PCG Generator for Building 3D Artifacts in Minecraft

Ye Xu
New York University
New York, USA
yx2534@nyu.edu

Hemanth Sai Grandhi
New York University
New York, USA
hg2315@nyu.edu

Arty Altanzaya
New York University
New York, USA
aa4928@nyu.edu

***Abstract*—We propose two procedural content generation (PCG) frameworks for creating 3D content in a Minecraft environment. For this project, we are exploring the use of procedural content generation via Reinforcement Learning (PCGRL) and a supervised learning model(PCGML) built on top of the 'Path of Destruction' paper [1] to generate the required dataset. Procedural Content Generation (PCG) has been effective in producing 2D content and several crucial research has been done to achieve the result. In this work, we experiment with the EvoCraft environment as it provides various features that allow us to test our frameworks. Specifically, we use our frameworks to procedurally generate houses that are diverse 3D structures and serve as a complex problem to tackle. We discuss in detail the framework architectures, data generation process, experiments conducted, challenges faced and the final results obtained.**

I. Introduction

Procedural Content Generation (PCG) is a method used in games to create content such as textures, items, and levels allegorically with generated randomness. One of the challenges that PCG poses is that the generated content has to be aesthetically suiting and also functional which is challenging in 3D environments such as Minecraft. Minecraft is a compelling domain for the development of reinforcement and imitation learning methods because of the unique challenges it presents: Minecraft is a 3D, first-person, open-world game centered around the gathering of resources and the creation of structures and items. In Minecraft, the procedurally-generated world is composed of discrete blocks that allow modification; throughout gameplay, players change their surroundings by gathering resources (such as wood from trees) and constructing structures (such as shelter and storage). Since Minecraft is an embodied domain and the agent's surroundings are varied and dynamic, it presents many of the same challenges as real-world robotics domains. We use EvoCraft which is a platform for experimenting with Minecraft features and contains useful pre-built functions to load, read and manipulate blocks from the Minecraft server.

As games become more complex and less linear, the use of PCG tools becomes more diverse, and the need for content generators that are reliable and trustworthy increases. Due to this, recent developments in Procedural Content Generation (PCG) can be divided into two crucial parts: PCG via supervised machine Learning (PCGML) and PCG via reinforcement learning (PCGRL).

PCGML refers to the use of machine learning through self-supervised learning on existing data to learn how to generate new content based on some preexisting content.
Supervised-learning content generators can be used to produce whole levels or structures (content). On the other hand, PCGRL provides a different way of using machine learning to learn content generation. Without training data, reinforcement learning is used to learn to generate by trial and error meaning with state and action pairs. In the environment, the generator is perceived as an agent that receives actions and changes the environment and the agent gets rewarded based on the quality of the content created by its actions. This paper shows how we can use both of these methods and apply them to a complex 3D problem statement and procedurally generate meaningful content.

II. Related Work

A crucial amount of literature has defined the area of PCG in recent years including [1], [2], [3], [4], and [5]. In this section, we focus on related studies in PCG via Machine Learning and PCG via Reinforcement Learning. In addition, we also take a close look at other research that succeeded in creating 3D artifacts and functional machines in a controlled environment [6]. In [7], they created an algorithm that can take a given Minecraft map and without supervision, generate a settlement on the map by placing and deleting appropriate blocks. There are successful research papers on using PCGRL and PCGML to create content in 2D environments such as level generation [1], [8]. Path of Destruction as stated in [1] has only been applied in 2D game map generation, here with Minecraft and Evocraft API we can create an environment to apply the technique into a 3D world generating 3D houses to show the method's versatility.

The majority of existing experiments inspect 2D content instead of 3D objects due to several different reasons including computation power, access to data, and complexity of training. Therefore, this research is unique in terms of scaling up the Path of Destruction method to larger 3D game artifacts.

Evocraft is a Minecraft API that allows us to use Python to interact with Minecraft game world on a local server. Images

[1]Repo: https://github.com/hsgrandhi/AIforGamesPCGRLMineCraft.git

in this project are rendered using Minecraft JAVA version game connect to the local hosts game world that is been directly modified by Evocraft. In data generation, Evocraft functions such as spawn blocks data in a region and reading a region's block data from [9] have been proven to be fast and reliable to ensure the speed and success of data generation. The rendering ability also makes visual debugging possible during both the generation and training process.

III. Methods

Even though the combination of ML and PCG shows great potential compared to classical PCG methods, PCGML is, however, limited by the lack of training data which is often the case in games. In this project, this could be potentially solved by utilizing The "Path of Destruction" method for this project. The "Path of Destruction" (PoD) technique [1] works by generating a training dataset by iteratively destroying a set of Goal content until it reaches respective random noisy levels. During each of these destructive sequences, a corresponding repair sequence is produced and added to the training data. This produced training data is afterward used to train an artifact generator that can make new houses in the Minecraft environment.

PCGRL methods on the other hand do not require training data, as compared to the PCGML frameworks. As proposed in [2], we can also solve the challenge in this project by proposing to train a PCG agent via reinforcement learning which frames tasks such as building diverse houses, as a game and then uses an RL agent to solve it, creating different houses in the process as generated content. The main goal of the research is to create an iterative content generator and apply it to 3D content in Minecraft through PCGML and PCGRL methods.

A. PCGML with Path of Destruction

For the Path of Destruction method, success lies majorly in the generation of correct data. As the training process can be done using any suitable supervised learning algorithm. For our approach, we use a convolution neural network to train our agent which is discussed more in detail in the below sections.

1) Goal Set Generation: We will need a set of goal states on which to run the Path of Destruction algorithm. For this purpose, we procured some pre-built Minecraft buildings in the form of NBT files. We explored other options and existing Minecraft building datasets, however, most of them consisted of huge buildings with a large number of cube types, which would make testing out the framework difficult. Therefore, we initially selected a simple village house, as seen on the left side of Fig. 1 as our starting goal state. We then selected a list of 7 cube types we would use in our model, to limit the dimension of our tests: Air, logs, Planks, Dirt, Glass pane(window), Stone stairs, and cobblestone. Initially, we had included a wooden door to this set, but due to an internal Minecraft implementation issue with rendering doors dynamically(causing it to fall off and make the game lag), we later removed it.

We then needed to process this initial house data and remove cubes from the NBT file which are not present in our set of 7 cubes(For example, remove the door and torch cubes from the NBT file on the left side (a) of Fig. 1 to get the processed house shown on the right side (b)) and replace them with one that is accepted to finally arrive at the goal set data. Moreover, due to limited resources, we pick small houses which fit in a (6 x 6 x 6) space. Doing so will give us training data with constant dimensions, which can be used to train the ML model. After processing, we now have goal houses that can be fitted into a defined space and consist of a predetermined set of cube types.

After initial experiments, which will be discussed in the training section, we designed two vastly different houses from our original village house using Minecraft's inbuilt structure block and save block functionalities and saved them as NBT files, also shown in Fig. 1 (d), which can be used to train a more complex model with multiple goal houses in the goal set. The village house is a basic wooden house with a stone base, a window at the side, stone stairs, and a block inside the house for sitting. The glasshouse is a modern-looking house with large glass windows, stone pillars, and wooden walls. The mud house has different dimensions and is larger, is made up of dirt blocks, and has a dirt balcony. The aim is to use these goal houses and attempt to generate new houses with a mix of these features.

(a) First house: Village House (b) Village house after processing

(c) Second house: Glass House (d) Third House: Mud House

Fig. 1. (a) The first house is the NBT data file that comes with torch on top which will later be removed by processing and is loaded using EvoCraft. We show the first house after our processing step in (b). The Second (c) and third (d) houses were created by our team, the above picture also displays how we saved the house data using Minecraft structure blocks and save block functionalities.

2) Destruction Agent: Once we have the processed goal house, the house will be moved to a constant location in the sky which has a coordinate of (0, 12, 0) for the data generation process. The house's block location data and block type data will be read and stored in a list that is used later. To achieve this, we used Evocraft API's client. readCube function that returns 'blocks' data. The 'blocks' data of Evocraft is a whole

set of blocks with their location and type stored as a data structure in a particular region. For later experiments, We do this same step for all three of the goal house data. Once this is done, we have a list of 'blocks' type data. Here, each 'blocks' data in the list indicates one of our goal house data which was processed and normalized in the sky. The reason we move the houses high in the sky is for the next step when we add the padding. This will ensure not to damage the game world since the game world's ground is also made with blocks, specifically grass blocks. Padding with air for a house on the ground will cause the ground to be torn open, which we want to avoid.

Now we have three goal houses data saved into a list as "blocks" data type. In the next step, we will start the "destruction" process to generate our training data. Here, we build a "Destruction Agent" that will iteratively go through every block in the goal house. The order will be sequential as the agent will be given the house's starting and an ending coordinate, that is, 5 blocks more than the house starting location, to cover a total of (6 x 6 x 6) area which is the boundary we've set while picking out all our goal house data. By doing so, we will ensure our training data is constant and in a unified shape.

The Destruction Agent will start from the house's starting location and move along the z-axis, once this row is done, the agent will move up by one block (moving long y-axis), then start from the minimum z value. It will again go through the row and follow the z-axis like before. An example would be to assume the agent starts at the minimum boundary of (x, y, z) of (0, 0, 0) location, and traverse to the max boundary of (5, 5, 5). Agent will first move in order of (0, 0, 0) $\rightarrow$(0, 0, 1) $\rightarrow$ (x, y, z). Once agent reached (0, 0, 5), the agent will move to (0, 1, 0) $\rightarrow$ (0, 1, 1) $\rightarrow$ (x, y, z) again and so on. Once it traverses this whole vertical plane in the game world, it will increment along the x-axis and do the same traversal for the next vertical plane and hence move through every block in the (6 x 6 x 6) area that contains our goal house inside.

Each time before the agent makes a move, the block type where the agent is currently standing will be recorded as an "Action". Also, every block inside the current (6 x 6 x 6) area state will be recorded as our current "State". Then, the action and the state will be written out as a row in a CSV file. Once this is done, the agent will have an 80 percent chance to change the current block it's standing on into a randomly selected block type in our list of accepted blocks, then moves on to the next block following the traversing order. Continue doing so will change the whole (6 x 6 x 6) house area into a random noise state made of the blocks in our accepted blocks list. One thing to note is, that the specific traverse order we mentioned above in order of Z $\rightarrow$ Y $\rightarrow$ X is important since this is also the same order of how Evocraft is reading the block data and spawning the blocks. So by keeping this traverse order we can make sure our steps to process the house won't conflict with Evocraft's function to read and write blocks in the Minecraft game world. Fig. 2 shows this process of loading the goal house state, having the agent move across the house while randomly replacing each cube in the space until it ends in a random noise state.

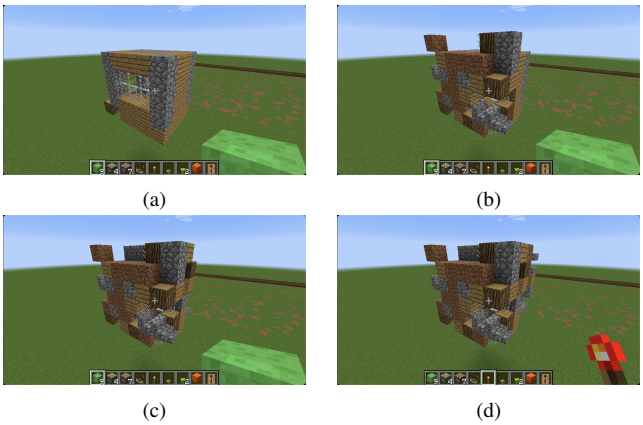
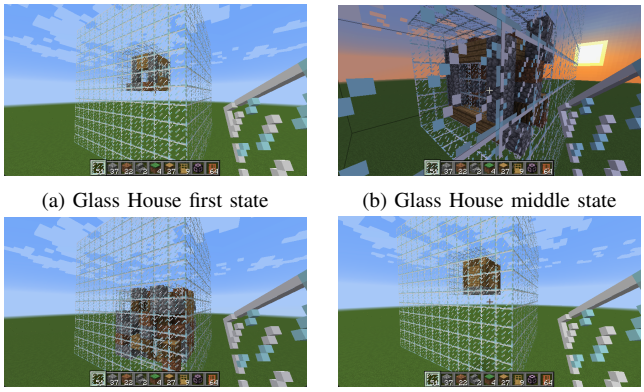


Fig. 2. These are 4 states which show the process of the agent traversing (destroying) the Glass House, as we can see from the image, the agent goes one vertical plane by another to destroy the original house into a noise state.

3) Encoding agent location with padding: After our initial tests, we found that the initial model did not do well and was predicting each action to repair as an air block, resulting in an empty building. This will be discussed in detail in the below section. This could be due to the model not knowing where the agent is currently located, thus, making it impossible to know which correct repair action to take. To fix this, we use the idea of padding to encode the agent's current location as used in the original PCGRL paper [2]. This process involves adding padding around the agent such that the agent is centered in the space(the padded region will need to be twice the size of the house state space in all dimensions to handle edge cases). While the agent moves around the house, each time before it makes a move, we will pad around the agent with air blocks, Since the whole goal house blocks space takes the size of (6 x 6 x 6), to cover the whole house with our padding, we will pad around the agent location for 6 blocks in each direction of our agent. Doing so will generate a padded area around the agent of size (13 x 13 x 13).

The reason we do this padding is to encode the agent location into our training data. Since the training process won't work well with directly passing 3D position data like the agent's current x, y, and z coordinate(as these would be random scalar points which should be independent of the process). Fig. 4 shows two examples of the agent's padding procedure, here we switch the air padding blocks to Glass blocks for better visibility. We can see in this padded area, that the agent is in the center location. With this padded area with the house data enclosed in it, we successfully encode the agent location (center of the padded region) into our training data. We then take this state and the previous block type as the action and then store it as a row in the CSV file.

Once our destruction agent is finished traversing the whole (6 x 6 x 6 area), We will have a CSV data file that contains 216 rows, each row will contain all the goal house's blocks within the padded area of (13 x 13 x 13), with our house of

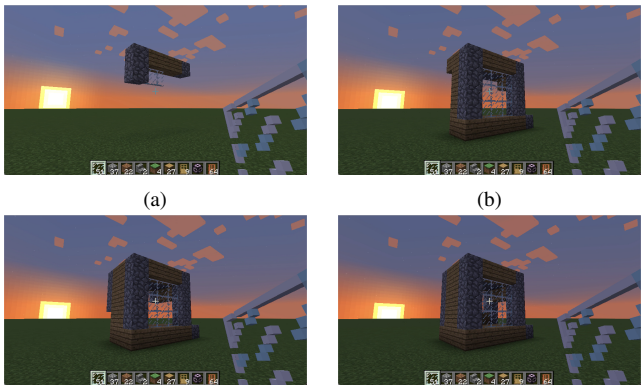


(a) Glass House first state (b) Glass House middle state

(c) Glass House final state (d) Village House first state

Fig. 3. It can be seen here that at the agent's starting location, the house is at the padding's upper corner while at the final location (pure noise state), it is moved to the bottom corner of the padding block, this is how the agent's location data is encoded in the data. The agent is always at the center of the padding block.

size (6 x 6 x 6) inside it. With these 216 rows, if we reverse it's order, we will be able to construct the house back to its original states by following the actions that are provided at the end of each row. Fig. 4 shows how we can build the original house with actions from the last row to the top using the opposite traverse order of the agent.



(a) (b)

(c) (d)

Fig. 4. These pictures are to showcase if we reverse the order of actions in the csv file. In opposite traverse order as how agent moves, it will built the original house back in the game world

4) Training data generation: Now we repeat the above process to generate enough data for our agent to train on. This means we need to run the agent on the same goal house again and again but generate different episodes(as the agent randomly picks which noise block to replace the original block with, each iteration will be different). In later experiments, since we have a total of three different goal houses, for our agent to learn a variation of house and build a house with no bias towards a certain house type, once a house episode is finished (216 rows) in the CSV file, the next house we spawn and let agent work on, will be randomly picked from the three houses we currently recorded. For later experiments, we modified this process to write the one-hot encoded block data to the file for training purposes and model improvement. This process can be expanded for N goal houses of varying sizes.

5) Model and training: For our supervised learning model, we use a 3D Convolutional Neural Network(CNN) model as it is the most applicable for our 3D problem. We can see the architecture and the shape of the intermediate steps in Fig. 5. The input shape is the (13 x 13 x 13 x 7) one-hot encoded current state of the padded house and the output is the predicted action to take(which block to replace the current location with). We use a Max-pooling layer of stride 2, three 3D convolutional layers with ReLu activation functions. We then flatten the data and then use a fully connected dense layer. Finally, we use a softmax activation function to get the prediction probabilities.

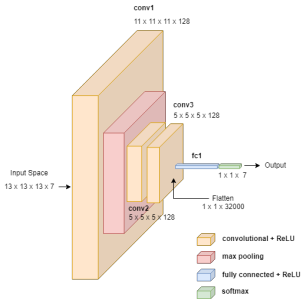


Fig. 5. The Model Architecture

Initially, while training, we used pandas to read all the individual CSV files and combine them into one large dataframe. We then processed this input and one-hot encoded all the data. This processed data was used as input to the model. Later, when we added padding and one-hot encoding to improve the model performance, we faced several issues. This post-processing step was taking a lot of time with large amounts of data. Furthermore, With padded one-hot encoded data, the dimensions of the data increased quite a lot, and we could no longer fit all the data at once into the memory.

To fix these issues, we implemented a function to store the on-hot value of the block while writing the data to the CSV file itself. We also changed our training process to an iterative process using data generators and batch training. Due to memory and time limitations, we found a batch size of 128 worked best. We ran various experiments with different batch sizes, amounts of data points, epochs, etc. This will be discussed further in the results section.

B. PCGRL 3D environment for content generation

Our goal in using PCGRL is to explore how we can train an agent to build structured artifacts in the 3D environment as there is not sufficient research that shares the same mission. This approach compared to PCGML, does not use the pre-existing dataset to train from, instead relies

heavily on the reward function to learn how to take the next action. We experiment with different reward functions to receive the desired outcome in the amount of time we have.

1) Problem: The problem class which is to build a house provides information about the environment including the type of blocks to use and reward weights. This module assesses the change in stats of the different actions and the quality of generated content after a certain action is done by the agent. Considering it is a 3D environment, it is important to have a reset function that removes all the prior-built artifacts at every run. As it is important to define a goal for our problem, we aimed to build a structured apartment or a house and experimented with several approaches to fine-tune our process.

2) Representation: For the PCGRL problem, we have to define the action state and observation space. The representation class is used for this purpose. Our representation space sets the problem while keeping a record of the current state and moves on to change the state based on the next action from the trajectory. We use the simplest way of representing the problem which is set to record 3D environments. It is the narrow representation where the agent is trying to modify the tile value of a certain selected position that is selected which is similar to cellular automata representation. [2] The representation observation object at the current moment is represented with x, y, z position of the current location and when an input action is received, we update the narrow representation with the input action.

IV. Results

A. Results for PCGML with Path of Destruction

In the initial experiment, we did not encode the agent location with padding and we loaded all the individual CSV files into a single dataframe and trained the model. Further, this version only used the village house as the goal set. This model, however, did very poorly and kept predicting the repair step as air, which resulted in an empty space. This could have been due to the model not knowing where the agent was and hence did not know where to repair with which block.

We then changed our data to include padding to encode agent location and one-hot encoded values for the blocks. We trained a model with a small sample of 1000 data points and while the model was not trained with enough data, it produced some interesting results as shown in Fig. 6. We can see that the model learned to keep a stone base and build wooden structures with stairs at the front.

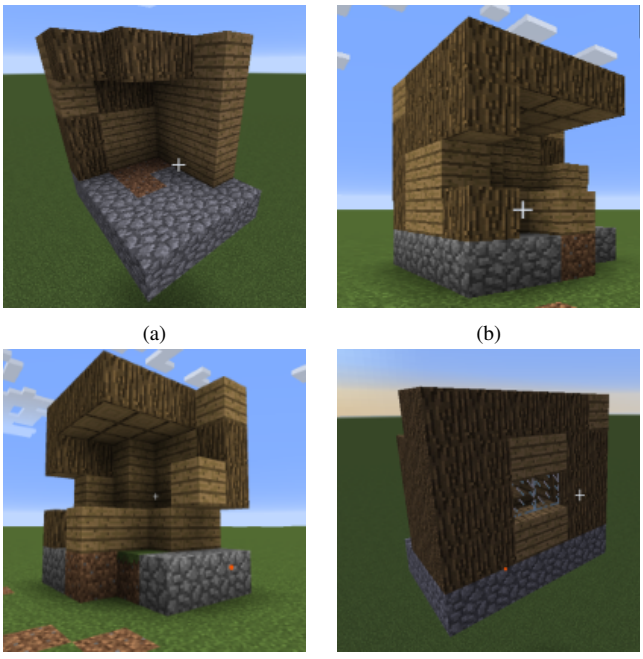
Once we confirmed the model was capable of learning, we trained the next model with 11,000 data points. This time we see that the model was able to learn to build walls and build houses with interior space. However, we still cannot transform the noise block completely to the goal state. This can be seen in Fig. 7.

When we tried to load more data to get a better model, we ran into memory issues, as the entire data with padding and one-hot encoding could no longer fit into memory. To tackle this, we changed the training process to use data generators and batch processing. We experimented with various batch sizes and found that the best case was a batch size of 128(any higher would cause memory allocation warnings). We then trained this iterative model for 500 epochs with 200,000 data points. The results of this are shown in Fig. 8 as well as the steps of the process, we can see the initial noise state, the intermediate state where the agent is still working, and the final result. We observe that we can get the original village house back. Even with varying starting noise states, we were able to consistently get the original village house.

As we had used only one goal house to generate the data until now, there is not much variation in the results from the model. To show how effective our framework actually is, we added two other goal state houses(namely the glasshouse and

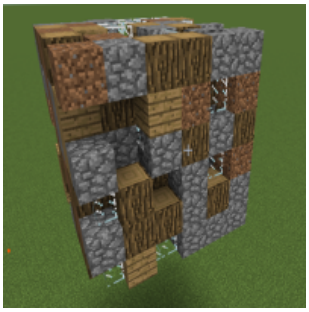
(a) (b)

Fig. 6. Results from the second model trained with 1000 data points. We can see that the model learned to keep a stone base and build wooden structures with stairs at the front of the house.

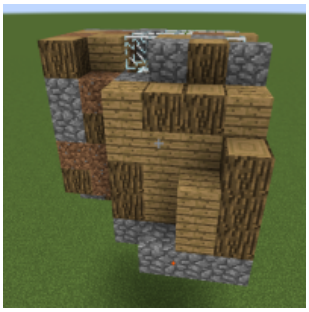


(a) (b) (c) (d)

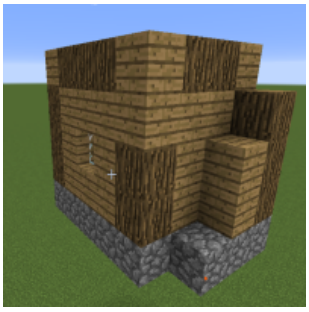
Fig. 7. Results from the third model trained with 11,000 data points. The model learns to build walls and have interior space.



(a) Starting noise state



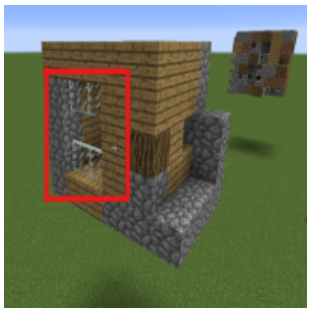
(b) Intermediate step in the process



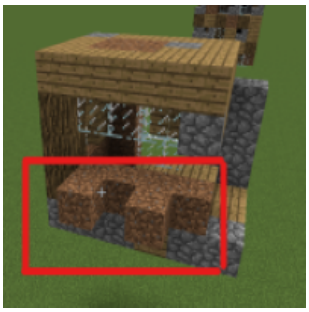
(c) Final resulting house

Fig. 8. We can see the starting noise state in (a), the intermediate step (b) where the agent is taking actions to repair the noise state to the house. Finally, we see the resulting house (c).

mud house, as seen in Fig. 1 (c) and (d)). We then trained for 500 epochs for 10,000 data points and got very good results, which can be seen in Fig. 10. We can see that the resulting house in Fig. 9(a) is a mix of the wooden house and glasshouse, with the large modern windows from the glasshouse and the wooden structure from the village house. In Fig. 9(b) we can observe it is a mix of all three houses, the wooden roof from the village house, the large glass window from the glasshouse, and the dirt balcony from the mudhouse.



(a)

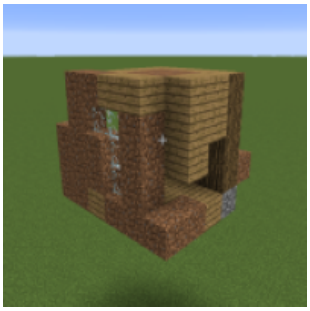


(b)

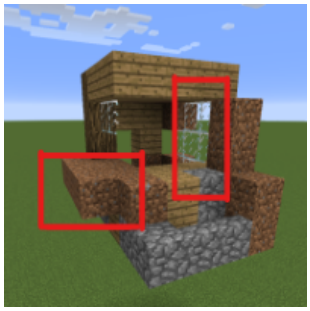
Fig. 9. Results from the multi-goal state model. We can see the large glass window in (a) from the glasshouse combined with the village house. In (b) we see the mudhouse balcony partly built combined with the glass window from the glasshouse and the wooden walls from the village house.

We saw some interesting results, in which each noise state generated different results each time with large variations. However, we see that the model struggles in fully building the dirt balcony(in Fig. 9(b)). This could be due to training with only 30,000 data points. In our final model, we trained with 400,000 data points for 500 epochs. This time we saw the model generate more mixed houses. One issue is that for some noise states, this model did not work as well, and struggled to complete the house, we believe this could be due to overfitting the training data. When the model, did work, it generated very livable houses with multiple features. The results can be seen in Fig. 10.

(a)



(b)



(c)

Fig. 10. Results from the final model. We see that there is a good mix of all the features from the various goal state houses. This model can even generate the dirt balcony feature on top of other house features.

Overall, we get good results from the final models, which generate houses from random states with a mix of features from the goal state houses. We were able to use the Path of Destruction algorithm to generate correct training data, and train a PCGML model to procedurally generate buildings with a variety of features. We can expand this process to use N number of goal states to get even more interesting and varying houses. Further, we can also train different models to generate houses of different sizes, by increasing or decreasing the initial state area.

B. Results for PCGRL

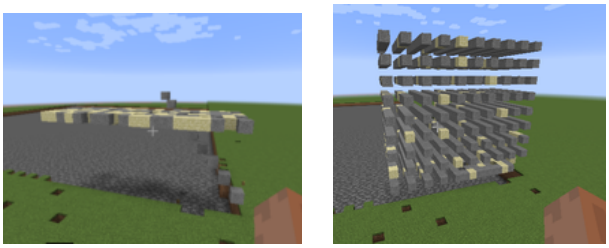
First, we experimented with having an initial state of a square of blocks and rewarded if the agent diversified the blocks and added noise to the building. This resulted in diverse types of blocks that were not structured.

Then, we experimented with using an agent to traverse from ground level to reach a higher level to build floors from the ground and the agent was optimized to make diagonal lines from the two points. In order to solve this problem, we moved on to implement wall building functions that optimize creating

(a) (b)

Fig. 11. Results from training with reward function that prioritizes using different types of blocks to build.

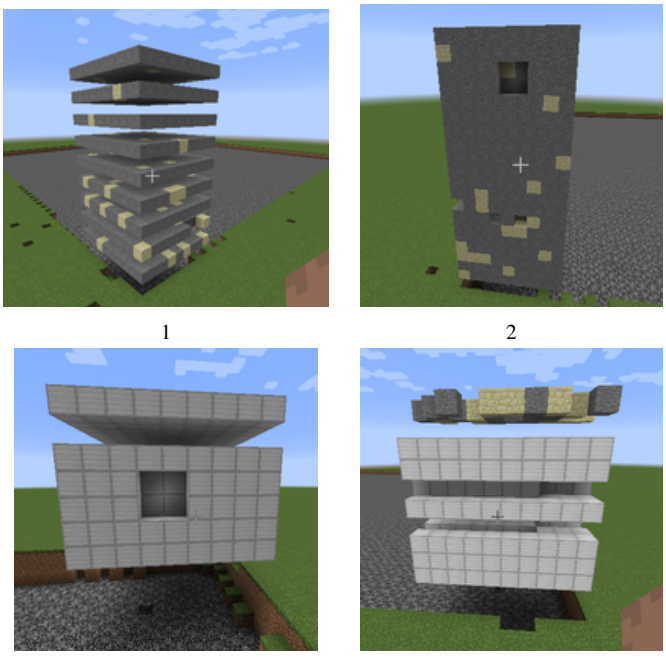
walls around a certain coordinate, however, the function did not give a proper result that can qualify as a house.



(a) (b)

Fig. 12. Results from training with an agent that traverses from ground to higher level and builds walls at a given coordinate.

For our research, the structure was a priority so we started aiming to build symmetrical and structured blocks as shown below.



1 2

3 4

Fig. 13. Our agent attempting build structured building with some noise.

Afterward, we started prioritizing house heights, and certain blocks that are suitable for house material and added some noise to the training. This gave a desirable outcome as the agent was building apartment complex like structures.

(a) (b)

Fig. 14. Results from training with an agent prioritizes height, blocks and structured noise

For training, we implemented an OpenAI Gym interface that allowed us to train without further obstacles. We use Proximal Policy Optimization (PPO) to train our agents. In our PCGRL environment, since our current representation space allowed the agent to take action only in the current location, we would like to experiment with using a representation space where the agent has full control over the location and block type which would allow us to be more creative in building larger structures.

C. Comparing results of PCGML and PCGRL

PCGRL results are dictated by the reward function and the representation space of how the agent can interact with the environment. Since RL does not use any prior dataset to train from or any previous knowledge to succeed, the PCGRL method could be used for more complicated tasks. Since building a house requires previous knowledge and certain prior rules that we need to inform the agent, we can achieve the same result using supervised learning without an agent. We attempted to eliminate the need for prior knowledge by adding a more tailored reward function, however, the end result gave us a house that was not really creative which matched the end result houses from PCGML. This means that with PCGRL we can get a more creative perspective on the problem that can be used to build larger structures in 3D environments and solve more complex tasks.

With PCGML using the Path of Destruction method to create its own training data, we will be able to take advantage of the supervised learning algorithm's advantages to quickly train the agent using training data to generate results. However, with training data, while the training process is greatly shortened, the house generated will strongly lean towards the styles of houses in the training data. Leading to good, but rather constrained training results. With more house data with diverse styles added, the agent would overcome this limitation. As when we compare the result using one house data vs three house data, We can already see how the agent will start utilizing more block types even trying

to build a balcony that is only featured in the third house data.

V. CONCLUSION AND FUTURE WORK

The main goal of the research was to create an iterative content generator using PCGML and PCGRL methods and apply it to generate 3D content in the Minecraft game. Using the Path of Destruction algorithm we were able to successfully generate training data and build several models which were able to generate interesting and viable living houses. We were able to use several goal state houses and build houses with a random mix of their respective features. We can generate varying houses from random noise states to get different results every time. Hence we can say that we can use this approach to generate 3D artifacts in a 3D setting, and the Path of destruction algorithm can be applied in a 3D setting.

Using different reward functions at the same time while balancing out the values and choosing which function to prioritize over made us question the quality of the training. Since the different reward functions are concatenated to return one state, the agent is not able to build different types of houses at every run that uses different reward functions at the same which can speed up the experimentation process. Moving on, we would like to experiment with using a target shape dataset of a sophisticated house by importing the dataset to have our agent aim toward building a similar structure. In the future we also wish to explore building different sizes of houses with more goal states for the PCGML approach.

REFERENCES

[1] M. Siper, A. Khalifa, and J. Togelius, "Path of destruction: Learning an iterative level generator using a small dataset," 2022. [Online]. Available: https://arxiv.org/abs/2202.10184

[2] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, "PCGRL: procedural content generation via reinforcement learning," *CoRR*, vol. abs/2001.09212, 2020. [Online]. Available: https://arxiv.org/abs/2001.09212

[3] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (PCGML)," *CoRR*, vol. abs/1702.00539, 2017. [Online]. Available: http://arxiv.org/abs/1702.00539

[4] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis, "Procedural content generation through quality diversity," *CoRR*, vol. abs/1907.04053, 2019. [Online]. Available: http://arxiv.org/abs/1907.04053

[5] T. Shu, J. Liu, and G. N. Yannakakis, "Experience-driven PCG via reinforcement learning: A super mario bros study," *CoRR*, vol. abs/2106.15877, 2021. [Online]. Available: https://arxiv.org/abs/2106.15877

[6] S. Sudhakaran, D. Grbic, S. Li, A. Katona, E. Najarro, C. Glanois, and S. Risi, "Growing 3d artefacts and functional machines with neural cellular automata," *CoRR*, vol. abs/2103.08737, 2021. [Online]. Available: https://arxiv.org/abs/2103.08737

[7] C. Salge, M. C. Green, R. Canaan, and J. Togelius, "Generative design in minecraft (gdmc), settlement generation competition," *CoRR*, vol. abs/1803.09853, 2018. [Online]. Available: http://arxiv.org/abs/1803.09853

[8] P. Bontrager and J. Togelius, "Learning to generate levels from nothing," 2021.

[9] D. Grbic, R. B. Palm, E. Najarro, C. Glanois, and S. Risi, "Evocraft: A new challenge for open-endedness," *CoRR*, vol. abs/2012.04751, 2020. [Online]. Available: https://arxiv.org/abs/2012.04751