Building a Textual Database for the Generative Design in Minecraft Competition:

Methods for textual labeling, building and room detection, and future work

EGAN GRAY, Earlham College, USA

ACM Reference Format:

Egan Gray. 2018. Building a Textual Database for the Generative Design in Minecraft Competition: Methods for textual labeling, building and room detection, and future work . 1, 1 (May 2018), 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 ABSTRACT

This paper presents the structure and design of a novel open-source textual database named the Brick Database for Minecraft settlements. Aimed at addressing the current lack of comprehensive datasets for model training in the Generative Design in Minecraft Competition (GDMC), this study explores the methodologies and potential challenges in creating a textual database and automatically detecting buildings and rooms.

2 INTRODUCTION

The Generative Design in Minecraft Competition (GDMC)[1] is a yearly event aiming to facilitate the creation of engaging settlements in Minecraft via procedural generation. Although GDMC does offer some beneficial tools for block placement and extraction, such as GDMC-HTTP[2] and GDPC[3], it does not supply a complete dataset of settlements or buildings for model training.

In addition, the GDMC Impressions[4] showed a common theme. Many different methods were applied, but every project had some variation of the same issue– the robots misunderstood why we place things in the contexts we do. Paths would be dangerously high and narrow, foundations would be unreasonably big, bridges would be built over tiny amounts of water, doorways would be blocked by leaves, etc. The overarching theme is that while the AI could imitate what a house with a doorway looked like, or what a bridge looks like, it couldn't imitate not the logic of why we have doors that can't be obstructed and bridges that we won't fall off.

This means that not only do we need a database of Minecraft buildings, but we also need semantic and textual labels for buildings. There is two open source database [5][6] for this purpose, but they were built under a time constraint and are generally low-quality building with poor labeling.

This article proposes a structure for an open-source textual database—dubbed the Brick Database—for Minecraft settlements. It will delve into the methodologies, challenges associated with the automatic detection of buildings and rooms, and potential applications of the Brick Database in the realm of generative design.

Author's address: Egan Gray, Earlham College, 801 W National Dr, Richmond, Indiana, USA, 47374, eewhite19@earlham.edu.

3 BRICK DATABASE FORMAT

This section outlines the textual labels and data structures used by Brick Database.

3.1 Textual labels

Brick Database uses 3 types of textual labels:

3.1.1 Aesthetic.

Aesthetic labels have two categories, *Meta-Aesthetic* and *settlement-aesthetic*. Aesthetic tags facilitate the establishment of contextual relationships between structural and aesthetic blocks in the settlements.

(1) *Meta-Aesthetic*: This is the broad aesthetic of the map, like "Modern", "Medieval" or "Futuristic".
(2) *Settlement-Aesthetic*: A settlement's aesthetic may be more specific than its *Meta Aesthetic*. For example, a Medieval Settlement's style could range from Nordic, Ruined, Dark Wood, etc.

3.1.2 Function.

Buildings, floors, and rooms all have their own unique function labels. Function labels are used to establish context on which rooms belong in which buildings, and which blocks should be placed in which room.

(1) *Building function*: General building purpose, like "Inn", "Hospital" or "Home".
(2) *Floor function*: These are very basic and labels are set by the number of floors in the building. These functions are useful later during room detection. A 1-level house level is saved as a "single floor", if the building contains more than 1 floor each floor is labeled hiercharly: *Multifloor bottom:Multifloor middle:Multifloor top*
(3) *Room Function*: Room functions represent the purpose of the room, like "bar", "bed room" or "shop", "craft room", etc. Rooms may also have sub-functions, like "craft room:alchemy" or "craft room:smithy".

3.1.3 Block Type.

Each block may be one of the four following types. The block type is used during building and room detection.

(1) *Structure*: Blocks typically used for the construction of the building. Stairs, wood, chisled stone, etc.
(2) *Function*: Blocks placed in a room depending of the function of the room. Ex: Beds for bedrooms, Furnace for kitchens.
(3) *Aesthetic*: Aesthetic blocks are placed depending on the settlement's aesthetic, like "cobwebs" and "skulls" for a ruined aesthetic, and "snow" for cold aesthetics
(4) *Trash*: Blocks we don't save as part of the building, like dirt, sand, and naturally occurring stone.

3.2 Data Structure

A settlement stores buildings use a hierarchical collection of groups. The settlement class contains the settlement's Meta Aesthetic and Sub-Aesthetic, as well as pointers of its children building classes and arrays. Each building has a function and pointers to all level arrays and classes inside the building. Each level class pointers to all room classes. Finally, the Room class contains the room's coordinates, its main and sub-functions, and lists of all functional and aesthetic blocks. All Minecraft maps, buildings, levels, and rooms are saved as NumPy arrays. All labels, arrays, and classes are saved in a .HPF5 database.

Input: 3D array of Minecraft Map
Settlement: Root Node (Position, Bounds, Aesthetics tags, Building list, Pallete) ->
Building: Node (Position, Bounds, Type, Function, Parent, Level list)->
Level: Node (Position, Bounds, Function, Parent, Room list)->

Room: Node (Position, Bounds, Function, Parent, Aesthetic tags, Blocks) ->
Block: Node (Block ID, Position, State, Data, Parent)

4 BRICK DATABASE BUILDER

The Brick Database Builder, not to be confused with the Brick Database, is a series of Python tools used to help us automate building the dataset. In this section, we will discuss palettes, building detection, and room/level auto-detection.

4.1 Palletes

Palettes are a way of saving time when checking block types. The Palette class in Brick Database Builder contains every Minecraft block ID sorted into 5 categories– *Structural*, *Trash*, *Functional*, *Aesthetic* and *Magic*. *Structural*, *Trash*, *Functional*, *Aesthetic* blocks we can safely bet the type of. For example, a bed will always be a *functional* block. *Magic* blocks are any blocks that can change depending on context. For example, sometimes flowers are *aesthetic* type blocks, but other times they can be *trash* blocks. Magic blocks ask for researcher input to specify which type the block id should be saved under during generation. Palettes are highly organized and reorderable, so they can be easily reconfigured.

4.2 Buidling Detection

Buildings have two types, houses and structures. Houses (Fig.1a) are any type of building that has levels and rooms. Structures (Fig.1b) are buildings that are less conventional, like bridges, castle walls and churches which may not be so easily described as arrays of levels and rooms. Building detection has two methods:

- *Selector-Detection*: Selector Detection involves manually marking each building that should be saved. House-type buildings are labeled with "Lime Wool" blocks, and Structure type buildings are labeled with "Cyan Wool" blocks. These blocks are then appended to a search list, which appends the block to a search queue. The search queue pops the top value it, and each adjacent[1] block that is *structural*, *functional*, or *aesthetic* type are appended to the search queue. All connected coordinates are saved as part of the building. Selector detection struggles with issue 4.5.1. This method involves hand-selecting data, so these issues may be addressed by deleting nearby blocks that could cause errors.
- *Auto-Detection*: This method is similar to Selector-Detection, except instead of wool blocks each building is found by a pre-set selector. This selector should be something most buildings have, like doors or windows. This method involves no input from the researcher, but issue4.5.1 and issue**??** are more common with this method, as the researcher may not hand-pick buildings and sanitize the surrounding areas for blocks that could cause errors.

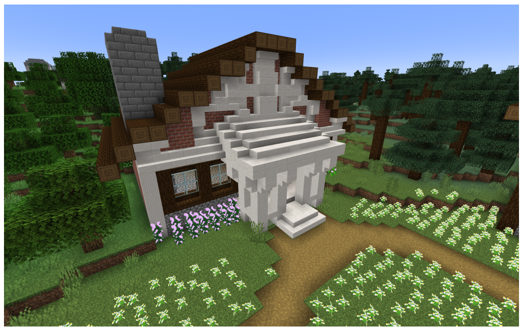
4.3 Level Auto-Detection

Fundamental Concept: The vertical axis (y-axis) of a given floor is likely to contain a larger proportion of building materials compared to the immediate areas above or below it.

The detection of floors is achieved by identifying y-coordinates within the building array that contain the greatest concentration of structure-type blocks. Consider a structure with three levels:

- **Level 1:** Contains 10% structure blocks, 20% function blocks, and 70% empty space.
- **Level 2:** Contains 70% structure blocks, 1% function blocks, and 29% empty space.

[1] *Adjacent*: block x(-1, +1),y(-1, +1), z(-1, +1)


(a) House

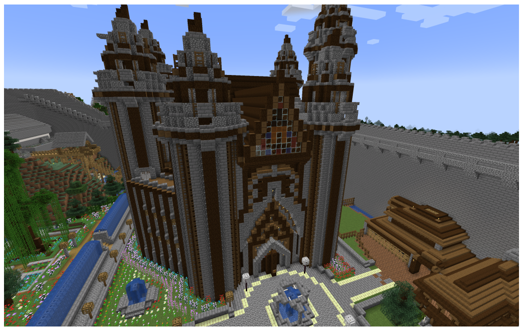

(b) Structure

Fig. 1. Building types

- **Level 3:** Contains 10% structure blocks, 30% function blocks, and 60% empty space.

Given that y-coordinates containing a floor will typically exhibit a higher concentration of structure blocks, it's reasonable to deduce that Level 2 includes a floor. When the coordinates of these floors are determined, the spaces between them are segmented into new "level" arrays, which are later employed for room detection.

This step is critical, as any inaccuracies may result in flawed room detection. Although this method proves generally successful in detecting floors, it's crucial to consider and address potential edge cases since incorrect level determination can lead to inaccurate room detection. Therefore, researchers are advised to verify or amend the automatically detected floor coordinates prior to proceeding with Room Detection.

4.4 Room detection

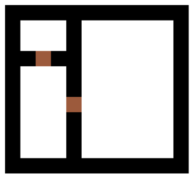
The procedure for room detection for each level is as follows:

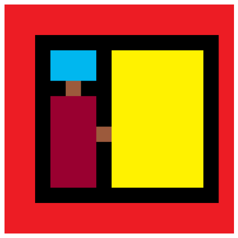
(1) **Generate floor plan** This is a 2D array created from the 3D level array, with each index of the graph being either a "wall" (Fig.2a black) or "not wall" (Fig.2a white)
(2) **Color empty areas** All white blocks are given separate colors depending on if area are connected with each other.
(3) For each colored area:
    (a) **Scan z-axis**: For each x coordinate of color, check (highest_Z-axis+1, x) and(lowest_Z-1, x) blocks are are walls. If they are, return true, else return false. Fig.2c
    (b) **Scan x-axis**: For each z coordinate of color, check (Z, highest_X+1) and(Z, lowest_X+1) blocks are walls. If they are, return true, else return false. Fig.2d

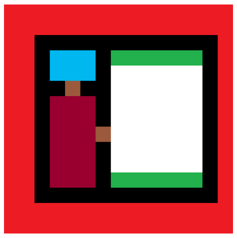
4.5 Issues

4.5.1 Connected buildings. If a building has a road made out of the same material as its building or has a fence connecting it to another building, the building selection will be faulty.

4.5.2 Open rooms. The room detection method only works on rooms that have walls on all four sides. Buildings like Fig.**??** contain exterior rooms that will not be detected, while buildings like Fig.**??** has a room with an open section, which means it would not be detected as an interior area.


(a) Generate floor plan


(b) Color separate areas


(c) Scan for walls(z axis)

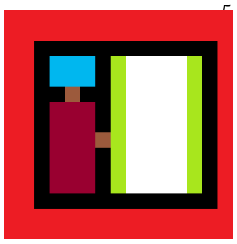
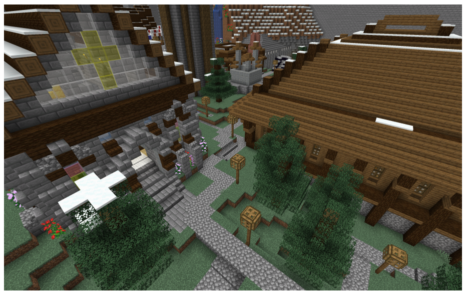

(d) can for walls (x axis)

Fig. 2. Room Detection process


(a) Buildings connected by paths

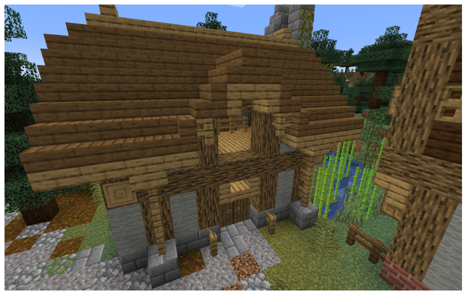

(b) Buildings connected by walls

Fig. 3. Room Detection process

4.5.3 To trash or not to trash. Some trash blocks are used as structural blocks, such as a settlement that uses gravel for both roads and structural blocks, or floors that have dirt, like in barns (Fig.4).

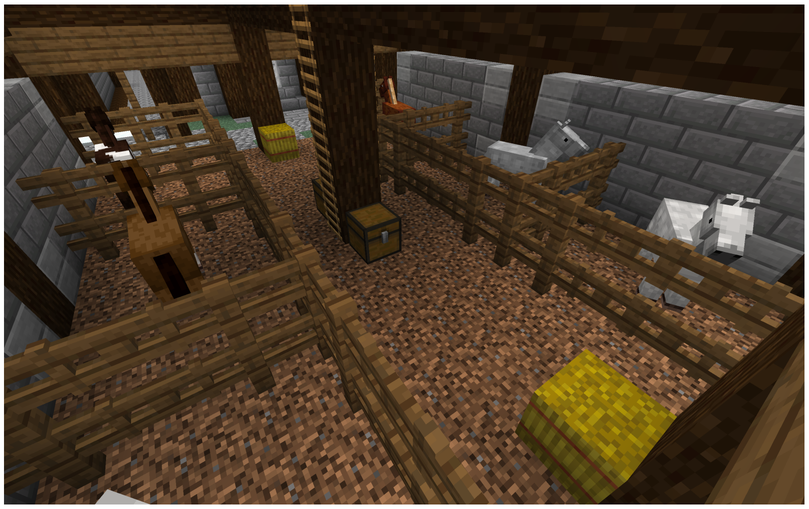


Fig. 4. Should dirt be saved?

4.5.4 Level detection . Level detection is mostly effective, but still occasionally fails, as the percentage difference of structural blocks between floors and walls varies depending on the size of the structure. If the target percentage difference is too low, it may detect roofs or rooms as floors, but if the percentage is too high, it may not detect any floors at all. Because correctly set levels are crucial for successfully detected rooms, all levels must be double-checked by the research when scanning before proceeding to room detection.

4.5.5 High quality maps. There are a vast number of Minecraft maps and building packs, but many of these buildings have empty interiors. Adventure maps meant for exploration tend to have more buildings with decorated interiors, but finding good adventure maps is difficult. Empty houses are still useful for exterior generation, but decorated interiors are required for Brick Method to as effective as possible.

5 FUTURE WORK

5.1 Automated Building Detection

As of writing, every building in the database requires manual building selection, level double-checking, and labeling of room and building functions. A building on average requires anywhere between 4-15 manual pieces of data entry per. At this rate, it would take anywhere between 40,000 - 150,000 manual entries for a 10,000-building database.

As discussed in 4.5, architecture is full of edge cases. While it is possible to create algorithms that work for 80% of buildings, edge cases make it near impossible for efficient detection that is completely effective. This naturally puts a bottleneck on the speed we can build our dataset with. Many generative methods such as GANs and CNNs are very data-hungry, and the more data we have the better our output could be.

For proper detection of buildings, we require a training a new training set. The training set should contain at least 5,000 valid examples of buildings and at least three times invalid examples. Invalid examples should range from partially selected buildings, roads, and terrain like trees and mountains. Once this data set is built we can use it to train a building classification agent using a variety of methods, like 3D Convulention, Recurrent or Reinforcement learning.

5.2 Possible uses for generation

Once fully built, there are many ways this dataset could be used to train generative systems to create new and interesting buildings. In this section we will be briefly discusses methods of generation that naturally lend themselves to this type of textual organized data. Recurrence neural networks could create buildings level-by-level, taking the initial function of a building and its first floor, then generating a 2D array for the y-axis immediately above it. This process is repeated until the building is complete. Another possible method to generation could be a Knowledge-Based System, using buildings as its knowledge base and the connection level sizes, room placement, and building function. Once automatic building classification is accurate, it could be used as a discriminator agent for a GAN model. A discriminator model and a generator model would allow self-teaching, which could result in more unique buildings.

Finally, once settlement generation has become automatic with the simple input of a map and the settlement aesthetic, the settlement generator could be paired with two “story-telling” language models. One model creates and maintains a story arch, while the second Dungeon Master model spawns settlements, mobs and resources depending on the story and player stats. This could mean creating an ambush when the player has gone too long without conflict, spawning a village when the player is lost and low on resources, or more. Advancements in language models, like Auto-GPT could allow fully interactive NPC’s as already demonstrated possible by cite. These NPCs remember previous interactions and could return personall, local and story karma score. Positive or helpful interactions that meet the NPC’s goal boost karma, and negative, aggressive, non-helpful, or nonsensical interactions decrease the player's Karma score. The amount of positive/negative interactions the player has total effects the story karma, which changes the plot of the story, the chances of different events happening and even the aesthetic of buildings.

6 A NOTE TO THE READER

Brick Database Builder will be open-source in June, 2023, and the success of the project is dependent on the size and accuracy of the dataset. Brick Databse Builder comes with a variety of modular detection tools that makes building and adding to this dataset very simple, requiring no coding experience to use. A dataset is only as good as it's datapoints, and if you a Minecraft builder or GDMC engineer, we urge you to add your data to the Brick Database, so our robots can build better houses.

7 CONCLUSION

In conclusion, the development of a comprehensive, textual database for Minecraft settlements, the Brick Database, is an important step towards automated and generative design in Minecraft. This work has provided an analysis of the complexities and challenges involved in the creation of such a dataset, including the intricacies of building and room detection, and the importance of aesthetic, functional, and structural elements.

While the Brick Database and its building tools are promising, they are not without their issues. The accurate detection of buildings is a challenge that we must continue to address. However, despite these issues, the Brick Database still stands as a valuable resource for researchers and developers in participating in GDMC.

As we look forward to the future of this project, the need for a robust, automated building classification system is evident. Such a system will not only aid in the rapid expansion of the datsbase but also improve the precision of the generative models, leading to more exciting creative output.

8 ACKNOWLEDGEMENTS

Special thanks to Charlie Peck and Malik Karim Barrett. Your assistence was invaluable.

REFERENCES

[1] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. Generative design in minecraft (gdmc): Settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, New York, NY, USA, 2018. Association for Computing Machinery.
[2] Niels-NTG and Nikigawlik. Minecraft http interface mod.
[3] Nikigawlik. Minecraft http interface python.
[4] Christoph Salge, Claus Aranha, Adrian Brightmoore, Sean Butler, Rodrigo De Moura Canaan, Michael Cook, Michael Green, Hagen Fischer, Christian Guckelsberger, Jupiter Hadley, Jean-Baptiste Herve, Mark Johnson, Quinn Kybartas, David Mason, Mike Preuss, Tristan Smith, Ruck Thawonmas, and Julian Togelius. Impressions of the gdmc ai settlement generation challenge in minecraft. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, FDG '22, New York, NY, USA, 2022. Association for Computing Machinery.
[5] Jonathan Gray, Siddharth Goyal, C. Lawrence Zitnick, Arthur Szlam, and Demi Guo. Minecraft house. Jul 2019.
[6] Jonathan Gray, Siddharth Goyal, C. Lawrence Zitnick, Arthur Szlam, and Demi Guo. Minecraft segementation. Jul 2019.