Universiteit Leiden

Master Computer Science

Terrain-Adaptive PCGML in Minecraft

Name:	Arthur van der Staaij
Student ID:	s2014149
Date:	07/08/2023
Specialisation:	Artificial Intelligence
1st supervisor:	Dr. Mike Preuss
2nd supervisor:	Dr. Christoph Salge

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This work makes a first step towards terrain-adaptive PCGML in Minecraft. We introduce an automated system and preprocessing tool to create "volume-to-volume" datasets suitable for machine learning by leveraging handwritten black-box Minecraft settlement generation algorithms. Using this system and tool, we create ten terrain-adaptive Minecraft ML datasets — including ones based on the current best-performing algorithm submitted to the Generative Design in Minecraft (GDMC) competition. Finally, we train and qualitatively evaluate various GAN-based volume-to-volume models on all ten of our datasets. Although we do not obtain good results in all cases, we demonstrate that terrain-adaptive PCGML in Minecraft is indeed feasible.

1 Introduction

1.1 Aim and scope

This work is a first step towards terrain-adaptive PCGML in Minecraft. That is, we aim to train machine learning models on the task of generating structures on top of a given piece of natural terrain. A central issue for this area of research is the lack of suitable datasets. Our goal therefore splits into two parts: we aim to both create such datasets, and to then use these datasets to train ML models.

The main idea behind our approach was to create the needed datasets by leveraging generative algorithms from the Generative Design in Minecraft (GDMC) competition, a yearly competition on settlement PCG in Minecraft [1]. This is what led us to our research question:

Is it possible to reproduce GDMC Minecraft settlement generators using machine learning?

In the remainder of this introduction, will first give more extensive background on the topic of our work (Section 1.2), then we will explain our approach in more detail and summarize our contributions (Section 1.3), and finally we will outline the structure of the rest of the text (Section 1.4).

1.2 Background

1.2.1 PCG(ML)

Procedural Content Generation (PCG) is a commonly-used game development tool that has been gaining increasing scientific interest. PCG is the technique of generating game content algorithmically, as opposed to creating it by hand. It is used for many reasons, such as to alleviate development work, increase replay value, save storage space, or even to dynamically adapt content to the player [2, 3].

The explosive growth of machine learning has also spurred scientific interest in Procedural Content Generation via Machine Learning (PCGML), which Summerville et al. define as the direct generation of functional game content using machine learning models [3]. This subgroup of PCG has however not seen much use yet in the game industry. Most forms of PCG used today lean heavily on domain-specific generation rules handwritten by humans.

1.2.2 Minecraft

A video game that is well known for making heavy use of (non-ML) PCG is Minecraft. It uses PCG to not only generate the natural terrain and biomes of the world, but also to place various terrain-adapted structures like villages (Figure 1). This way, it provides ever-diverse worlds to play in.

Minecraft has grown to be a prominent testbed for all kinds of AI research, including PCG. This is likely fueled by its popularity and the wide availability of modifications that allow you to interact with the game programmatically, some even supported by Microsoft itself (the present owner of Minecraft) [4].

From the viewpoint of PCG research, the game has many interesting challenges to offer. It has a fairly unique token-voxel-based world (it consists of 3D pixels that have non-continuous *token* values like "stone" or "air"), it provides randomly generated terrain that can be used to evaluate *terrain-adaptive* generative algorithms, and it offers gameplay context to evaluate the functionality of generated artifacts. Furthermore, due to the game's popularity, a huge amount of human examples is available, which is useful both as inspiration for handwritten algorithms and as training data for machine learning systems.

1.2.3 GDMC

To promote more research towards PCG in Minecraft, Salge et al. introduced the Generative Design in Minecraft competition[1] in 2018 [1]. In this competition, the goal is to write an algorithm that automatically generates a believable Minecraft settlement that adapts to the pre-existing natural terrain. All submitted entries are evaluated by human judges in four categories: *Adaptation*, *Functionality*, *Evocative Narrative* and *Aesthetics*. The competition has been running annually since its foundation, and has led to consistent development in the field. [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]

The GDMC competition is completely open in regard to PCG methods, and all submissions — from intricately handwritten algorithms to scientific experiments — are judged on an equal playing field. Entries are scored on absolute scales, which makes it possible to compare results across all iterations of the competition. This makes them a great way to compare new experimental generation methods with more established ones.

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

Figure 1: A regular Minecraft village, generated on top of the natural terrain.

Thus far, the best results have consistently been achieved by algorithms that primarily consist of a large amount of constructive generation rules, such as "build houses on flat ground", "connect houses with roads", "place furniture in houses", etc. Although PCGML-based methods have been attempted in the competition [18], their results have not yet paralleled these more manual methods.

1.2.4 Minecraft PCGML

Outside the GDMC competition, there have been several Minecraft PCGML approaches that have achieved better results [19, 20, 21, 22], but these all come with limitations. Most notably, they are all unable to condition their generation on existing Minecraft terrain. This limits them to generating artifacts in isolation.

The automated settlement generator algorithms brought about by the GDMC competition provide an interesting opportunity in this regard, one which we aim to capitalize on with this work. An important problem for PCGML is the lack of sufficient training data [3]. Creating a PCGML system that can adapt to pre-existing terrain requires data on such pre-existing terrain, which significantly complicates the collection of a suitable dataset. Although an enormous amount of human-built settlements is available, they often heavily alter the original terrain, or do not provide an easy way to obtain it.

Machine learning systems that can learn to convert from one image-like domain to another (e.g. from terrain to settlements) in an unsupervised manner do exist [23], but they can be difficult to train. However, using settlement generator algorithms submitted to the GDMC competition, it is possible to automatically generate a practically unlimited amount of terrain-settlement pairs that can be used to train a supervised learning algorithm.

1.3 Our approach

In this work, we develop an automated system to run external settlement generation algorithms in a Minecraft world and extract the resulting terrain-settlement pairs to a dataset that is suitable for machine learning. We then use this system to create several such datasets, including one for the current best-performing GDMC generator [17]. Finally, we use these datasets to train various machine learning models, and we evaluate their performance.

Our main contributions can be summarized as follows:

1. We introduce a system that can automatically create a machine learning dataset using a black-box Minecraft settlement generation algorithm written for the most common GDMC submission method, and which can easily be extended to other types of settlement generation programs.
2. We introduce several datasets of Minecraft terrain-settlement pairs, primarily ones made with variants of the current best-performing GDMC algorithm [17], which can be used for future research.[2]
3. We create machine learning systems for 3D spatial token-based image-to-image translation, train them using the aforementioned datasets, and evaluate their performance.
4. We introduce a way to handle Minecraft *block states* in machine learning systems.
5. We introduce a new *frequency weighting* technique for mapping from an embedding space back to a categorical space, and investigate its effectiveness.

[2]Datasets are available via correspondence with `a.j.w.van.der.staaij@umail.leidenuniv.nl`

1.4 Thesis structure

This work is organized as follows. In Section 2, we will discuss related work in the areas of Minecraft PCG settlement generation, Minecraft PCGML and image-to-image translation. In Section 3, we will describe the design of our automated dataset creation system and how we use it to generate several datasets. In Section 4, we describe how we preprocess these datasets to prepare them for our ML models. In Section 5, we describe the ML models and related systems we used for experimentation. In Section 6, we describe the experiments we performed and their results. In Section 7, we summarize our work. Finally, we present our conclusions in Section 8 and make suggestions for future work in Section 9.

2 Related work

2.1 Minecraft PCG settlement generation

The GDMC competition has led to a substantial amount of research in the area of PCG in Minecraft. The organizers and judges have published two reports that discuss the developments made in the 2018 and 2021 competitions [5, 6]. We will give a brief overview as well, limited to the published work but including some of the more recent developments. Note that we leave out PCGML approaches here, as we will discuss them in Section 2.2.

Tools for settlement generation. Green et al. have described a technique for generating organic floor plans using a constrained growth method and cellular automata, which can improve internal building variety [7]. A different kind of planning algorithm is described by van Aanholt et al. [9]: they generate 3D building plans for general voxel buildings using architectural profiles and answer set programming. Their method is not limited to Minecraft, but has been used successfully in a GDMC submission.

Full settlement generation. Naturally, the GDMC competition has led to many works about full settlement generation. Fridh et al. describe a settlement generator that combines several techniques, such as surface adaptation with graph grammars, house floor plan generation and furnishing and a road system using global and local roads [8]. Iramanesh et al. developed AgentCraft, which introduced the concept of using *agents* for settlements. They simulate a society of in-game "players", whose movements and actions define the settlement. [10]. The concept of using agents was used by various subsequent generators, though they differ in the exact definition of an agent. Esko et al. describe an agent-based algorithm with more abstract agents that makes use of existing city generation techniques. [11]. Van der Staaij et al. take this concept even further with an *iterative* agent simulation, and combine it with several other PCG techniques [17]. Their algorithm currently holds the highest score in the GDMC competition, and is hence the one we primarily use in this work.

Evaluation of settlements. There has also been research into the evaluation of PCG Minecraft settlements. Hervé et al. have investigated the applicability of a large amount of general automated PCG performance metrics to the Minecraft settlement generation domain. While they note that many metrics can be adapted to Minecraft, they did not find strong evidence for a correlation with GDMC judge evaluations for most metrics. And for the metrics with do show correlation, they remark that they appear to be unsuitable to optimize for directly: a settlement needs to be "okay" first [12]. The same authors moved on to investigate automatic metrics based on isovist theory. While this seemed promising, they were limited by a lack of sufficiently detailed human evaluation data to compare with [13]. Finally, Hervé et al. analyzed the specific themes that GDMC judges tend to base their scores on, giving more insight into what aspects make a settlement generator "good" according to humans [16]. Since automated settlement quality metrics do not quite seem ready yet for direct optimization, data-driven methods like ours remain important for Minecraft PCGML.

2.2 Minecraft PCGML

When it comes to PCGML in Minecraft, there are a number of notable related works. One of the earliest is an attempted GDMC submission [18]. Its author developed a 3D version of DCGAN and trained it using samples of human-made Minecraft buildings. They converted their samples to only differentiate between `air` and non-`air` blocks to simplify the problem. Their results were unsatisfactory, but were an important first step for Minecraft PCGML. Better results were achieved by Fumagalli [19] using a similar architecture. They were able to generate simple villages when again only differentiating between `air` and non-`air`.

A major step up was achieved by the authors of *World-GAN* [20], an approach for generating Minecraft world samples based on a single input sample. One of the reasons that the previous works were limited to differentiating between `air` and non-`air` only (which we will refer to as *binary* generation from now on), was the large number of blocks in Minecraft. Version 1.19 of Minecraft (released on June 7, 2022) contains 825 different blocks, and this number continues to grow with each update. This large amount of blocks poses a problem for neural network-based approaches: since block types are categorical, they need to be converted to one-hot vectors before they can be fed into a network. This can require huge amounts of memory for large world samples. For example, a single $64 \times 64 \times 64$ sample of Minecraft 1.19 blocks using 32-bit floating point numbers would require $64 \cdot 64 \cdot 64 \cdot 825 \cdot 32$ b $= 825$ MB to store. This number can be reduced by only considering the blocks that actually appear in the data, but it remains large. Furthermore, it will only increase as more blocks are introduced in the game.

The World-GAN approach solves this problem by introducing *block2vec* [20], a technique inspired by word2vec [24, 25]. They train a two-layer skip-gram model that learns

to predict the surrounding blocks (context) for each block in their input sample. This results in an *embedding vector* for each block: a list of floating point numbers of fixed length that identifies the block. The resulting embedding space can then be used for neural network in- and output. The authors of World-GAN used 32-number embeddings, greatly reducing the required storage for each sample. This way, they were able to support all Minecraft block types. Our work also leverages the word2vec technique.

The authors later extended their method to *Wor(l)d-GAN* [21], which can additionally make use of a pretrained language model (BERT) instead of block2vec. The idea is to embed the English display names of the blocks (e.g. "stone bricks") using the pretrained model, rather than training a custom block2vec model. By using some additional tricks such as prompt editing, the authors were able to achieve performance similar to block2vec. This method is however not directly applicable to our problem, since we use a different notion of "block" that contains more data than just the name.

Both World-GAN and Wor(l)d-GAN train on a single world sample, and output samples that are similar to the input sample at multiple scales. They use a generative adversarial network (GAN) architecture based on SinGAN [26] to achieve this. Both methods achieve good results for landscapes and organic structures, but perform worse for complex structures such as Minecraft villages.

A completely different approach to Minecraft PCGML is taken by Chen et al. [22]. Instead of using a model to generate whole samples at once, they instead train a model to predict the next block to be placed based on a sequence of previous blocks. They collected a substantial dataset of structures with ordered block placements by recording the actions of human players using a modified Minecraft server. Their results still have some flaws, but they are generally quite impressive. However, the need for ordered block placement data is a significant limiting factor: this data is unavailable in most Minecraft datasets. Our dataset lacks order data as well, so this approach is not applicable in our case.

An important difference between our approach and all of the aforementioned methods, is that we attempt to condition the generation on the initial Minecraft terrain. The aforementioned methods only train a model to generate structures in isolation, without taking initial terrain into account. Although World-GAN [20] is able to generate samples based on an input sample, it is still not quite terrain-adaptive: it can generate variations of its input sample or refine a structure manually added to the input sample, but it cannot learn to convert from one sample distribution to another. We aim to leverage the terrain-adaptiveness of GDMC submissions to train a terrain-adaptive PCGML model.

Another limitation that all existing Minecraft PCGML approaches share, is that they only support a subset of all possible Minecraft blocks. This is obvious for the binary approaches, but even the methods that support all block *types* still do not quite support all possible "configurations" a Minecraft block can have. Our dataset creation system extracts additional block information that brings the number of representable block configurations much closer to the full amount. We describe this in more detail in Section 3.2.

2.3 GAN Image-to-image translation

To achieve terrain-adaptive generation, we need a model that can learn to translate a 3D Minecraft sample to another 3D Minecraft sample. While there is little prior work on "volume-to-volume" translation, a substantial amount of work exists for the 2D equivalent — *image-to-image translation*. This work is generally focused on regular three-channel color images, but is often applicable for our many-channel problem as well.

Many image-to-image approaches are based on the Generative Adversarial Network (GAN) architecture [27]. We will give a brief overview of these here. There is also research on other techniques for the problem, such as diffusion models and direct regression, but we do not explore these in this work.

An important baseline for image-to-image translation is pix2pix [28], a general GAN architecture that can be applied to any image-to-image task. It uses a generator with a "U-net" structure, which shortcuts low-level information from the early layers of the network to the later ones [29].

Several GAN approaches followed pix2pix, each achieving improved performance at the image-to-image task. These include pix2pixHD [30] (which is actually quite different from pix2pix architecturally), Sketchygan [31], StyleGAN [32], SPADE [33] and CoCosNet v2 [34]. Except for CoCosNet, these approaches are however all evaluated on more specific image-to-image tasks, such as image generation from segmentation maps [30, 33], style transfer [32] or sketch-to-image translation [31]. Bissoto et al. have also semi-recently written a survey that touches on GAN image-to-image translation [35].

There are also GAN architectures that can learn the harder task of *unpaired* image-to-image translation — learning to translate from one image domain to the other based on two completely separate sets of images. The first of these was CycleGAN [23], which is still an important baseline. A more recent architecture is UVCGANv2, whose authors list a more comprehensive overview of GAN approaches [36]. We do not require an unpaired image-to-image system, since utilizing GDMC generators allows us to create paired examples.

As we touched upon in the previous section, SinGAN (the architecture behind World-GAN [20]) addresses another variation of the image-to-image translation problem: it can learn to generate variations based on only a single learning image [26]. It does so by training a pyramid of GANs on multiple scales of resolution. We do not require a single-example method like SinGAN either, since our method allows us to create many examples in an automated fashion.

Most works on GAN image-to-image translation can in theory be extended to 3D. This concept is explored by Cirillo et al.: they implement a 3D equivalent of pix2pix called vox2vox, and evaluate it on brain tumor segmentation problems [37]. We also experiment with vox2vox, though we implement a slightly modified version.

3 Dataset creation

In order to train machine learning models to replicate GDMC generators, we first needed to construct a dataset of generation examples.

3.1 Automated dataset generator

3.1.1 Overall design

Other works that explore generative modeling in Minecraft have created datasets by manually extracting buildings from public Minecraft maps using Minecraft map editors like MCEdit. [20, 21, 18, 19]. However, since we aimed to create a huge amount of examples in an automated fashion, such a manual method would not have been suitable.

Zhuoyuan Chen et al. used a modified Minecraft server that automatically records all block placements. [22], including the order of placement. This method was however not suitable for our purposes either. It does not allow us to capture the original terrain, and it requires the use of a non-standard Minecraft server that might not support the modifications that the GDMC generators need to run.

To overcome the limitations of the existing methods, we developed our own automated dataset creation system. Its overall design is shown in Figure 2. The system is capable of pregenerating natural Minecraft terrain, running an external settlement generator program at multiple locations in the world, and saving the world's blocks in a convenient format. It combines many existing technologies to automate the entire process, and it contains extensive checkpointing and retrying logic to handle any errors that may occur.

The final result is two datasets of 3D minecraft samples, one for the original terrain and one for the terrain with the settlement generator's output on it. By matching up samples from each of these datasets index-wise, we can then obtain a dataset of before-after sample pairs.

3.1.2 Supported settlement generators

The system currently only supports settlement generator programs that are based on the GDMC HTTP Interface, and only the version that was used in the 2022 GDMC competition (though it can easily be extended to other input methods). The GDMC HTTP Interface is a submission method for the GDMC competition that "allows you to interact with a live Minecraft world using standard web requests" [40]. The interface is implemented by GDMC-HTTP [41], a modification ("mod") for Minecraft. We chose to focus on supporting generators based on the GDMC HTTP Interface because it was by far the most commonly used GDMC submission method in 2022, being used by ten of the eleven submissions.[3]

We initially planned to "mock" the HTTP interface (i.e. listen to the same endpoints to intercept all block placements), but we dropped this idea for various reasons. It would have required us to re-implement multiple live Minecraft effects, such as the flowing of water and the adjustment of certain blocks to their neighbors. It would also have strictly limited our system to GDMC HTTP Interface-based generators. Instead, we decided to automate the process of running the generators in an actual Minecraft server instance. This achieves the best accuracy, and makes the system far easier to extend to other world interaction methods than the GDMC HTTP Interface.

3.1.3 Minecraft server mangement

Our system manages a Minecraft server by running the server software as a subprocess. It can shut down and restart the server as needed (to recover from crashes, for example). The server software provides a command-line interface, which we manipulate by reading and writing to the process' input and output. The command-line interface was however not designed for this usage, which led to some difficulties. For example, messages can be posted in any order, making it difficult to isolate the output of an executed command. To get around this, we also made use of Minecraft's RCON protocol, which allows you to remotely send commands to a running Minecraft server and get their output back.

3.1.4 Terrain pregeneration

The first task of the dataset generator system is the pregeneration of the natural terrain using Minecraft's own terrain generator. Here, we were somewhat in luck: this task is also commonly performed by Minecraft server administrators to reduce lag when players are exploring the world, so many tools were readily available.

We considered using the modified Minecraft server programs Spigot[4] and Paper[5], which support efficient asynchronous terrain pregeneration, but these server programs turned out to be incompatible with the GDMC-HTTP mod.[6] Instead, we chose to use the *Chunk-Pregenerator* mod [42], which we control via RCON. This mod also allowed us to reset (regenerate) parts of the world, which was very useful for error handling purposes. We did run into some memory issues with the mod, but these could be fixed by allocating more server memory, recursively splitting the areas to pregenerate into smaller batches and periodically restarting the server.

[3] https://gendesignmc.engineering.nyu.edu/results
[4] https://www.spigotmc.org/
[5] https://papermc.io/
[6] The GDMC-HTTP mod is a Forge mod, and Forge is in principle not compatible with Spigot or Paper.

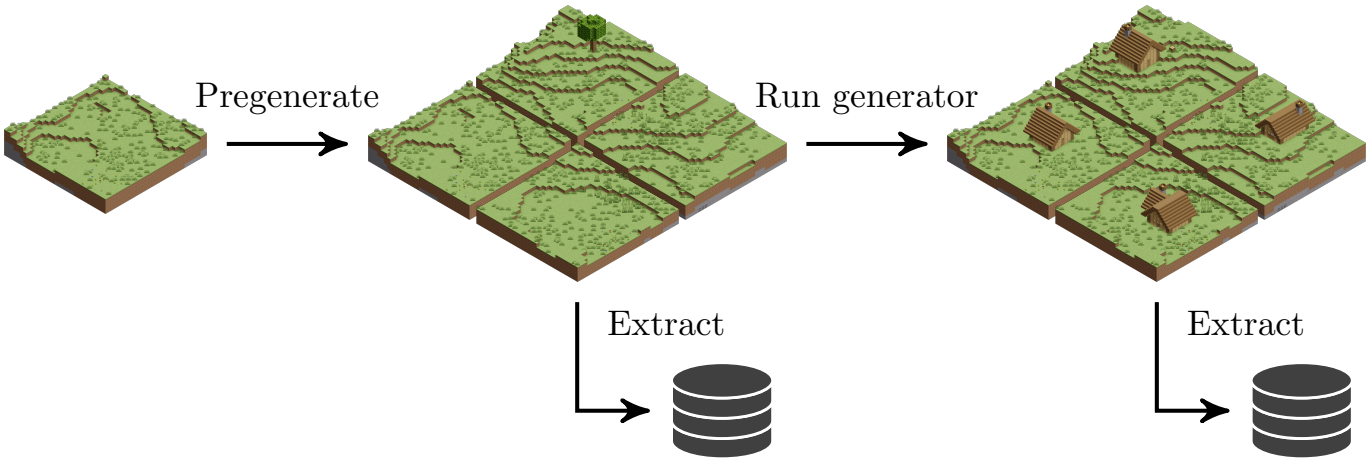


Figure 2: A high-level overview of our automated dataset creation system. It pregenerates the natural Minecraft terrain, extracts it, then runs a settlement generator at multiple locations and extracts those as well. The final result is a dataset of before- and a dataset of after-samples, which can later be combined into a paired dataset. All Minecraft renders are made with Mineways [38] and Blender [39].

3.1.5 Settlement generation

The second task is to run a settlement generator program in the managed server. As we stated before, the system currently only supports GDMC HTTP Interface-based generators. To run these generators, we installed the GDMC-HTTP mod in the managed server. The GDMC-HTTP mod provides a command to control the area where the settlement generator should build (the *build area*), which we again call using RCON. This way, we can sequentially execute the generator in multiple areas of the world. Just like the server, we run the settlement generator programs as isolated processes.

3.1.6 Data extraction

The last task is to extract samples from the world and save them to disk in a convenient format. This needs to happen both after terrain pregeneration and after settlement generator execution. Prior works that store Minecraft world samples have used Minecraft's regular world format [20, 21] or made use of the unofficial *schematic* format that most Minecraft world editors use [18, 19]. We instead chose to design a custom format that is much easier to manipulate programmatically for preprocessing and machine learning purposes. We describe it further in Section 3.2. To extract the blocks from Minecraft's world format, we made use of Amulet Core [43], a Python library that provides the core functionalities of the Amulet world editor [44].

3.1.7 Other design aspects

Both terrain pregeneration and settlement generator execution can be disabled, and it is possible to specify an initial world file. These features make it possible to first create a dataset of natural terrain samples, and then re-use the pregenerated world to run multiple different settlement generators.

When executing a settlement generator, we run at grid-aligned locations to use as much of the natural terrain as possible. The result of this can be seen in Figure 3.

Because of the way that Minecraft saves worlds, we only used build areas that are a multiple of 16×16 blocks in size. Minecraft's world format separates worlds into 16×16 units (spanning the entire world height), which are called *chunks*. The game generates terrain one whole chunk at a time, which means that 16×16 is the minimum size at which we can generate or reset terrain using terrain manipulation tools like *Chunk pregenerator*. Reading or writing blocks is also significantly more performant when working with full chunks at a time.

3.1.8 Error management

The dataset creation system includes many checks and failsafes to ensure it can keep running for long periods of time. The datasets we used for experimentation required time in the order of weeks to be generated (see Section 3.3.3), which made these error handling measures essential.

The entire generation is performed in a checkpointed fashion: the system regularly saves intermediate results, and can recover from a checkpoint in the case of a complete crash. Furthermore, Minecraft server commands are always sent with a timeout. If the server times out or if it crashes, it is automatically shut down and restarted. Settlement generators are also executed with timeouts, and they too are restarted on timeout or crash. When a settlement generator is re-executed at the same grid position, the terrain of that grid position is first reset to ensure a clean slate. There is also a limit on the number of generator retries to guarantee an upper bound on the dataset creation time. In Figure 3, the bottom right settlement failed to generate in time and was thus skipped.

Figure 3: Grid arrangement of multiple small settlements generated by the algorithm from Van der Staaij et al. [17], as produced during the settlement generation step of our dataset creation system.

3.2 Dataset format

Our dataset format is quite simple. We store all encountered unique blocks in a JSON palette file, which allows us to store all world samples in a single NumPy array file of palette indices. The NumPy array is five-dimensional, with the first two axes indicating the 2D sample position and the last three axes indicating the 3D block position within the sample. This format is conceptually very similar to Minecraft's own world format, but it is much easier to parse and manipulate.

We need to go on a small detour to explain what "blocks" are, exactly. In Minecraft, every grid position is occupied by a block ("empty" positions contain air blocks). All blocks have an *ID*, like `stone` or `oak_planks`[7], but some blocks have additional information attached to them. In general, blocks consist of three components: a block ID, optional *block states*[8] and optional *block entity data*. Block states are simple key-value properties that usually denote basic variations in the state of a block. For example, a stairs block can be facing in one of six possible directions. Block entity data is used by only a few block types to store particularly complex data, such as the items in a chest or the text on a sign. Figure 4 shows a screenshot of a stairs block and its block ID and block states.

Because of the complexity of Minecraft blocks, many other works that explore generative modeling in Minecraft make some simplifications. Most commonly, only the block ID is used [20, 21, 22]. Some works simplify even further by only considering a subset of all block IDs [19], or even only differentiating between `air` and non-`air` blocks [18, 19]. To our knowledge, there is no work that attempts to model block states or block entity data.

Figure 4: A screenshot of a minecraft block and some of its technical data. We have highlighted the block ID in red and the block states in yellow.

We have chosen to leave out block entity data, but include block states in our dataset format. Most block states are fairly unimportant, but some can be very noticeable. For example, the models of [20, 21, 22] always place stairs blocks in the same direction. From this point on, when we speak of blocks, we mean the tuple (block ID, block states).

3.3 The final datasets

3.3.1 Generators

We used our dataset creation system to create a dataset of samples for original Minecraft terrain and a number of different "settlement" generators, including two trivial generators that we made to serve as simple tests for our machine learning systems. The full list of generators for which we made datasets is as follows:

[7] Technically, a *namespaced* ID (`minecraft:stone`), but this is only relevant when using mods that add blocks

[8] Block states are sometimes called *block properties*, and the tuple (block ID, block states) is also sometimes referred to as a *BlockState*. The terminology is confusing.

1. **Original terrain** (build areas 64×64 and 96×96). The built-in Minecraft terrain generator. An example is shown in Figure 5a. All other generators are executed on top of this pregenerated Minecraft terrain. We also trained our machine learning systems using sample pairs where both the before- and after-sample were the original terrain, in order to evaluate whether they can learn to simply replicate the input.
2. **Ring** (build area 96×96). A test generator that adds a ring (actually, a rectangle) of `red_concrete` blocks around the generation area at a fixed height, independent of the original terrain. An example is shown in Figure 5b. We made this generator to evaluate whether our machine learning systems can learn to add a fixed global structure while still maintaining the original terrain.
3. **Adaptive ring** (build area 96×96). A test generator that adds a terrain-adaptive ring of `red_concrete` blocks on the ground. An example is shown in Figure 5c. We made this generator to evaluate whether our machine learning systems can learn to replicate a trivial terrain-adaptive generator.
4. **Mike's Angels** (build area 96×96). The generator created by Van der Staaij et al. [17] (team name "Mike's Angels"). It uses an iterative agent-based algorithm to generate large medieval settlements. Examples are shown in Figure 5d and Figure 3. We found 96×96 to be the minimum (multiple of 16) size at which the generator produced representative results.
5. **Mike's Angels wall** (build area 64×64). An edited version of the previous generator that only builds the city wall. An example is shown in Figure 5e. Note that we disabled tree removal as well. We made this version because we did not manage to achieve good replication results for the full generator (Section 6), and this limited version served as a simpler but still non-trivial alternative.

We used Minecraft 1.16.5 to generate the original terrain, which is the version that was used in the GDMC 2022 competition. Minecraft allows you to submit a seed for the random number generator when creating a world. We used a handpicked seed that resulted in a world which contained nearly all biomes (Minecraft terrain types like "plains" and "forest") in the area over which we collected samples[9].

3.3.2 Datasets

Using these generators, we created six (unpaired) raw datasets. For the *Ring* and *Adaptive ring* generators, which do not display much variance, we created datasets of 1024 samples. For the other generators, we collected 16 384 samples. The largest of our datasets span a surface area of $12\,288 \times 12\,288$ blocks, and cover multiple billions of blocks in total. Detailed properties of our datasets are shown in Table 1.

Most dataset creation jobs had none or very low amounts of failed samples. The exception is *Mikes Angels wall*, where approximately 36% of the samples failed. This is likely due to the fact that the *Mikes Angels* generator was intended for much larger build areas, causing it to fail quite often when only given 64×64 build areas.

3.3.3 Performance

The dataset creation process is quite time-consuming. We will briefly discuss the system's performance. We ran all dataset creation jobs on a (shared) machine with 64 *Intel Xeon E5-4667v3* cores (2.00GHz), 1 TB RAM and local SSD storage.

Pregenerating the natural terrain and collecting the dataset of 16 384 samples of size 96×96 took 60 hours in total. Although we do not have exact profiling data (because this would have slowed down the process further), it seems that a large amount of time was spent stopping and restarting the managed Minecraft server. We needed to make the server regularly restart because we found that the *Chunk pregenerator* mod could freeze up otherwise. Using a different chunk pregeneration method might help to speed up this step. Furthermore, the data extraction could perhaps be sped up with parallelization.

Out of all generator datasets, the *Mike's Angels* dataset took the longest to complete, at approximately 18.2 days. We tightened certain timeouts during the generation process to speed it up. If we had used these settings from the beginning, we project the process would have taken about 13.4 days. Most of this time was spent executing the *Mike's Angels* generator itself, for which the performance is out of our control.

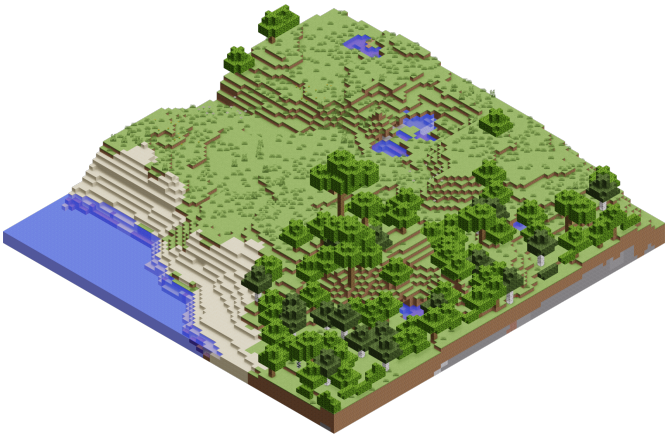
4 Data preprocessing

We performed several pre-processing operations on the raw datasets described in Section 3.3 before using them to train models. We consider this preprocessing separate from the dataset creation process itself, because it is more specifically targeted at preparing the data for our particular machine learning models, and because some of them are destructive in nature. The preprocessing step is also where we "stack" datasets together to create datasets of *paired* examples.

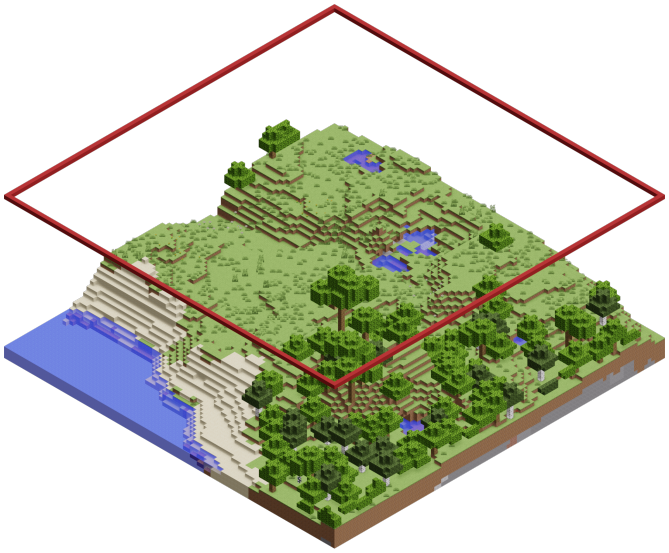
4.1 Preprocessing tool

We have developed a tool to perform multiple kinds of preprocessing, most of which can be chained together. We will describe the various modes of this tool in the order that we typically apply them.

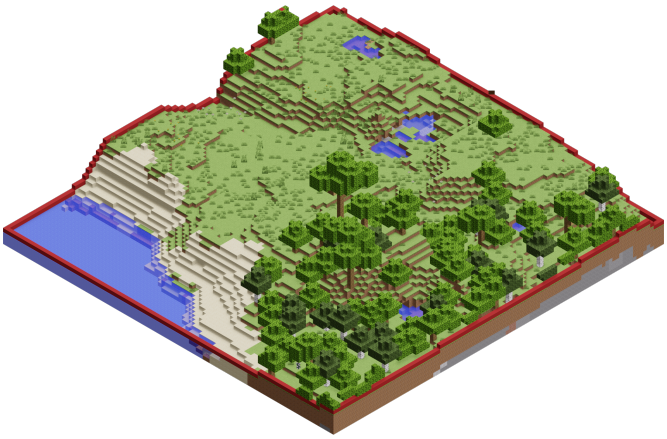
[9] We used seed `8700554514277685781`, and we collected samples in square areas from $(x, z) = (208, 208)$ towards positive x and z.



(a) Original terrain



(b) Ring



(c) Adaptive ring

(d) Mike's Angels

(e) Mike's Angels wall

Figure 5: Example outputs of the generators that we used to construct datasets, and which we attempted to replicate via machine learning. Note that we used a 64×64 build area for the *Mike's Angels wall* generator, even though the depicted area is 96×96.

Table 1: Properties of the six raw datasets we created using our automated system. Only the successfully generated samples are counted for *Number of samples* and *Total block count*.

Generator	Build area size (blocks)	Sample height (blocks)	Number of samples	Total block count	Number of different blocks	Size on disk (GiB)
Original terrain	64×64	64	16 384	4 294 967 296	1171	8.1
Original terrain	96×96	97	16 384	14 646 509 568	1322	28.0
Ring	96×96	97	1024	915 406 848	1323	1.8
Adaptive ring	96×96	97	1024	915 406 848	1323	1.8
Mike's Angels	96×96	97	16 382	14 644 721 664	2170	28.0
Mike's Angels wall	64×64	64	10 565	2 769 551 360	1264	8.1

1. **Stack datasets.** "Stacks" two datasets into one dataset of paired examples. Given two datasets with block arrays where the first two axes indicate the 2D sample position and the last three axes indicate the 3D block position within the sample (as described in Section 3.2), the result is a dataset with a block array where the first axis indicates the flattened sample index, the second axis indicates the pair side (0 or 1) and the last three axes again indicate the 3D block position within the sample. The block palettes of the two input datasets are merged, and the palette indices of the input datasets are updated accordingly.

   Corresponding samples from each input dataset are paired according to their sample position. That means the input datasets should be generated starting from the same origin point in Minecraft, which is the case for our raw datasets. If one of the input datasets contains more samples than the other, it is possible to specify where to overlay the smaller dataset on the larger one. The operation flattens the sample axes into a single one at the end of the stacking operation because the distinction is no longer needed afterwards.

2. **Slice array.** "Slices" the block array axes according to the given specification, reducing the dataset's size. We used this mainly to reduce the size of each sample in the Y-axis (the upward direction in Minecraft), as we generated most raw datasets with extra large Y-ranges for safety. For example, slicing the sample Y-axis to `16:-32` would remove the bottom 16 block layers and the top 32 block layers of each sample.

3. **Remove invalid samples.** Removes samples from the dataset that contain a block index outside the range of its palette. We used this to fix some malformed datasets early in our development process. It is no longer required, but we still apply it for safety.

4. **Prune unused blocks.** This operation and the next two have a similar purpose: to reduce the size of a dataset's palette. As explained in Section 3.2, we distinguish blocks by not only their *ID*, but also their *block states*. This leads to a much larger number of different blocks to deal with: our raw *Mike's Angels* dataset contains 2170 distinct blocks, even though Minecraft 1.16.5 only has 681 block IDs in total. This large amount of blocks makes training ML models more difficult, so we attempt to reduce the palette sizes in various ways.

   This operation reduces the palette size in a mostly non-destructive way: it removes all palette entries that are not actually referenced in the array. Such "dangling" entries can occur when one of the previous three preprocessing operations removes all references to an entry.

5. **Remove automatic block states.** This operation removes palette entries more aggressively. It removes all block states (Section 3.2) that we have classified as *automatic*, and merges palette entries for blocks that have become identical after these removals.

   We consider a block state to be automatic if Minecraft automatically re-assigns its value on block placement. Typical examples are the block states that govern the shape of stairs and fences: these states are determined by the block's surroundings when it is placed. This behavior is depicted in Figure 6. Our machine learning models do not need to learn how to handle these kinds of block states, so we can safely remove them.

   We have identified a total of 20 automatic block states. Note however that we do make a simplification here: the operation removes block states solely based on their name (e.g. `shape` for stairs), even though the same block state name can occasionally have different meanings for different block IDs. We believe this to not be a huge concern.

6. **Remove flow blocks.** Removes all blocks that represent flowing liquid by replacing them with air. Minecraft has two liquids: water and lava. We will not discuss the game's liquid mechanics in detail, but the basics of it are that there are *source blocks* from which liquids originate, and flow blocks that disappear when all their source blocks are removed. This preprocessing operation removes all liquid blocks except for the sources.

   Liquids are not guaranteed to flow in the exact same way when the flow blocks are recalculated, so this operation is destructive. We consider the removal of a few more palette entries to be worth this cost.

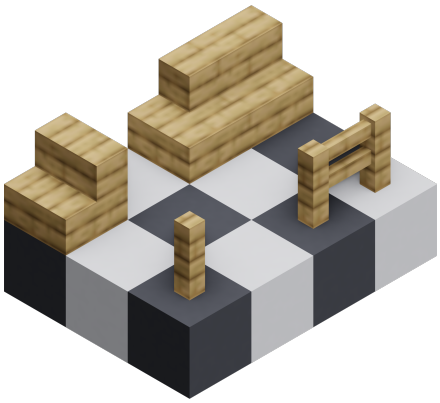


Figure 6: Stairs and fences adjust their shape according to the adjacent blocks. The differently-shaped versions are considered distinct by our dataset creation system, but are merged during preprocessing.

7. **Remove equal pairs.** Can only be applied on a stacked dataset. Removes all pairs of identical samples.

   Our dataset creation system can skip samples when they fail to generate too many times (Section 3.1). However, it does not actually leave out these samples. It still extracts the samples as normal, though they will equal the original terrain. This preprocessing step removes failed samples by removing pairs of identical samples. Naturally, we do not apply this operation when we intentionally create a dataset of equal before- and after-samples.

8. **Sort palette by frequency.** Sorts the dataset's palette by block frequency, and reindexes its array accordingly. We used this mainly for debugging and visualization purposes. It does not functionally alter the dataset.

9. **Convert to binary.** Converts a dataset to *binary* form. This removes the block palette entirely, and turns the index array into a binary map that only indicates whether a position contains `air` or non-`air`. We used this to train models on binary versions of the settlement generators.

Some of the (intermediate) NumPy arrays handled by the preprocessing tool are very large, and can even exceed available memory — our raw dataset arrays already have sizes of up to 28 GiB (Section 3.3.2). To manipulate arrays that do not fit in memory, we made use of NumPy's memory mapping feature, which uses the array file on disk as a "backing cache" at the cost of performance. To minimize the performance hit, the preprocessing tool dynamically decides whether to memory map or to directly read the arrays depending on how much memory is still available.

4.2 Preprocessed datasets

We used our preprocessing tool to create ten preprocessed datasets of paired examples. We only used natural Minecraft terrain as the first pair half. For each of the five generators described in Section 3.3.1 (including the original terrain), we stacked their raw datasets with an original terrain dataset, and created both a categorical and a binary preprocessed paired dataset. For the paired dataset where both pair halves are original terrain, we used the 96×96 version. We will refer to each of these pair datasets using the name of the generator that was used for the second pair half, except for the double original terrain dataset, which we will refer to as *Terrain copy*. The pairings we used, and the names we use for them, are also shown in Table 2.

Table 3 shows the properties of our ten preprocessed pair datasets. Note how the amount of different blocks has been significantly reduced during preprocessing. For example, the raw *Mike's Angels* dataset contained 2170 different blocks, while the preprocessed *Original terrain – Mike's Angels* dataset contains only 1297.

5 Model development

Like most of the existing work in image-to-image translation and Minecraft PCGML, we make use of generative adversarial networks (GANs) [27]. We implemented two volume-to-volume GAN architectures, and experimented with several hyperparameter configurations.

5.1 Learning task

For both architectures, we trained both binary versions, which only learn to differentiate between `air` and non-`air`, and categorical versions, which differentiate between all blocks. In all cases, we trained on the task of converting the first sample from a pair from one of our datasets to the second sample. This means that our models needed to learn to both copy the terrain from the first sample, and to add the correct structure or settlement on top of it.

5.2 Block embeddings

5.2.1 block2vec

To train models on the categorical datasets, we need a way to represent categorical Minecraft blocks as numbers. As we have mentioned in Section 2.2, it is not feasible to simply use one-hot vectors. There are too many different blocks, making one-hot vectors cost too much memory. This problem is even more prominent for our work than for existing Minecraft PCGML works due to our increased number of distinct blocks: storing one sample of the *Mike's Angels* dataset with 32-bit floating point one-hot vectors would require $96 \cdot 96 \cdot 64 \cdot 1297 \cdot 32b \approx 22.8GiB$.

We address this problem using *block2vec*, a technique proposed by the authors of World-GAN [20] that is inspired by the word2vec method from natural language processing [24, 25]. The NLP word2vec method involves training a model to predict a word from its context (continuous bag

Table 2: Overview of the raw dataset pairings we used.

Raw dataset 1	Raw dataset 2	Pair name
Original terrain (96 × 96)	Original terrain (96 × 96)	Terrain copy
Original terrain (96 × 96)	Ring (96 × 96)	Ring
Original terrain (96 × 96)	Adaptive Ring (96 × 96)	Adaptive ring
Original terrain (96 × 96)	Mike's Angels (96 × 96)	Mike's Angels
Original terrain (64 × 64)	Mike's Angels Wall (64 × 64)	Mike's Angels wall

Table 3: Properties of the ten preprocessed pair datasets we used for machine learning.

Name	Build area size (blocks)	Sample height (blocks)	Number of samples	Total block count	Number of different blocks	Size on disk (GiB)
Terrain copy	96 × 96	64	16 384	9 663 676 416	825	37.0
Binary	96 × 96	64	16 384	9 663 676 416	–	19.0
Ring	96 × 96	64	1024	603 979 776	580	2.3
Binary	96 × 96	64	1024	603 979 776	–	1.2
Adaptive ring	96 × 96	64	1024	603 979 776	580	2.3
Binary	96 × 96	64	1024	603 979 776	–	1.2
Mike's Angels	96 × 96	64	16 382	9 662 496 768	1297	36.0
Binary	96 × 96	64	16 382	9 662 496 768	–	18.0
Mike's Angels wall	64 × 64	64	10 565	2 769 551 360	843	11.0
Binary	64 × 64	64	10 565	2 769 551 360	–	5.2

of words model) or the context from a word (continuous skip-gram model) and extracting fixed-length *embedding vectors* from the model's embedding layer. In block2vec's case, *words* are replaced with *blocks*, and a block's *context* is represented by its surrounding blocks.

The resulting dense embedding vectors can be used in place of sparse one-hot vectors. This is possible because the distances between blocks in each embedding space dimension are meaningful. Blocks that often appear close together in the data will be embedded to vectors that are close together as well.

5.2.2 block2vec configuration

Similar to World-GAN, we use a two-layer skip-gram model. We used embedding vectors of size 32, a size that the authors of World-GAN found to give good results [20]. This means that, in the case of the *Mikes Angels* dataset, we reduce the size of each block vector from 1297 (one-hot vectors) to 32 — a 97.5% decrease.

We also follow World-GAN by using a context radius of 1, which means that the context of each block consists of the 26 blocks that touch it directly or diagonally. We do not sample blocks on any of the edges of a dataset example, since we cannot form a full context around them. Edge blocks can still appear as context for other blocks, however.

World-GAN trains a skip-gram model to predict only one context block, and averages the resulting losses for all the ground truth context blocks. We instead predict each of the 26 context blocks separately, and compute separate losses. We expect that this helps the model to, for example, differentiate between blocks that often have some other block below them and blocks that often have the same other block above them.

The NLP word2vec has a feature called *subsampling*, which increases the probability of sampling a rare word when training by discarding words based on their frequency [25]. The World-GAN authors use a variant of this feature for block2vec as well, to reduce the chance of sampling common blocks like `air` [20]. Unlike word2vec, block2vec does not remove skipped blocks from the dataset entirely for the rest of the epoch, nor does it use subsampling for a block's context. We expect this is because shifting tokens to fill the gap of a removed one is much more involved in 3D than it is in 2D.

We also employ subsampling. Like World-GAN [20], we follow the formula from word2vec [25][10]:

$$P(b_i) = 1 - \sqrt{\frac{f(b_i)}{t} + 1} \cdot \frac{t}{f(b_i)}$$

Here, b_i denotes a block (e.g. `stone`), $f(b_i)$ denotes the block's relative frequency in the data, and $P(b_i)$ denotes the "probability"[11] of discarding any b_i. The hyperparameter t controls the aggressiveness of the subsampling. We used $t = 0.001$, which results in relatively strong subsampling.

[10] This formula actually comes from the word2vec implementation; the formula described in the paper is slightly different.

[11] The "probability" can become negative for very rare tokens, in which case the token is never discarded.

5.2.3 Our training process

We train block2vec on only the second pair half of each of our pair datasets. Using at least the second pair half is necessary, since it often contains many blocks that do not appear in the first half (such as the various materials of the city wall for *Mikes Angels wall*). While we could use the first pair half as well, we expect that the block distribution in the first pair half carries less useful information than that of the second pair half.

Even with this limitation, we have far more data available to train block2vec than World-GAN. World-GAN learns from only a single example, and therefore trains block2vec using a single example as well [20]. We instead train on datasets of up to 16 384 samples, which are each similar in size to the largest samples used for World-GAN. Our largest pair dataset, *Terrain copy*, contains a staggering $(96-2)^2 \cdot (64-2) \cdot 16384 = 8\,975\,679\,488$ possible block-context examples.

Because our amount of examples is so large, we do not train on all of them — let alone run multiple epochs. For each dataset, we instead train on the fixed amount of 560 979 968 examples (randomly chosen). This is equivalent to 1024 full $96 \times 64 \times 96$ samples. Subsampling can however still cause more block-context examples to be "visited". Since we do not train on all blocks in the data directly, subsampling becomes even more important to decrease the chance that we miss a rare block.

Figure 7 shows our learned embeddings for the 25 most common blocks in the *Mikes Angels wall* dataset, reduced to two dimensions with a dimensionality reduction algorithm. We can clearly see some patterns, indicating that the algorithm has worked. All leaf types are mapped closely together on the top left; dirt, grass blocks and grass are grouped in the middle; blocks generally become more "underground" as they are placed lower in the graph; and the blocks that constitute the city wall are all near the top right.

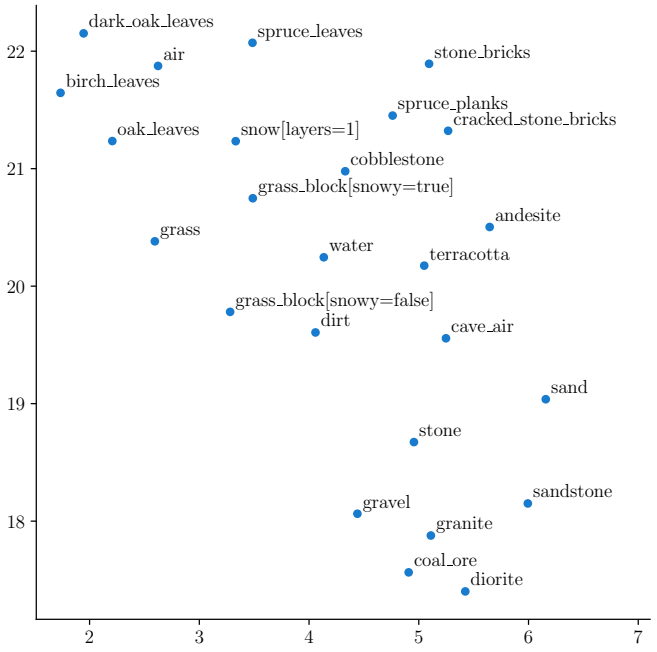


Figure 7: Embeddings learned by block2vec [20] for the 25 most common blocks in the *Mikes Angels wall* dataset. Texts in square brackets indicate block state values. The embedding vectors were reduced from 32 dimensions to two dimensions using Uniform Manifold Approximation and Projection (UMAP) [45]. Note that UMAP is not deterministic: the reduction can vary between executions.

5.2.4 Converting back to blocks

Although we can use block embeddings throughout the entire machine learning process, we do need to convert back to categorical blocks when we want to build an output in Minecraft. A common way to map vectors from an embedding space back to categorical values (in this case, blocks) is to map them to the value whose embedding is closest according to some distance metric. This is also what World-GAN does (using Euclidean distance as the metric) [20]. We use the same technique, but we extended it with an additional feature.

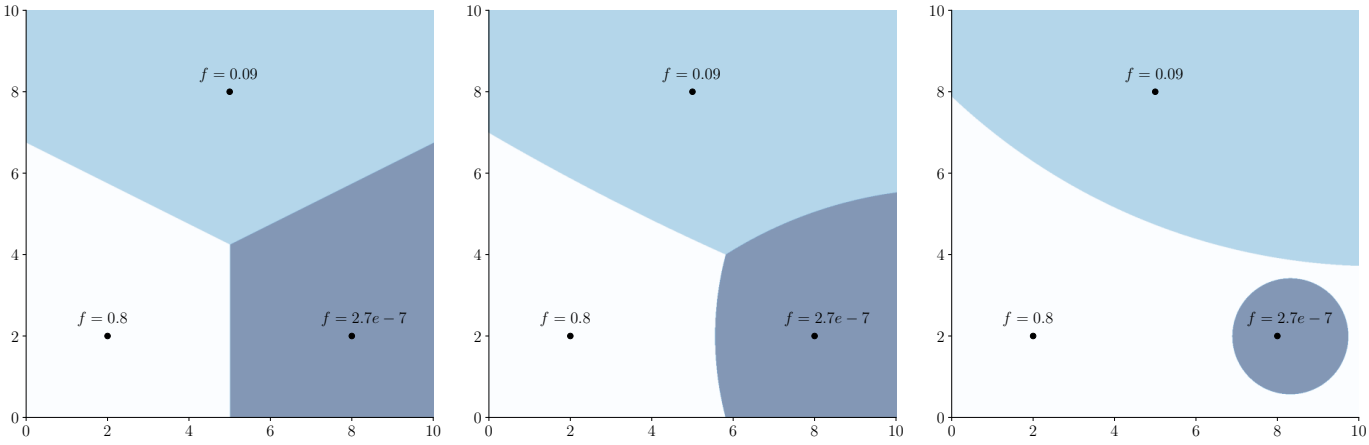
During early experimentation, we found that considering only distance would result in excessively many rare blocks. To address this, we made an addition that we call *frequency weighting* (or *block frequency weighting* in the context of Minecraft): we give a greater "weight" to blocks that were more common in the input dataset.

The full formula we use to determine which block b^* to map an embedding vector e to is as follows:

$$b^* = \operatorname*{argmin}_{b \in B} \sqrt{\sum_{i=1}^{32} (e(b)_i - e_i)^2} \cdot f(b)^{-w}$$

Here, B denotes the set of all the blocks in the dataset, $e(b)$ denotes the block2vec embedding of block b, and $f(b)$ denotes the relative frequency of b. The hyperparameter w ("frequency weighting exponent") determines how much to bias towards more common blocks: $w = 0$ means no bias and $w > 0$ causes a preference for common blocks. The hyperparameter w needs to be an exponent (rather than a factor) because the frequency weighting already works multiplicatively. The frequency weighting feature is not limited to Minecraft: it could be applied to other domains as well.

The mapping defined by this formula is shown visually in Figure 8, in particular its frequency weighting component. We found that values for w in the range $[0, 0.075]$ worked best for our models: any higher, and nearly all vectors got mapped to `air` (the most common block). Negative values for w promote more rare blocks, which we found to not be helpful for our models. We give more details about the effect of different values of w on our model output in Section 6.



(a) $w = 0$ (b) $w = 0.05$ (c) $w = 0.1$

Figure 8: Visualization of how the embedding space gets mapped back to categorical values, for increasing frequency weighting exponents w. The three points in each diagram denote hypothetical embeddings of three categorical values, and are annotated with their (hypothetical) relative frequency. The colored regions indicate the vectors that are mapped to their corresponding categorical values. As w increases, more common values "attract" more vectors.
To better show the effect of specific w values on datasets similar to ours, we used frequencies from our *Mike's Angels wall* dataset: an extremely common block (`air`, $f \approx 0.8$, white region), a common block (`stone`, $f \approx 0.09$, light blue region) and a rare block (`melon`, $f \approx 2.7 \times 10^{-7}$, dark blue region).
The graphs are equivalent to multiplicatively weighted Voronoi diagrams with weights f^w. We approximated them discretely at 1000×1000 resolution.

5.3 vox2vox

The first architecture we implemented and experimented with was *vox2vox* [37], a slightly modified 3D version of pix2pix [28].

Both architectures use a *U-net* architecture [29] for the generator. This architecture receives the condition image as an input. In vox2vox's case, the U-net consists of four convolutional downsampling blocks that each cut the spatial size of the input in half while doubling the number of samples, then four residual convolutional blocks that keep the spatial size constant, and finally four convolutional upsampling blocks that bring the spatial size back to that of the input. There are shortcut connections between downsampling blocks and upsampling blocks with the same size (for example, the first downsampling layer shortcuts to the last upsampling layer), which give the network a U-shape. All convolutional blocks consist of a $4 \times 4 \times 4$ convolution layer followed by instance normalization, dropout (20%) and a normal or leaky ReLU operation.

The main advantage of the U-net architecture is that it allows later blocks to access low-level information from the input, a property that we expected to be useful for the replication of the original terrain. Unlike many other GAN architectures, vox2vox (and pix2pix) do not feed a noise image to the generator. Rather, they introduce randomness in the generated samples by applying dropout at evaluation time. While this could limit the degree of variation in output for a single input image, it is not a huge concern as this type of variation is not very important for our learning task.

The architecture's discriminator receives a concatenation of a (real or generated) target sample and the condition sample as input, and applies four downsampling convolutional blocks followed by one convolutional block that reduces the feature count to 1. Because the discriminator is fully convolutional, its output may contain more than one element. By default, the receptive field of an output pixel is $94 \times 94 \times 94$, though we experimented with different values. During training, GAN losses are averaged over all output pixels. The pix2pix authors call this technique *patchGAN*, since the model only learns structures at the scale of $(94 \times 94 \times 94)$ patches.

An advantage of using a fully convolutional generator and discriminator is that the model can be trained with and applied to samples of varying size. We do however not make use of this feature in our experiments.

We used vox2vox mostly as-is, except for the following modifications:

1. We use a slope coefficient of 0.2 for all Leaky ReLU layers, following the original pix2pix paper [28].
2. We train with LSGAN loss [46], following the original pix2pix paper [28].
3. We disable instance normalization layers if their input has only one spatial element (they would crash otherwise). Vox2vox's U-net reduces the spatial size of the input samples by factors up to 16. Our modification therefore allows us to train with samples with a minimum size of $16 \times 16 \times 16$, which conveniently coincides with Minecraft's chunk size. This modification is mostly relevant for the vox2vox adaptation we describe in Section 5.4.

4. We make the number of downscaling convolution layers in the discriminator model configurable, allowing us to experiment with different receptive fields.

5.4 Multiscale vox2vox

While we modified vox2vox itself only slightly, we also implemented a novel coarse-to-fine multiscale variant of vox2vox, which we straightforwardly call *multiscale vox2vox*.

Coarse-to-fine architectures are used in many prominent GAN architectures [47, 48, 49, 30, 32, 26, 33, 34]. The idea is to first train to generate images at a low resolution, and then train to upscale those images to higher resolutions. This can be done in various ways, such as training separate models for each resolution [47, 48, 26], or gradually adding layers to a model during training and eventually training the entire model as a whole [49, 30].

We use a multiscale architecture based on SinGAN [26], which falls in the first category. It consists of a configurable amount of scales N with scaling factors $s_0, \ldots, s_{N-1}$, and uses a vox2vox generator-discriminator pair G_n, D_n at each scale. The vox2vox model pairs at each scale are trained independently, starting from the bottom at scale 0 and moving upwards.

At each scale n, we downscale condition images and ground truth output images to s_n. The training process is different for the first scale (scale 0) and all subsequent scales. At scale 0, we train vox2vox as normal with the downscaled condition and target images. At scale 1, we first pass the condition image through G_0 and then upscale the result to s_1. The G_1 model then receives a concatenation of both the condition image downscaled to s_1, and the previous generator's output upscaled to s_1. The output of G_1 is added to the upscaled previous generator's output element-wise, and is then fed to the discriminator. That is, G_1 learns to generate a refinement for G_0's upscaled output, a technique called *residual learning*. This process continues all the way to G_{n-1}, with each $n > 1$ generator refining the output of the previous one.

In essence, our multiscale vox2vox architecture uses the SinGAN multiscale architecture [26], but extends it with support for condition inputs and replaces the inner generator and discriminator models with those from vox2vox [37]. By implementing a multiscale variant of vox2vox rather than adapting a different multiscale architecture, we can better evaluate the impact of the multiscale architecture in isolation.

We train multiscale vox2vox only on categorical tasks, because binary samples cannot cleanly be down- or upscaled. Down- and upscaling categorical samples is only possible thanks to the use of block2vec embeddings: with one-hot vectors, we would have ran into a similar problem. This ability to down- and upscale was in fact one of the main motivations of the World-GAN authors for introducing the block2vec tool [20].

6 Experiments and results

6.1 Setup

We trained our vox2vox variant and multiscale vox2vox system on all ten of our terrain-to-settlement datasets (Table 2 and Table 3), except for the combination of multiscale vox2vox with a binary dataset (for reasons outline in Section 5.4).

Due to the large amount of combinations this already entailed, and due to long training times (in the order of days for some configurations), we had to keep most hyperparameters constant. We did however experiment with several amounts of downscaling convolution layers in the discriminator models, since this has a direct relation with the model's receptive field, and we experimented with the block frequency weighting exponents w (Section 5.2.4). We performed a simple grid search for these two hyperparameters. For the constant hyperparameters, we selected the values based on the best-performing ones in related work [37, 26, 20] and some initial experiments.

Our hyperparameters were as follows[12]:

- **Epoch count (constant)**: 100. For multiscale vox2vox, we trained each scale for 100 epochs individually.
- **Batch size (constant)**: 32 for binary models, 4 for categorical models.
- **Optimizer (constant)**: Adam with learning rate 0.0002 and betas $(0.5, 0.999)$.
- **Multiscale vox2vox scales (constant)**: $(0.5, 1.0)$ for 96×96 datasets (all but *Mike's Angels wall*), and $(0.5, 0.75, 1.0)$. For the 96×96 datasets, scale 0.75 is not possible because $0.75 \cdot 96$ is not divisible by 16.
- **Discriminator downscaling convolution count (DDCC) (three values)**: 4, 3 and 2 (receptive fields $94 \times 94 \times 94$, 46^3 and 22^3 respectively).

  For multiscale vox2vox, we used only DDCC 2. We expected this value to be optimal because the multiscale architecture already increases the effective receptive field size in the lower scales, and the refinement performed by higher scales should not need a large receptive field.
- **Block frequency weighting exponents (three values)**: 0, 0.025 and 0.05. Only applies to the categorical models.

In total, we investigated 75 different (dataset, model, hyperparameters)-configurations: 15 for binary datasets (5 datasets, vox2vox with 3 hyperparameter configurations), and 60 for categorical datasets (5 datasets, vox2vox with 9 hyperparameter configurations and multiscale vox2vox with 3 hyperparameter configurations).

[12] For multiscale vox2vox, we used identical hyperparameters for each scale.

6.2 Evaluation

Unfortunately, our results are quite weak for many of the evaluated categorical settings. For this reason, we believe that most quantitative quality metrics such as block distribution comparisons [20] are not meaningful. Instead, we evaluate our trained models qualitatively by manually inspecting their output.

To visualize our model outputs, we created a tool to build them in Minecraft itself. It can build model outputs both in live Minecraft worlds using GDMC-HTTP [40, 41] (Section 3.1) and GDPC [40, 50] (a Python library for the GDMC HTTP interface), and in closed worlds on disk with Amulet Core [43].

We use three types of input terrain in this evaluation, which are shown in Figure 9: A plains biome, a forest biome, and a taiga biome. The plains biome serves as a simple, neutral type of terrain. The forest biome is a little more difficult, since trees are often removed or partially ignored by the generators we trained on. The taiga biome makes this challenge even more prominent: it is covered by a layer of `snow` blocks, which are ignored by all of our generators that build on the ground.

For each of our 75 configurations, we created one sample on all three terrains, resulting in a total of 225 samples. We rendered these samples with Mineways [38] and Blender [39], and manually evaluated the results — moving between terrains first, hyperparameter configurations second, model architectures third, and datasets last.

We first describe the performance of our binary models (Section 6.3), and then move the categorical ones (Section 6.4). In each case, we discuss our models in the order of our pair datasets. Finally, we discuss some overall observations in Section 6.5

6.3 Binary models

For binary models, our tool only builds the differences between model output and the original terrain to improve clarity. Note however that the binary models do also include the original terrain in their output. Added blocks are represented by `red_concrete`, and removed blocks are represented with `glass`.

6.3.1 Terrain copy

In the binary mode, our vox2vox models learn the *Terrain copy* task near-perfectly on all evaluated terrains. We obtain the best results with DDCC 4 — the amount with the highest receptive field. Lower amounts result in the introduction of a handful of noise blocks.

6.3.2 Ring

The *Ring* task is also learned quite well. In this case, we obtain the best results with DDCC 3 (Figure 10). With DDCC 4, significantly more noise is introduced. With DDCC 2, we observe that some sides of the ring are built at incorrect heights. We expect this happens because the terrain surface and correct ring position are not contained in a single receptive field region with DDCC 2

6.3.3 Adaptive ring

The *Adaptive ring* task turned out to be quite a bit harder than the non-adaptive one. While some of our models produce recognizable results, there is always a significant amount of noise. We obtain the best results with DDCC 3 — an example is shown in Figure 11.

We observe that the *Adaptive ring* models usually perform better on top of water, as can be seen in the example. We suspect this is because Minecraft always generates water at a fixed height (sea level). This means that if a column of blocks has water near the top, the height at which the ring should be placed is always the same.

Our models are unable to learn to replace the top layer of `snow` blocks in the taiga terrain. This is easily explained: in the binary mode, it is nearly impossible to distinguish these `snow` blocks from solid ground. Both are equally non-`air`. There are some hints to the presence of snow, such as terrain with snow being higher overall, trees with snow being thicker in the height axis and snow not being there below trees, but these are all extremely subtle.

For the same reason, the ring blocks are often placed on top of grass and leaves in Figure 11. Compared to snow, these blocks are however somewhat recognizable: both stick out from the flatter ground, and leaves are usually separated from the ground by at least one `air` block.

Note that the model actually performs better in relation to grass than it may seem from Figure 11: we only show *added* blocks as `red_concrete`, so pieces of grass in the ring still indicate a correct output of non-`air` at those positions.

6.3.4 Mike's Angels

The *Mike's Angels* task was, as expected, by far the hardest. Our models cannot reproduce recognizable buildings. They do however produce "clumps" of blocks on the ground, which is somewhat in line with how buildings should be placed. This makes the results almost seem like ancient, ruined variants of *Mike's Angels* cities. The scales of these clumps seem to correspond with the DDCC: a lower DDCC leads to smaller clumps. An example with DDCC 3 is shown in Figure 12.

Again, the models do not learn to replace the top layer of `snow` blocks. The tree removal feature of the *Mike's Angels* generator is however decently well replicated: trees

(a) Plains (b) Forest (c) Taiga

Figure 9: Terrain samples on which we evaluate our trained terrain-adaptive models.

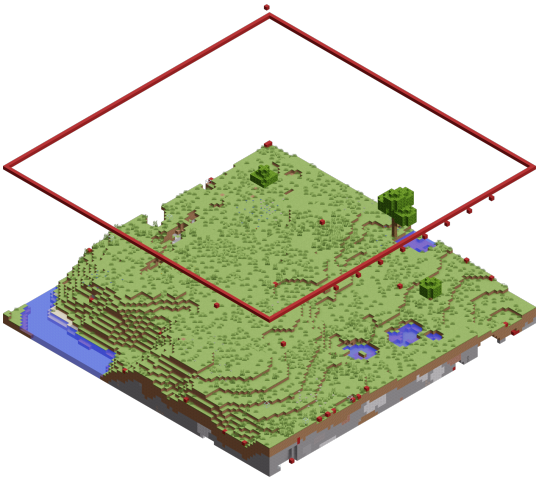


Figure 10: Output of our best-performing binary vox2vox `Ring` model on the plains terrain.

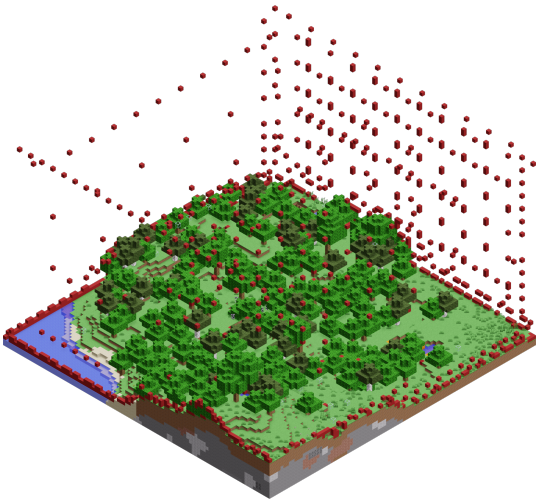


Figure 11: Output of our best-performing binary vox2vox `Adaptive ring` model on the forest terrain.

are usually at least partially removed. This indicates that the model does learn to somewhat recognize trees, even in binary form.

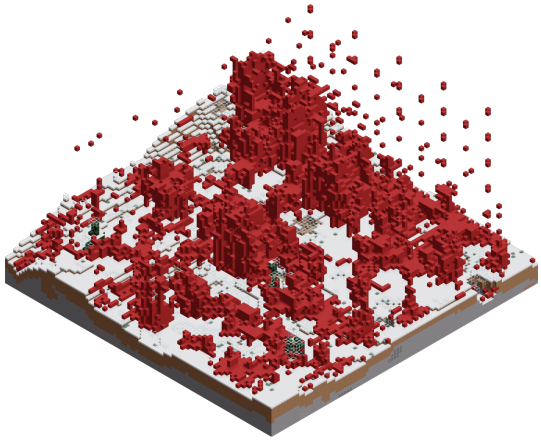


Figure 12: Output of one of our binary vox2vox `Mike's Angels` models on the taiga terrain.

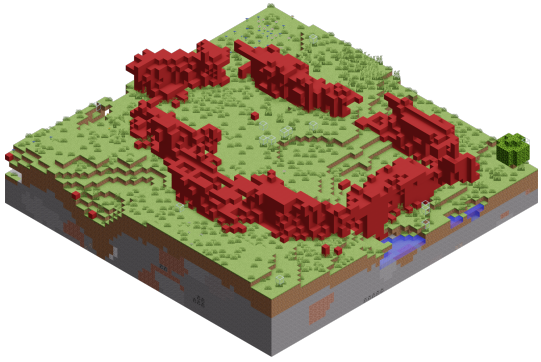


Figure 13: Output of one of our best-performing binary vox2vox `Mike's Angels wall` models on the plains terrain.

6.3.5 Mike's Angels wall

We obtain surprisingly good results for the *Mike's Angels wall* task. While the fine details of the wall are still not quite there, our models replicate its overall shape fairly well. We get the best results with DDCC 3 and 2. With DDCC 3, the overall shape is replicated a little better, but with DDCC 2, some areas have slightly more accurate details. An example with DDCC 3 is shown in Figure 13. Here too, the results seem akin to a ruined version of a *Mike's Angels wall*. With DDCC 2, we can even recognize the battlements in some places.

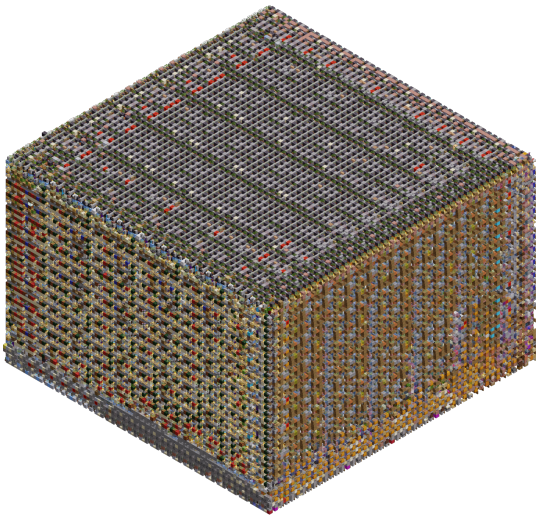


Figure 14: Example of the nonsensical noise output that many of our categorical models produce.

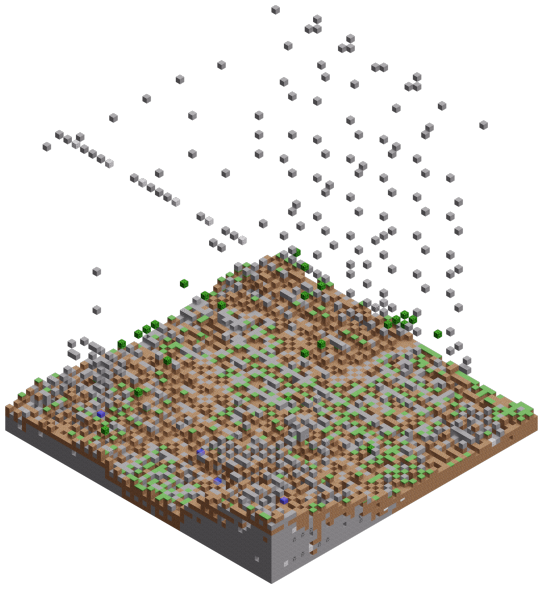


Figure 15: Example of the output of a noise-emitting categorical model with a high block frequency weighting exponent (0.05) on the forest terrain.

6.4 Categorical models

6.4.1 Noise output

Unfortunately, our categorical models completely failed to converge on the datasets *Terrain copy*, *Mike's Angels* and *Mike's Angels wall*, leading to outputs that appear to be entirely noise. An example of this is shown in Figure 14. Increasing the block frequency weighting exponent causes this noise to consist of more standard blocks, but it remains noise. We did get non-noise results for the *Ring* and *Adaptive ring* datasets, which we will discuss.

We do however observe that there is usually a difference in the makeup of the noise between the former `air` and the former `non-air` areas, and increasing the block frequency weighting exponent sometimes leads to the original terrain becoming recognizable. This can be seen in Figure 15.

It is suspicious that our models performed much worse on the *Terrain copy* task than on the *Ring* and *Adaptive ring* tasks — the latter two are strictly more difficult. Since all models, both vox2vox with various DDCC values and multiscale vox2vox, performed worse for *Terrain copy*, the problem does not seem to be that the *Terrain copy* models happened to perform worse by chance.

This leads us to believe that there might be something wrong with our categorical *Terrain copy* dataset, and perhaps with the *Mike's Angels* and *Mikes Angels wall* datasets as well. However, our binary datasets are directly derived from the categorical ones, and our binary models work fine on these datasets. The issue also does not seem to lie with the block2vec embeddings: these all seem plausible. We are therefore unsure of what exactly is causing the problem.

6.4.2 Ring

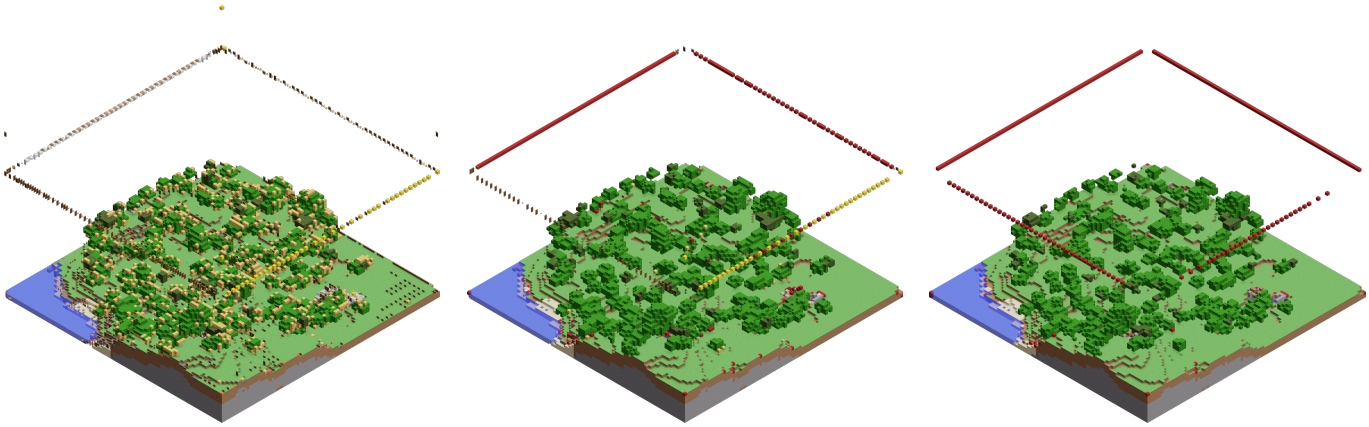
With the correct block frequency weighting exponent w, our DDCC 3 vox2vox model is able to learn the *Ring* task very well. This is shown in Figure 16. With $w = 0.05$, results are quite convincing. The terrain is slightly distorted, the trees are somewhat mangled and lack stems, and the grass is missing, but the ring is nearly there.

Figure 16 also shows the effect of block frequency weighting very well. When the feature is disabled ($w = 0$), we observe that the result contains a lot of erroneous rare blocks. The ring is made of incorrect, rarer blocks as well.

The yellow blocks that litter the trees in Figure 16a are `beehives` — blocks that Minecraft only generates in trees, and whose embedding vectors are therefore close to those of leaves. The blocks that replace the ring are all blocks that are very close to `air` in the embedding space, likely because the ring is placed in a very `air`-heavy area. As the block frequency weighting exponent is increased, these errors gradually decrease. Higher exponents lead to too much `air`, however.

Our DDCC 2 multiscale vox2vox model did not outperform regular vox2vox on this task. It does produce beginnings of the ring and replicates the terrain somewhat decently, but introduces a lot of noise as well. Furthermore, it often outputs multiple partial rings at different heights.

A reason why the multiscale vox2vox model performs poorly might be that the DDCC 2 value is too low. Another reason could be that the ring is too thin to learn well: when a sample is downscaled to a factor of 0.5, voxels that cover a ring piece cover three times more `air` blocks than `red_concrete` blocks on average.



(a) $w = 0$ (b) $w = 0.025$ (c) $w = 0.05$

Figure 16: Output of our best-performing categorical vox2vox *Ring* model on the forest terrain, with increasing block frequency weighting exponents w.

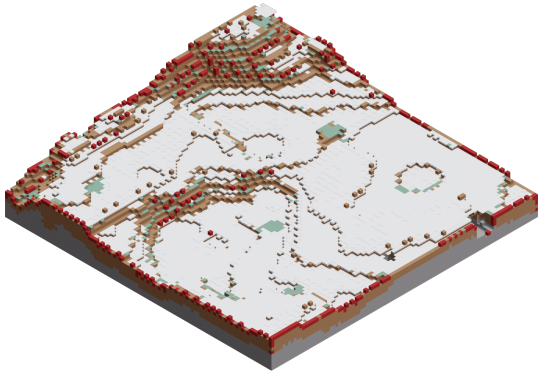


Figure 17: Example output of our best-performing categorical vox2vox *Adaptive ring* model on the taiga terrain.

6.4.3 Adaptive Ring

The results for the *Adaptive ring* are quite good as well. As expected, the results are not as good as those for the binary *Ring* task, but a few categorical *Adaptive ring* models actually improve on the binary ones in some ways.

With the vox2vox model, we obtain the best results with DDCC 3 and 4. The DDCC 4 models sometimes produce more solid ring segments than the DDCC 3 ones, but the DDCC 3 models are more consistent overall and replicate the original terrain much better. With DDCC 4, a block frequency weighting exponent of 0.025 performs best. With DDCC 3, the exponent 0.05 works slightly better. An example with DDCC 3 and block frequency weighting exponent 0.05 is shown in Figure 17.

In Figure 17, the terrain is certainly mangled and the trees are gone, but the ring is quite recognizable. Compared to the best-performing binary model (Figure 11), the ring is less solid and the terrain is less accurate, but there is much less noise. Unlike the binary models, the categorical models do actually learn to replace `snow` blocks correctly.

Again, our multiscale vox2vox model does not outperform our regular vox2vox models. It does learn to replicate the original terrain somewhat decently, but it only produces very faint signs of the ring. We suspect the same causes as for the *Ring* task.

6.5 Overall observations

6.5.1 Training instability

We observed that many of our models were quite unstable during training, a known problem with GAN architectures. Occasionally, our models would reach the best performance halfway through training, then suddenly start outputting garbage and start to slowly improve again. This issue could perhaps be prevented with an early stopping policy, but this would require performance metrics or mid-training human evaluation.

It is possible that some of our best-performing models in Section 6.3 and Section 6.4 performed better at an earlier stage during training, but it was not feasible to manually evaluate all intermediate stages. For consistency, we used only the final, 100-epoch versions.

6.5.2 Best-performing models

For the binary tasks, we conclude that the vox2vox model architecture with DDCC 3 performs best overall. DDCC 4 sometimes results in less noise and DDCC 2 sometimes yields better details, but DDCC 3 is the most consistent and produces the best results for most of the tasks.

For the categorical tasks, we again conclude that the vox2vox architecture with DDCC 3 performs best. We only trained the multiscale vox2vox architecture with DDCC 2, in which case it did not outperform regular vox2vox. We do however suspect that the only datasets on which categorical training worked at all (*Ring* and *Adaptive ring*) are particularly hard for multiscale vox2vox due to the thin structures.

Our block frequency weighting trick significantly improves the results for all evaluated models. Out of our evaluated exponent values, we find 0.05 to perform best overall. We do however expect that the optimal value depends on the quality of the trained models: better-performing models might benefit from a lower exponent. After all, there is no block frequency weighting during training.

7 Summary

We introduced an automated system to create machine learning-ready "volume-to-volume" Minecraft datasets using black-box Minecraft settlement generation algorithms. It currently only supports generators written against an old version of GDMC-HTTP [40, 41], an API for the Generative Design in Minecraft competition [1], but it can easily be extended to other input methods. Unlike other existing Minecraft machine learning datasets, the datasets produced by our system also include Minecraft *block states*.

Furthermore, we introduced a tool to preprocess these datasets to make training easier. Most importantly, it features three ways to reduce the amount of distinct blocks that alter the data only in minimal ways. This makes it much more feasible to actually train models that can deal with the increased amount of blocks caused by the inclusion of block states.

Using our automated dataset creation system and our preprocessing tool, we created ten new terrain-adaptive Minecraft machine learning datasets. These cover both binary and categorical versions of five terrain-to-structure tasks: three simple tasks for testing purposes and two tasks based on the 2022 GDMC submission by Van der Staaij et al. [17] (team name *Mike's Angels*).

We implemented a modified version of the vox2vox GAN [37] and a novel coarse-to-fine multiscale variant of vox2vox based on the multiscale architecture of SinGAN [26].

In addition, we applied the *block2vec* technique introduced by Awiszus et al. [20] to create embeddings for our categorical datasets, and demonstrated that the method is also effective for datasets that include block states. Furthermore, we extended the method with an evaluation-time block frequency weighting feature, which we showed to significantly improve results in some cases.

Finally, we trained and qualitatively evaluated various modified vox2vox and multiscale vox2vox models on all ten of our datasets. To our knowledge, this is both the first attempt to perform terrain-adaptive PCGML in Minecraft, and the first attempt at training Minecraft PCGML models that can deal with block states.

8 Conclusions

8.1 Research question

We started out this work with the following research question:

Is it possible to reproduce GDMC Minecraft settlement generators using machine learning?

In the end, we did not manage to fully reach our goal of replicating a GDMC generator, but we did get quite a bit of the way there.

Our trained models showed varying degrees of success. We obtained decent to good results for both binary and categorical versions of our three test datasets, but we could only train models for the binary versions of the tasks derived from the *Mike's Angels* GDMC generator. These binary models did however show promise. In particular, we achieved rather good results on a simplified wall-only version of the generator.

Seeing how our binary models showed a good amount of promise on the *Mike's Angels* tasks and how our categorical models were able to learn a simple terrain-adaptive task to a decent degree, we are of the belief that the goal is indeed achievable — there are still many machine learning approaches to try (we give some suggestions in Section 9). We therefore answer our research question with: *probably yes*.

8.2 Dataset creation system

We believe our dataset creation system to be a good approach for obtaining terrain-adaptive datasets, especially because it results in a large amount of *similar* examples. It does have one fundamental limitation: it only allows us to create datasets for translation tasks that we could already perform with the source generator. Still, we believe the datasets to be useful, challenging benchmarks for terrain-adaptive Minecraft PCGML, or even volume-to-volume translation in general. On top of that, they may also enable the training of co-creative settlement *completion* models (Section 9).

Managing an actual Minecraft server to run settlement generators was quite a hassle. Its only input/output system is a text-based command-line meant for interactive use. This causes several issues: the output includes a prompt string with a dynamic position; all output messages are posted in a single output stream and in an inconsistent order, making it hard to isolate the output of a command; and all commands are asynchronous while there is no easy way to wait for them to complete. The server also occasionally fails to start or stop.

Using RCON was essential to work around the server's single output channel, and extensive timeout, retrying and force-killing logic was needed to avoid crashes. We do

however believe that using an actual Minecraft server was worth it, as it can theoretically support any Minecraft settlement generator algorithm.

The *Chunk pregenerator* mod [42] was useful, but we did run into quite a few issues with it — likely because it was not really designed to be invoked from outside the game. This component of the system could perhaps be swapped out to improve pregeneration performance.

Both the server management and terrain pregeneration issues could perhaps be alleviated using a third-party server program like Spigot[13] or Paper[14], but we found that these were usually not compatible with Forge mod that many GDMC settlement generators require.

Block extraction with Amulet core [43] worked very well. It was very performant, and we encountered no significant issues.

Initially, our dataset creation system created datasets of before-after pairs directly, rather than separate before and after datasets. We however realized that re-computing the before-samples for each settlement generator was unnecessary, which is why we switched to creating separate datasets and "stacking" them during preprocessing.

Before separating the datasets, we used a single flat NumPy array axis for the samples, but switching to two axes (sample X and sample Z) made the process of matching up samples from the before- and after-datasets easier and less error-prone.

8.3 Machine learning

We found the block2vec method introduced by Awiszus et al. [20] to be very useful. Without it, we would not have been able to train categorical models with our build area sizes. We did however find that our categorical models outputs often contained excessively many rare blocks.

The original World-GAN approach trains block2vec embeddings on only one example and does not consider block states [20], so it has much fewer different blocks to deal with. Perhaps the block2vec method works less well when more blocks are involved, or perhaps larger embedding sizes are needed in that case. Either way, we were largely able to solve the rare block issue with our block frequency weighting feature.

Still, categorical models remained very difficult to train compared to binary ones. As we have shown, it is not impossible, but perhaps the jump from binary to categorical with block states was too much. In Section 9, we give some in-between options that we could have considered.

We believe the image-to-image translation literature was a good place to start for our 3D volume-to-volume problem. The 3D variant of pix2pix [28] called vox2vox [37] proved to be quite effective, and the multiscale architecture from the 2D SinGAN [26] worked in 3D as well. Image-to-image translation is an active field of research, and the improvements made there will likely be useful for terrain-adaptive Minecraft PCGML as well.

9 Future work

Our contributions open up many avenues for future work. We group these into two categories: *Datasets* and *Machine learning*.

9.1 Datasets

Using our automated dataset creation system and our preprocessing tool, more datasets could be created. We created datasets based on the current best-performing GDMC settlement generator, but it turned out to be difficult to train machine learning models for them. There may be other (GDMC or non-GDMC) settlement generators that are more feasible to replicate. Similarly, if more progress in terrain-adaptive Minecraft PCGML is made, our systems could be used to create more varied or more difficult learning tasks as well.

Another interesting possibility is to create pair datasets where the first pair half is something other than the original terrain. For example, a dataset that represents the task of completing a partial settlement.

Given a terrain-to-settlement dataset, one could obtain paired examples for a settlement completion task by "cutting" out parts of a settlement and "pasting" them onto the original terrain. This way, it might be possible to leverage handwritten settlement generators to achieve co-creativity, a promising application for PCGML [3] that has been described by the GDMC competition organizers as one of their future goals [1, 6].

Furthermore, our dataset creation system and preprocessing tool could be improved in various ways. For example, more ways to reduce the amount of distinct blocks could be implemented, or dataset augmentation features such as rotating and flipping examples could be added.

9.2 Machine learning

There is still much room for improvement with regard to training machine learning models on our datasets. There are many directions that could be taken to attempt to improve the results: more hyperparameter configurations, longer training, better architectures, better training process analysis, etc.

There are many more image-to-image architectures that could be adapted to 3D. In the area of GANs, one recent example is CoCosNet v2 [34]. And of course, GANs are not the only possibility. Some examples of different techniques are transformers and diffusion models. In some cases, 3D

[13] https://www.spigotmc.org/
[14] https://papermc.io/

versions already exist. Furthermore, for all the aforementioned techniques, unpaired variants could be investigated as well.

The learning task could also be adjusted. To give some examples, one could attempt to train on only the differences between the input and target sample, or perhaps train some kind of recurrent system to generate samples Y-layer by Y-layer.

Another possibility is to consider a mode somewhere between binary and full categorical with block states. Some possibilities are categorical without block states, or categorical with a small number of manually curated categories such as "stone-like" and "wood-like".

A learning mode between binary and full categorical could also be considered

Furthermore, different ways of embedding blocks could be investigated. One could train block2vec [20] with different settings, or perhaps extend the LLM-based method used in Wor(l)d-GAN [21] to blocks with block states. With a large amount of memory or a simplified learning task, regular one-hot vectors could be considered as well.

Finally, the evaluation of trained models could be improved. We performed only a qualitative evaluation because we felt our results were not yet ready for quantitative measures, but such measures do exist — though they do not always reflect quality. Examples include block distribution comparisons [20] and TPKLDiv [51, 20]. If results are sufficiently good, human evaluation could be considered as well.

References

[1] Christoph Salge et al. "Generative design in minecraft (GDMC)". In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*. ACM, Aug. 2018. DOI: 10.1145/3235765.3235814. URL: https://doi.org/10.1145%2F3235765.3235814 (cit. on pp. 2, 21, 22).

[2] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016. DOI: 10.1007/978-3-319-42716-4 (cit. on p. 2).

[3] Adam Summerville et al. *Procedural Content Generation via Machine Learning (PCGML)*. 2018. arXiv: 1702.00539 [cs.AI] (cit. on pp. 2, 3, 22).

[4] Matthew Johnson et al. "The Malmo Platform for Artificial Intelligence Experimentation". In: *25th International Joint Conference on Artificial Intelligence (IJCAI-16)*. AAAI - Association for the Advancement of Artificial Intelligence, July 2016 (cit. on p. 2).

[5] Christoph Salge et al. "The AI Settlement Generation Challenge in Minecraft: First Year Report". In: *KI - Künstliche Intelligenz* 34 (Jan. 2020). DOI: 10.1007/s13218-020-00635-0 (cit. on pp. 2, 4).

[6] Christoph Salge et al. *Impressions of the GDMC AI Settlement Generation Challenge in Minecraft*. 2021. arXiv: 2108.02955 [cs.OH] (cit. on pp. 2, 4, 22).

[7] Michael Cerny Green, Christoph Salge, and Julian Togelius. "Organic building generation in minecraft". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 2019, pp. 1–7 (cit. on pp. 2, 4).

[8] Marcus Fridh and Fredrik Sy. *Settlement generation in Minecraft*. 2020 (cit. on pp. 2, 4).

[9] Levi van Aanholt and Rafael Bidarra. "Declarative procedural generation of architecture with semantic architectural profiles". In: *2020 IEEE Conference on Games (CoG)*. 2020, pp. 351–358. DOI: 10.1109/CoG47356.2020.9231561 (cit. on pp. 2, 4).

[10] Ari Iramanesh and Max Kreminski. "AgentCraft: An Agent-Based Minecraft Settlement Generator". In: *AIIDE workshops* (2021) (cit. on pp. 2, 4).

[11] Albin Esko and Johan Fritiofsson. "Multi-Agent Based Settlement Generation In Minecraft". MA thesis. Malmö University, Faculty of Technology, Society (TS), Department of Computer Science, and Media Technology (DVMT), 2021 (cit. on pp. 2, 4).

[12] Jean-Baptiste Hervé and Christoph Salge. *Comparing PCG metrics with Human Evaluation in Minecraft Settlement Generation*. 2021. arXiv: 2107.02457 [cs.AI] (cit. on pp. 2, 4).

[13] Jean-Baptiste Hervé and Christoph Salge. *Automated Isovist Computation for Minecraft*. 2022. arXiv: 2204.03752 [cs.AI] (cit. on pp. 2, 4).

[14] Sebastian S. Christiansen and Marco Scirea. "Space segmentation and multiple autonomous agents: a Minecraft settlement generator". In: *2022 IEEE Conference on Games (CoG)*. 2022, pp. 135–142. DOI: 10.1109/CoG51982.2022.9893679 (cit. on p. 2).

[15] Michael Beukman et al. *Hierarchically Composing Level Generators for the Creation of Complex Structures*. 2023. arXiv: 2302.01561 [cs.AI] (cit. on p. 2).

[16] Jean-Baptiste Hervé, Christoph Salge, and Henrik Warpefelt. "An Examination of the Hidden Judging Criteria in the Generative Design in Minecraft Competition". In: (May 2023). DOI: 10.36227/techrxiv.22698484.v1 (cit. on pp. 2, 4).

[17] Arthur van der Staaij et al. "Believable Minecraft Settlements by Means of Decentralised Iterative Planning". In: *2023 IEEE Conference on Games (CoG)*. 2023, to appear (cit. on pp. 2–4, 8, 9, 21).

[18] BluShine. *Minecraft-Gan-City*. 2020. URL: https://github.com/BluShine/Minecraft-GAN-City-Generator (visited on 11/19/2022) (cit. on pp. 3, 4, 6–8).

[19] Aldo Fumagalli. *Minecraft Settlement 3D-GAN*. 2022. URL: https://github.com/ikros98/Minecraft-settlement-GAN (visited on 01/05/2023) (cit. on pp. 3, 4, 6–8).

[20] Maren Awiszus, Frederik Schubert, and Bodo Rosenhahn. *World-GAN: a Generative Model for Minecraft Worlds*. 2021. arXiv: 2106.10155 [cs.LG] (cit. on pp. 3–8, 12–14, 16, 17, 21–23).

[21] Maren Awiszus, Frederik Schubert, and Bodo Rosenhahn. "Wor(l)d-GAN: Towards Natural Language Based PCG in Minecraft". In: *IEEE Transactions on Games* (2022). DOI: 10.1109/TG.2022.3153206 (cit. on pp. 3, 5–8, 23).

[22] Zhuoyuan Chen et al. "Order-Aware Generative Modeling Using the 3D-Craft Dataset". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 1764–1773. DOI: 10.1109/ICCV.2019.00185 (cit. on pp. 3, 5, 6, 8).

[23] Jun-Yan Zhu et al. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. 2020. arXiv: 1703.10593 [cs.CV] (cit. on pp. 3, 5).

[24] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL] (cit. on pp. 4, 12).

[25] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: 1310.4546 [cs.CL] (cit. on pp. 4, 12, 13).

[26] Tamar Rott Shaham, Tali Dekel, and Tomer Michaeli. *SinGAN: Learning a Generative Model from a Single Natural Image*. 2019. arXiv: 1905.01164 [cs.CV] (cit. on pp. 5, 16, 21, 22).

[27] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML] (cit. on pp. 5, 12).

[28] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. arXiv: 1611.07004 [cs.CV] (cit. on pp. 5, 15, 22).

[29] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV] (cit. on pp. 5, 15).

[30] Ting-Chun Wang et al. *High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs*. 2018. arXiv: 1711.11585 [cs.CV] (cit. on pp. 5, 16).

[31] Wengling Chen and James Hays. *SketchyGAN: Towards Diverse and Realistic Sketch to Image Synthesis*. 2018. arXiv: 1801.02753 [cs.CV] (cit. on p. 5).

[32] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2019. arXiv: 1812.04948 [cs.NE] (cit. on pp. 5, 16).

[33] Taesung Park et al. *Semantic Image Synthesis with Spatially-Adaptive Normalization*. 2019. arXiv: 1903.07291 [cs.CV] (cit. on pp. 5, 16).

[34] Xingran Zhou et al. *CoCosNet v2: Full-Resolution Correspondence Learning for Image Translation*. 2021. arXiv: 2012.02047 [cs.CV] (cit. on pp. 5, 16, 22).

[35] Alceu Bissoto, Eduardo Valle, and Sandra Avila. *The Six Fronts of the Generative Adversarial Networks*. 2019. arXiv: 1910.13076 [cs.CV] (cit. on p. 5).

[36] Dmitrii Torbunov et al. *Rethinking CycleGAN: Improving Quality of GANs for Unpaired Image-to-Image Translation*. 2023. arXiv: 2303.16280 [cs.CV] (cit. on p. 5).

[37] Marco Domenico Cirillo, David Abramian, and Anders Eklund. *Vox2Vox: 3D-GAN for Brain Tumour Segmentation*. 2020. arXiv: 2003.13653 [cs.CV] (cit. on pp. 6, 15, 16, 21, 22).

[38] Eric Haines. *Mineways*. version: 11.00. 2023. URL: https://www.realtimerendering.com/erich/minecraft/public/mineways (visited on 07/20/2023) (cit. on pp. 7, 17).

[39] Blender Foundation. *Blender*. version: 3.6.1. 2023. URL: https://www.blender.org/ (visited on 07/20/2023) (cit. on pp. 7, 17).

[40] GDMC wiki editors. *Submission Method: HTTP Interface - Generative Design in Minecraft*. 2023. URL: https://gendesignmc.wikidot.com/wiki:submission-httpserver (visited on 04/07/2023) (cit. on pp. 6, 17, 21).

[41] Niels NTG Poldervaart and Niki Gawlik. *Minecraft HTTP Interface Mod*. 2023. URL: https://github.com/Niels-NTG/gdmc_http_interface (visited on 04/07/2023) (cit. on pp. 6, 17, 21).

[42] Speiger. *Chunk Pregenerator*. 2023. URL: https://www.curseforge.com/minecraft/mc-mods/chunkpregenerator (visited on 01/16/2023) (cit. on pp. 6, 22).

[43] Amulet team. *Amulet Core*. 2023. URL: https://github.com/Amulet-Team/Amulet-Core (visited on 01/16/2023) (cit. on pp. 7, 17, 22).

[44] Amulet team. *Amulet Editor*. 2023. URL: https://www.amuletmc.com/ (visited on 01/16/2023) (cit. on p. 7).

[45] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML] (cit. on p. 14).

[46] Xudong Mao et al. *Least Squares Generative Adversarial Networks*. 2017. arXiv: 1611.04076 [cs.CV] (cit. on p. 15).

[47] Emily Denton et al. *Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks*. 2015. arXiv: 1506.05751 [cs.CV] (cit. on p. 16).

[48] Han Zhang et al. *StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks*. 2017. arXiv: 1612.03242 [cs.CV] (cit. on p. 16).

[49] Tero Karras et al. *Progressive Growing of GANs for Improved Quality, Stability, and Variation*. 2018. arXiv: 1710.10196 [cs.NE] (cit. on p. 16).

[50] Arthur van der Staaij, Blinkenlights, and Niki Gawlik. *Generative Design Python Client (GDPC)*. 2023. URL: https://github.com/avdstaaij/gdpc (visited on 07/29/2023) (cit. on p. 17).

[51] Simon M. Lucas and Vanessa Volz. "Tile Pattern KL-Divergence for Analysing and Evolving Game Levels". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, pp. 170–178. ISBN: 9781450361118. DOI: 10.1145/3321707.3321781. URL: https://doi.org/10.1145/3321707.3321781 (cit. on p. 23).