Exploration vs exploitation - Investigating the effects of novelty on a NEAT population

Abstract

Introduction

Procedural content generation (PCG) is a technique used in video game design and other creative fields, such as art and music. The technique allows creators to generate a near infinite amount of content based on a few rules, constraints, and parameters. This allows games to be highly replayable, feeling more unique and unpredictable. An example of effective PCG is in No Man's Sky ("No Man's Sky" 2016), a universe exploration game which can generate 2^64 (~18x10^18) worlds, each roughly 78.5 miles^2 (~203 sqkm) (Newhouse 2016). Comparing this to one the biggest non-generated games, ARMA 3 (2012), which has a map size of 270 sqkm (Carlson 2013), barely bigger than one of the quintillions of the worlds in No Mans' Sky.

PCG does come with some downsides, mainly sacrificing quality control. A traditional game would have game designers' hand-designing all aspects of the game, meaning the output is predetermined and easily controllable. Since PCG is designed to be unending, it is impossible to completely quality check. Unlike other types of PCG, like art and music, the design of content in games is first and foremost based around playability, with artistic qualities being a secondary objective. Many existing machine learning PCG techniques allow for small variations in the content generated, which in a medium like music makes no difference to the overall quality, but small variations in game design can lead to the game being unplayable. (###Example about platformers?). But playability is not a binary scale, as (Koster 2013) explains there is a "golden section" of game design which is the perfect balance novelty and familiarity, which is pleasing for our brains, therefore better to play.

Another drawback to PCG is processing costs and generation speed. According to the steam hardware survey ("Steam Hardware and Software Survey 2023" 2023), which is the most comprehensive hardware census and continuously updated, there is still a massive range in hardware capabilities being used, with strongest graphics card being ~2400% stronger than the weakest ("Intel Uhd

Graphics 630 (Desktop Coffee Lake I5 I7) Vs Nvidia Rtx 3060-Ti," n.d.). The higher the hardware requirements for a game, the larger the proportion of users unable to run the game, alienating a portion of the customer base.

Outside of games, machine learning based PCG has made massive leaps forward. Models such as DALL-E 2 (Ramesh et al. 2022) can generate completely unique photo-realistic images from a text description, and ChatGPT ("ChatGPT Chat" 2022) can generate text answers to almost any question. These models have been so successful due in large part to the enormous amount of data they have been trained on and the huge complexity of the models, which presents problems for PCG in games. For DALL-E/ChatGPT to generate whatever is requested, they needed to have seen something similar to base their response off of. This is problematic for game world generation, as there exists no dataset big or diverse enough to adequatly trai these models (Summerville et al. 2018). Large-scale deep learning models are also extremely computationally expensive, heavily affecting the hardware requirements .

Another issue is the idea of creativity. Models such as DALL-E and ChatGPT do produce unique outputs, but the output is based on many existing sources. For these models this is not a big problem because of the huge variety of training data which allows the models to draw on many sources to generate new content ("The Quest for Ai Creativity" 2015). Games on the other hand generally have the same recognisable styles and artifacts throughout the world. A model splicing these together could lead to a disjointed world and recognisable from other games, leading to lower immersion.

The fundemental problem with PCG is the the *state-explosion* problem, where as the size of the generated content increases, the search space increases exponentially and brute force solutions become intractable (Godefroid and Khurshid 2002). Since it is impossible to verify if a solution is the best solution, an approximate solution is required. One solution to this is through the use of genetic algorithms (GA). GA's take inspiration from the biological process of natural selection and use it to evolve a solution to a problem by using a heuristic approach to move towards a better solution. Accomplished by removing ineffective algorithms from the population and allowing effective models to proceed and evolve further. Evolving a solution instead of training from past solutions comes with certain advantages. By having an algorithm which is controlled by the rules of a system rather than being trained on existing examples allows the model to come up with new unique solutions, rather than rehashing existing solutions.

This paper will be experimenting with NEAT (Stanley and Miikkulainen 2002), which is a type of genetic algorithm, and how the performance is affected by different levels of novelty. I will be investigating this within the context of the EvoCraft (Grbic et al. 2020) challenge, a PCG challenge for MineCraft (Persson 2023). Theses concepts will be explained in more detail.

Aims and Objectives

Aim: Investigate how varying levels of novelty affects the (quality, realism??) of procedurally generated cities in Minecraft Objectives: 1. Investigate other competition entries for the EvoCraft challenge 2. Research procedural content and novelty search techniques 3. Implement fitness functions to evaluate single structures 4. Experiment with varying levels of novelty 5. Evaluate multi-structure cities to see effects of novelty

Background

Minecraft

Minecraft is a 3D, open-world, sandbox, voxel-based video game. Each voxel, called a block, can be broken and replaced to build structures, allowing players to apply their creativity. Minecraft uses PCG and world seeds to create a unique world which is 3.6 billion blocks^2 (Whitworth 2021), allowing players virtually infinite space to explore and build. Because of the open-ended nature of the game and the simplicity of interactions with the world, Minecraft has become a platform for many AI challenges, including mineRL. This competition focused on an agent within Minecraft which has to complete a variety of tasks in an unknown environment. Because there is no one defined task, the algorithm has to be able to complete many smaller problems, with the eventual goal of improving research into general intelligence.

For interactions between Python and Minecraft world I will be using a Minecraft server with a Bukkit (2010) plugin called RaspberryJuice installed and the McPi Python library to communicate with it. Bukkit is a server modification tool with an API which allows users to easily create server plugins. The plugin converts Python commands into Java commands which can be processed by the Minecraft server. The McPi simply sends Python commands to the RaspberryJuice plugin.

EvoCraft

To investigate genetic algorithms for PCG, I am working on the EvoCraft Challenge. The EvoCraft challenge brief is to create an open-ended algorithm which is capable of creating novel and increasingly complex structures in MineCraft. These algorithms have to be unending and should aim to diverge over time rather than slow down and become repetitive. One of the drawbacks of PGC was the lack of quality control, and the problem with infintely generating content becoming repetitive over time. This challenge aims to use evolving algorithms to keep generating content which keeps diverging and becoming more interesting.

Other EvoCraft Entries

Evocraft PCGNN ***Michael Beukman, Matthew Kruger, Guy Axelrod, Manuel Fokam, Muhammad Nasir***

[ref] were came runners up in the EvoCraft competition with their endless city generator. Their approach broke a city down into the component 'levels', starting from the lowest level, the house and garden. To generate a house and garden they broke this down into 4 components: the house structure, roof, decorations, and garden. They then used a PCGNN (Procedural Content Generation using Neat and Novelty search) to generate each of these components. The house, roof, and decorations are all generated as 3d tilemaps to be placed in-world. The house consists of walls, empty space, and enterance, the roof consists of a design covering the area beneath it, and the decorations consists of decoration blocks filling floor space inside the house. The garden works slightly differently as it is a 2d tilemap covering an area with flowers, grass, and trees. They then used these component houses and gardens to create a town. A town is its own generated 2D tilemap of houses, gardens and roads @(**???**) in appendix@@, where are road connects all the houses in the town. They then placed many towns together to create a city, which could grow endlessly. @@ Put about successes and drawbacks? About other details in paper @(**???**) in appendix@@ https://github.com/Michael-Beukman/Evocraft22_PCGNN

simple_minecraft_evolver ***real_itu***

The simple Minecraft evolver is basic NEAT implementation which aims to build a tower towards a gold block in the air. Early generations start by building towers in random directions, but as they evolve they move more towards the gold block. https://github.com/real-itu/simple_minecraft_evolver

https://github.com/GoodAI/EvocraftEntry

Neural Networks

A neural network is a type of machine learning technique which is modelled after the human brain. A neural network consists of neurons, shown in figure @@. Each neuron has a weighted connection to other neurons, a bias, and an activation function.

$$y = f(\sum_{i=0}^{n} w_i x_i + b)$$

(Equation for neuron output) w: input weights from previous neurons x: input values from previous neurons b: bias Sigmoid activation function:

$$f(x) = \frac{1}{(1 + e^{-x})}$$

Shown is the the sigmoid activation function as an example, but many different activation functions exist. In a neural network many of these neurons are fully

connected together to create a network of neurons, an example shown in figure @@. A typical neural network has an input layer, some hidden layers, and an output layer. To make a prediction from a neural network, some input values are given to the input layer neurons and those values are used to calculate the values in the next layer, then the next, until reaching the output layer.

NEAT + Novely Search

NeuroEvolution of Augmenting Topologies (NEAT) is type of genetic algorithm (GA) which mimics biological evolution to increase the complexity of neural networks. Traditional neural networks have a fixed structure of input, hidden, and output layers, usually fully connected by weights and biases. To improve the performance of the model the network is given examples of inputs and corresponding outputs, and techniques such as back propogation are used to update the network. NEAT works in a much different way. Instead of having a fixed structure inside the genome (the name for a network in NEAT context), the genome structure evolves over time. The genome is initialized with basic connections inside the genome, resulting in essentially random output. A population of individuals, each with a copy of the genome, is generated. This population then "evolves" through a process called "mutation". Each individual in the population will have random changes made to their internal structure. Each individual is then evaluated using a fitness function, which evaluates an output and assignes a score. The top individuals are then taken and the others are destroyed. The population is then recreated, to create a next "generation", using these top individuals and the mutation process is repeated. The goal of the process is, through random mutation, to move towards a network which can produce perfect output. NEAT, and other GA's, are especially useful when the size of the search space is extremely large and cannot be solved through exhaustive search techniques ##Needs rewriting. Terminology is wrong and incomplete. Include other terms such as crossover use https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

There is however a problem with this evolution process, a population's fitness score can becomes stuck in a local maxima. This comes from a deeper ideological difference between biological evolution and genetic algorithms. The purpose of an GA is to reach this global maximum fitness, whereas biological evolution aims to both evolve individuals which will survive (high fitness), but will also spread out and diverse over generations. If a species does not diversify then the species is more vulnerable to diesases and changed to the environment. For an GA to get out of a local maxima it must first drop in fitness score before finding another gemone structure which would allow it to reach the global maxima. Traditional GA algorithms don't allows for this as any drop in fitness score would kill the individual and stop it from evolving further. Here we take a note from biological evolution and promote diversity, called novelty search. Novelty search gives a higher fitness to models, which may have a lower calculated fitness, but have genetically mutated

and diverged from the other models and its parents. This allows models to drop in fitness and explore other methods of getting to a global maxima. https://www.biologicaldiversity.org/programs/biodiversity/elements_of_biodiversity/

Reinforcement Learning PCG

Reinforcement learning (RL) is a type of machine learning composed of three key elements: an agent, the environment, a reward. RL uses a trial-and-error method with an agent interacting with an environment. An agent makes an action within and environment and will either be rewarded, if the the action was a positive action, and punished, if the action was negative. The eventual goal for a RL agent is to learn a policy, which is a mapping from input states to output actions. The agent starts with random actions and getting experience, in the form of state-action-reward, which is used to update the policy of the agent. Eventually the agent aims to maximise the cumulative reward signal by maintaining a balance between exploring, to learn new experiences, and exploitation, leveraging existing techniques.

RL gains many of the same benefits as genetic algorithms. Just like GA's, RL's require no large dataset and therefore don't have any of the intrinsic biases and creativity issues which come with it. There are some subtle differences between the two algorithms. RL can sometimes suffer trying to reach a global maxima in reward/fitness. Because RL is one agent learning over many iterations, if the agent chooses a strategy which works well, but only reaches a local maxima, it would have to unlearn that entire strategy and come up with a new one for it to reach a global maxima. A GA on the other hand has many individuals in a population, each which can explore their own routes (promoted through novelty search). Each route that ends in a local maxima will be killed in favour of a route which produces a global maxima, therefore having a higher likelihood of reaching the global maxima. While the RL model approach has the potential to be effective in this project, the properties of NEAT make it more desirable.

https://pythonistaplanet.com/pros-and-cons-of-reinforcement-learning/

General Adveserial Networks

A General Adveserial Network (GAN) is type of deep learning network, composed of two neural networks: A generator and descriminator. The role of a generator is, once trained on the training data, to generate more examples which resemble the training data. The role of the descriminator is to tell the difference between examples from the training data and the examples generated by the generator. A common analogy for this the relationship between an art forger and art appraiser. The appraiser ensures that only the most convincing art forgeries survive and sell art, improving the quality of the forged art.

GAN's have some powerful advantages which make them very powerful in certain situations. Since the training data is only used for examples, there is no

need to label the data. Once a GAN is fully trained, both parts of the network can be used. A well trained generator can create very realistic and unique creative works, in many different styles. DALLE-2, a realistic image generator, uses a GAN-like network structure to create unique photo-realisic images. The descriminator can also be used to detecting abnormalities, for example medical imaging, quality control, and fraud detection. There are some isses with GAN's. They require vast amounts of training data and the wider the range of outputs being generated, the higher the size of the training data required. They are also a black box and very hard to reason why it came up with what it did, making it very hard to fix problems like diversity of output and garbled data. GAN's naturally lose detail from the input to the output, which makes the generated content lose some finer details. This is not a problem for creative works as some minor variation has no affect on the quality of the output, but does have an effect on PCG for games. Some minor changes to game generation can leave it unplayable and useless. The combination of not being able to generate at a very fine level, being a black box, and requiring huge amounts of training data, it can be very hard to reliable produce quality controlled content like games. Because of these disadvantages a GAN approach is unsuitable for this project.

https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8877944/ https://jonathan-hui.medium.com/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b

Implementation

As previously mentioned, I will be experimenting with novelty in NEAT populations to create structures in MineCraft. I am taking a note from @(**???**) PCGNN@@ and breaking down the solution into 2 models, one to produce the base of the structure and the other to create the roof. This each approach can be broken down into three main objectives: genome creation, the fitness functions, and novelty experimentation. The structural base model, which I will call the house model, will create a 3D tilemap of the blocks to place. The roof model will output a 2D heightmap, fig@@, which details the height of each block to place. I chose to do the roof model as a 2D heightmap as a roof only needs to be one layer thick and it is much more computationally efficient. I have chosen to do this project in Python. While the competitors (Java, C++, R...) are much faster than the core Python modules, there are libraries, such as NumPy, which are written in C and are extremely quick. This means it can perform well on large datasets, making it a favourite for data scientists. Because it is a popular data science language, there is also a large collection of machine learning libraries supported to help development.

Genome creation

A genome is an individual within a population. It contains the information needed to create a neural network, which can be used to create structures. The

aim is to create a genome which, given some configuration information (building dimensions, type of blocks…), can output a structure which follows the configuration instructions. To implement the NEAT algorithms, I am using a python library called python-neat. This library manages the encoding and mutation of the population, the only implementation that is required is giving the evoluation hyperparameters and evaluating the fitness of each individual.

Evolution hyperparameters

Hyperparameters are the parameters which control the learning process for each population, such as mutation rate and network activation functions. One of these parameters is the input/output sizes of the genomes, which must be a fixed value for all individuals in a population at all times. The simplest way to use a genome to create a structure is to map each output neuron to a block to be placed in the structure. The disadvantage of this strategy is that since the output size is fixed, the size of the structure must be fixed, which limits the possible creativity and could easily become boring. A different strategy is to have the output not be a whole structure, but one block in the structure, and run the model many times. Using this technique the output size of the genome is each block possible to place. The genome will then choose the most likely block that should be placed, denoted $\hat{y}$. Unfortunately MineCraft contains over 1050 blocks/items, many of which cannot be placed or are dependant on other blocks around it, and there is no list of all structural blocks. Therefore I went through each blocks and made a list of each block which can be placed, which came out to 220 blocks.

The house model genomes should be able to take in some configuration options, to control which blocks should be predicted. The information given will be: the blocks surrounding $\hat{y}$ (denoted x_surr), the dimensions of the structure (height, length, width), the seed blocks which should be used, and the coordinates of $\hat{y}$ (relative to the starting point). The genome will predict the building from the bottom up and input is padded by 1 to enable x_surr to be found anywhere. Figure @@ shows a diagram of an example x_surr. Because the genome is predicting $\hat{y}$ from the bottom up, any blocks above $\hat{y}$ are not taken into account, because they will not have been predicted yet and will provide no information. The current $\hat{y}$ coordinates are required to give the genome more information about the placement. The seed block are required to give more control over the block styles of the building and to add some randomness to the input.

The roof model genomes work in a similar way, but in 2D. A height map consists of the heights of blocks to place, not the blocks themselves. This ensures that all spaces are filled in the roof and still allows to interesting structures. The process is similar to the house model, where each height is generated individually ($\hat{y}$) with the surrounding heights (x_surr) used for inputs, shown in figure @@. Same as the house model, any heights which haven't been generated have the value -1. The other information given to the genome is the dimensions of the roof (length, width, height) and the coordinates of $\hat{y}$.

Fitness Functions

A fitness function is used to evalute the quality of a candidate solution, to help it reach the desired solution. According to @(**???**)@@ a fitness function should be:
- Clearly defined: it should be easy to understand and provide meaningful insight into the performance. - Intuitive: Better solutions should get a better score and visa-versa - Efficiently implemented: NEAT requires many generations to evolve a good solution, so the fitness function should not be a bottleneck - Sensitive: Should be able to distinguish between slightly better and slightly worse solutions to allow a gradual movement towards a better solution https://towardsdatascience.com/how-to-define-a-fitness-function-in-a-genetic-algorithm-be572b9ea3b4 For each model the fitness is calulated by combining the novelty score and structure score (explained in more detail below). The ratio of these combinations decides the level of novelty in the population.

Structure Scoring

House Model

To start calculating a house score, the components of the desired structure have to be broken down. An example structure is shown in figure @@. When deciding fitness functions it is important to limit the scope of what can be expected, there is always more detail that can be added to the scoring. The purpose of the scoring is to highlight the important parts of a structure and guide the genome towards the desired output, but leaving enough flexibility to allow for creativity. A balance between control and creativity. Each of the component scoring functions generate a value between 0@(**???**)@(**???**) and the average is used to calculate the final overall score. The fitness functions I decided on were: 1. A bounding wall An important part of a house is a complete wall with no airgaps and one block thick. 2. Airspace Inside the wall space there should be a maximum amount of air space inside to ensure the structure can be used. I am reducing the scope of the project and not trying to decorate the structure inside; I am only concerned about the external view. 3. A door There should be a door on ground level to enter the structure. This is the only score which is either 0 (no usable door) or 1 (usable door). 4. Seed blocks The seed blocks inputs are to control the types of blocks which are generated by the structure. This scores models higher which prioritise using blocks from the three seed blocks. For each seed block which is in the top 6 blocks used, 1/3 is added to the score. I chose top 6 as this almost always includes an air block in the top 5, which will never be a seed block. 5. Symmetry In almost everything considered beautiful, there is some element of symmetry. It has been well documented that humans find symmetry much more attractive and soothing to look at. Instead of limiting my model and forcing symmetry in one axis, I am checking both xy axes and chosing the one which has the highest symmetry. The symmetry is calculated

by comparing the percentage of equal blocks on both sides of the axis, giving a value between 0 and 1. https://www.nature.com/articles/s41598-018-24558-x https://www.psychologytoday.com/gb/blog/beastly-behavior/201907/why-are-symmetrical-faces-so-attractive

Roof Model

Since the roof is a simpler structure than the house, less scoring metrics are required. Since this is a simple structure it is very easy for the fitness functions to be too controlling. For example, if the scores maximise for sloped rooves, then only sloped rooves will be generated, therefore it was very important to have gentle control. The scores I decided on were: 1. Compliance This ensures the heightmap generated fits within the limits I have set. For example, if the maximum height set is 5 blocks and the heightmap contains a height greater than 5, then the solution doesnt pass the compliance check. This fitness is either a 0 (failed the check) or 1 (passed the check). 2. Symmetry Just like the house model, I am maximising for symmetry in the xy axes. 3. Visual Complexity This is the fitness for quality checking and making the solution more interesting. To make the solutions more intersting I am maximising for complexity in the surface. This is achieved by rewarding changes in height over flatness. This is accomplished by counting the ratio of changes in height of 1 vs no changes in height. I am purposly not counting changes in height of more than 1, as that will leave gaps and ruin the look of the roof. An example of low complexity vs high complexity is shown in figure @@. I chose this solution over rewarding specific types of structure, like flat or triangle-sloped rooves, to allow for more creativity.

Novelty Scoring

A novelty score encapsulates how much an individual in a new generation has changed from the previous generation. A higher novelty score means an individual has significantly changed from the previous generation, leading to a higher fitness score. The hope of this is for individuals to be able to make changes to their genotypes which may not improve the structure score in the short term, but will lead to long term improvements. In populations with a high novelty score to structure score ratio, the individuals will be able to spread out try more novel solutions without being killed. If novelty is too high then the structure score becomes less and less important, leading to the solutions being novel but useless. The model still has to enforce the structure scores.

When finding the novelty of a genome, it needs to be compared to its closest relations. If a genomes if very different to its closest relations, then it will have a high novelty score. If there are two genomes which are completely different then they will always have a high novelty score, regardless of the changes made to the genomes. The easiest way to compare a genome to the closest relations is by taking the average novelty between the k-nearest-neighbours (KNN). To improve the novelty score, a new genome can also be compared to the previous

generation by archiving them. A novelty score is then composed of the comparison between neighbours and archived ancestors. To promote the highest novelty, only individuals above a threshold,

$$\rho_x$$

, are added to the archive. From some simple testing,

$$\rho_x = 0.85$$

archived the right amount of the highest novelty genomes.

Dynamic Novelty

One issue with a fixed level of novelty is that it is not always suitable for every situation. When a population is stuck in a local minumum a very high novelty ratio is needed to help promote potential solutions which move the the population out of the local minumum. On the other hand when the population is quickly improving, a low novelty ratio is needed to promote the individuals which are improving the most. Having a high novelty ratio in this circumstance is not bad but increases the number of generations needed converge to a solution. An improvement to this is a dynamic novelty system which looks at how quickly a population is improving and adjusts the novelty accoringly. To get the speed of improvement, the gradient of n-previous generations structure score's are taken and the inverse is used to calculate the novelty required.

Evaluation

Future Work

References

Carlson, Patrick. 2013. "Arma 3 Map Might Be Bigger Than You Ever Imagined." *Pcgamer*. PC Gamer. https://www.pcgamer.com/arma-3-map-might-be-bigger-than-you-ever-imagined/.

"ChatGPT Chat." 2022. *ChatGPT*. OpenAI. https://chat.openai.com/.

Godefroid, Patrice, and Sarfraz Khurshid. 2002. "Exploring Very Large State Spaces Using Genetic Algorithms." In *Tools and Algorithms for the Construction and Analysis of Systems*, 266–80. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-46002-0_19.

Grbic, Djordje, Rasmus Berg Palm, Elias Najarro, Claire Glanois, and Sebastian Risi. 2020. "EvoCraft: A New Challenge for Open-Endedness." http://arxiv.org/abs/2012.04751.

"Intel Uhd Graphics 630 (Desktop Coffee Lake I5 I7) Vs Nvidia Rtx 3060-Ti." n.d. *UserBenchmark*. https://gpu.userbenchmark.com/Compare/Nvidia-RTX-3060-Ti-vs-Intel-UHD-Graphics-630-Desktop-Coffee-Lake-i5-i7/4090vsm356797.

Koster, Raph. 2013. *A Theory of Fun for Game Design*. O'Reilly Media.

Newhouse, Alex. 2016. "Making Sense of No Man's Sky's Massive Universe." GameSpot. https://www.gamespot.com/articles/making-sense-of-no-mans-skys-massive-universe/1100-6441344/.

"No Man's Sky." 2016. *No Man's Sky*. Hello Games. https://www.nomanssky.com/.

Persson, Markus. 2023. "Welcome to the Minecraft Official Site." *Minecraft.net*. Mojang Studios. https://www.minecraft.net/en-us.

Ramesh, Aditya, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. "Hierarchical Text-Conditional Image Generation with Clip Latents." http://arxiv.org/abs/2204.06125.

Stanley, Kenneth O., and Risto Miikkulainen. 2002. "Evolving Neural Networks Through Augmenting Topologies." *Evolutionary Computation* 10 (2): 99–127. https://doi.org/10.1162/106365602320169811.

"Steam Hardware and Software Survey 2023." 2023. *Steam*. Valve Corperation. https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam.

Summerville, Adam, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. "Procedural Content Generation via Machine Learning (Pcgml)." http://arxiv.org/abs/1702.00539.

"The Quest for Ai Creativity." 2015. *IBM Cognitive - What's Next for AI*. IBM. https://www.ibm.com/watson/advantage-reports/future-of-artificial-intelligence/ai-creativity.html.

Whitworth, Spencer. 2021. "How Big Is a Minecraft World?" *Sports News*. Sportskeeda. https://www.sportskeeda.com/minecraft/how-big-minecraft-world.

2010. *Bukkit*. Mojang. https://dev.bukkit.org/.

2012. Bohemia Interactive. https://arma3.com/.