft is almost perfectly suited to make use of procedural content generation (PCG). The Generative Design in Minecraft (GDMC) competition presents an interesting challenge by requiring participants to create systems that generate settlements into an existing Minecraft world, and then subjectively evaluating them on four separate criteria. Since there is no one correct solution, the problem is much more difficult than it initially appears. In this thesis, we will propose the use of Poisson Disc Distribution to place the buildings pseudo-randomly and A* pathfinding to create paths between them and show the results of applying them in randomly generated Minecraft worlds.

Acknowledgments

Thank you to my supervisor, Dr. David Churchill, for his advice and guidance during this whole process, especially in the final few weeks.

I would also like to thank my friend and classmate James Hudson for helping me design most of the structures used in this project.

Finally, I would like to thank my parents for supporting me as I was forced to complete the past three semesters from home due to the COVID-19 pandemic, as well as for raising me as well as they have in the first place.

Table of Contents

List of Figures

Chapter 1:

Introduction

Minecraft is an open-world sandbox game that allows players to break and place blocks to shape the world how they please, while at the same time not confining them to following a predetermined narrative.

Procedural content generation (PCG) is the process of video games algorithmically creating their own content. Perhaps the most famous example of this is the world generation in Minecraft itself, but it is also used in creating everything from procedural dungeons in many roguelike games such as The Binding of Isaac, to procedural guns in Borderlands 3, to supposedly an entire 3D universe in No Man's Sky.

The Generative Design in Minecraft (GDMC) competition is a programming competition in which participants are asked to design and develop an algorithm that generates a settlement in an existing Minecraft world. To determine a winner each submission is judged by several independent judges in four separate categories and then all of the scores are averaged together to determine an overall score. What makes this a difficult and interesting challenge is that each of the four aspects that the submissions are judged on are completely subjective. As a result of this, there is no one "correct" solution

to the problem and many participants over the years have tried to maximize their scores in a multitude of different ways.

Some common features of GDMC entries include semi-random placement of houses and other buildings, roads or paths that connect them, and walls that encircle the settlement in its entirety to protect against the dangers of the outside world.

For our settlement, we want the following: multiple types of houses that are placed pseudo-randomly, paths that connect the houses to each other as well as other buildings, several unique structures that are only placed once per settlement, some way to explain how the former inhabitants got the food they needed to survive and the materials to construct the settlement, and some simulated inhabitants so that we can generate patterns specific to the various families (family crests) and place them in the houses. The placement of the houses will be determined by Poisson Disc Distribution. To determine which buildings to make paths between we will make a spanning tree using the entrances of the buildings as the nodes, and to actually determine the blocks that will comprise the path we will use A* pathfinding. We will have a structure that contains an abundance of farmland with crops already planted to explain the inhabitants' food situation. Since we want to be able to use certain materials that are only available in the Nether, we will also have a structure that contains a Nether Portal. With regards to the simulating of inhabitants, we will create a custom algorithm that will simulate things like marriage and childbirth.

Chapter 2:

Background

2.1 Minecraft

Figure 2-1: Minecraft being played in Survival Mode

Minecraft is an open-world sandbox game that can be played alone or with others through the use of multiplayer servers. All of the actual gameplay in Minecraft takes place within distinct and separate “worlds” that are procedurally generated from random seeds when they are initially created and continue to generate during gameplay as the player(s) explore.

A Minecraft world is primarily made up of three distinct components: blocks, entities, and tile entities. Blocks are the traditional static blocks that normally do not directly interact with the player outside of collisions. Entities are all of the dynamic, moving objects and creatures in a Minecraft world. Tile Entities are simplified entities that are bound to blocks, and although not usually changing their appearance, they do provide their block with some additional functionality.

2.2 Procedural Content Generation (PCG)

PCG is the process by which some video games generate content algorithmically as opposed to the developers designing it all manually This process usually involves some sort of randomness so that the experience can be different for each player/play session.

2.3 The Generative Design in Minecraft (GDMC) Competition

The GDMC Competition is a programming competition in which participants/teams are asked to develop algorithms to procedurally generate settlements in existing Minecraft worlds (what defines a settlement is up to the participants). Each submission is judged based on four criteria: adaptability, functionality, narrative, and aesthetic:

Adaptability: How well the settlement adapts to the landscape, this includes actual structures respecting the terrain as well as the structures themselves being made up of materials that are readily available at the location of the settlement.

Functionality: A measure of how well a settlement would be able to meet the needs of an in-game player or its hypothetical former inhabitants.

Narrative: How much of a story or history is woven into the settlement.

Aesthetics: How much the buildings look like they could be actual buildings, and how well the visuals of the settlement reflect the narrative that it is trying to convey (for example, if it's trying to be a city and it looks like a city the score would be high).

The way in which we will be implementing this algorithm is by writing a filter for MCEdit.

2.4 MCEdit

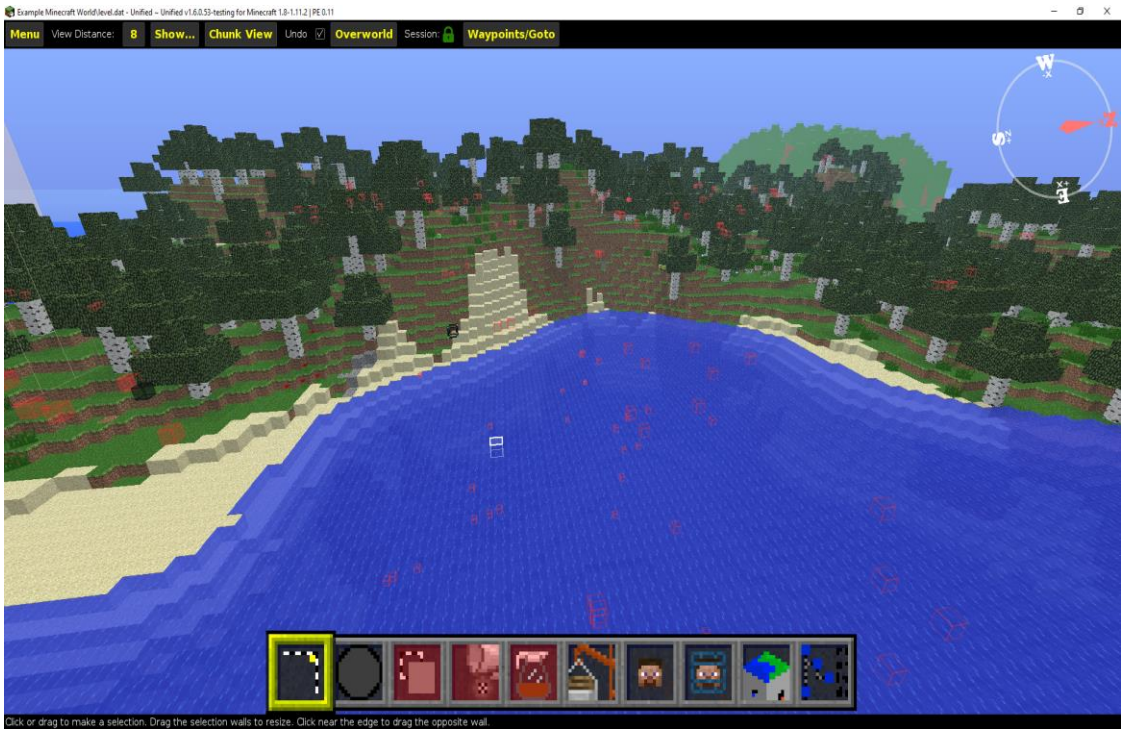

Figure 2-2: The MCEdit Graphical User Interface (GUI)

MCEdit is a third-party tool used to edit Minecraft maps outside of gameplay. Some things that can be done with MCEdit are normally impossible within the game. The two functionalities that we are concerned with are running filters and importing schematics.

2.4.1 Filters

Filters are python scripts that can be run on specified areas of the map to change that portion of the map in some way.

2.4.2 PyMCLevel

PyMCLevel is a python library for reading and editing Minecraft worlds. It is usually used through the GUI of MCEdit, however, through the use of filters, it can just be imported and used directly.

2.4.3 Schematics

Schematic files are files that store the NBT data for a specific section of a Minecraft world. While they are mainly used to transfer structures from one world to another, they can also contain entities and tile entities (allowing for the transfer of in-game items through chests and other containers).

2.5 A* Pathfinding

2.5.1 Pathfinding Algorithms

A pathfinding algorithm is an algorithm that finds a path from one point to another (most often in a 2D grid). In the case of the settlement generation problem, a pathfinding algorithm will be used to find the shortest path from the entrance of one structure to the entrance of another. Once found, these paths will then be shown in-game by the block at ground level (as defined by the heightmap) being changed to the “path” block.

2.5.2 A* Pathfinding

A* is a popular pathfinding algorithm that makes use of heuristics. It represents positions on a grid with nodes that contain the position as well as a reference to its parent node, and

its G and H values. The “G value” is the sum of the G value of its parent node and the cost of the action to get to a node from the parent node, and the “H value” is an estimate of the distance from the node to the goal (expressed through one of many different heuristic functions). The sum of G and H values is referred to as the “F value”.

The algorithm starts by keeping track of a list of both open and closed nodes (both of which would be empty at the start). A node representing the starting position is then added to the open list. The main loop of the algorithm then commences. First, a check is done to see if the open list is empty, and if so, the algorithm concludes with no path being found. After that, the node in the open list that has the lowest F value is removed from the list and stored in a variable. A check is then done to see if that node is the goal node. If it is the algorithm returns the position of the node, and the position of its parent node, so on and so forth until the start node. These positions collectively form the path from the start to the goal. If the node was not found to be the goal, a check is done to see if the node stored in the variable is currently in the closed list and returns to the start of the loop if so. The algorithm then checks to see if any of the adjacent nodes are already in the closed list, and if not adds them to the open list (with appropriate G, H, and F values). Following this, the node that is currently stored in the variable is added to the closed list and the loop begins again.

The pseudocode that the implementation of A* used in this project is adapted from is presented here:

Algorithm 1 A*

```
function A*(problem, h)
    closed = {}
    open = PriorityQueue(Node(problem.initial_state), f=g+h)
    while (true)
        if (open.empty) return fail
        node = pop_min_f(open)
        if (node.state is goal) return solution
        if (node.state in closed) continue
        closed.add(node.state)
        for c in Expand(node, problem)
            open.add(c)
```

2.6 Poisson Disc Distribution

Poisson disc distribution is an algorithm that populates a grid with random points, while ensuring those points are at least a certain distance from other points but not too far away either. The algorithm takes as inputs a domain in which we are trying to place the points, the number of dimensions (n) of the grid (we will assume 2D), the radius (r) that the points will be at least r apart but not 2r apart, and also a constant (k) that will be the number of samples to choose before "rejection". The algorithm for finding those points is as follows: First, calculate the size of a given grid cell with the formula r/n. After that, make an n-dimensional array where the sizes are the sizes of that dimension in the domain divided by the grid cell size. Once you have the grid, choose a point at random from somewhere in the domain, and put it in its appropriate place in the grid as well. Following that, create an "active" list and add the point that you just chose to it. Then if the active list contains at least one point pick a random index from the active list and try to place up to k points at distances from r to 2r away from the point at that index. Use the grid to check the distances from these new points to any other points on the grid and place the ones that are not too close to anything else in the grid. If a point gets placed on the grid add it to the active list,

and after trying to place k points close to the point at the specified index remove it from the active list and move on to another index. See pseudocode for a 2D implementation below:

Algorithm 2 Poisson Disc Distribution

```
function PoissonDiscDistribution(problem)
    cellSize = r/sqrt(n)
    grid = Grid2D(problem.width / cellSize, problem.height / cellSize)
    grid.fill(null)
    point = Vec2(random.float * problem.width, random.float * problem.height)
    grid[point.x / cellSize][point.y / cellSize] = point
    active = []
    active.append(point)
    while (active.length > 0)
        randIndex = random.float * active.length
        currentPoint = active[randIndex]
        for i in range(problem.k)
            angle = random.floatInRange(0, 2*PI)
            dist = random.floatInRange(problem.r, problem.r * 2)
            offsetX = cos(angle) * dist
            offsetY = sin(angle) * dist
            possiblePoint = Vec2(currentPoint.x + offsetX, currentPoint.y + offsetY)
            gridPos = Vec2(possiblePoint.x / w, possiblePoint.y / w)
            notTooClose = true
            for x in range(-1, 2)
                for y in range(-1, 2)
                    nGPos = Vec2(gridPos.x + x, gridPos.y + y)
                        if (grid.isInBounds(nGPos) and grid[nGPos.x][nGPos.y] != null)
                        if (possiblePoint.dist(grid[nGPos.x][nGPos.y]) < problem.r)
                            notTooClose = false
            if (notTooClose)
                active.append(possiblePoint)
                grid[gridPos.x][gridPos.y] = possiblePoint
        active.pop(randIndex)
    return grid
```

2.7 Spanning Trees

Spanning trees are a subset of undirected vertex-and-edge-based graphs in which all vertices are connected using the minimum number of edges. This means that no cycles can exist within the graph. The algorithm used to get a spanning tree in this project is as follows:

Start with a list of all vertices (including their positions on a 2D grid) and begin to loop through them. For each vertex create an edge connecting it to the vertex that is nearest to it that has yet to be checked. See pseudocode below:

Algorithm 3 Spanning Tree Algorithm

```
function spanningTree(problem)
        vertexList = problem.vertices
        edgeList = {}
        for i in range(vertexList.length)
                minDistance = infinity
                closestVertex = None
                for j in range(i + 1, vertexList.length)
                        dist = vertexList[i].dist(vertexList[j])
                        if (dist < minDist)
                                minDist = dist
                                closestVertex = vertexList[j]
                edgeList.add(Edge(vertexList[i], closestVertex))
        return edgeList
```

Chapter 3:

Methodology

3.1 Heightmap

Since this project mostly involves generating a settlement within a pre-generated world, one of the more important bits of data that we need is a height map. Most of the structures that are generated as part of the settlement are placed on top of the terrain that already exists, so knowing the height of the terrain at those locations is essential. There is a function in PyMCLevel that gets a heightmap for a given volume of blocks. There is also functionality that lets you apply masks of certain block types to those volumes before getting the heightmap. So first we get a chunk of blocks that are in the selection, then we apply a mask so that we are ignoring any blocks that we do not consider part of the terrain (Wood blocks from trees for example). After that, we get the heightmap from that volume and repeat that process for all chunks that are included in the selection. In the end, we append all of the chunk heightmaps together, and then we are left with a complete heightmap of the selection. This process is done twice in parallel, once for a heightmap including water, and once for one not including water. This is because in some cases we want things to rest on top of the water, while in others we want them on the ground beneath.

3.2 Poisson Disc Distribution

The next stage is getting the positions for the buildings to be placed. This is done through the use of Poisson Disc Distribution. We pass in the dimensions of the selection and the distance that we would like between the buildings, run the algorithm and get a collection of points. We then place buildings at each of the points that are returned from the algorithm and record the positions of the entrances of each of those buildings. Also, when placing the buildings, we add “walls” to the pathfinding grid in the shape of the footprints of the buildings to make sure that no paths end up running through the buildings.

3.3 Paths

After that paths are created between the entrances of the buildings. These paths are made by finding a path from one entrance to another using A* pathfinding and changing the blocks at ground level at the x and z positions (and adjacent positions) of the found path. We decide which buildings to build paths from by constructing a spanning tree using the entrance positions as the vertices. We also place streetlights two blocks away from each actual path position, one in each direction that would be perpendicular to the direction of the path itself.

3.4 Family Simulation

One of the main features of this project is the simulation of former inhabitants of the settlement. The simulation works by starting with 2 people in each of n families and then simulating marriage and childbirth for 3 generations. There are no genders associated with the simulated inhabitants and as a result, the way marriage is simulated needed to be altered slightly from the traditional way that it happens in the real world. A person will approach

a random individual from a random family and if the other person's family isn't the same as theirs and the other person isn't married yet, they get married. Otherwise, the person asking gets "rejected" and after 10 consecutive rejections, the person stops trying to get married. If a marriage occurs the person who asks is considered the dominant parent and the other person takes their last name and joins their family. With regards to childbirth, each married couple has 1-3 children who get added to the next generation of their family.

3.4.1 Cemetery

There is a cemetery structure constructed in one of the corners of the settlement. There is a tombstone created for every simulated person of a previous generation (before the third simulated generation) in this cemetery. Each tombstone has a sign on the front detailing the name of the person, their dominant parent, and their first child, as well as a banner of the family's crest above it.

Figure 3-1: A tombstone in a generated cemetery

3.4.2 Flags and Indoor Banners

Each house is assigned an inhabitant and has a flag erected somewhere on the house that shows their family crest sideways. There is also at least one banner with the inhabitant's family crest somewhere in every house.

Figure 3-2: A matching flag and banner attached to a house

3.5 Importing Schematics

The main way structures get placed in the Minecraft world is by importing schematics constructed beforehand and saved as separate files. The importing of schematics is done through the use of a specific function from PyMCLevel. It takes the position that the structure will be placed, the path to the schematic file, and the bounds of the schematic (the dimensions, in blocks, of the structure that the file contains) as parameters and places the structure in the world when run.

3.6 Building-Specific Randomization

Some buildings have additional aspects randomized to further differentiate them from each other. All of the houses have the color of their beds and the carpets immediately surrounding them randomized (the bed and the carpets are the same color, but that color is random). Some specific houses also have flower beds on the outer walls and the flowers within them are also randomized.

Figure 3-3: Randomized flowers on the exterior of a house and randomized bed color visible through the window

3.7 Unique Structures

Four unique structures get placed in effectively the same place every time, and they all serve to try and increase the project's score in the narrative category. Three of those structures are placed in corners of the settlement. The unique structures include the cemetery, the Nether portal building, the farming building, and the central shrine.

Cemetery: See 3.4.1.

Nether Portal Building: A building that houses a "Nether Portal". A Nether Portal is a portal to a different section of a Minecraft world that is not accessible otherwise. Some certain items and materials can only be obtained in this separate section of the world, and some of those materials are used in the construction of the structures

in the settlement. Therefore, this structure exists to try and increase the narrative score by providing an in-world explanation for how those materials were obtained.

Farming Building: A building that contains multiple plots of farmland. This tries to increase the narrative score by providing an in-world explanation of where the inhabitants of the settlement got food.

The Central Shrine: Essentially a fancy enclosure for a single chest. Inside this chest is a written book that contains an in-world explanation for the pillars which connect the buildings to the ground.

Both the farming building and the Nether Portal building also serve to increase the functionality score of the project by giving any hypothetical players that would encounter the settlement access to the resources they contain.

3.8 Final Details

The final details added to the settlement are the pillars that connect the buildings to the ground. This was mainly done because with the Poisson Disc Distribution, the positions of the buildings are decided without considering the heights of the terrain. This results in buildings floating above the ground in some places if we place them at the height of the highest block that is beneath them and leaving unsightly chunks taken out of the surrounding ground if we chose to embed them in it (this occurs because all schematics are treated as cubes and as a result the “air” that is in the corners of the building schematics will also be placed). A byproduct of this pillar system is that the entrances of the buildings end up being several blocks above the ground and are therefore inaccessible to the players

and the hypothetical past inhabitants. To account for this there are ladders that extend down from the entrances of buildings to ground level.

Chapter 4:

Results

4.1 Screenshots

Figure 4-1: Several screenshots of a sample settlement

4.2 Expectations

When beginning this project, I expected to have far simpler structures. I thought that I wouldn't even use schematics, but instead place every block of every structure one by one, and in doing so allow the buildings to somehow adapt to their structure to their surroundings. I also expected to have some system in place that would change the blocks that the buildings were made of into blocks found in the biome they are located in. The paths turned out the same as I had expected, using the path blocks and having some sort of streetlight on either side. I had no thoughts of inhabitants or simulating families at the beginning, it wasn't until after the houses and paths were complete that I even considered those as possibilities. I also expected the buildings themselves to be placed entirely on the ground, but due to my lack of any ideas as to how to accomplish that (without heavily deforming the terrain) I ended up putting all of the buildings on pillars.

4.3 Experiments and Results

4.3.1 House Placement

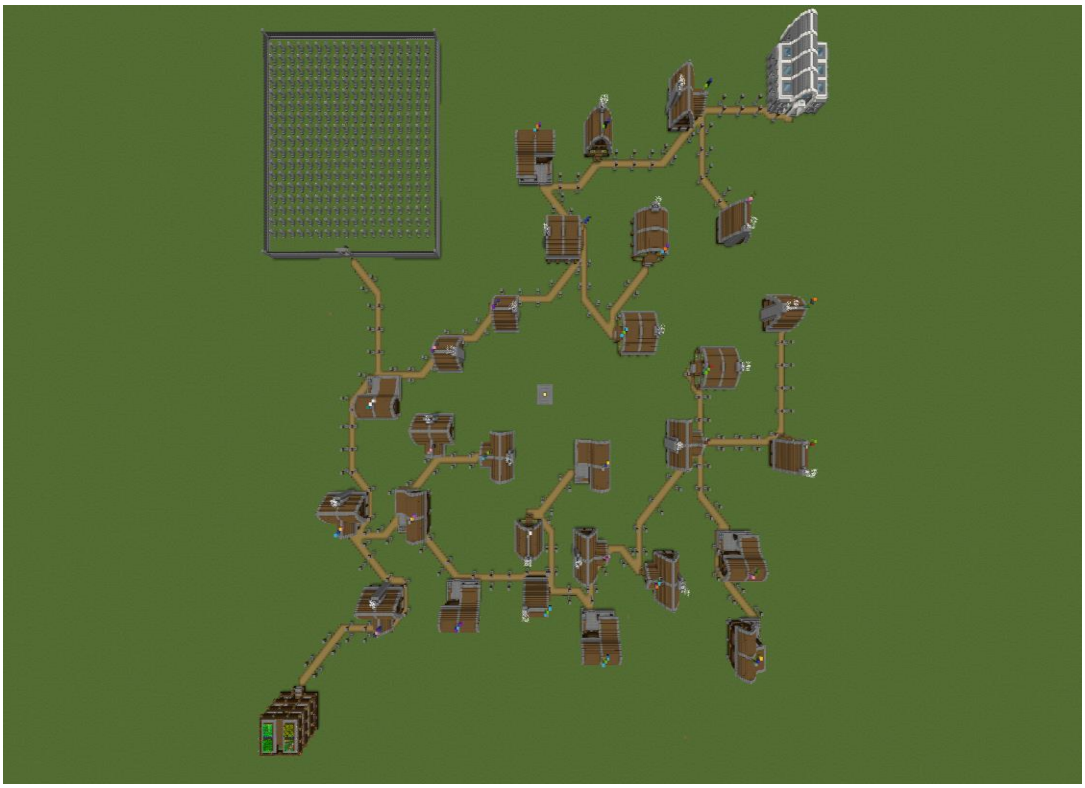

Figure 4-2: A settlement generated on a flat Minecraft world viewed from above

The determining factor for whether the houses were placed "correctly" is if there is enough space between the structures for the paths to be generated correctly. As we can see in the figure above all of the paths connecting the buildings go under any of the buildings or intersect with each other. Also, the fact that the generation completed at all implies that paths could be found between the entrances that needed to be connected.

4.3.2 Path Placement

By looking back to figure 4-2 we can see that none of the paths intersect. This shows that the correct buildings were chosen to be connected.

4.3.3 Path Generation

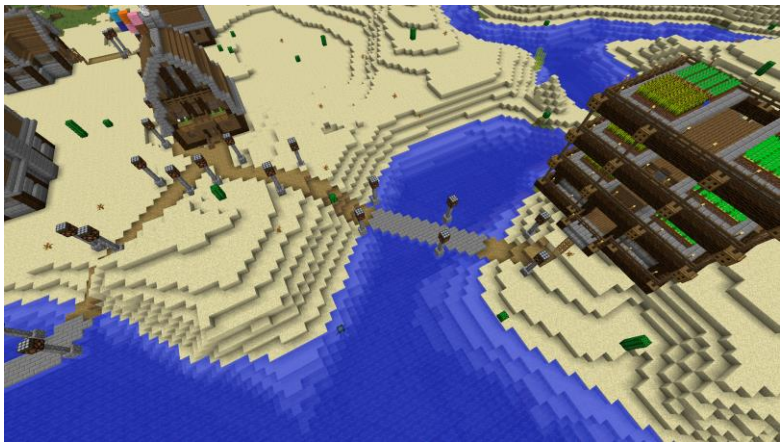

Figure 4-3: A path in a generated settlement that crosses a river

In figure 4-3 we see that the paths generate normally on the sand and turn to stone bricks when over water. We also see that there are streetlights placed every so often on the sides of the paths. Finally, if we look back to figure 4-2 we see that every structure that should be attached to a path is connected (the one structure that is not supposed to be connected to a path is the central shrine and it is not shown to be connected in the figure).

4.3.4 Building-Specific Randomizations

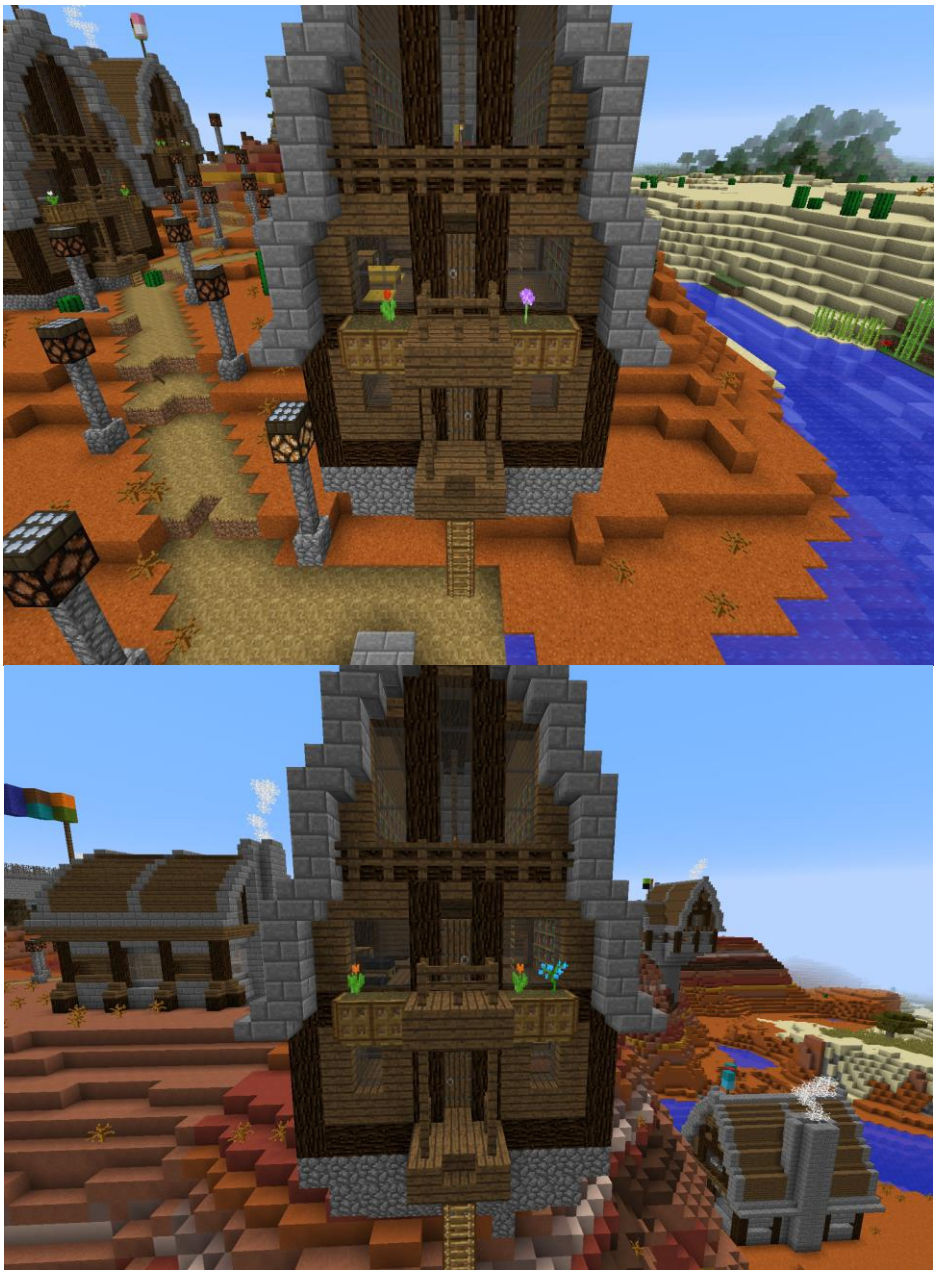

Figure 4-4: Two houses of the same type with different building-specific randomizations

In figure 4-4 we see images of two separate houses of the same type in the same settlement. We see that the flowers in the flowerbeds are different, and the color of the beds and carpets visible through the windows differ as well.

4.3.5 Cemetery Generation

Figure 4-5: Three tombstones in a generated cemetery showing three generations of a family

In figure 4-5 we can see that the cemetery has been generated correctly because we can trace a family line through three generations. The reason that the last name and banner of the last person differ from those of the first two is that that person has married into another family.

4.3.6 Flag and Indoor Banner Generation

Figure 4-6: A house with a matching flag and indoor banner

In figure 4-6 we can see that the flag on top of the house matches the banner inside the house visible through the middle window. If we look back to figure 4-5 we see that the pattern on the flag and the banner of this house matches the banner on the tombstone in the bottom image indicating that the pattern is that of a family that is present within the settlement.

4.3.7 Unique Structure Presence

Figure 4-7: A generated settlement viewed from above, showing all unique structures

In figure 4-7 we see all four unique structures are indeed present in the settlement. We can also look back to figure 4-2 and see this more clearly.

4.3.8 Pillar Generation

Figure 4-8: Multiple buildings connected to the ground with pillars and one not

In figure 4-8 we see multiple buildings that are connected to the ground with pillars. There is one building on the left side of the image that has been generated on flat ground and it does not appear to have a pillar connecting it to the ground.

4.3.9 Ladder Generation

Figure 4-9: Two buildings with ladders connecting their entrances to the ground

In figure 4-9 we see that both visible buildings have ladders connecting their respective entrances to the ground.

Chapter 5:

Conclusion

We believe that we were successful in generating a settlement. The Poisson Disc Distribution generated points that were appropriate to place buildings on, and the spanning-tree made it so that none of the paths intersected each other, while A* generated the paths themselves correctly. The cemetery was generated correctly and so were the flags and indoor banners, which all collectively indicates that the family simulation was successful. The building-specific randomizations were all sufficiently random, and all of the buildings that were not completely attached to the ground had pillars and ladders generate appropriately. As a result of all of these things as well as the fact that all four of the unique structures are always present, I believe that the project was a success.

5.1 Future Work

The main purpose of this project was to create a competitive submission for the GDMC competition. While it would do rather well in its current state, a few things could be added that would further improve its performance.

5.1.1 Palette Swapping

Any given column of a Minecraft world is assigned a Biome. The biome of an area dictates the color/presence of grass on the ground, the presence of the various types of trees, and

the abundance of certain other resources. Implementing an algorithm that replaces the blocks of the schematics and other generated structures with ones available in the biome that the settlement is generated in would certainly lead to an increase in the adaptability score of the project.

5.1.2 Mines / Underground Structures

In most GDMC submissions the part of the map below the surface is almost always neglected. Adding a unique structure to the settlement that serves as an entrance to a series of underground mines would lead to an increase in both the narrative and functionality scores, by providing an in-world explanation for how the former inhabitants of this settlement acquired the materials needed to construct it and allowing opportunities for a hypothetical player to collect the materials themselves.

5.1.3 Family Occupations

By adding the capability to attribute certain occupations to certain families to the current family simulation system, you could populate the currently empty chests located in the houses with items related to the occupation of their family.

5.1.4 Chest Contents Randomization

Adding randomized contents to the empty chests in the houses (from a predefined set of items) would lead to an increase in narrative score by making it appear as though someone had been living in the house at one point in time. This could be increased even further by making the randomized items relevant to the family occupation of the resident of the house (if implemented alongside 5.1.3).

5.1.5 Height Respecting Paths

In Minecraft, a player can only jump up inclines of at most one block unassisted. Making it so that the pathfinding algorithm takes this into account would increase the functionality score of the project by making it so that all paths are fully usable by players.

Bibliography

[1] Bridson, Robert. “Fast Poisson Disk Sampling in Arbitrary Dimensions.” *ACM SIGGRAPH 2007 Sketches on - SIGGRAPH '07,* 2007, doi:10.1145/1278780.1278807.

[2] Hart, Peter, et al. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968, pp. 100–107., doi:10.1109/tssc.1968.300136.

[3] Salge, Christoph, et al. “Generative Design in Minecraft (GDMC).” *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 2018, doi:10.1145/3235765.3235814.