Using GDMC Framework to Build Bridges in Minecraft

Steven Guan
Computer Science @ University of California, Davis
Source code can be found here: `https://github.com/stguan2/Master-Project`

May 21,2021

1 Abstract

The Generative Design in Minecraft (GDMC) framework allows the automation of settlement generation in Minecraft through the use of AI. Over the past few years, many people submitted work to the GDMC competition using this framework. None of the submitted work in the 2018 GDMC competition dealt with water very well. There were a lot of improvements made in the most recent 2020 GDMC competition submissions, but none of these dealt with water very well either. This project aims to improve the way structures work together and adapt to the environment by adding bridges over huge gaps and water through the use of A* search algorithm, grid-based clustering, and 3D rasterization.

2 Introduction

Minecraft is a game where you can build and create structures using different types of blocks. The Generative Design in Minecraft (GDMC) framework uses AI to automate this process and to create multiple structures that fit well together, creating a settlement. The AI Settlement Generation Competition is a competition that started in 2018 where many people use this framework to build a settlement to be scored. This competition "is about writing an algorithm that can create a settlement for a given, unknown Minecraft map[12]." The algorithm created should be "adaptive towards the provided map, create a settlement that satisfies a range of functional requirements - but also look good and evoke an interesting narrative[12]." What the algorithm should do is produce something that a human can produce.

The GDMC competition grades every entry using four main categories: adaptability, functionality, narrative, and aesthetics. Each category is scored between 0 to 10, where 0 is when the AI does not consider the category at all, and 10 is where the AI created something that a superhuman would create. 5 is comparable to a naive human (someone not experienced but still human)[11].

During the 2018 GDMC competition, all the entries scored below 5 and none of the entries dealt with water well. Some of the entries had their settlement split by a river with no way to get to the other part of the settlement (meaning there were 2 disconnected settlements instead of 1). Some had unrealistic building placements where the doors would be underwater[10]. Bridges can improve the first issue by allowing connecting settlements that are crossed by a river. This would increase the functionality and adaptability of a design. Another way to increase adaptability is to build water structures like harbors and ships if there is a large body of water.

Although the 2020 GDMC competition submissions did improve a lot compared to the 2018 submissions, these submissions did not deal with water too well. In the 2020 competition interior furniture was added, moving objects using command blocks were used, and underground structures were generated, but there is still room for improvement by incorporating water structures and bridges[1, 13].

The submissions between the start of the GDMC competition to the most recent 2020 GDMC competition all struggled when the Minecraft map used for the competition had water in them. For this reason I create something that did deal with water, specifically when a river splits two settlements apart. This will improve the adaptability of the submissions, which in term will improve the scores. With

this in mind, building bridges over specific areas will fix the issue where a river can split two settlements apart. Extending from this, building bridges over any large gaps can keep settlements from being split apart by these gaps (gaps like Minecraft ravines). Therefore, the aim of this project is to improve the scoring of submissions by building bridges over water and large height gaps.

3 Related Work

This project uses rasterization for transforming a 3D images into a 3D voxelized image and grid-based clustering to create a clustering of tree blocks.

3.1 3D Image Rendering

3D rasterization is used to create a 3D voxelized image of a bridge from a 3D image. There are many different ways to rasterize a 3D image, but one method is to use an octree. In this method we "store only the voxels that are intersected by mesh triangles instead of a full grid[5]." Since the 3D images used for the bridge construction algorithm has many more points needed for constructing the bridge, using this octree-based sparse voxelization is sufficient.

Ray tracing is another approach at rendering 3D images and has been considered to be a completely different rendering approach to rasterization[6]. Research on this topic has shown that a new rasterization technique is much faster than the traditional 2D rasterization which companies the concepts of both rasterization and ray tracing [6]. Although my usage of rasterization is on a smaller scale, the usage of this new technique can improve runtimes of generating 3D voxel images on a larger scale.

3.2 Clustering

Although this project uses grid-based clustering, a density based clustering algorithm could work as well.

- DBSCAN[7] is a density based clustering algorithm by Ester, Kriegel, Sander, and Xu. This is one of the more popular clustering algorithms used and it uses two parameters: epsilon and minPts. Epsilon is the maximum distance between two points for the two points to be considered neighbors (most important parameter) and minPts is the minimum points to form a dense region/the minimum number of points for a point to become a core point.

- OPTICS[4] is another density based clustering algorithm by Ankerst, Breunig, Kriegel, and Sander. The idea of OPTICS is similar to DBSCAN but it addresses DBSCAN's weakness of detecting meaningful clusters in data of varying density. It also uses the same parameters as DBSCAN (epsilon and minPts).

- DeLi-Clu[2] is a clustering algorithm by Achtert, Bohm, and Kroger. DeLi-Cli is an improvement to OPTICS by incorporating ideas from both OPTICS and single-linkage clustering. DeLi-Clu does not require the epsilon parameter and is more efficient and robust.

There are also many different types of grid-based clustering as well.

- STING[14] is a grid-based clustering algorithm by Wang, Yang, and Muntz. This algorithm is a top-down approach, checking each layer to see if it passes a certain threshold before splitting into four children and doing the same.

- CLIQUE[3] is another grid-based clustering algorithm by Agrawal, Gehrke, Gunupulos, and Raghaven. This algorithm is a grid-based algorithm that looks at the density of specific areas in the grid.

- There is also another grid-based clustering algorithm that uses an adaptive mesh refinement[9] by Liao, Liu, and Choudhary. This algorithm's approach is very similarly to STING's approach where it uses a top-down approach and if it passes a certain threshold it looks at its children. This algorithm also looks at the density of the children and clusters groups based on density depending on the mesh used.

STING most resembles the clustering algorithm used in this project.

4 GDMC Submission Tools

The GDMC competition has two main ways of submitting entries. One of the ways to submit work is with the use of an HTTP Server. There is a Minecraft mod called HTTP Interface that opens up an HTTP Server. This submission method allows for new ways of working with Minecraft (like being logged into the world while editing using the HTTP Server). This method also allows for editing Minecraft versions past 1.12.

The other method to submit work is with the use of MCEdit. Unlike HTTP Interface, which is a Minecraft mod, MCEdit is a standalone application that allows users to edit Minecraft worlds outside of the game. Although it does not allow editing while logged in and it does not support Minecraft versions past 1.12, it does not require the user to hook an agent into the game.

There is a third method to submit work which is to create your own script and schematics but this is for more advanced users. This method gives the user the most freedom and it is very portable since the schematics are not tied down to a mod or an application. This method is not very common and requires you to contact the GDMC staff before submitting.

4.1 MCEdit

MCEdit was the method of choice to edit Minecraft worlds for this project. The HTTP Server method is still fairly new and was only a proof of concept at the end of 2020. It was released in 2021 for submissions to GDMC competition. Since I started working on this project before this release, it was not feasible to use the HTTP method.

All instructions to install MCEdit are on the GDMC wiki page. The short overview of this installation process is to first install Anaconda and create an anaconda environment (highly recommended) for python 2.7 (note that MCEdit does not work on Python 3). Then install PyOpenGL, numpy, pygame, pyyaml, pillow, and ftputil==3.4. There are specific installation methods for Windows and Mac. One thing to note is that when installing pygame, the version installed should be 1.9.4 or the movement keys for moving the camera and character around will not function properly in MCEdit.

Once installed, running MCEdit only requires going into the GDMC file found here `https://github.com/mcgreentn/GDMC` and running python mcedit.py in the console. Any scripts created get put into the stock-filters folder and can be used in the filter section of MCEdit (once a world has been selected to be edited).

When using filters (scripts) from MCEdit, the user needs to select a box first before using the filter. This box is the area that the filter can view and modify in the Minecraft world. Although the user can code in a way that views and modifies the outside surrounding area of the box, it is generally not preferred since the area chosen should be the bounding box (box where the filter works in and not outside). For this reason, the algorithm used for this project does not view the outside areas of the box. There are some cases where my algorithm that builds blocks outside of the box, but it is rare and it does not build too far out of the box.

5 Design

Since I want to build bridges over large gaps and over water, I need a method of finding these large gaps and water. The main reason for building these bridges is to connect settlements that are may be separated by large gaps or water. Since the main focus of the project is not about inland structures, the algorithm will condense two settlements into two center points. These two center points will be used as a way to represent two settlements that may be separated by large gaps or water. From these two center points, the algorithm will try to build as many bridges as necessary. This algorithm is split into two main portions: (1) finding suitable locations to build a bridge, and (2) building the bridge itself.

5.1 Finding Suitable Locations

To find suitable locations to build a bridge, the algorithm needs to find a path for the bridge to be built. If there is a river between two settlements, there should be a bridge that gets built through this river. The path that is generated should also be the most optimal path to go from one settlement to another. This means the path should go around mountainous/hilly

areas, and should go around forests/places with a lot of trees. The algorithm best suited for this task is A*[8]. As stated before, my algorithm condenses two settlements into two center points. These two center points will be used as the start and end points for the A* algorithm.

I use A* here because there is a clear heuristic which will speed up the path search. The pathfinding algorithm should show that it is getting closer to the end point (one of the two settlement's center points) from the starting point (the other settlement's center). In A* this is shown by the heuristic. When the heuristic gets smaller, the score gets lower. When the score gets lower, the algorithm is getting closer to the end point. Another similar algorithm is Dijkstra's algorithm but Dijkstra's algorithm is just A* without a heuristic (or a heuristic that is equal to 0). Dijkstra's can also be used here but it is just less efficient because it does not predict if it is getting closer to the end point or not.

Like mentioned before, there is a clear heuristic for my A* algorithm. To depict the algorithm getting closer to the end point, the algorithm uses Squared Euclidean distance $(x_2-x_1)^2+(z_2-z_1)^2$ (the square root is omitted because the comparison between each score is the same with or without rooting). A* also has a scoring value that shows all the previous scores (the sum of the path of the score) summed with the cost of moving to the next cell. The cost of moving to the next cell will have two main parts: (1) height difference to avoid mountainous/hilly areas and (2) forest/tree avoidance.

1. To include the height difference the algorithm checks the y-coordinate of the next block to move to with the current block. For this to work a height map is created. The height map should only show heights of actual blocks and ignore tree blocks, ferns, flowers, and snow. This can be done by creating a list of blocks to ignore (called nonsurface). Moving through water is a lot worse than moving through land so for water blocks, the height is reduced by 5. This is done to create a larger y-difference between water blocks and normal surface blocks. Water blocks are also included in the nonsurface list so the height map does not pick up on water (it will keep moving down until it sees land). What this does to the height map is whenever it sees water, it will note that it touched water and keep moving until it touches the ground, then it will subtract 5 to its height value and save that into the height map. Since moving up/down one block is the same as moving up/down 0 blocks in Minecraft, the algorithm will not penalize a y-difference of one.

2. In Minecraft, trees are constructed with wood blocks and leaves so a forest is an area with a lot of wood blocks and leaves clumped together. Since lots of wood blocks and leaves are clumped together, using a clustering algorithm allows the algorithm to find the size of the forest. Since Minecraft is divided into chunks, a grid-based clustering algorithm to divide these chunks is used for forest/tree avoidance. To find out where the tree blocks are located, the algorithm creates a tree map. The tree map's creation is similar to the height maps creation except instead of displaying the heights of every point, it will display 1 if there is a tree block and 0 if there is no tree block. Once the tree map is created, the grid-based clustering algorithm will split this into groups of size 8x8 and check if there are sufficient amounts of tree blocks inside each 8x8 cell. If there are, the algorithm will go into the next level by splitting these 8x8 cells into four smaller cells and do the same for these smaller cells. After doing this the algorithm will be left with clusters of 2x2 cells with sufficient tree blocks in them. From these clusters, the algorithm will score every cell by doing the following:

   - If a cell is not in any cluster, the score is set to 0
   - Otherwise it is set to the size of the cluster/16 (since the algorithm splits the 8x8 cells into 16 2x2s).

   The score is used for tree avoidance shown later. Algorithm 1 is the pseudocode to the grid-based clustering algorithm. Note that in the pseudocode, adjacent cells is equal to $(cells[0]$+/-$group\_size, cells[1])$ and $(cells[0], cells[1]$+/-$group\_size)$. Also, divideCluster is another function that takes

in the current cell, the group_size, and the threshold and returns all cells greater than the threshold. This is where the splitting into four cells happens. The first time divideCluster is called, the parameters are $(p, map_t, group\_size/2, threshold/4)$.
The second time the parameters are $(p, map_t, group\_size/4, threshold/16)$.

Algorithm 1 Grid-Based Clustering

Require: box, map_t
Ensure: $cluster\_score$
$clusters \leftarrow \emptyset$
$group\_size \leftarrow 8$
$threshold \leftarrow 32$
$x\_size \leftarrow (max_x - min_x)/group\_size$
$z\_size \leftarrow (max_z - min_z)/group\_size$
$cells \leftarrow \emptyset$
for x in x_size **do**
  for z in z_size **do**
    $cells \leftarrow (x * group\_size, z * group\_size)$
  end for
end for
$i \leftarrow 0$
$visited \leftarrow \emptyset$
while $len(cells) \neq len(visited)$ **do**
  $bfs \leftarrow \emptyset$
  $curr\_cluster \leftarrow \emptyset$
  if $cells[i]$ in visited **then**
    $i \leftarrow i + 1$
  else
    $bfs \leftarrow cells[i]$
    $i \leftarrow i + 1$
  end if
  while $bfs \neq \emptyset$ **do**
    $c \leftarrow bfs[0]$
    $bfs \leftarrow bfs[1:]$
    if c not in visited **then**
      $visited \leftarrow curr$
      $amount \leftarrow 0$
      for j in group_size **do**
        for k in group_size **do**
          $amount \leftarrow map_t[c[0]+j][c[1]+k]$
        end for
      end for
      if $amount > threshold$ **then**
        $points \leftarrow divideCluster$
        for p in points **do**
          for c in divideClusters **do**
            $curr\_cluster \leftarrow c$
          end for
        end for
        $append\_cells \leftarrow adjacent_c ells$
        for $cell$ in append_cells **do**
          if $cell$ in cells **then**
            $bfs \leftarrow cell$
          end if
        end for
      end if
    end if
  end while
  if $curr\_cluster \neq \emptyset$ **then**
    $clusters \leftarrow curr\_cluster$
  end if
end while
$cs \leftarrow [len(map_t[0]][len(map_t)]$
for $cluster$ in clusters **do**
  $score \leftarrow len(cluster)/16.0$
  for pnt in cluster **do**
    for row in group_size/2 **do**
      for col in group_size/2 **do**
        $cs[pnt[0] + row][pnt[1] + col] \leftarrow score$
      end for
    end for
  end for
end for
return cs

Every time the algorithm makes a step (in any direction), the algorithm will add to its score. This stops the algorithm from going into an infinite loop of walking back and forth between two blocks of equal height with not tree blocks. Combining all this gives me the scoring method of my A* algorithm shown in Algorithm 2.

Note that the height difference is cubed and the tree avoidance is squared. The reason for this is because the algorithm should heavily penalize a bigger height difference and a forest with more trees. Once the path is generated, the next part of finding a suitable location to build a bridge begins.

In order for a bridge to be constructed, the height

Algorithm 2 A* scoring

1: $g_{child} \leftarrow g_{current} + .1 + max(0, height - 1)^3 + cluster\_tree\_map[nextposition]^2$
2: $h_{child} \leftarrow child.position - current.position$
3: $f_{child} \leftarrow g_{child} + f_{current}$

difference between the start and end points of the bridge have to be the same height. If the height difference is very small (less than 4), then the algorithm can easily compensate by moving the start and end points of the bridge to be the mean/average of the two points. To find these points the algorithm will walk along the path found in the previous step and look at the height values. If the height difference is large along the path, then there is either a big dip or a big jump. If the height jumps, then we move along this path because we cannot build a bridge through a mountainous/hilly area. If the height dips, then we save the point right before it dips, because this can be a potential bridge. Every point after this is then compared to this saved point. Once the algorithm finds a point that is close to this height, then the algorithm will look at the Euclidean distance between these two points. If the distance is large enough, then we save these two points as a possible place to build a bridge. This is because the heights of these two points is similar, the two points will contain a hole (a dip), and the two points are far enough away for a bridge to be built.

Note that when comparing new points to the saved point, if the new point's height value is much larger than the saved point, we scrap the old save point and set a new save point equal to the new point. This is so the algorithm does not build bridges with big start and end point height differences.

After obtaining points to build a bridge, the algorithm needs to score the bridge to see if it should be built or not. The scoring uses two points and looks at the line between those two points. From this line, the algorithm finds the height values and compares these height values with the height values of the two points used to create this line. If any of the height values on this line ever exceed the height values of both points, then we cannot build a bridge here because we cannot build bridges through blocks. The algorithm also checks if the distance between the two points is too small because we do not want to build a bridge that is too short. The algorithm deals with these two cases by setting the score to -1.

If both these conditions are passed, then the algorithm will look at the heights of every point on the line and compare them to the height value of the two points chosen to create the line. The algorithm will not build a bridge if the difference in height values between all the points and the two points chosen are too small because building a bridge over a height distance of 1 does not make a good bridge. The larger the height difference the higher the score. If the score passes a certain threshold, then the algorithm will build the bridge. Note that this threshold changes depending on the distance of the bridge because there are more height values to compare. This will naturally increase the score and to compensate, we increase the threshold. The algorithm also puts more weight on bigger height differences so the height difference is squared. The final scoring equation is shown below where the $\sum y$ is the height of all the points on the line and y_1, y_2 are the heights of the chosen two points.

$$(\sum y - max(y_1, y_2))^2 > threshold * distance$$

Note that the heights described here should not be modified by water blocks. This means that when creating the height map adjusted for water and tree map, the algorithm will also create a normal height map.

After finding the points to build the bridge on, the next step is to actually construct the bridge.

5.2 Bridge Construction

Using two points from the previous part, the algorithm will convert the two points into a scale (using the distance of the two points), the midpoint of those two points, and an angle (arcsine of x distance/scale). There are two different parts to building the bridge: (1) Converting a 3D image into points and (2) Using those points to put into Minecraft.

Converting a 3D image into points will be discussed later in the paper. Just note that this part will give the algorithm a file containing all points of the 3D voxelized image. After obtaining these points, the algorithm will adjust these points to fit into Minecraft. The first step in this is to scale the points by normalizing the points using the maximum

x, y, or z value (absolute values to ignore negative signs), and then dividing all points by this max value. Then the algorithm will use the scale number obtained from the previous part and multiply the scale with the normalized values to get a scaled version of the bridge. The angle is then used to rotate the bridge so that it is in the proper rotation.

After scaling and rotating, the algorithm will remove all unnecessary points. The points in the file obtained are not integers, but Minecraft coordinates are integers. In order for the algorithm to work, it will need to truncate all the points and then remove all the repeated points. If this happens before scaling or rotating there can be holes/missing blocks when building the bridge.

After removing all the unnecessary points the algorithm will move the points up/down to the right area. This is done by finding the largest amount of points in a specific height and moving that to the height given from the midpoint. This is because the largest amount of y-value points on the bridge is the road of the bridge (since the bridge is flat/not curved). Without this, the bridge will be built a little off of the desired height value.

Once all this is done the algorithm will finish the supports of the bridge by building the supports down to the ground. Finding the lowest point of the bridge is trivial. After finding these points the algorithm looks at if the block underneath the lowest point of the bridge is a ground block or not. If it is not the algorithm will build down until it is a ground block.

The last portion is to build the road using a different block so the bridge is not all the same block. Just like moving the bridge up/down, the algorithm will find the largest amount of points in a specific height. As stated before, this is the road.

5.3 PyOctree Package

Going back to the first step in bridge construction, the algorithm converts a 3D image into a 3D voxelized image of points. This is done by using the PyOctree.py package, which is a package that converts a 3D .stl file image into an octree. After obtaining the octree, it uses ray casting to find the intersections on the octree from rays and output that into a file containing all the points of the intersections. This is where the file with the points is obtained.

PyOctree requires Cython, a C++ compiler, and vtk. Since vtk requires Python 3.6, this part will not be compatible with the MCEdit portion. A workaround for this is to create another anaconda environment for Python 3.6 and install all these packages. Then run PyOctree and a 3D image and generate a file with all the points of the intersections and use just the file for Minecraft.

6 Testing

There are three main parts to the whole algorithm that needs to be tested: (1) A* path finding algorithm, (2) Bridge finding algorithm, (3) Bridge construction algorithm. Once all components have been tested individually, I will test them together using randomly generated Minecraft maps.

6.1 A* Path Finding Algorithm

The A* path finding algorithm has two parts to its scoring (tree avoidance and minimizing height difference). I will test each part separately and then merge them together.

6.1.1 Tree Avoidance

To test this part, I will initially create a massive amount of tree blocks in a flat Minecraft testing world and see if the path goes around the blocks or if it goes through the blocks. Once this initial test phase is complete, I will go into more real world examples by generating different Minecraft worlds and seeing how the path is generated if there were lots of trees in specific areas like in Figure 3.

6.1.2 Height Difference

Testing this part is similar to testing tree avoidance where I will initially create a massive height difference in a flat Minecraft testing world by generating a lot of blocks in a certain area (with a huge y-coordinate difference) and seeing if the path will go around this area or not shown in Figure 1. Just like the tree avoidance part, I will do this test in actual Minecraft worlds after the first part and pick areas with lots of hills, mountains, or ravines.

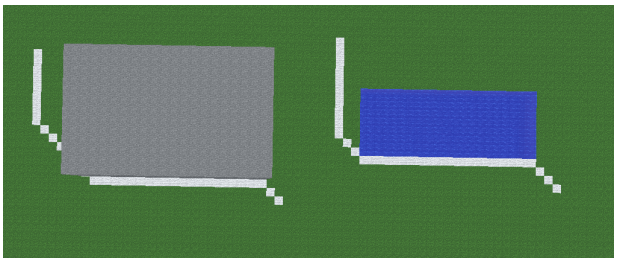

Figure 1: Testing A* path finding algorithm on a simple flat Minecraft world. The left side shows the algorithm dealing with height and the right side shows the algorithm dealing with water. The white wool generated is the path that A* finds most optimal.

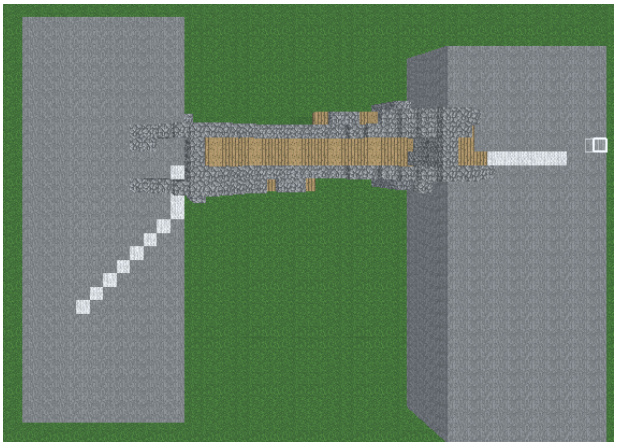

Figure 2: Testing bridge finding and building algorithm on a simple flat Minecraft world.

6.2 Bridge Finding Algorithm

Once a path is found, I can use it to test this portion. Similar to how the A* path finding algorithm was tested. I first test this part in a flat Minecraft world by creating a hole that is a large enough distance away for a bridge to be built and see if the bridge finding algorithm detects this and generates the two points required to build a bridge shown in Figure 2. Once it passes this initial phase of testing, I will generate Minecraft worlds and see if the bridge finding algorithm is able to generate bridge points based on the path found by the A* algorithm like in Figure 3.

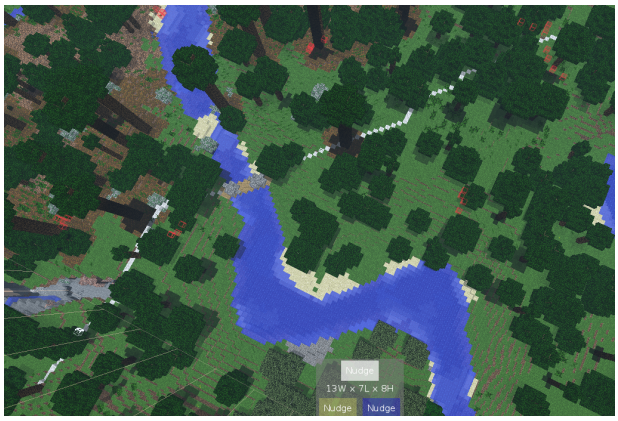

Figure 3: Testing tree avoidance and bridge finding together in a Minecraft world with lots of trees and a river. The white wool does not just go straight and attempts to avoid as much of the trees as possible.

6.3 Bridge Construction Algorithm

To test this part I will first give it two points and see if the bridge gets generated properly. Once this is successful, I will use the points generated from the bridge finding algorithm to test the bridge construction algorithm like in Figure 3.

To test the PyOctree part (which is still part of bridge construction), I first test if the octree generated it correct. To do this I use a software that can read .vtu files and overlay the octree on top of the actual 3D image. The octree and 3D image should fit perfectly together. Once this part is correct, run the bridge construction algorithm to see if the object put into Minecraft looks like the 3D image itself.

7 Challenges

7.1 Clustering Issue

There are some small issues that were discovered from testing and from how the implementation of the project was done. When doing the grid-based clustering, the grid is first split into cells of size 8x8 (This was chosen because a chunk in Minecraft is 16x16). The main issue with this is that if the box created was not divisible by 8, then there would be extra points. The way I dealt with this is to just ignore any points past the last number that was divisible by 8 on the (on

the x axis and the z axis). In order for this to affect the algorithm, the edges of the box have to contain a large number of tree blocks. To get around this, I can increase or decrease the selected box to either not include the tree blocks or to make sure the tree blocks do not get truncated.

7.2 Height Difference

Another challenge from this algorithm was dealing with height differences. Since the scoring of each cell only looks at its neighbors, even if there is a large discrepancy of height in between two points, the scoring for this mountain (or more specifically the path through the mountain) could be small/optimal. This might happen if the mountain is fairly flat with small inclines going up and down the mountain. This has happened in one of my tests where there was a pillar and the path ended up going through the pillar instead of around the pillar. In most cases, however, this does not happen.

8 Conclusion and Future Work

The usage of multiple factors to determine the path between two settlements means the path is more optimal. After finding specific points to build a bridge on, scoring the bridge to make sure the gap between the bridge and the ground meant that bridges would never be built over small gaps of height.

Although there were many different tests to make sure the algorithm was generating proper bridges, none of tests done checked the different methods of finding a path. Instead of using a grid-based clustering method, using a density-based clustering method might be more optimal.

This project mainly focuses on bridge placement. Finding a way to use different blocks to build the bridge by looking at the terrain/surrounding area or by looking at the 3D image's color can improve bridge construction. My algorithm is not able to deal with any curved bridges since it uses the y-coordinate's mode to determine the road of the bridge. Using a different method of finding the road can allow for curved bridges to be used instead of just flat ones.

Another extension to bridge building is to procedurally generate the bridges. Instead of using the PyOctree package and manually having to adjust the bridge by scaling, rotating, and translating, finding a way to procedurally generate a bridge given two points will allow the constructed bridges to be more varied and look better.

Building bridges over water and over large gaps is one way of improving the score on GDMC submissions. Another method of doing this is to deal with water by adding water structures like ships and harbors. This was mentioned in the 2018 GDMC competition results.

References

[1] Fdg2020 september 17, 18:30 19:30 demos and competitions.

[2] E. Achtert, C. Böhm, and P. Kröger. Deliclu: boosting robustness, completeness, usability, and efficiency of hierarchical clustering by a closest pair ranking. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 119–128. Springer, 2006.

[3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 94–105, 1998.

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: Ordering points to identify the clustering structure. *ACM Sigmod record*, 28(2):49–60, 1999.

[5] C. Crassin and S. Green. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, pages 303–318, 2012.

[6] T. Davidovič, T. Engelhardt, I. Georgiev, P. Slusallek, and C. Dachsbacher. 3d rasterization: a bridge between rasterization and ray casting. In *Proceedings of Graphics Interface 2012*, pages 201–208. 2012.

[7] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clus-

ters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[9] W.-k. Liao, Y. Liu, and A. Choudhary. A grid-based clustering algorithm using adaptive mesh refinement. In *7th workshop on mining scientific and engineering datasets of SIAM international conference on data mining*, volume 22, pages 61–69, 2004.

[10] C. Salge, M. C. Green, R. Canaan, F. Skwarski, R. Fritsch, A. Brightmoore, S. Ye, C. Cao, and J. Togelius. The ai settlement generation challenge in minecraft. *KI-Künstliche Intelligenz*, 34(1):19–31, 2020.

[11] C. Salge, M. C. Green, R. Canaan, and J. Togelius. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[12] C. Salge, M. C. Green, R. Canaan, J. Togelius, and C. Guckelsberger. Generative design in minecraft.

[13] T. Thompson. Generative design in minecraft competition 2020 - live judging (part 1).

[14] W. Wang, J. Yang, R. Muntz, et al. Sting: A statistical information grid approach to spatial data mining. In *VLDB*, volume 97, pages 186–195, 1997.