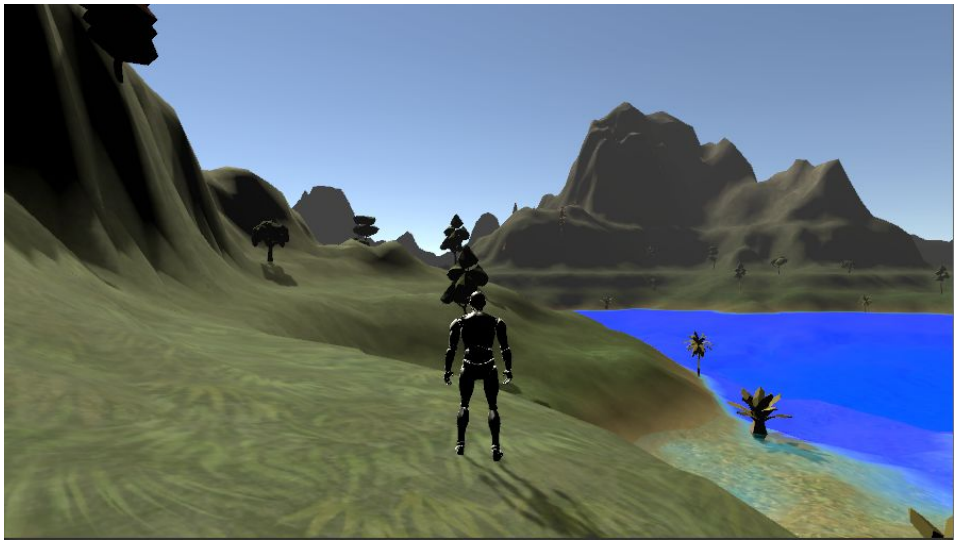
Abertay University

Procedural Generation of Terrain using Machine Learning Guided by Analysis



DISSERTATION

Rohan Menon, MSc Computer Games Technology
School of Design and Informatics,
Abertay University

ABSTRACT

Procedural Content Generation (PCG) can be used by a relatively small team to create multiple levels with a short amount of turnaround. The example within this project is terrain generation using a height map based on Perlin noise. This also means that there are a lot of output possibilities for Procedural Generation which can be overwhelming to the design/development team. Artificial Intelligence can be used to go through this large variety of output data and pick up patterns in the positioning of level objects that have better engagement or playability.

A Machine Learning(ML) agent can be used to look at multiple generations of randomized terrain data and create steps that would make the noise generation into something more approachable for level design. This can be done by looking at the output of the noise generation parameters and attempting to change them so that the generation is more meaningful or better for gameplay. An object generator behaviour can then be used to check the ideal position of gameplay elements for a given noise generation. This allows for further streamlining of the design process when a designer is trying to manipulate the noise generation algorithm to create relatively playable terrain, the ML Agent can help them arrive at a more balanced or ideal output.

The aim of this project was to look into PCG with ML and how it can be used to ease or improve the work of a designer. The integration with a designer would be key to ensure that the project/tool would supplement a designer and also show new or unique ways to solve a problem. Different learning strategies for PCG with ML were understood and compared, in particular, how to generate a tool to create meaningful terrain from a combination of noise data. The end result was different Neural Network(NN) files to help a designer that can be incorporated to generate or refine terrain noise parameters to be more viable for better gameplay.

KEYWORDS

Procedural Content Generation, Machine Learning Agents, Pattern Recognition, Gameplay Engineering, Unity, Nav Mesh Agents

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

3.1 Procedural Generation

Procedural generation is a method of creating data algorithmically as opposed to manually, typically through a combination of human-generated assets and algorithms coupled with computer-generated randomness and processing power(Smith, 2015)[1]. Fractals, for example, are a method of procedural generation where a mathematical function, algorithm or “procedure” can be used to define shapes, patterns or spaces. It can be a very useful tool to speed up asset generation that would traditionally be made by a relatively large group of artists. This can be to automate part of the creation process or be used as a base to build up additional detail on top of it. This automation not only reduces the time for content generation but also allows for the randomization as well as personalization of content according to certain playstyles. This improves replayability of the game in question as well as the chances of gameplay being repetitive is significantly reduced. As a simple example, when creating a level, we can place reward pickups in areas of the map and generate enemies around it.

This can be used for a wide variety of games ranging from shooters to role playing games as well as puzzle games. It allows a relatively small team(or in some cases just an automated script for some puzzle games), to create a vast variety of gameplay. It has now been used for games from Minecraft (Persson, 2009)[46], to create different terrain, resources and enemies based on multi layered perlin noise, indie hit Spelunky (Yu, 2008)[47] to create levels composed of mini-structures combined together to form a complex and many unique platformer experiences, to something like Candy Crush Saga (King, 2012)[48] for dynamic difficulty adjustment in how the puzzles for various levels are generated. Some other notable games using this technique are No Man’s Sky (Hello Games, 2016)[49] to generate an entire universe based on noise data and one of the earliest, genre defining examples being, Rogue (Wichman, 1980)[50] to create unique layouts for a multi layered dungeon.

Usually, like in Rogue or Spelunky, procedural generation systems are used to create a large sample set of levels and an evaluation function is used to select a level that is most fit. Others remove areas that turn out to be unreachable. One particular example of this kind of generation are cellular automata to create regions of procedurally generated tile maps. (Johnson et al., 2010)[8]

Rogue

Minecraft

No Man's Sky

Figure 1. Some games using procedural generation

Procedural generation can be used to not only create new and varied content for each gameplay session but also as an additional tool for designers to guide the level generation process. Different procedurally generated content can be used as a template that designers can then tweak and create better maps or simply as a way to generate a baseplate that can be used to create more varied levels faster.

In the current case, procedural generation is being used to generate terrain and place objects over it in positions that would lead to interesting gameplay. The noise based algorithm allows one to create crests, mountains, valleys and water areas in the terrain. An object distributor is then used to spread out the generation of in game assets like trees or ruins that the player can visit.

Noise Based Terrain Generation

Noise is the random or pseudo random generation of a set of values in some dimension. The PCG used in this project is similar to games like Minecraft wherein multi layered Perlin noise(Perlin, 1998)[12] is used to generate and populate a terrain with objects. These octaves create layers of large scale and small scale noise that are layered together, using values of lacunarity (how much the layers interact with each other) and persistence (how much each octave affects each other). The layers of noise maps generated are made using a seed and offset that are assigned when the noise is generated. This is then combined to create the overall height map. An additional curve is used to lengthen or shorten heights and a falloff curve can be applied to generate additional details to the generation.

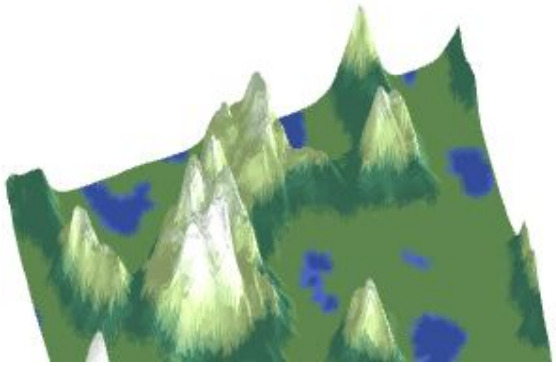


Figure 2. Example terrain from making maps with noise functions (Patel, 2015)[58]

This height map is then used to generate a terrain mesh that can have a level of detail applied to it to skip some vertices so that far away terrain meshes can be spawned with lower detail. This increases the efficiency of the generated terrain mesh as far away terrain won't need a very high amount of vertex resolution anyway.

Distributing Objects Across Terrain

Objects can then be distributed across this heightmap. This is either done randomly using a scatter algorithm like poisson distribution(Yang et al., 2015)[3] or a region based algorithm like voronoi regions(Aurenhammer, 1991)[4]. Another ML agent that is an object creator can be trained to generate check points for a terrain generation based on its noise parameters, seed and offset. The checkpoints will need to be arranged in such a way that they do not cross over each other's paths. Additional parameters like the next checkpoint being visible from the current one can be enforced in the machine learning algorithm as well. Obstacles can be created to wall off potential paths that break the flow of how checkpoints should be visited.

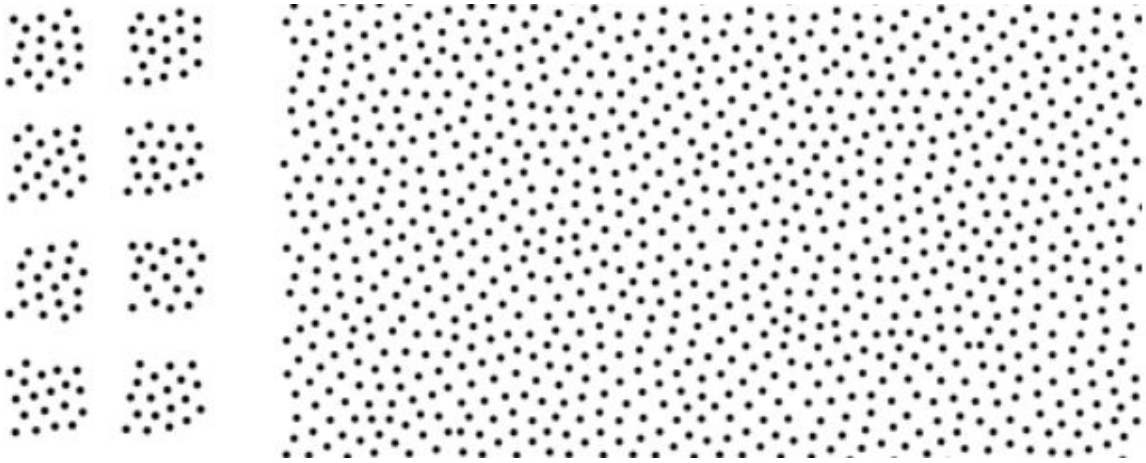


Figure 3. Blue noise sampling

3.2 Machine Learning (ML Agents)

Machine Learning is the concept of training a neural network to make observations in an environment and making it take actions to arrive at ideal conditions. Reinforcement Learning using Recurrent Neural Networks have been applied in a variety of situations, from games and simulations of real world phenomena to robotics and analysis.

The ML Agents Toolkit provided by Unity allows someone to create environments for an “Agent”, an object that trains a neural network by observing values in its environment, and defining decisions for a neural network based on these observations to take actions and evaluate the outcome of the action based on a reward defined by the user. This is particularly useful for extremely tasks that require complex input and have a variety of possible outputs based on the input but have some way of defining an “ideal” outcome. For example, training a walker object with multiple motors acting as legs to walk (Haarnoja et al., 2019)[5] or training humanoid gameobjects to run an obstacle course or use active ragdoll effects(Booth et al., 2019)[6]. These scenarios show how extremely complex environments allow Machine Learning to shine as it is able to crunch large amounts of experimental data to create an outcome that would be hard to arrive at through regular processing. Machine Learning agents are also known to find unique and unpredicted outcomes, also known as emergent behaviour to a simulation as showcased by a paper using multiple agents to play hide and seek(Baker et al., 2020)[7]. This is very relevant to an area like PCG where unique and interesting outcomes would be useful for creating complex gameplay variations.

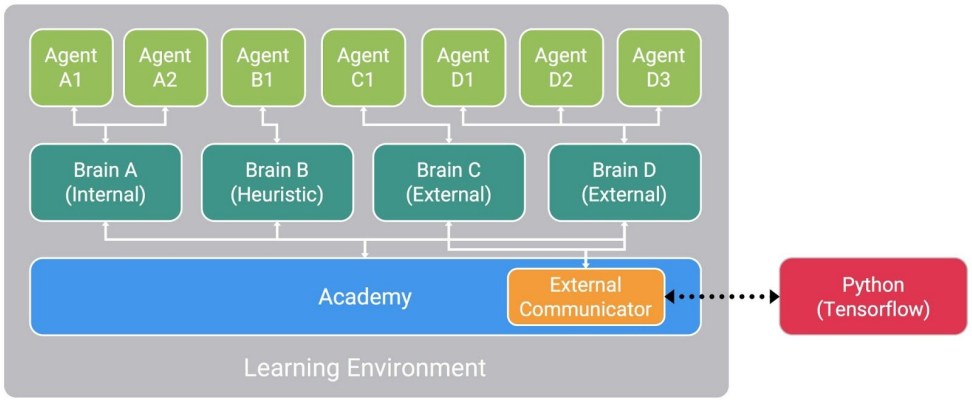


Figure 4. Learning Environment for ML Agents

There are a variety of strategies available for RL, two notable ones when it comes to ML Agents are PPO and SAC which will be discussed further later on. A key component of RL is tuning hyperparameters based on the complexity of the task at hand. The main drawback of reinforcement learning is that training time can take hours but the output of the training can shorten the workload considerably for users of the agent itself.

The aim of this project will be to discuss and evaluate strategies on how to train a machine learning agent for PCG. With particular focus on how the ML agent can be used to ease the weight of the content generation process, just like a good PCG system is used as a way to enhance or support a designer or developer rather than replace one. To look at whether there are ways to integrate ML with PCG to not only create many different generations of content but also add meaningful structure and refine these generations as well.

The aim of the project is, how can PCG with ML be used to improve the job of a designer, providing a range of content while ensuring correctness, improvement and promoting emergent behaviour that they might not have thought of?

BACKGROUND & LITERATURE REVIEW

4.1 Procedural Generation

Level Design and Procedural Content Generation

Level Design is the creation or placement of game objects in a 2D or 3D space in such a layout that players are able to get a sense of progression through the game. Procedural Generation has been studied by a variety of papers for generating multiple levels with infinite replayability. Many different strategies have also been implemented in commercial games as well with varying degrees of success across a large number of genres.

Some games focus on procedural generation for in-game items and loot, for example World of Warcraft (Blizzard Entertainment, 2004)[51] or Borderlands (Gearbox Software, 2009)[52] which add complexity, surprise and player choice in the items they collect. Genres like *roguelikes*, with the genre defining Rogue (Wichman, 1980)[50] and Diablo(Blizzard Entertainment, 1996)[54] using procedurally generated dungeons and rooms to add surprise and a feeling of exploration in their playthroughs. Other *roguelikes* like FTL (Subset Games, 2012)[53] use a complex set of encounter generators and grammars to create gameplay that progressively rises in difficulty but also ensures that a playthrough has significant variance to an earlier one. Even puzzle games like Candy Crush Saga (King, 2012)[48] use procedural generation to create interesting and varied gameplay easily and in very large numbers allowing for increased replayability. Open world games like Minecraft (Persson, 2009)[46] and No Man's Sky (Hello Games, 2016)[49] use PCG to create vast open spaces, dungeons and planets along with interesting flora and fauna to populate these worlds that are created using a noise generation algorithm. Of particular note in the last category is the fact that these games were created by relatively small teams but offer a massive playspace for players to explore with the added bonus of near infinite replayability.

Map and world generation are of particular focus for this project and there are papers that discuss different ways for world generation. Some ways that are commonly used are listed below:

1. Generative Grammars
   Defining a set of tasks, called a grammar to accommodate into a level design, in most cases, sequentially. The level is then created taking this grammar into account, for example an enemy protecting a key which then leads to a locked door. The levels are placed in such a way that the second room or area will be dependent on the first. A key point of this form of generation is that it defines the

set of actions first and then creates the level around these actions rather than populate a pre generated level with a sequence of actions. A few approaches are cited here. (Dormans et al., 2011)[2]

2. Cellular Automata
   It is dividing the level generation into various cells and using something like a noise function to arrange the cells in interesting ways. It leads to highly randomized levels and is often used in conjunction with another form of procedural generation to give some structure to gameplay. (Johnson et al., 2010)[8]
3. Monte Carlo Tree Search
   It is an algorithm used to explore decision spaces, making use of different heuristics to weight the potential value of a given state in a decision tree. Can be used to have an advanced generation that is populated in logical steps under a "tree" of rules. It is particularly useful to create highly structured levels with very defined outcomes for example, a puzzle game. (Graves et al., 2016)[9]
4. Search Based Procedural Content Generation
   Unlike the above PCG methods where the aim is to generate a single iteration quite similar to randomized generation using a seed, this approach uses different grades for judging a generated output and using this data to "score" a level generation. If the generated level has a good score it is accepted and the map is selected. (Togelius et al., 2010)[10]
5. Genetic Algorithms
   Genetic algorithms use generative policies and a fitness function to generate multiple kinds of outputs and select the outputs that are ideal respectively. They use a combination of randomized mutation and crossover functions as generative policies. They are very useful for creating closed structures and rooms to add variance to gameplay. Alternatively they can also be used for large open spaces to create a multitude of new maps. (Brown et al., 2018)[11]

Fractional Brownian Motion Noise

Perlin noise is a tried and tested noise generation algorithm developed by Ken Perlin in 1983. It is a good way to define a 2D array of float values ranging from -1 to 1, in such a way that it creates a smooth gradient between any 2 points. It is very efficient and can be used to create an infinitely movable and scalable array of noise data by using a function to query a position in the noise. Perlin noise was very useful to create textures for CGI in the early days. In video games, a more recent and efficient form of Perlin noise called Simplex noise is used because of it's faster generation (useful in real time applications) as well as better scalability in higher dimensions. (Perlin, 1998)[12]

Simplex noise has the very useful quality of being function driven, this means when creating the heightmap, a function can be used to extract pixels for the heightmap positions required. This offers substantial pros over other noise generation patterns that either do not create the high quality noise that perlin noise is able to create or require a larger amount of memory overhead. It is slightly more processor intensive when using the function than something like value noise but since the noise pattern is generated only when a heightmap for a mesh is being created the low memory usage of perlin noise wins over it being more processor intensive. Some other forms of noise generation algorithms are Diamond Square Algorithm, Value Noise and Worley Noise.

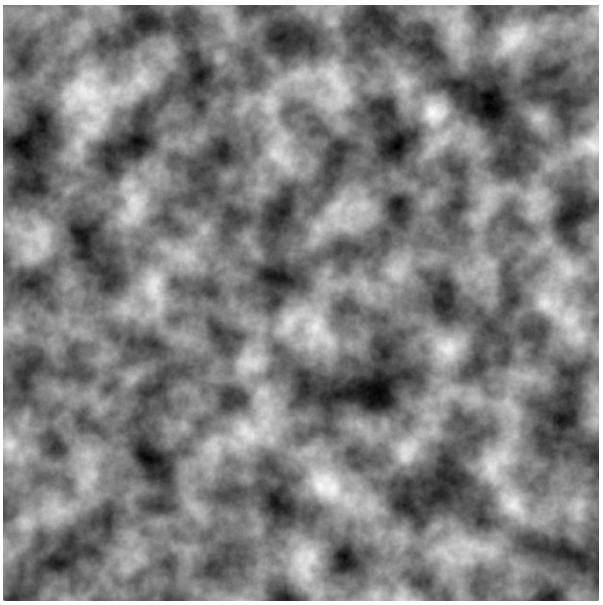


Figure 5. Simplex noise

In the project, multiple layers of perlin noise are used to create more interesting and controllable noise outputs based on parameters like number of layers (also known as octaves), lacunarity, persistence, noise offsets and a random seed to generate interesting noise outputs. This form of layering of noise patterns is known as Fractional Brownian Motion noise or 1/f noise. It is useful in many cases like creating procedurally generated clouds, fog, object density maps and as in this project, terrain. (Rose et al., 2016)[13]

Terrain Generation

The height map mentioned in the earlier section can be used as a reference to create vertices for every pixel value generated. A height map is a 2D texture that has a float value for each pixel in the range of 0 to 1. This can be used to generate a vertex mesh where the X and Y coordinates of the heightmap represent the X and Z coordinates of the vertex respectively, while the float value of each pixel represents the Y value of the

vertex. This generates relatively realistic looking terrain especially due to the layering of noise to create fBm noise. But it also leads to a uniformity in the noise to a certain extent.

One way to solve this issue of uniformity is to simulate erosion in the noise overtime using the GPU. This is done by simulating thousands of particles in the noise starting from the top and "flowing" down the noise from higher heights to lower heights. The terrain material at higher heights is "dissolved" and transported to lower positions in the noise map where it is "deposited". This when done over time creates very realistic erosion patterns that spread across the terrain generation. This form of noise generation can be very useful in creating many different types of erosion as well, depending on the flow rate and deposition of the erosion particles. (Olsen, 2004)[14]

The problem with this approach is that even with multithreaded GPU processing instead of using CPUs, each large scale mesh of terrain will require 3-5 extra seconds for simulating the erosion particles. This leads to a slow down in the generation algorithm and also causes significant issues for the purpose of this project as we need the terrain to be generated multiple times as quickly as possible for machine learning to understand and process the output.

Hence, an alternative approach suggested in a video tutorial series by Sebastian Lague (2016)[15] is used. In this approach, a Unity AnimationCurve(simply a mathematical curve function defined in 2 dimensions), is used and multiplied with the height value at each point of the animation curve. This allows us to create a terrain where larger heights are a lot steeper and lower heights are flatter and more rounded. This creates a good enough simulation of how erosion would function even if it does not have the "fractal" nature of true erosion generation algorithms. This also creates variance in the terrain to allow certain parts of it to be impassable due to its steepness. For more variance of this kind, the curve is bumpy in nature so that layers of plateaus and slopes are created using the noise function.

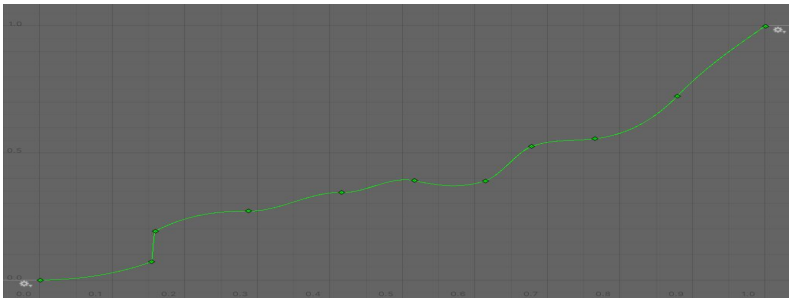


Figure 6. Animation curve for creating plateaus and peaks

Poisson Disc Sampling

Scattering game objects across a space uniformly is similar to a problem in computer science known as sampling, where the aim is to sample a random set of points in a given space. If the aim is to create points that are a certain amount of distance from each other and fill the plane with the points in the quickest way possible, Blue noise sample patterns are used.

“Blue noise sample patterns—for example produced by Poisson disk distributions, where all samples are at least distance r apart for some user-supplied density parameter r—are generally considered ideal for many applications in rendering”. (Yang et al., 2015)[3]

Most blue noise sampling algorithms were used to gather a set of points for ray tracing purposes or for sampling points for example in an MRI scan. This can then be used to generate points in a 2D or 3D euclidean surface. The example used within this project is scattering trees across the generated terrain where the terrain is our 3D euclidean surface.

There have been many refinements in efficient blue noise sampling, one key and often used approach described in (Dunbar et al., 2016)[16] where they show that breaking up the sample space into a grid of tiles or “scalloped region” based on the minimum sampling distance r. This reduced computation and greatly improved the speed of a poisson disc sampler as the sampler can definitively say that if a point is within a grid at x and y coordinates, it needs to only be worried about the surrounding 8 grid cells for a point to be less than the minimum distance.

How ML can help PCG

Machine learning is a very useful tool when it comes to controlling data sets that are inherently spread out and require a large number of sampling. This allows for the ML agent to be able to gather a lot of varied input for the neural network, thus increasing its efficiency. This also adds a larger amount of time required for training the agent as many procedural generation systems are inherently random or pseudo random. Creating a generator that can train itself on these random and pseudo random inputs is a potentially very useful tool. There is a name given to this method and it is called PCGML (Procedural Content Generation via Machine Learning).

“The difference to search-based and solver-based, PCG is that while the latter approaches might use machine-learned models (e.g. trained neural networks) for content evaluation, the content generation happens through search in content space; in PGCML, the content is generated directly from the model. By this we mean that the

output of a machine-learned model (given inputs that are either drawn from a random distribution or that represent partial or previous game content) is itself interpreted as content, which is not the case in search-based PCG." (Summerville et al., 2016)[17]

PCGML has been used to not only create new levels but also to create game content itself, from interactive fiction games, card games as well as some puzzle based games. There has been a lot of research into creating PCGML levels for Super Mario Brothers, some focusing on designing black box systems, some that are focused on player expression (Summerville J et al., 2016)[18] and others that are focused on how individual designers create levels (Guzdial et al., 2018)[19]. There has also been significant exploration in interactive fiction to create expert systems that generate story data from crowdsourced material (Guzdial et al., 2015)[20] and an NLP AI generator created by OpenAi that are used to create articles from online data sources from scratch (Radford et al., 2018)[21]. (Raffe, 2014)[22] has an expansive list of how procedural generation has been used in conjugation with many learning algorithms including genetic algorithms, Adversarial Neural Networks(ANNs) and Generative Adversarial Networks(GANs). It also goes into detail about how procedural generation can be helped by complex learning algorithms. Including showcasing an Evolutionary Terrain Tool that uses Genetic Algorithms to define parameters for noise sampling and terrain generation. (Volz et al., 2020)[23] talks about how machine learning can be used to understand local and global patterns in a level output and although there was a limit in the sample size, it showed how there were ways Machine Learning algorithms like GANs were able to detect some patterns like vertical symmetry in generated levels for Candy Crush Saga.

This key difference means that the ML algorithm learns from random distribution, patterns that output a good implementation for the procedural generation. Unlike rule based procedural generation which in many ways creates relatively predictable and not very varied outputs, players can notice patterns in the generation after multiple playthroughs, the hope is that PCGML will allow for more varied outputs that offer higher replayability along with the potential for emergent behaviour to create added gameplay opportunities not thought of earlier.

4.2 Unity

Unity is a real-time 3D development platform that consists of a rendering and physics engine as well as a graphical user interface called the Unity Editor. Unity has received adoption in Games, Architecture, Engineering and Construction, auto and film industries and is used by creators to create a wide variety of interactive simulations. Unity has always focused on being able to create many different genres of games, from simple 2D puzzle games to complex shooter and strategy games as well. Recently many new

tools have been introduced in Unity of particular importance to this project. These tools are listed below.

Shader Graph

Shader Graph is a useful tool to make shaders with visual representations. Nodes can be used to define shader behaviour by vertex or by pixel. In the case of this project, a shader graph is used to create terrain effects including slope based and height based texturing as well as another for creating effects for the water in the project. (Unity Technologies, 2020)[43]

ML Agents Toolkit

"Modern game engines are powerful tools for the simulation of visually realistic worlds with sophisticated physics and complex interactions between agents with varying capacities. Additionally, engines designed for game development provide user interfaces which are specifically engineered to be intuitive, easy to use, interactive, and available across many platforms." (Juliani et al., 2020)[24]

The Unity ML-Agents Toolkit is an open source project which enables researchers and developers to create simulated environments using the Unity Editor and interact with them via a Python API. It allows the creation of "Agents" that are trained to use and can be used to train a neural network "Brain" based on observations of its environment and by taking decisions and actions that lead it to a more favourable outcome(a better reward signal). Its usage of continuous vector controls have been used in a variety of applications including training a humanoid ragdoll simulation to understand how to use it's joints to move across a layout of levels. (Booth et al., 2019)[6]

Nav Mesh Agents

Nav Meshes are navigation meshes created in a 3D space using NavMesh Surfaces, Obstacles and Agents that traverse this space. Nav meshes can be used to create paths for agents usually using an A* algorithm. Usually NavMeshes are baked beforehand for a level but there are ways to create NavMesh Surfaces in real time using NavMesh Modifiers even though this can be computationally expensive. (Unity Technologies, 2020)[44]

4.3 Machine Learning

Artificial Intelligence and video games have been co dependent for a very long time. AI had been used in games from humble beginnings like Rule Based Systems to Finite State Machines and currently Neural Network agents. AI research as well has been dependent on Game Engines to simulate complex environments effectively.

This started with Arcade Learning Environment(ALE) which was used to simulate Atari 2600 games. It was used for domain independent AI technology and reinforcement learning simulations in arcade games and is responsible for much of the resurgence of interest in Machine Learning. (Bellemare et al., 2013)[25]

DeepMind Lab was built using the Quake III engine that was released to improve the DeepMind learning environment. It allowed for simulating bots in a 3D learning environment which allowed them to be trained for animal psychology as well as application based robot simulators. Since the bots are being trained according to the environment that is created for them, it was a domain specific environment. (Beattie et al., 2016)[26]

Project Malmo was a machine learning tool created using MineCraft. It was used for a long time in a wide variety of AI research as a domain-specific tool to create agents. But it faced certain limitations due to the minecraft engine itself due to it's simplistic physics simulations and low polygon, pixelated visuals, it was not an ideal tool for real world simulations or for good looking terrain. (Johnson et al., 2016)[27]

VizDoom was a visual learning platform based on the game DOOM. It was used to create a number of impressive approaches to Deep Reinforcement Learning like using curriculum based learning, novel algorithm design and memory systems for recurrent neural networks(RNNs). Because it was built on the DOOM engine it was limited to a first person perspective and had limitations due to it being a relatively old game as well. (Kempka et al., 2016)[28]

This brings us to the Unity ML Agents toolkit mentioned in the previous section. Due to Unity's general purpose approach to game design it can be used to train a wide variety of Agents playing 2D or 3D games as well. Using Unity's asset based structure like Scenes to define learning environments, it allows many different implementations and has now been used in many research projects for allowing training in simulated game environments as well as uses in real world simulations as well.

Training Algorithms provided by ML Agents

Unity ML Agents toolkit provides a wide variety of training algorithms for general purpose learning.

1. PPO

Proximal Policy Optimization, a family of policy optimization methods that use multiple epochs of stochastic gradient ascent to perform each policy update. It has some of the benefits of Trust Region Policy Optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). (Schulman et al., 2017)[29]

PPO has been showcased to be able to learn over multiple iterations and is actually able to consistently showcase higher reward signals than SAC in Unity's example learning environments as showcased in the comparisons in Unity's ML agents paper. (Juliani et al., 2020)[24]

2. SAC
   Soft Actor Critic(SAC) is an off policy maximum entropy actor-critic algorithm, which provides for both sample efficient learning and stability. This algorithm extends readily to very complex, high-dimensional tasks. The introductory paper (Haarnoja et al., 2019)[30] mentions how much more efficient (or at least faster) SAC is when it comes to complex tasks like humanoid walkers, with the same being showcased in Unity Example Environments for Walker and Crawler(Juliani et al., 2020)[24]. Though SAC is faster than PPO, it does have the side effect of "plateauing" while learning as it looks at the entire sample space of training rather than focusing on epochs like PPO.

3. GAIL
   Generative Adversarial Imitation Learning(GAIL) is a method where demonstrations of expert play are used to extract a policy from a given dataset. A certain instantiation of the framework draws an analogy between imitation learning and generative adversarial networks, from which we derive a model-free imitation learning algorithm that obtains significant performance gains over existing model-free methods in imitating complex behaviors in large, high-dimensional environments (Ho et al., 2016)[31]. It is mentioned that it has limitations when the actions to be taken are more complex and requires a lot of environment interaction.

4. Behavioural Cloning
   Behavioural cloning is the process of reconstructing a skill from an operator's behavioural traces by means of Machine Learning techniques. It uses an input dataset of demonstrations that are cloned over multiple iterations with slight variances, allowing a policy to emerge from it. This is usually used in conjugation with GAIL by creating more datasets for it. (Hussein et al., 2017)[32]

5. Self Play
   Self Play is pitting multiple agents against each other and using what each agent learns to train a singular neural network. It is particularly effective when it comes to complex games that are being played between two different agents. Self Play has been showcased by training 2 similar agents in this paper to learn many tasks against each other. (Bansal et al., 2018)[33]

Curiosity in ML

Many times when training an agent, tasks can have extremely sparse rewards. With respect to the current project, the reward is quite sparse because there are relatively few positive outcomes in the terrain generation algorithm. This problem of sparse rewards is an active area of research in many Machine Learning papers. For example, Hindsight Experience Replay wherein a non-constant negative reward is suggested to shape the learning experience closer towards solutions so that the agent approaches faster towards the goal of it's training. (Andrychowicz et al., 2018)[34]

Another approach that is also provided with the ML agents toolkit is an Intrinsic Curiosity Module. This was proposed as a way for agents to navigate environments with no or very little rewards (Pathak et al., 2017)[35]. This allowed an agent to train using exploration of possible spaces or states that it can visit. This means the agent is trained to be "curious" and has an intrinsic reward for exploring multiple states and inputs in its environment including the extrinsic reward of the agent's behaviour. This is particularly useful in our case as we want the trained agent to be able to handle many different states of the terrain generation algorithm and understand how to act best on these states.

Curriculum Learning

Humans as well as animals are able to learn a lot quicker when concepts are incrementally introduced to them, both in terms of number of concepts as well as complexity of those concepts. The splitting of topics into subjects and increasing difficulty is known as a curriculum in schools as well as institutions. (Bengio et al., 2009)[36] introduces the concept of curriculum learning for machine learning agents and puts forth some ways that curriculum learning will be useful for training an agent.

Though not always an improvement on traditional learning strategies there have been many cases where curriculum not only helped the learning process but also led to emergent behaviour. Starting off from relatively simpler tasks and increasing

complexity is also useful in handling sparse reward as the agent can be rewarded for easy tasks more frequently.

Self play has also led to curriculum like behaviour, it was particularly showcased in the hide-and-seek self play demonstration(Baker et al., 2020)[7] where both agents went through stages of different strategies that forced each other to adapt. This phenomenon in self play is known as *autocurricula* (Leibo et al., 2019)[37] and it has been observed in agents playing Go as well as Dota. Observing agents usually using this form of generated arms race when training suggests that it is an effective tool for teaching machine learning agents how to train if used correctly. The paper will also discuss the effectiveness of different strategies for curriculum learning and how it helps tackle the problem at hand.

METHODOLOGY

5.1 Creating Procedurally Generated Terrain

The approach used in this project to create procedurally generated terrain was using fractional Brownian motion noise to create a heightmap. This heightmap is used to generate a vertex mesh for each terrain chunk. An animation curve is applied along with this generation to amplify the heights and slopes of certain sectors to create mountains as well as plateaus in the generated terrain mesh. A Level Of Detail system is applied along with ways to merge the chunks together between different levels of detail. A more in depth discussion of each of the components follows:

1. **Using fractional Brownian Motion Noise to create a Height Map**
   The fBm noise is generated in the Noise.cs script using the GenerateNoiseMap() function using multiple octaves of perlin noise. This function uses parameters like the width and height of the noisemap, the NoiseSettings class to define properties about the noise generated and the sample center of the current chunk being generated.

   The Noise.cs script also contains the NoiseSettings class being passed to the GenerateNoiseMap() function which has the properties:
   1. Normalize Mode: Whether the generated height values on the noise map should be scaled to just the current chunk or a global parameter using the noise estimator variable
   2. Noise Estimator Variable: This is used in the globally normalized noise generation to calculate what the maximum possible heights for the terrain chunks could be.
   3. Scale: The scale of the noise map being generated, a larger value will create larger features using the noise.
   4. Octaves: The number of octaves or layers that need to be used in the noise generation.
   5. Persistence: The variance of amplitude across the different octaves or how much lesser the layers affect the final heightmap as they are progressively generated
   6. Lacunarity: The variance of frequency across the different octaves or how the noise scales to smaller values as the layers are progressively generated.
   7. Seed: The random seed being used to generate the current noise map using a pseudo random number generator to pick different locations of the noise map.

8. Offset: A Vector2 value defining a user defined offset to move the noise in the horizontal and vertical direction.

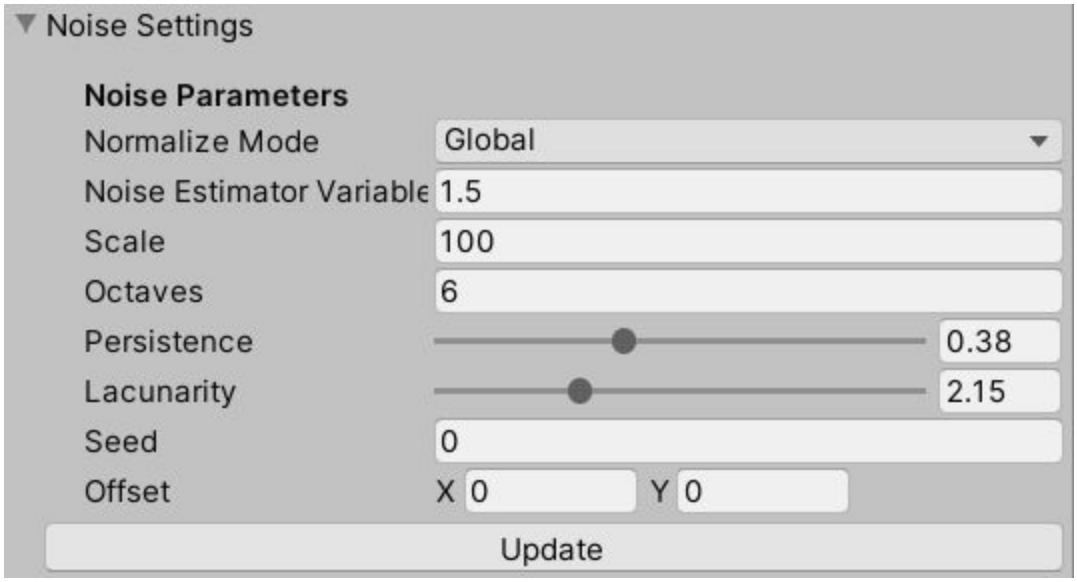


Figure 7. Noise settings scriptable object

This Noise Map that is generated is then used in the HeightMap.cs script. The GenerateHeightMap() function combines the noise map with a height curve to simulate erosion and a falloff map if we want it to generate terrain that has a gradual slope on it, for example to create patches of islands or lakes for each chunk. The properties for this function are stored in HeightMapSettings:

1. Height Multiplier: This defines the maximum height of the generated features.
2. Height Curve: The animation curve that modifies the height according to a slope value that can be used to add steeper mountains and plateaus to simulate erosion.
3. Use FallOff: A boolean value that says if the fallOff curve should be used to create a gradual slope in the overall generation.
4. FallOff Curve: An animation curve that is used to define an overall gradual slope in the generated terrain.
5. Noise Settings: The Noise Settings for this height map generator.

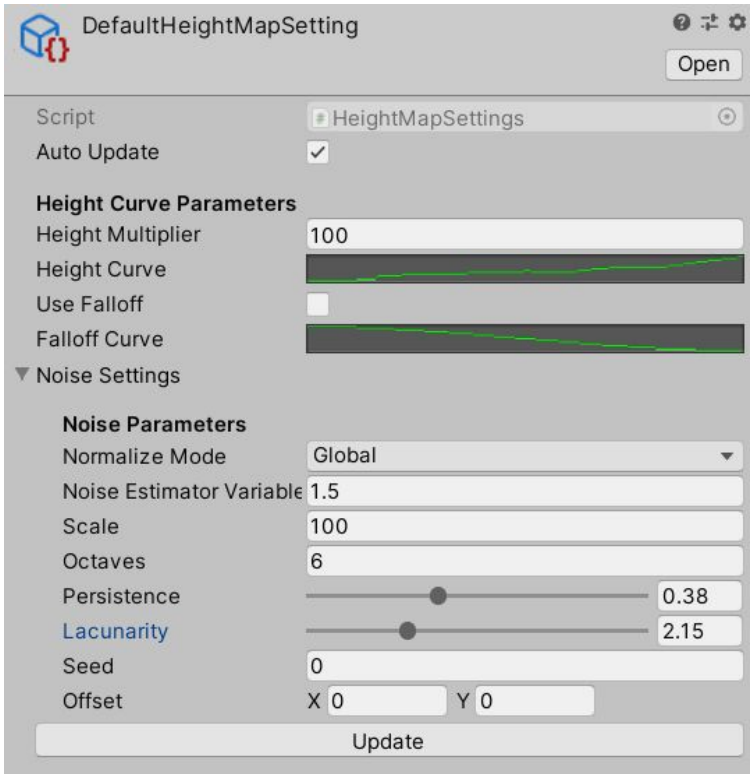


Figure 8. Height Map Settings object

2. Mesh generation and normal calculation

The MeshGenerator.cs script is used to generate a mesh for each TerrainChunk.cs class. These terrain chunks are arranged according to a Level of Detail(LOD) system using the TerrainGenerator class which is used to spawn infinite terrain and also update the LOD on each terrain chunk. This script is also used for calculating the normals (in unity, the orientation/normal of each vertex, not the triangle) for the generated mesh.

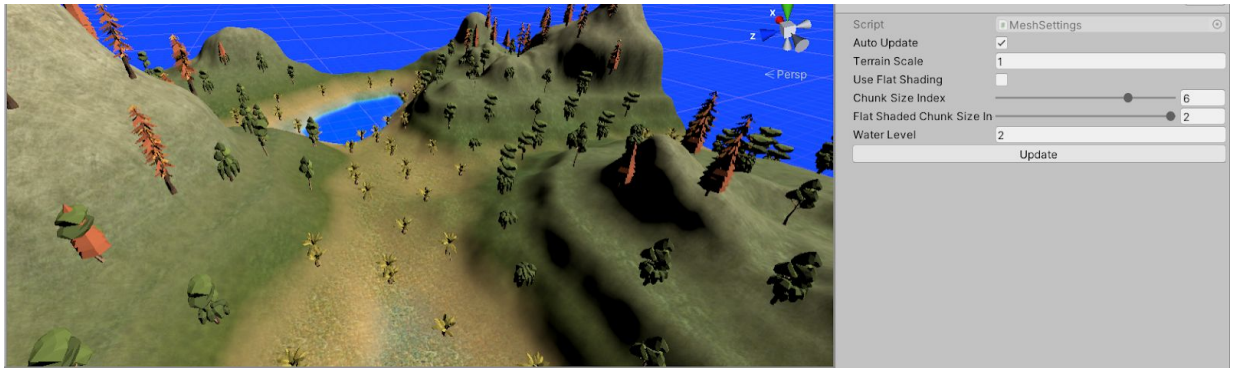


Figure 9. Generated sample terrain mesh

Flat shading can also be used to generate terrain that has more defined and blocky shading for each triangle. The problem with generating this kind of shading in unity is that since the normal data is defined per vertex, vertexes need to be duplicated 3 times for each connecting triangle. Unity has a limit on the number of vertices that can be generated for a mesh at 65535 for 16 bit mesh index format, which is supported by platforms like Android. Even though there is a 32 bit mesh index format that supports 4 billion meshes, this constraint was taken into account. Hence the possible size of each terrain chunk is limited to 96x96 for flat shaded chunks and 240x240 for regular shaded chunks.This also provides a slight boost in processing each mesh as well.

Figure 10. Flat shaded terrain mesh

The mesh generation including requesting for the heightmap, creating terrain chunks, baking normals and creating a collider are multi-threaded so that the endless terrain is generated very quickly and also so that it doesn't block the regular movement of the player in the environment when it is creating a mesh.

The MeshGenerator.cs script uses the MeshSettings.cs class to define what kind of meshes are generated for the terrain chunk, an overview of these follow:

1. Terrain Scale: The transform scale of the terrain chunks being generated
2. Use Flat Shading: If flats shading should be used
3. Chunk Size Index: The current size of regular chunks
4. Flat Shaded Chunk Size Index: The current size of flat shaded chunks
5. Water Level: The height at which water for the generated terrain is placed

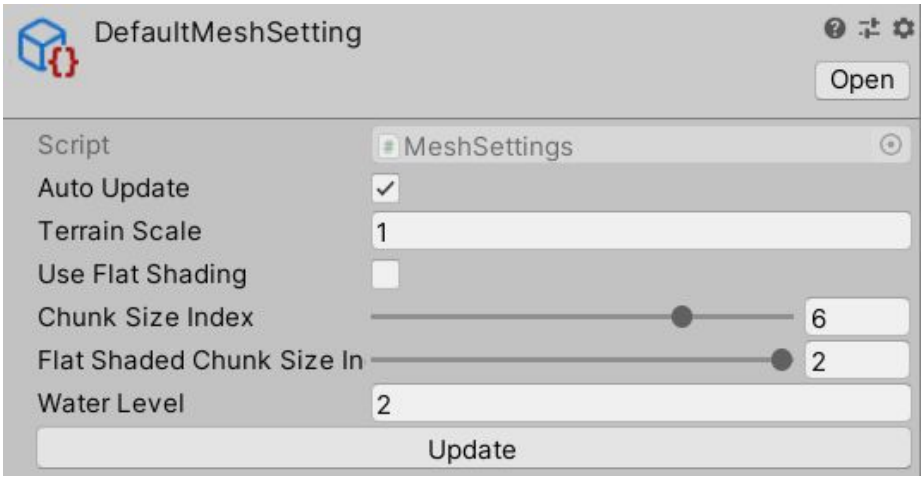


Figure 11. Mesh settings scriptable object

3. Collision Detection

Collision detection for the mesh is done by baking a Mesh Collider for each terrain chunk when the player is reasonably close to the chunk. This ensures that no unnecessary chunk colliders are generated.

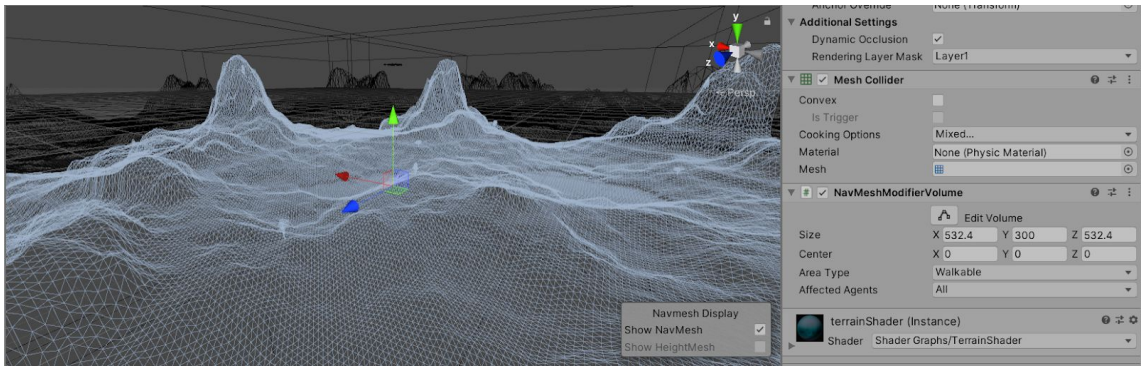


Figure 12. Generated Collision Mesh

Raycasts are used to create trees for each terrain chunk as well and can only be done when the collider for the terrain chunk has been generated. Raycasts are also used for non third person control, intended to be used for the object generator to place objectives on the map.

For non-third person control(when using a Nav Mesh Agent to traverse the level) a nav mesh modifier is used to also create the Nav Mesh Surface for the world when a collider for the Terrain Chunk is generated. This effectively ensures that the mesh surface is generated only when necessary for traversal and when the player is relatively close to a chunk to be able to navigate it.

4. Implementing Level of Detail

A level of detail system is used to render chunks that are farther away with a lower resolution so that it is less processor intensive to generate them. These chunks skip 1, 2, 3 or 4 vertices when generating the mesh. This also means that the overall vertex size of all terrain chunk meshes need to be a multiple or 2,3 and 4.

Since the LOD of the meshes mean that meshes are slightly different when it comes to the boundary between two different mesh chunks, a padding of width 1 vertex was used to ensure that meshes stitch together properly and do not cause artifacts that show gaps in the terrain generation.

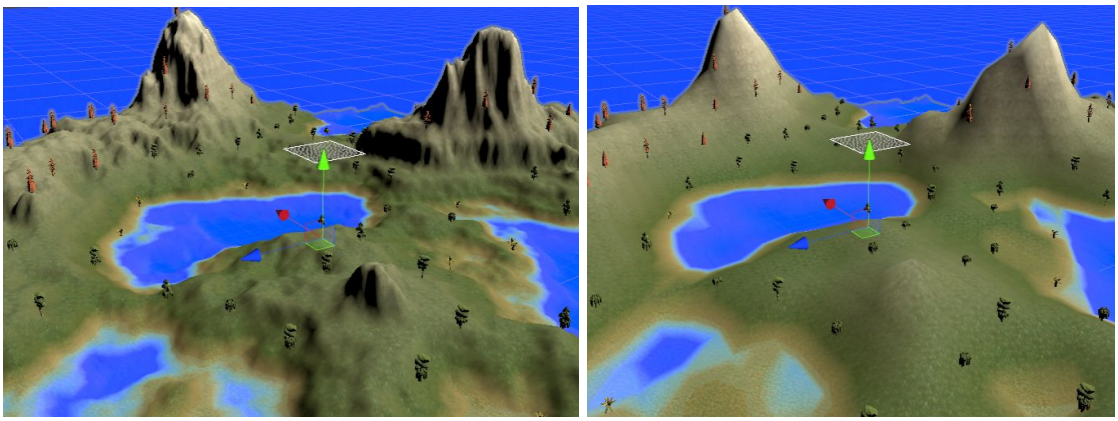


Figure 13. High and low LOD generation for the same mesh

5. Poisson Distribution

The PoissonDiscSampling.cs script is used to generate a set of 2D points that are a particular distance r away from each other. It uses the approach mentioned in (Dunbar et al., 2016)[16] of creating a grid of cells with width r that reduces the number of points to be checked limited to only the 9 cells surrounding the point.

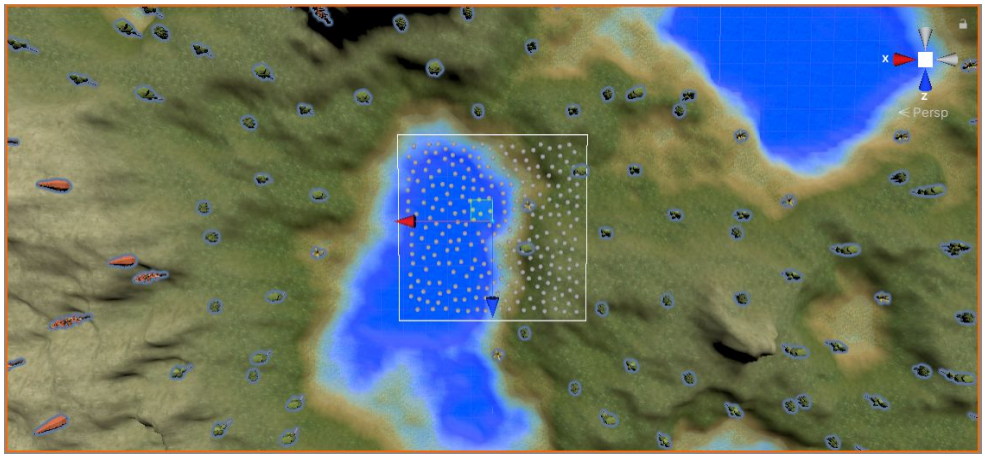


Figure 14. Poisson distribution of trees on a terrain mesh with a preview

The points returned by the PoissonDiscSampling are used by the ObjectCreator.cs script to raycast above a terrain chunk and create trees based on the height that the ray intersects the terrain. This is done for every terrain chunk and the points returned by the poisson disc sampling are stored for each chunk so that they can be recreated when the player revisits an earlier chunk. The parameters for it are:

1. Radius: The minimum distance between each point
2. Region Size: The space size for the generation
3. Rejection Samples: How many times it should retry creating points
4. Fill Percent: What percentage of overall space should be filled
5. Offset: The XZ offset for the distribution
6. Tree Settings: This controls the number of trees to be stored in a pool, The minimum tree height for each type of tree and the prefabs associated with each tree type.

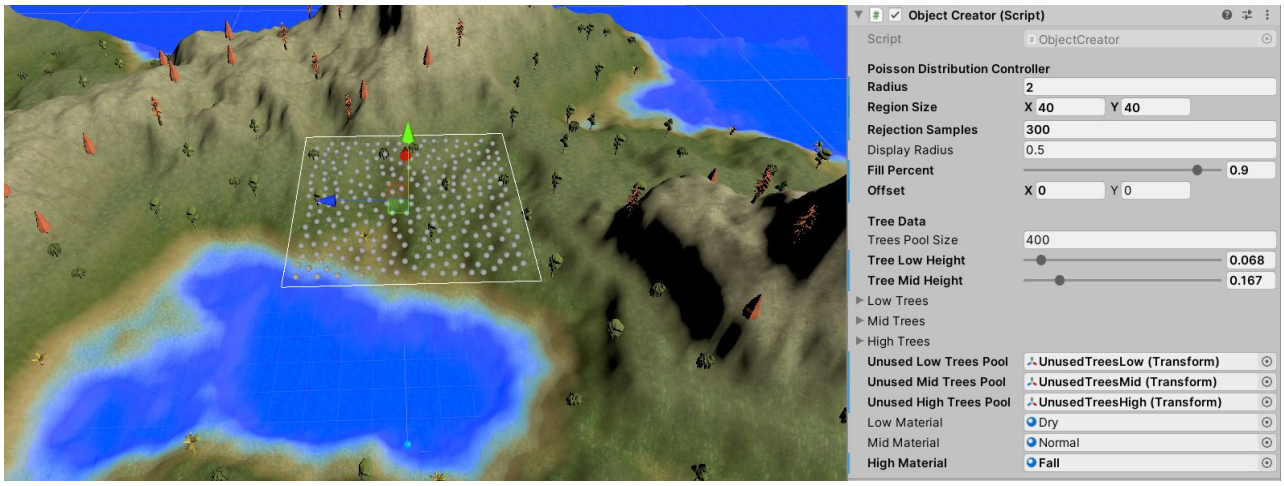


Figure 15. Object creator script showing Poisson distributor settings

6. Unity Shader Graph

Unity Shader Graph is a powerful tool to create shaders for meshes using vertex and pixel shaders. They have been used to generate textures for the terrain as well as generate vertex shaders for moving water for the game. It allows a user to also define customized parameters for the shader that can have different values, from float values to textures to be rendered.

a. Height Based Terrain Texturing

Height based texturing is a relatively simple approach wherein a texture is selected based on the height of the terrain vertex being rendered.

Crossfading between textures is also done so that the terrain looks like the textures are merging together.

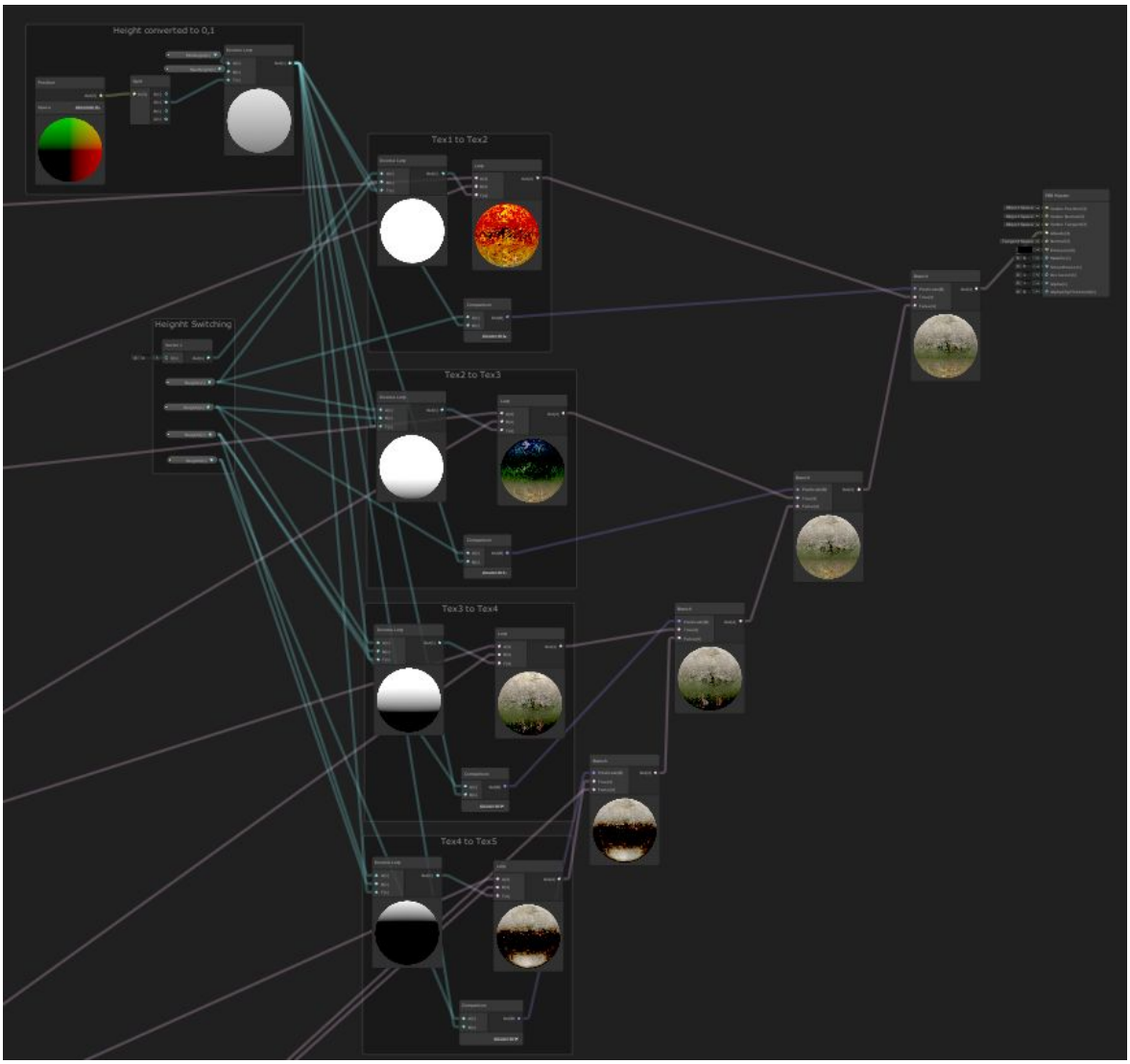


Figure 16. Shader Graph height based terrain shader

b. Slope Based Terrain Texturing
Slope based terrain is accomplished by looking at the orientation of the normal values of the mesh. In the case of our shader graph, a slope is seen by checking the normal Y value and the texture selected for height based texturing is used based on this value.

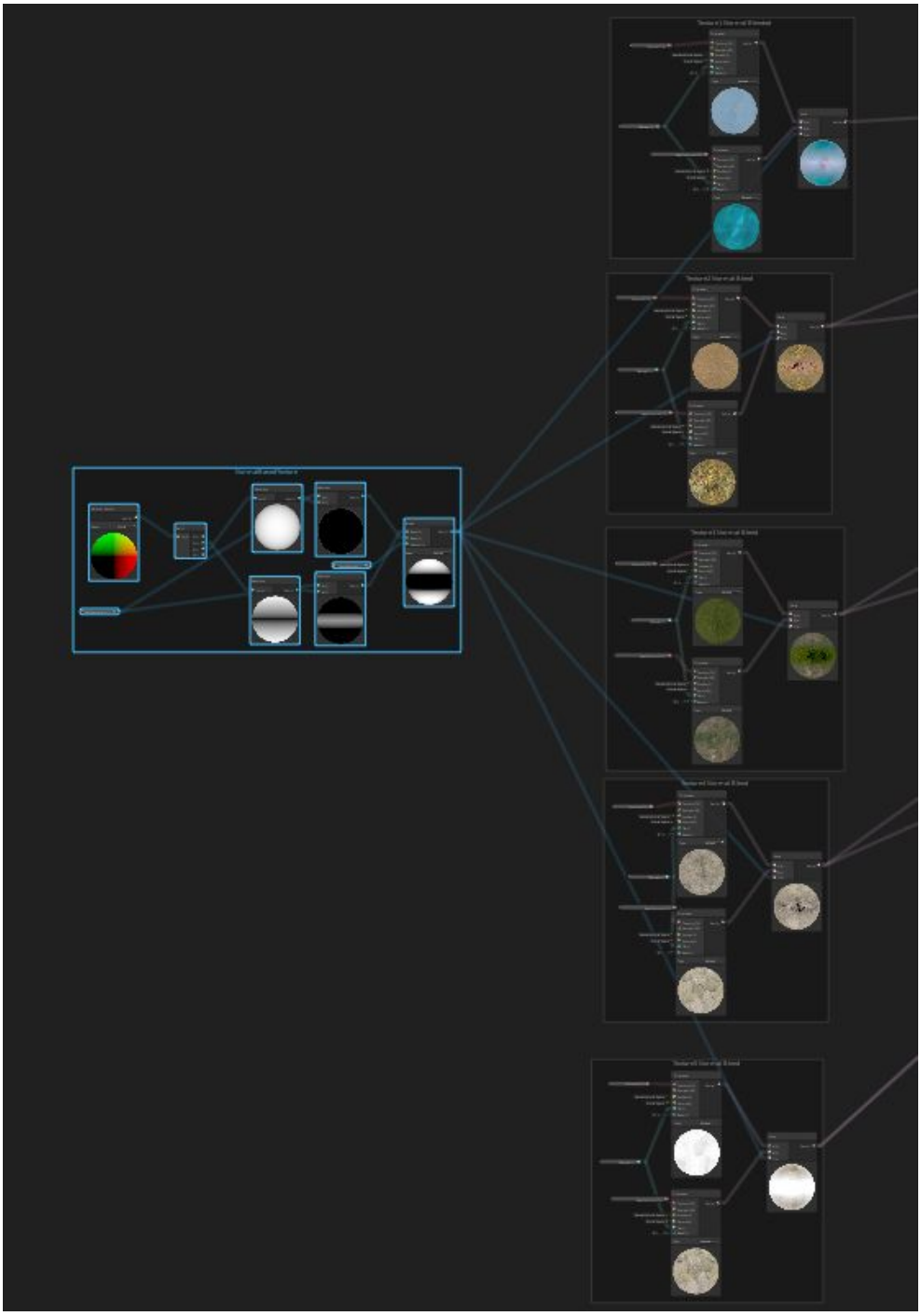


Figure 17. Shader Graph slope based terrain shader

c. Water Shading

Water is also implemented in the game using a depth buffer that allows us to see how far away a vertex is from the camera. This allows to implement fog like characteristics and simulate water depth using a gradient. Intersecting normal maps are used moving in different directions

to simulate moving waves intersecting with each other. A vertex shader is used to simulate tides with crest and troughs for the water as well.

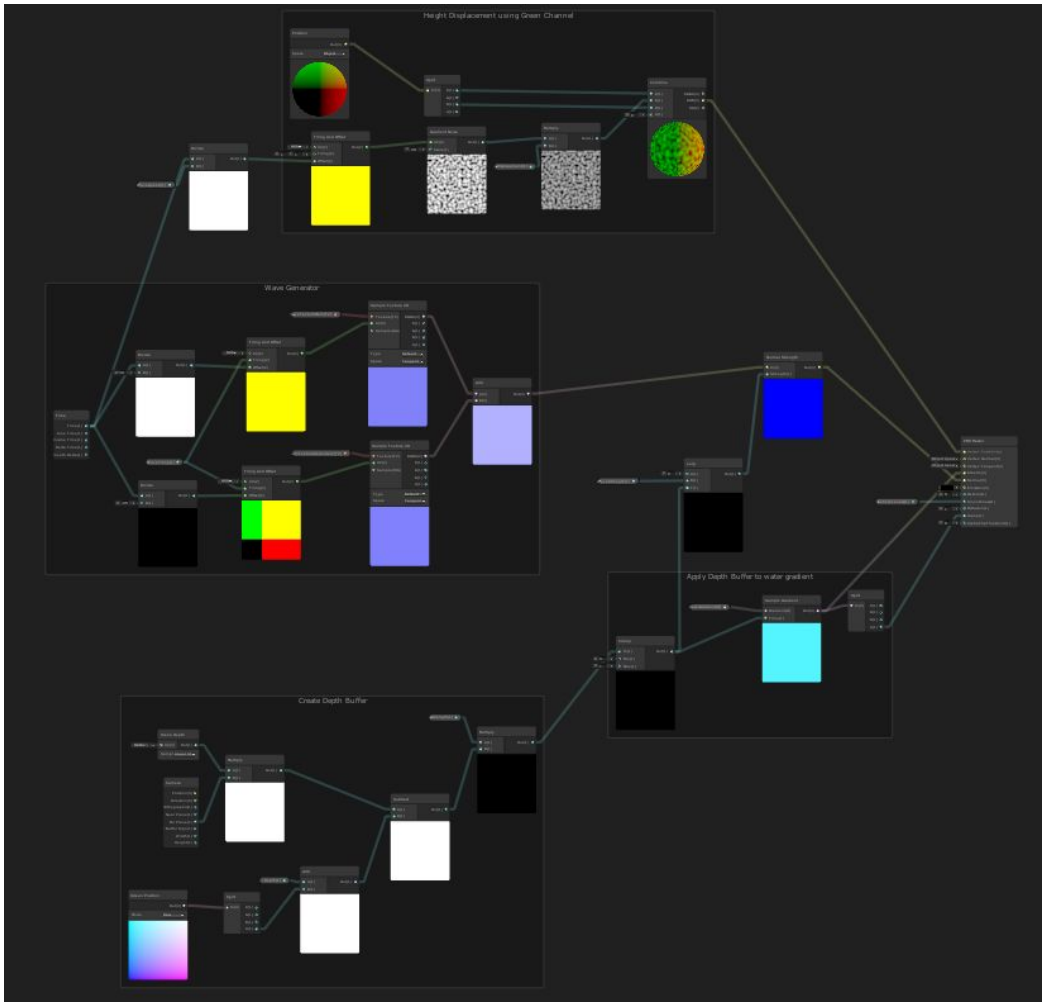


Figure 18. Shader graph water shader

7. Unity Scriptable Objects

Unity scriptable objects are classes that can be stored as a .asset file. Scriptable Objects don't need to be attached to a GameObject in a scene. Most often, they are used as assets which are only meant to store data, but can also be used to help serialize objects and can be instantiated in scenes. In the case of this project, the settings mentioned in Part 1 and Part 2 of this section are stored as asset files that can be modified by our machine learning agent when generating multiple iterations of the terrain. Different iterations of these files can be generated for simulating GAIL as well.

a. HeightMap Settings
b. Mesh Settings

Figure 19. Example asset files

5.2 Navmesh Agents

The Unity NavMesh System is used to simulate navigational AI in unity. It uses Nav Mesh surfaces to define a traversable mesh that a Nav Mesh Agent can function on. Usually, nav meshes are baked into a level that has been created.

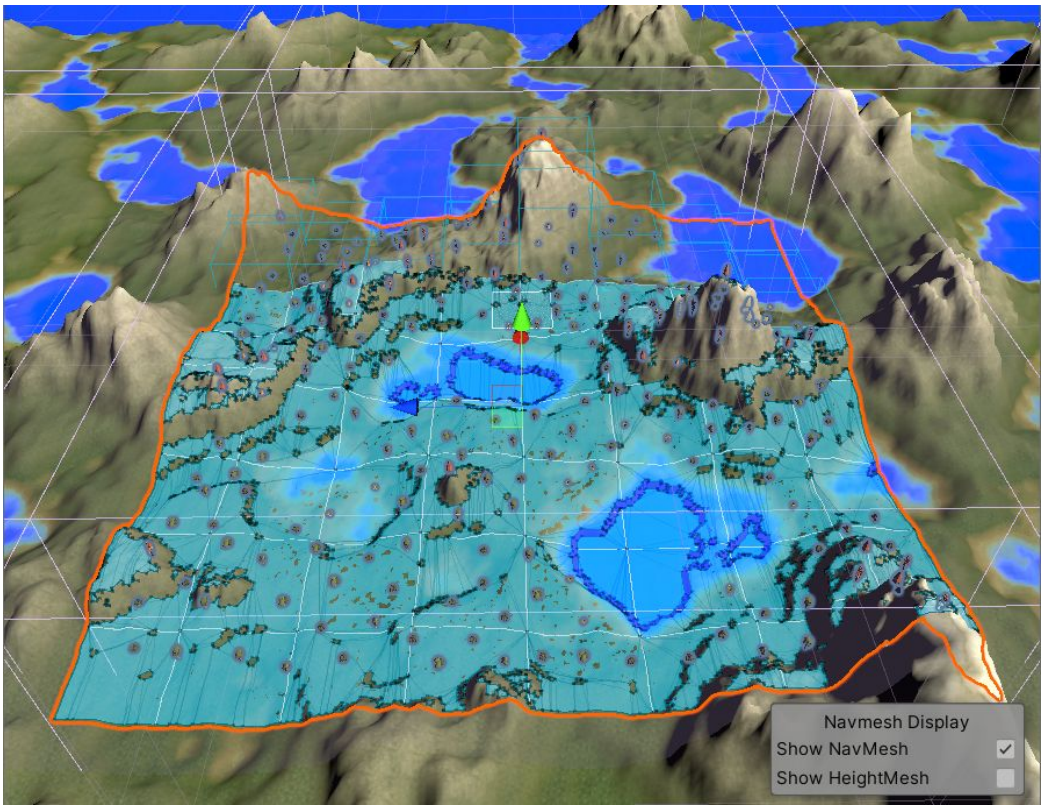


Figure 20. Runtime nav mesh surface generation

1. **Baking vs Generation of NavMeshes**
   Baking a nav mesh is usually done when designing a game in the editor because it is computationally expensive. But Unity allows runtime baking of a navmesh using the NavMeshSurface and NavMeshModifier classes.

2. **Nav Mesh Surface**
   NavMeshSurfaces define the walkable area for a NavMeshAgent. They can have different properties and weights assigned to them to make them harder to traverse. Whether or not a surface is traversable is defined by the slope of the surface and the width of the agent trying to navigate the nav mesh. These properties can be defined in the Navigation Panel in the Unity editor. For real time NavMesh baking, NavMeshModifiers can be created for child objects in the NavMeshSurface object. NavMeshSurfaces can then be baked everytime a new NavMeshModifier has been created.

3. **Nav Mesh Modifier**
   NavMeshModifiers are used for real time baking of NavMeshSurfaces. They can render based on the MeshRenderer of the object or the MeshCollider for the object. In the case of this project a single NavMeshSurface is defined for the TerrainGenerator object and every TerrainChunk has an associated NavMeshModifier. When the colliders for a terrain chunk are created a callback is sent to the TerrainGenerator to rebake it's NavMeshSurface which allows a NavMeshAgent to use the newly created mesh collider as part of it's NavMeshSurface.

5.3 ML Agents Toolkit

The ML agents toolkit uses the Agents, Observations made by the agent in an Environment and creates a neural network that can be trained to make decisions based on the Actions it makes in this Environment, using Observations it gathers and the rewards associated with the outcome of its decisions.

In the project, the ML agents toolkit is used to sample different settings of noise and mesh data to train a neural network that will be able to take actions that can make the noise or mesh data currently fed in better.

The GeneratorAgent script is what handles the machine learning behaviour for generating better terrain. It acts on the terrain generator class which contains the heightMapSettings and meshSettings for generating terrain.

1. **Integration of the terrain generator with ML Agents**
   a. Defining Ideal Terrain
      There needed to be some way to define good terrain. One approach that was decided is by measuring properties for the generated mesh and

comparing average values for each chunk that is generated. Some of the properties that were being measured are:

1. Percentage of land that is underwater: The water height in the mesh settings parameter could be used to increase or lower the water level of terrain chunks. Observing the percentage of land that is underwater would help in defining a good generation of land.
2. Normal values: The X, Y and Z normals for each vertex in the generated terrain mesh can be a good indication of how smooth or rough the terrain is. The normal values are added to a total when the normals are baked for mesh calculation. The vertices that are above the water level should be taken into account because those below the water will be flat and will be non navigable by the player.
3. Average slope: The average percentage of the terrain which is within a range of slope that will be traversable by the player as well as the nav mesh agent. This should also be calculated based on how much of it is above the water level.

Algorithm for calculating properties of the terrain
1. Wait for mesh generation 2. Define averageNormals, averageSlope, totalVertices and waterVertices as 0 3. Loop for each normal in mesh.normals a. Increment totalVertices b. If corresponding vertex is below water level i. Increment waterVertices c. Else i. Add x,y,z of normal value to averageNormals ii. If angle between normal and Vector3.Up < 45 1. Increment averageSlope 4. Divide averageNormals, averageSlope by (totalVertices - waterVertices) 5. Average water vertices = waterVertices/(totalVertices - waterVertices) 6. Send callback to increment total values in terrainGenerator

b. Approximating Generated Terrain

These values are calculated on a randomly generated number of terrain chunks. This is so that the noise settings should be a good fit on a large

number of terrain chunks being generated. The agent should be able to handle many different terrain chunks, being able to detect a pattern across a wide amount of generated terrain. A range for valid values is decided, which will then be used to guide the terrain generation agent. The farther the agent is from the average minimum and maximum value of the range, higher the negative reward for that particular generation.

2. **ML Environment**

Initially the steps that were being taken for the agent were being done for every fixed update since the ML Agents toolkit is designed for creating agents in a real time environment, taking into account physics and movement. This led to very noisy learning with the agent often getting confused based on how long the terrain took to be generated especially after many steps. It also had severe performance constraints as well.

Diving deeper into the ML agents documentation showed that they allow for user defined environment steps as well. This means the academy can be told when an environmental step should occur and take observations. It can also be told when it should make a decision and request an action from the brain. This allows for better performance as the minimum number of terrain can first be generated and then observations can be made whether or not the agent took the right decision. It also means that the agent gets more predictable observations and hence it can make much better decisions. (Unity Technologies, 2020)[38]

a. EpisodeBegin()
   This function is called everytime a new episode for training starts. Because we want the agent to handle a large number of different noise values and we want it to take action on them, everytime the academy resets or a new episode begins, the parameters that the agent controls are set to a random value between the minimum and maximum value possible for them. This ensured that the agent learned from a large number of possible sample spaces and it understood how to modify them.

b. RequestDecision()
   ML agents usually use the DecisionRequester class to ask the agent to make a decision. This class usually asks a decision from the agent every x times per fixed update. Because we want the agent to take a decision only after the terrain has been generated, the Agent.RequestDecsion() can be called after a minimum number of terrain chunks has been generated for it to decide when it should try to make a decision.

c. Inference
   Inference is when the trained neural network can be used to take actions in the environment. Currently there is no way to run inference in editor mode because the inference engine uses a compute shader that only runs in play mode. Hence the decision was taken to use inference in real time to modify .asset files by taking an environment step on every key press. This is useful to have a gradual number of steps taken based on how much designers want the ML agent to modify the input terrain.

d. EnvironmentStep()
   Academy.automaticStepping can be set to false so that we can decide when the agent should consider any observations it should make by explicitly calling EnvironmentStep() only after the minimum number of terrain chunks have been generated.

3. Iteration/Experiments of possible strategies

The project had many ways to assign a good reward to the outcome. There were also challenges in seeing how the agent handled randomized generation. Some of the experiments and challenges of the training algorithm are being discussed in the following sections:

a. Sparse Reward
   Sparse reward during machine learning is when there are very few solutions to the problem being stated. One of the problems when it comes to this is that the machine learning agent may never come across a solution, which means it will never have a positive cumulative reward. A lot of research suggests a non negative negative reward for each iteration to guide the machine learning agent towards a solution. This is known as reward shaping and is a widely used tool to solve the problem of sparse reward.

b. Negative Reward
   When a machine learning agent is trying to find out a solution to a problem, usually a positive reward is given when the agent accomplishes a solution. But a negative reward when a machine learning agent fails is also used to make it avoid situations that are not helpful to arrive at the solution. Shaped rewards are when this negative reward increases the more wrong a solution is. In the project's case a negative reward is assigned everytime the solution is not within the ideal values for the

parameters, a negative reward is awarded based on how far away the value is from the ideal midpoint of the ideal range.

c. Ending Episodes Early
Some machine learning experiments end an episode when the agent achieves its goal. This is usually with a high positive or negative reward at the end of the episode. It was attempted but having the episodes longer allowed for better learning, the machine learning agent in this case needed to make as many observations as possible. This also meant that the agent learned how to stabilize at a good value and understand how to stay at it rather than only attempting to reach a solution and ending the experience. Hence, ending episodes early was not useful.

d. PPO vs SAC
PPO and SAC were both used to see which was useful in training the agent. SAC is a lot faster at arriving at learning operations quickly but also has the issue of plateauing much earlier than PPO since it uses the entirety of the learning experience rather than epochs like PPO. This is challenging when it comes to curriculum learning as well because it does not have the adaptability that PPO has. But taking that into account PPO is very slow and requires a lot more hyperparameter tuning.

SAC took 9 minutes for every 500 steps while PPO took 15 minutes for the same. Because multi agent training was difficult due to the amount of hardware required to run multiple iterations of the terrain generator, SAC made a lot more progress in training than PPO.

Figure 21. Cumulative reward comparison between PPO(pink) and SAC(orange) with valid values check initially

e. Hyperparameters and Curiosity

Hyperparameter training is tuning values that control learning and is an important part of training models. (Unity Technologies, 2020)[39] provides a good overview of how hyperparameters should be calculated by using the tensorflow tool. Observations should be made on different training statistics and parameters should be modified accordingly.

A baseline template that was used is the walker and crawler environments provided in the Unity example environments. This is because they have a similar issue of having many continuous inputs and very sparse rewards.

In particular, batch size and time horizon had a significant effect on learning behaviour. Learning rate was closely related to the number of steps and actions an agent was able to take to start arriving at a good outcome, e.g early ending of episodes and when the number of steps the agent can take to arrive at a good outcome were increased. Due to the large number of continuous actions and relatively sparse reward a relatively large number of hidden units as well as layers in the neural network made sense. A list of tensorflow graphs are shown below showing comparisons of different hyperparameters and how they affected training.

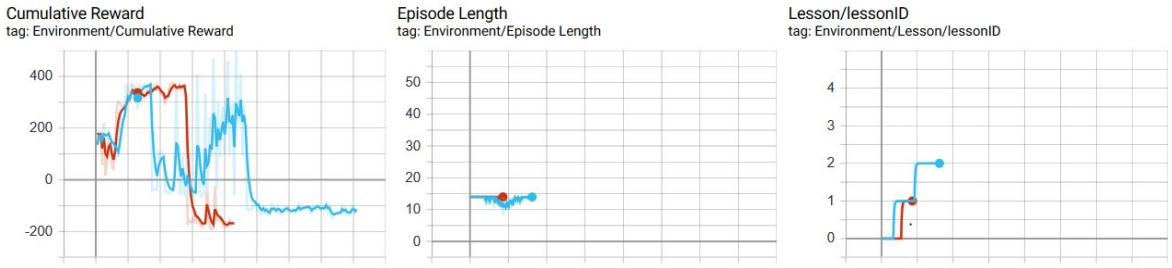


Figure 22. Cumulative reward between batch size 8 (red) and batch size 32 (blue). Batch size 32 was able to go to the second task curriculum as well

Curiosity in training was especially useful due to the sparse reward of the experiment as mentioned earlier and the ICM was used to make the trainer explore different values and arrive at a reasoning that helped it become better at finding solutions as well as find varied solutions to different initial conditions.

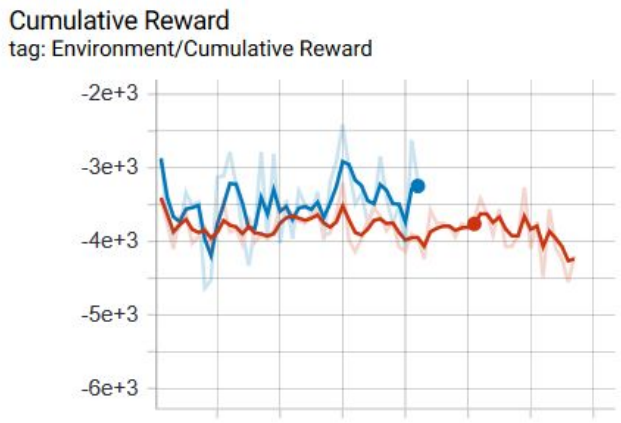


Figure 23. Initial configuration showing no training due to sparse reward

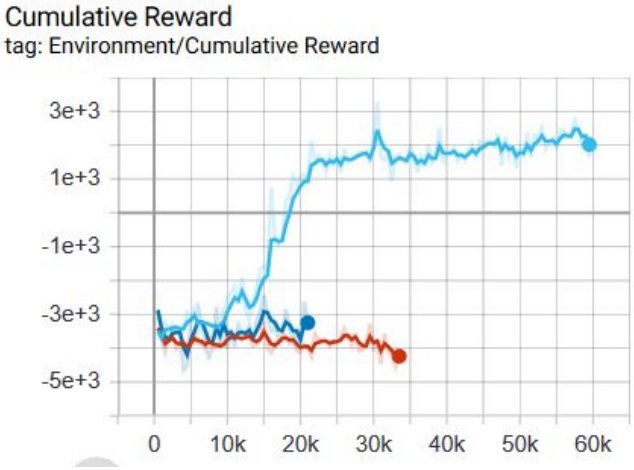


Figure 24. Significant increase in cumulative reward when using curiosity (light blue)

f. Curriculum Learning

Curriculum learning has been suggested by many research papers as a solution to sparse reward. Devising a strategy for good curricula is an important step here and one of the more difficult parts of the training process. (Hacohen et al., 2019)[45] goes into a list of possible strategies, some suggesting increasing task difficulty while others suggest broadening the scope of the problem as the curriculum proceeds.

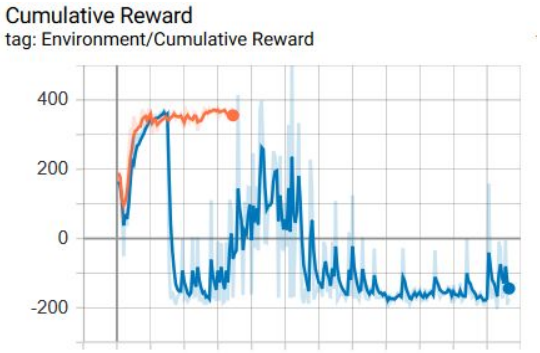


Figure 25. Curriculum training varies rewards significantly as the curriculum progresses (blue) compared to non-curriculum training(orange)

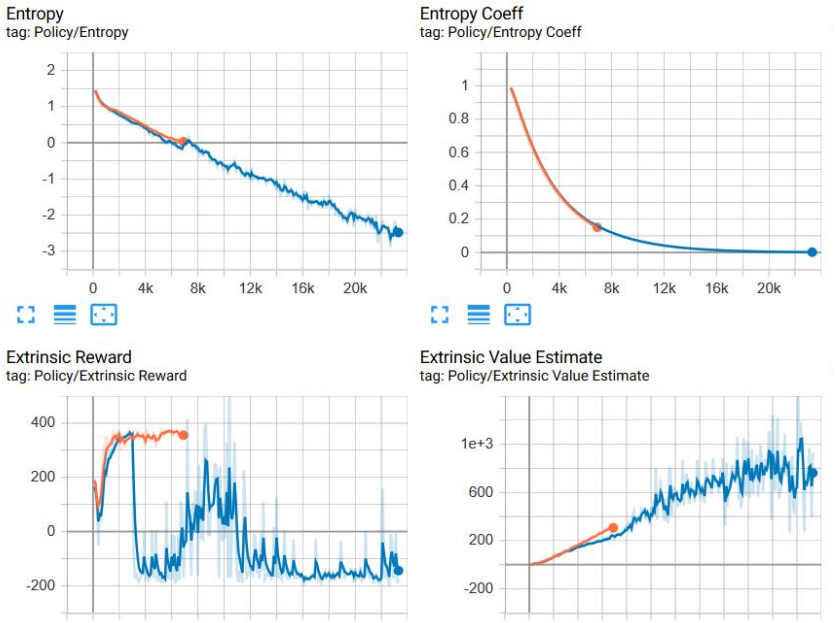


Figure 26. A better comparison in that case is the policy entropy(how many random actions are taken) and the policy's extrinsic value estimate

A "lesson ID" variable is used in the hyperparameters that progresses according to the curricula. They have been showcased here:

i. Modifying ideal terrain constraints
   Modifying the range of ideal constraints was one curriculum strategy that was looked into. Using larger ranges for a good scenario and reducing these ranges closer to the desired outcome as the curriculum progressed. This proved difficult for SAC to adapt to due to the plateauing of learning rate. This resulted in slightly predictable terrain generation as the agent would always try to

default to one way of solving the issue and would never move forward from there. Adding a slight increase in reward for when the agent was able to generate “actual” ideal terrain in spite of the constraints being loosened improved the efficiency of the agent as well. The repeatability can partly be attributed to curiosity from the ICM falling very quickly in the experiments and would prevent the agent from learning new behaviours. Further experiments where the curiosity decay is slower can lead to better results in this case.

ii. Modifying input to noise parameters

Modifying the possible values that the agent can take was also looked into as part of curriculum training. Starting off with much smaller inputs and gradually increasing the range of possible inputs as the curriculum progressed. This led to even more predictable results as compared to modifying output parameters because the agent would always default to the initial input ranges because it knew those lead to relatively good terrain. It was not a viable strategy at all.

iii. Defining Tasks

Another strategy was to define stages of tasks that the agent would need to accomplish as it proceeds in its curricula. This provided more interesting results than the first two methods and also showed a progressive understanding of how the agent should be trained.

The first step in the curricula was to make sure that the agent understands how to tweak the noise parameters within a valid range, ignoring the terrain that was generated. This was done by assigning a positive reward when the agent selected a valid value and a negative reward when the agent tried to move the parameters beyond the acceptable range. This also helped the agent learn when to not modify a value and the ICM during initial training was able to accommodate a lot more inputs.

The next step focused on the agent being able to create terrain that had the correct normal values for terrain generation. In this case setting the reward for a correct selection was ignored and a positive reward was given if the agent was able to achieve viable

normals. When the agent was unable to achieve viable normals a negative reward compared to the error in the normal values was added.

The next step was being able to create terrain with a valid amount of water as well as correct normals, since the agent had already learned how to achieve good normals, this was relatively quick. With the agent understanding that the only vector action it needed to tweak for this input was the water level in the mesh settings. Similar to earlier, a positive reward was assigned when the water amount was within the permissible range and a negative reward based on the error was assigned when it was not.

The final step to the training was achieving a good slope value as well as a good water amount and normals values this step took particularly long for the SAC algorithm to calculate as it had reached the learning plateau, this led to some confusion for the agent and a reduction in the threshold value for minimum reward was necessary for it to move onto the next step. The same calculations for error and valid average slope was done to define a good negative and positive reward respectively.

Lastly, regular training in which a positive reward was calculated based on the height of the terrain and the scale of the noise value along if it met with all the criteria for a valid terrain and a negative reward was assigned based on how far away it was from the ideal criteria. Over many iterations, this made the agent start arriving towards ideal criteria in fewer environment steps and also ensured that it stabilised on a relatively good terrain generation with good variance as it was difficult to create with a high noise scale and height multiplier.

iv. Ordering Tasks

Experiments were done to see which ordering of the curricula made more sense. When the agent curricula demanded it assign valid water before assigning valid normals, there was too much noise in the reward algorithm where the agent was not able to arrive at the idealised terrain shape because it had already learnt that any terrain shape was fine as long as it had a valid amount of

water. This led to confusion and never allowed the agent to learn how to arrive at valid normals due to the extremely sparse reward for arriving at that outcome. Hence the decision was taken to calculate normals first and then decide on what would be a valid output for water amounts.

4. **ML Actions: OnVectorAction()**

   OnVectorAction is the function that is called on an agent when it can receive a vector of different continuous float values or discrete integer inputs. These values define what actions the agents take. In the case of the TerrainAgent behaviour, this was how to change the input for the noise parameters and mesh parameters. The total number of Vector Actions to be taken was 10. One for each of the parameters below:

   1. HeightMapSettings-> HeightMultiplier
   2. HeightMapSettings-> NoiseSettings-> NoiseEstimatorVariable
   3. HeightMapSettings-> NoiseSettings-> Scale
   4. HeightMapSettings-> NoiseSettings-> Octaves
   5. HeightMapSettings-> NoiseSettings-> Persistence
   6. HeightMapSettings-> NoiseSettings-> Lacunarity
   7. MeshSettings-> WaterLevel
   8. HeightMapSettings-> NoiseSettings-> Seed
   9. HeightMapSettings-> NoiseSettings-> OffsetX
   10. HeightMapSettings-> NoiseSettings-> OffsetY

   When using curriculum learning, it would also add a negative reward for not having valid values in the first lesson. The strategies for changing these parameters are listed below:

   a. Specific Values

      The first attempt was to normalize the vector inputs and directly set it as a function between the minimum and maximum values between each parameter. This led to some progress, but later on it was realized that designers might want the agent to take iterative steps towards what the neural network considered to be ideal output rather than just defaulting to a value that it thought best.

   b. Relative Values

      This led to the actions that the agent takes being relative to the current value of the parameter rather than specific to a value between the minimum and maximum. This meant that based on the action vector, the agent would either increase or decrease the parameters by a certain

amount based on how divisions were set. Hence the need for step 1 in curriculum learning to make the agent learn when it was or wasn't allowed to change the parameters(to ensure the changes were constrained to the minimum and maximum values of the parameters). This also meant that the agent was able to learn how to “stabilize” towards an ideal set of inputs.

c. Minimum division for increment of parameters
   There was some amount of thought to decide what the minimum division of increments for the parameters should be. The number of divisions also decided how many steps would need to be taken per episode. The stacking of observations(how much in the past the recurrent neural network should keep track of observations) also depended on the size of divisions.

   For a large number of divisions but a small number of steps its possible that the agent could never arrive at a solution for the generation. If there are a small number of divisions, the agent can overshoot and never arrive at a good value for the parameter except purely by chance. If the stacked observations are small and the divisions are large, the agent is not able to understand a relationship between the observations and the actions it is taking.

   Having 10 divisions and keeping track of 5 stacked observations (5 steps in the past) gave the best results for an agent to learn. Along with this, having a very high number of environment steps, something like 50, made sure that the agent understood that it was overshooting and it should learn how to stabilize towards ideal values to receive greater rewards.

5. ML Observations: CollectObservations()

Observations are vector values that the agent looks at to make sense of the environment. These are values that the agent looks at to decide what kind of actions it should take i.e make a decision. There can be any number of values that the agent can observe and there can be stacking of these observations as well to make an agent look at a history of observations so that it can understand a series of actions it takes.

a. Observations to be looked at
   The list of observations that the agent is provided below:
   1. Number of chunks generated

2. Action Values (all the input parameters listed above)
3. Average X, Y, Z Normals for vertices
4. Average Valid Terrain Slope
5. Average Terrain Water Levels

b. Normalizing Observations
Normalizing observations between the minimum and maximum value proves to be beneficial to the agent understanding a clearer relationship between the actions it is taking and what the observed values are and what would be “good” for the agent to have a good reward when it starts arriving at them. (Unity Technologies, 2019)[40] Since a lot of the input as well as output variations were limited to a minima and maxima, normalizing both sets of observations lead to considerably better results.

6. ML User Defined Functions

a. CalculateRewards()
This is the function that is called on every environment step to calculate rewards by the last terrain generation of the agent. It also decides reward values based on curriculum learning and assigns the positive reward on a successful outcome and a negative reward for error values in the current curriculum. Rewards were added in different iterations checking for different methods of curriculum learning.

Algorithm for curriculum based on ideal constraints

1. Define new variables for expanded ranges of ideal constraints based on lessonID
2. Calculate if the generation has got valid values of normals, slope and water for expanded ranges
3. Calculate if the generation has got valid values of normals, slope and water for regular ranges
4. If has valid values for expanded ranges
  a. Add positive reward
  b. Double amount if also correct for regular ranges
5. Else
  a. Add negative reward based on error of the values from the average value of expanded ranges

Algorithm for curriculum based on tasks
1. Calculate if the generation has got valid values of normals, slope and water for regular ranges 2. Switch lessonID: a. Case -1: i. Add small reward for valid values (On vector action checks for reward and punishment based on valid values) b. Case 1: i. If has validNormals 1. Add positive reward ii. Else 1. Add negative reward for Y error and half the negative reward for X,Z error c. Case 2: i. If has ValidNormals and validWater 1. Add positive reward ii. Else 1. Add negative reward for water level error d. Case 3: i. If has ValidNormals, validWater and validSlope 1. Add positive reward ii. Else 1. Add negative reward for slope error e. Case 4: i. If has ValidNormals, validWater and validSlope 1. Add positive reward based on: ((noise persistence * noise lacunarity) / noise scale) ii. Else 1. Add large negative reward

b. OnGenerationComplete()
This is the function that is called when the terrain generator has finished generating the minimum number of terrain chunks desired for calculation. It is responsible for calculating rewards, taking an environment step and requesting decisions.

5.4 Evaluating Strategies

Training strategies as well as the neural networks generated were judged based on:

1. **Learn how to manipulate noise data**
   Whenever the negative reward of a generated agent was shaped according to how close it was to ideal parameters, the generated NN was able to make an estimated guess to handle randomized terrain data. This didn't necessarily mean that the new terrain was ideal but that it would be closer to ideal than what the agent acted on in the last step.

   Some agents, for example, those trained with the wrong curriculum order or those that did not have a wide range of possible values to train on, had a very narrow scope where they were able to do something useful with the parameters. They got "confused" by data that they had not seen before. This was observed in a much lesser degree when it came to agents that had been trained for a longer time. Agents trained for a longer time had a more discernible pattern to try to improve noise parameters, even if it wasn't correct(Agents that were confused by early noise data).

   The order, variables manipulated and progress with curriculum learning also had a large effect on how the agent operated, some agents that were not able to move onto the second or third task of the curriculum still managed to create terrain parameters that can be made useful with some simple tweaks. This was also observed for agents that were trained where the curriculum variable being affected was the "ideal terrain constraints". Even if the terrain

2. **Working with a user/designer**
   Some learning behaviour like setting specific values in OnVectorAction or behaviours where the number of steps the agent takes was very low meant that the agent when using the NN and reproducing behaviour would arrive at solutions too fast. This meant that the behaviour did not suit use by designers because the agent would not approach a solution but would instead "jump" to a solution. Even if the solution was right, it would mean that the agent would be very predictable in its behaviour.

   Longer steps using relative actions ensured that the agent being trained arrived at a good solution more gradually, allowing a designer to see multiple iterations of what would be good terrain. It also allowed the agent to learn how to stabilize at good terrain so that when the designer selects a good set of noise parameters, the agent understands that it doesn't need to vary these parameters greatly.

RESULTS AND EVALUATION

6.1 Procedural Generation

The procedural generation of terrain, creating objects for this terrain using a scatter algorithm and defining a way to generate game objects in this terrain generation were completed successfully and can be useful tools to increase the scope of the project in the future.

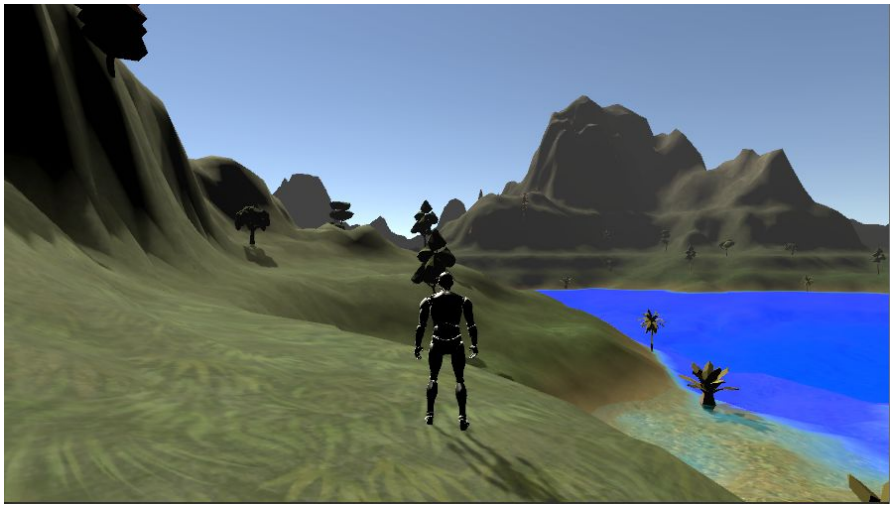


Figure 27. Third person game on generated terrain

Creating 2 playable modes for the example game, one involving a playable character with a third person controller as well as being able to click and guide a NavMeshAgent from a top down view on endlessly generated terrain were accomplished. This can also be used for the object generator behaviour that is suggested as further research.

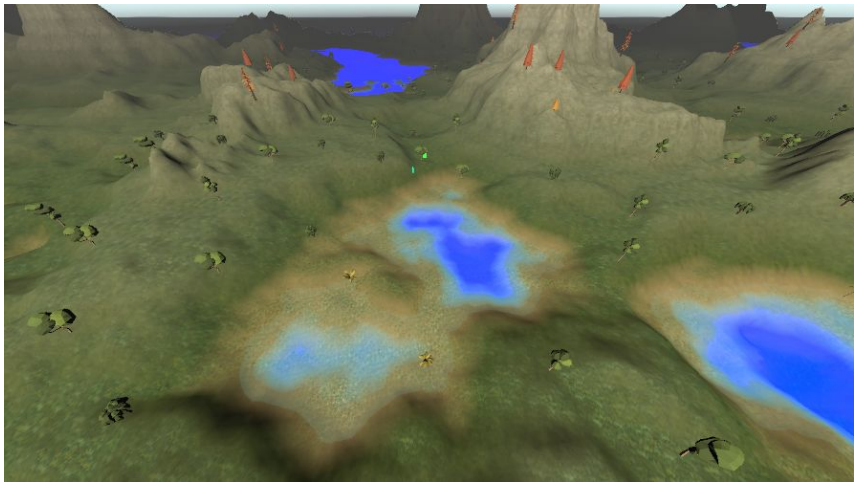


Figure 28. Nav mesh based top down game

6.2 ML Agents Results

An ML agent to help a designer defining noise parameters for the generated terrain was trained. The resulting method to use a neural network to guide or supplement a designer's ability to create useful terrain generations, can be used on many different concepts as well. From creating different ways to customize loot in a role playing game or procedurally creating good levels for a shooter by making bots play in different generations of levels. These concepts can also be used to make an already created level better through iterative training.

A discussion of various learning strategies for generative AI was done. Including the effects of

a. HyperParameters
   The number of possible actions the agent can take and the learning rate were intricately linked. Having a relatively simple agent that set specific values and had very few environment steps would train much faster with a higher learning rate but was prone to being inaccurate and defaulting to certain values when it found them. A more complex agent that took a larger number of environment steps would need a slower learning rate so that it doesn't overshoot.

   Generally, a large number of continuous inputs meant a large number of hidden units as well as layers was required. Keeping the batch size to near half the number of iterations the agent took seemed to improve understanding of increasing and decreasing reward better.

b. Algorithms
   SAC seemed to be more consistent with the output of learning due to it needing less hyperparameter optimization. But once it plateaued, there were certain cases that would stump it and it would never learn past them. This can also be because of underfitting of the initial random state in BeginEpisode().

   PPO was able to handle more use cases but was extremely slow to train and also needed better hyperparameter optimization. It was slightly better at understanding task based curriculum than SAC but still introduced noise in the output if the tasks were not relevant to each other. More tuning and better efficiency in the number of generations might lead to more understandable results.

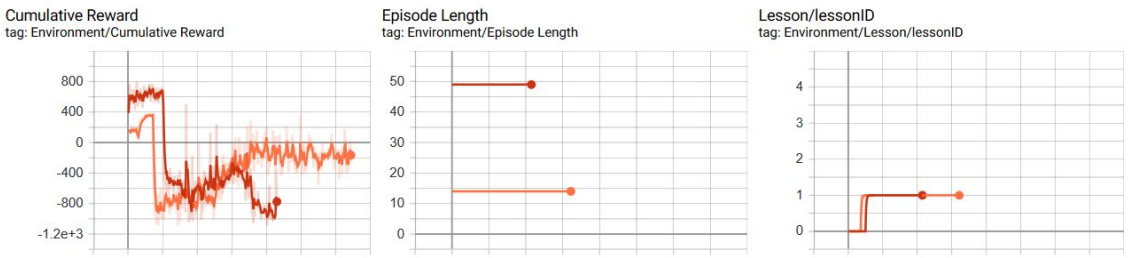


Figure 29. SAC(orange) with curriculum learning learnt faster than PPO(red) with curriculum learning and had better overall rewards too

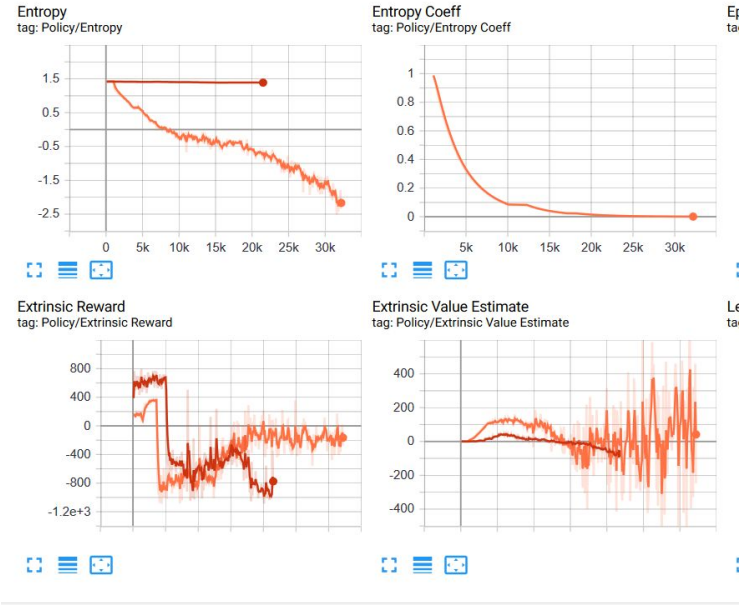


Figure 30. This is also showcased by the fact that SAC(orange) had a better entropy fall as the curriculum progresses as well, even if the estimates it made were more varied than PPO(red)

c. Possible actions the agent can take

Relative actions were a much better strategy than the agent setting specific values in the noise generation. It worked better for how the agent trained, how the agent stabilised and learnt what good outcomes were and also led to it working better for a designer working with the agent. Removing noise seed and noise offset as actions modifiable by the agent significantly improved the consistency of the agent predicting the right output. Hence the suggestion that this control should be given to a separate object generator agent as these parameters don't affect the shape of the noise as much as they affect the positioning. This can also be because the agent creates a random set of chunks within the minimum amount possible, making it difficult to make observations.

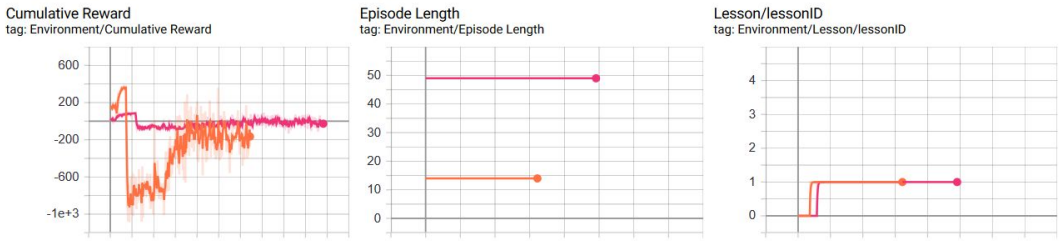


Figure 31. Removing the vector actions to control noise seed and noise offset(pink) showed significant improvement in consistent reward as compared to with control to modify seed and offset (orange)

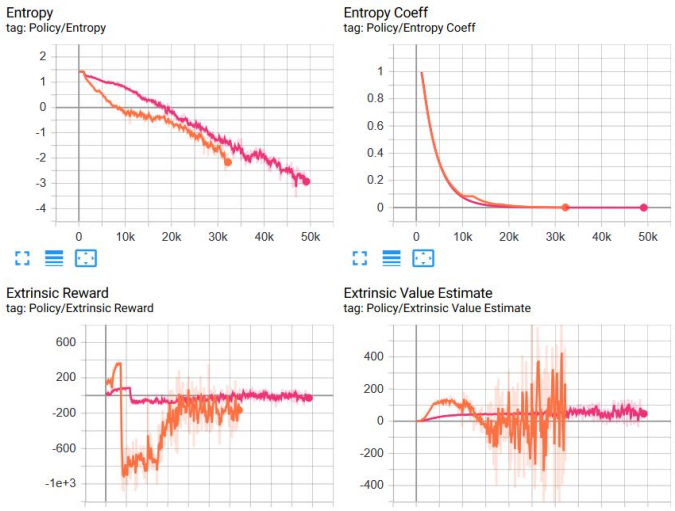


Figure 32. Even though the entropy loss is similar in both cases, the extrinsic value estimate shows how much more stable no noise seed and offset values make the learning

d. Calculating Rewards

Variable negative reward shaping was a very effective tool to improve terrain generation. In some cases the agent was able to prevent a bad outcome even if it couldn't generate terrain within "ideal" constraints. The agent had learnt to minimize the negative reward. Smaller rewards led to the agent learning faster. Larger rewards initially improve learning speed but do not work when used along with differently tasked curriculum.

Ending episodes early with a large positive reward payoff is not a good strategy as iterative positive rewards when the agent arrives at a good outcome reinforce agent behaviour as well as make the agents understand how to stabilize on a good outcome, something that is necessary when it is working with a designer.

Due to the large amount of continuous inputs, along with relatively straight forward sparse reward, intrinsic curiosity plays a major role during initial training for the agent to look into all the possible states of the input(all possible noise parameters) so that it improves learning later on as well.

e. Curriculum strategies
   Task based curriculum led to some agents learning better ways to create terrain but in many cases led to the agent getting confused due to the significant difference between each task. Having each task be an additional step rather than an iterative one improved this. Reducing thresholds on a lesson to go to the next step improved this. But that also meant that the agent took time to understand how to fulfill all criteria. SAC worked well with task based curriculum learning with lower threshold values. PPO worked better with task based curriculum learning when each task was an additional condition to the last rather than dealing with each condition step by step.
   Both had some success with curriculum affecting what an ideal terrain would be, especially with reward shaping for when it was producing actually ideal terrain.
   Curriculum affecting the possible input to noise parameters is a bad idea because it ensures that the agent would always default to an initial value of the noise parameters rather than actually iterate to values that are more effective even if randomization of initial parameters are there.

   A comparison below shows how much more unfocused learning becomes when the initial vector actions are not trained to set valid values (LessonID = -1 of the curriculum). When this is not done, training becomes a lot more random with no particular reward pattern being visible, possibly because the agent is not able to understand when it takes actions that are outside valid limits and no significant change is seen in the terrain generation results.

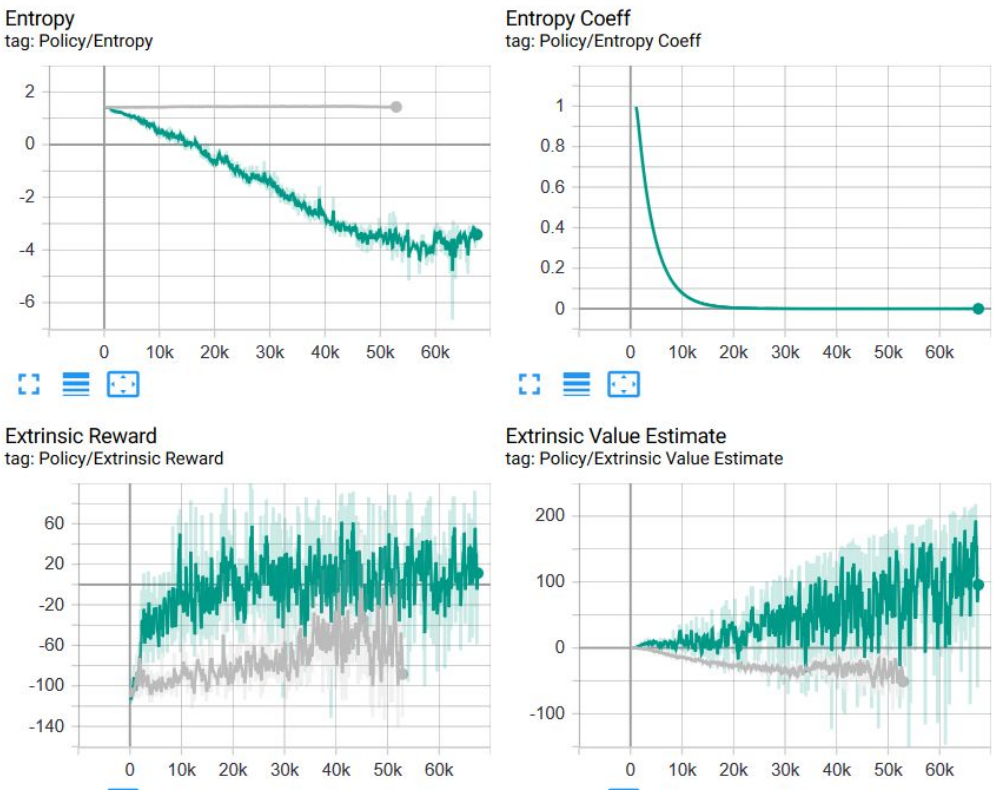


Figure 33. Training the agent without the initial curriculum of limiting values showed better results with SAC(green) than with PPO(grey) though both took significantly longer to go through the curriculum. Training them on valid values first was better.

CONCLUSION

7.1 Discussion

1. **Results**

   The aim of this project was to find a way to integrate Procedural Content Generation with Machine Learning in such a way that it can be used to improve upon, ease or compliment the work of a designer. The results of different training strategies as well as possible improvements for the same have been discussed.

   Some strategies prevented well rounded neural networks, like some that were unable to handle certain input/initial values while others would always default to certain outputs. There were still a significant number of them that had interesting observations and with further and more understood methods of training can have interesting results for a terrain generator optimizer. But every functional neural network actually has its own way of solving how to arrive at a solution. This also means that it provides options for a designer to use differently trained brains(Neural Networks), others based on different curricula, some based on different methods of solving a problem and others still based on different algorithms.

   Machine Learning in PCG can also then be used as a way of increasing the variance of PCG itself. Each new method of training a neural network defines a new way of understanding the problem statement, and each differently trained agent is customized in a way to solve the problem of what would be an ideal generation and the steps it takes to arrive at it in its own way. This allows a scope for further research in this field.

2. **Uses in the games industry**

   Procedural Generation has long been used by the games industry to add variety to gameplay. One of the first concrete examples, Rogue, coming out 40 years ago. It borrowed in many ways from Dungeons and Dragons-esque tabletop role playing games(Smith, 2015)[1]. Wherein a human dungeon master would define a world based on set rules defined within the game and create new stories based on those systems and rules. It is possible to imagine Machine Learning taking over from this human dungeon master when it comes to modern digital games. With many different ML agents working together to improve upon or populate a world. Using gents to create complex worlds and spaces is conceivable looking at how far PCG and ML have already come.

The concept shown here can also be used for other purposes, like populating points of interest on the map or creating sets of enemies in a more equalized spread across a shooter level or buildings across a strategy game with rewards being assigned on multiple playthroughs rather than just the definition of ideal terrain for the terrain generator. Machine Learning has already penetrated the industry with there being a lot of game as well as non-game research for Machine Learning in both the Unity engine and Unreal engine. Machine Learning is being used to generate new imagery for old games by texture upscaling (Edelsten, 2017)[41] and companies like NVIDIA looking into generating 3D data from 2D images (Finkle, 2019)[42].

ML has been very useful in understanding images and large maps of data as well, the concept can be used in heatmaps for MMORPGs to improve positioning of buildings. It can also improve slope based cover terrain in shooter games based on a heat map of how many players die at that location. Machine Learning is a powerful tool that is already used as AI and inference in games, with self play showing emergent behaviour with agents solving problems that were not even initially conceived of. Just as Machine Learning research is being done for fun gameplay experiences for players, it should be used to improve and ease the lives of designers and developers too.

In many ways, in the same way PCG has improved the lives of artists, designers and developers to create, improve and ease the development of assets for the games industry, ML with PCG can be used to improve them again. Due to Machine Learning's superior understanding of abstract patterns and data, it only makes sense to use it to generate new content procedurally as well.

3. Challenges Faced

1. Iterating through generations quickly
   One of the major issues faced was improving the speed at which terrain chunks are generated. This was required so that the agent could take environment steps more often.
   a. Multi Threading
      Requesting heightmap data, creating meshes and creating colliders were multithreaded so that the generated terrain was produced faster and in parallel. This also meant that any data that was being generated earlier as well as the associated threads for older data needed to be terminated.
   b. Nav Mesh Surface Calculation

Making sure that the navmesh was baked everytime using a nav mesh modifier class ensured that the generated mesh surface was continuous between multiple terrain chunks. This also ensured that there was no need for off mesh links that would hamper the playability of such a level.

2. Issues when multiple iterations are made
   When multiple iterations of terrain chunks were being generated simultaneously for hundreds of thousands of steps for each run of the terrain generating agent, it needed to be ensured that there was no increase in the amount of memory required due to old data. This meant calling the garbage collector manually but also meant looking into these issues over many generations.
   a. Memory Leaks
      Memory leaks needed to be fixed, for example instances of materials generated for each terrain chunk did not get cleared even when the garbage collector was called after the relevant gameobject was destroyed, this is because the terrain chunk class, to implement multithreading, could not have been a monobehaviour and kept references to the gameobjects as well. This also meant clearing mesh data and the relevant colliders before the garbage collector can be called.
   b. Disposing of old generation data
      There were issues where threads trying to generate old data would cause interferences with new generated data, especially if the noise parameters conflicted with each other. It needed to be ensured that no remaining data would clog the garbage collector or cause an increase in the amount of memory allocated to the application as this would increase with every step that the generator was run.

3. Machine Learning
   Trying to find the correct strategies to train the machine learning agent was a challenge of itself. As mentioned earlier PPO training took a considerably long amount of time due to a lack of parallel agents being trained because of resource constraints (the project was running on a laptop). And SAC had issues with respect to plateauing in its learning rates.

   Assigning rewards properly was a challenge as well as tuning hyperparameters properly also took considerable amounts of time. But some iterations did lead to fruitful results. Below is a discussion of how these were overcome:
   a. Limiting terrain generator values

Limiting the possible noise parameters that the agent could set within a minimum and maximum helped in ensuring that the agent would have a relatively usable set of data to work with. This also ensured that the properties that were being used to calculate the ideal terrain generation were calculable rather than them resulting in arbitrary values that would further confuse the already sparse reward for the agent.

b. Minimum number of meshes to be generated
   Since the agent should have found general purpose solutions to the terrain generation problem, the number of terrain chunks being generated before the agent reset to make a new decision were limited.

   First when the agent was looking at setting valid values for the noise generation, to a very small number so that iterations were quick. Then to a number that was a fraction of the total number of chunks that the agent would generate, ensuring that the samples of the terrain chunks it came across were varied.

c. Deciding proper sequencing for curriculum learning
   Deciding a proper curriculum learning strategy for continuous inputs might not have a right answer but it certainly has many wrong answers. This was compounded with understanding how to assign reward for the curriculum learning as well.

   Observations were made where the agent would get confused when it had a relatively simple task like limiting the amount of water in the terrain and then came across much more complex tasks like deciding the correct normal input in the next lesson of the curriculum. This is because the agent had already learnt that the wrong output for generating normals was fine as long as the water level was correct. This made the observations it made very noisy and it was never able to understand how to create relatively good terrain using this curriculum.

7.2 Further Research

There is a lot of scope for building on top of the results for this project. Some of these are discussed below:

a. Object Creator as an additional behaviour
   An additional behaviour planned along with the terrain generator was an object creator that would arrange points of interest on the map that a Nav

Mesh agent would then be able to move through (Using the generated NavMesh surface) using iterations that a trained terrain generator neural network would produce.

The terrain generator would no longer control the seed and offset of the noise parameters and this would be offloaded for the Object Creator behaviour to decide. The object creator will then define positions for points of interest for different seed and noise offset values and be able to emerge with a neural network that can place these objects properly for any kind of noise generation of the terrain. This proved difficult under the time constraints as the terrain generator agent took a lot of time to train.

Using the object creator proposed as an additional behaviour/agent in conjugation with the terrain generating neural network can help in creating much more emergent gameplay than just the terrain generator. New methods for scoring the output of the terrain generator besides just the average viable slope can be used like the number of areas possible for enemies and number of paths possible between them. The object generator will be trained on different noise seeds and offsets and can be used to procedurally generate different levels for any gameplay. Some use cases are:

1. Generating roads
2. Populating an area with enemies or buildings

b. Using GPUs for terrain generation
Currently the generation of the terrain meshes is multithreaded using a CPU, but geometry shaders running on a GPU can increase the efficiency of the terrain generation script significantly, especially to create meshes for rendering and colliders much faster. This can also allow for training multiple generator agents simultaneously and in parallel, something that was a huge limiting factor on the current implementation of the project.

c. Generative Adversarial Network (GANs)
Using the concept of the Object Creator above, the object creator agent can be used as a discriminator agent in a Generative Adversarial Network. In this case, the TerrainGenerator agent script will produce multiple iterations of terrain that the ObjectCreator can then rate on its viability and consequently modify the values of what would be an ideal terrain to generate for the TerrainGenerator. This process will possibly lead to the

Generator and Discriminator working in tandem to create much more possibilities of interesting levels.

d. Generative Adversarial Imitation Learning (GAIL)
Multiple iterations of already viable noise parameters decided by a developer or designer can be used as the raw dataset to train a GAIL agent. This would mean that the neural network will have examples of already viable or interesting terrain as a dataset and it can detect patterns or information that is useful for the agent to reproduce. This would also mean that the dataset would have to be significantly large but methods like Behavioural Cloning can help with simplifying this requirement.

ACKNOWLEDGEMENTS

8.1 Credits

1. My supervisor, Gareth Robinson at Abertay University
2. Unity Technologies
3. Creators of the unity packages
    a. ML Agents Toolkit (Juliani et al., 2020)[24]
    b. 3D assets from the unity asset store
        i. ADG Textures (A dog's life software, 2016)[55]
        ii. Broken Vector Low Poly TreePack (Broken Vector, 2018)[56]
    c. Invector Third Person controller (Invector, 2020)[57]
4. Sebastian Lague's excellent tutorial on procedural landmass generation

8.2 References

1. Gillian Smith, Foundations of Digital Games (2015), "*An Analog History of Procedural Content Generation*", Available At: http://www.fdg2015.org/papers/fdg2015_paper_19.pdf

2. Joris Dormans and Sander Bakkes (September 2011), "*Generating Missions and Spaces for Adaptable Play Experiences*", Available At: https://ieeexplore-ieee-org.libproxy.abertay.ac.uk/stamp/stamp.jsp?tp=&arnumber=5763766

3. Dong-Ming Yang, Jian-Wei Guo, Bin Wang, Xiao-Peng Zhang and Peter Wonka, Journal of Computer Science and Technology (2015), "*A Survey of Blue Noise Sampling and its Applications*", Available At: http://archive.ymsc.tsinghua.edu.cn/pacm_download/38/276-2015_JCST_BNSurvey.pdf

4. Franz Aurenhammer, ACM Computing Surveys Vol 23, No. 3 (September 1991), "*Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure*", Available At: https://www.cs.jhu.edu/~misha/Spring16/Aurenhammer91.pdf

5. Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker and Sergey Levine, (June 2019), "*Learning to Walk via Deep Reinforcement Learning*", Available At: https://arxiv.org/pdf/1812.11103.pdf

6. Joe Booth and Jackson Booth, AAAI-2019 Workshop (Feb 2019), "*Marathon Environments: Multi-Agent Continuous Control Benchmarks in a Modern Video Game Engine*", Available At: https://arxiv.org/ftp/arxiv/papers/1902/1902.09097.pdf

7. Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew and Igor Mordatch, ICLR 2020 (Feb 2020), "*Emergent Tool Use From Multi-Agent Autocurricula*", Available At: https://arxiv.org/pdf/1909.07528.pdf

8. Lawrence Johnson, Georgios N Yannakakis and Julian Togelius, Workshop on Procedural Content Generation (June 2010), "*Cellular automata for real-time generation of infinite cave levels*", Available At: http://julian.togelius.com/Johnson2010Cellular.pdf

9. Matthew Graves, University of Austin Texas (December 2016), "*Procedural Content Generation of Angry Birds Levels Using Monte Carlo Tree Search*", Available At: https://repositories.lib.utexas.edu/bitstream/handle/2152/46264/GRAVES-MASTERSREPORT-2016.pdf?sequence=1&isAllowed=y

10. Julian Togelius, Mike Preuss and Georgios Yannakakis, (June 2010), "*Towards multi objective procedural map generation*", Available At: https://www.researchgate.net/publication/228679513_Towards_multiobjective_procedural_map_generation#read

11. Joseph Alexander Brown, Bulat Lutfullin, Pavel Oreshin and Ilya Pyatkin, 9th Computer Science and Electronic Engineering Conference (CEEC 2017), "*Levels for Hotline Miami 2: Wrong Number Using Procedural Content Generations*", Available At: https://www.mdpi.com/2073-431X/7/2/22/htm

12. Ken Perlin, ACM Transactions on Graphics (July 2002), "*Improving noise*", Available At: https://mrl.nyu.edu/~perlin/paper445.pdf

13. Thomas J. Rose and Anastasios G. Bakaoukas, 8th International Conference on Games and Virtual Worlds for Serious Applications (September 2016), "*Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques*", Available At: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7590336

14. Jacob Olsen, (October 2004), "*Realtime Procedural Terrain Generation*", Available At: https://web.mit.edu/cesium/Public/terrain.pdf

15. Sebastian Lague, Procedural Landmass Generation (January 2016), "*Tutorial on Procedural Landmass Generation*", Available At: https://www.youtube.com/watch?v=wbpMiKiSKm8

16. Daniel Dunbar and Greg Humphreys, Association for Computing Machinery (2006), "*A Spatial Data Structure for Fast Poisson-Disk Sample Generation*", Available At: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.78.3366&rep=rep1&type=pdf

17. Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen and Julian Togelius, (February 2017), "*Procedural Content Generation via Machine Learning (PCGML)*", Available At: https://arxiv.org/pdf/1702.00539.pdf

18. Adam J. Summerville and Michael Mateas, (March 2016), "*Super Mario as a String: Platformer Level Generation Via LSTMs*", Available At: http://www.digra.org/wp-content/uploads/digital-library/paper_129.pdf

19. Matthew Guzdial, Joshua Reno, Jonathan Chen, Gillian Smith and Mark Riedl, Fifth Experimental AI in Games Workshop (September 2018), "*Explainable PCGML via Game Design Patterns*", Available At: https://arxiv.org/pdf/1809.09419.pdf

20. Matthew Guzdial, Brent Harrison, Boyang Li and Mark O. Riedl, Foundations of Digital Games (2015), "*Crowdsourcing Open Interactive Narrative*", Available At: http://www.fdg2015.org/papers/fdg2015_paper_06.pdf

21. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei and Ilya Sutskever, OpenAI (2019), "*Language Models are Unsupervised Multi Task Learners*", Available At: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

22. William L. Raffe, RMIT University (August, 2014), "*Personalized Procedural Map Generation in Games via Evolutionary Algorithms*", Available At: https://researchbank.rmit.edu.au/view/rmit:160863

23. Vanessa Volz, Niels Justesen, Sam Snodgrass, Sahar Asadi, Sami Purmonen, Christoffer Holmgard, Julian Togelius and Sebastian Risi, IEEE Conference on

Games (May 2020), "*Capturing Local and Global Patterns in Procedural Content Generation via Machine Learning*", Available At: https://arxiv.org/pdf/2005.12579.pdf

24. Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar and Danny Lange, Unity Technologies (May 2020), "*Unity: A General Platform for Intelligent Agents*", Available At: https://arxiv.org/pdf/1809.02627.pdf

25. Marc G. Bellemare, Yavar Naddaf, Joel Veness and Michael Bowling, Journal of Artificial Intelligence Research 47 (June 2013), "*The Arcade Learning Environment: An Evaluation Platform for General Agents*", Available At: https://arxiv.org/pdf/1207.4708.pdf

26. Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg and Stig Petersen, (December 2016), "*DeepMind Lab*", Available At: https://arxiv.org/pdf/1612.03801.pdf

27. Matthew Johnson, Katja Hofmann, Tim Hutton and David Bignell, International Joint Conference on Artificial Intelligence (2016), "*The Malmo Platform for Artificial Intelligence Experimentation*", Available At: https://www.ijcai.org/Proceedings/16/Papers/643.pdf

28. Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek and Wojciech Jaskowski, IEEE Conference of Computational Intelligence in Games (September 2016), "*ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning*", Available At: https://arxiv.org/pdf/1605.02097.pdf

29. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov, OpenAI (August 2017), "*Proximal Policy Optimization Algorithm*", Available At: https://arxiv.org/pdf/1707.06347.pdf

30. Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel and Sergey Levine, (Jan 2019), "*Soft Actor-Critic Algorithms and Applications*", Available At: https://arxiv.org/pdf/1812.05905.pdf

31. Jonathan Ho and Stefano Ermon, (June 2016), "*Generative Adversarial Imitation Learning*", Available At: https://arxiv.org/pdf/1606.03476.pdf

32. Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan and Chrisina Jayne, ACM Computing Surveys, Vol. V (April 2017), "*Imitation Learning: A Survey of Learning Methods*", Available At: http://www.open-access.bcu.ac.uk/5045/1/Imitation%20Learning%20A%20Survey%20of%20Learning%20Methods.pdf

33. Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever and Igor Mordatch, ICLR (March 2018), "*Emergent Complexity via Multi-Agent Competition*", Available At: https://arxiv.org/pdf/1710.03748.pdf

34. Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel and Wojciech Zaremba, OpenAI (February 2018), "*Hindsight Experience Replay*", Available At: https://arxiv.org/pdf/1707.01495.pdf

35. Deepak Pathak, Pulkit Agrawal, Alexei A. Efros and Trevor Darrell, ICML (May 2017), "*Curiosity-driven Exploration by Self-supervised Prediction*", Available At: https://arxiv.org/abs/1705.05363

36. Yoshua Bengio, J´erˆome Louradour, Ronan Collobert and Jason Weston, Journal of the American Podiatry Association (January 2009), "*Curriculum Learning*", Available At: https://ronan.collobert.com/pub/matos/2009_curriculum_icml.pdf

37. Joel Z. Leibo, Edward Hughes, Marc Lanctot and Thore Graepe, DeepMind (March 2019), "*Autocurricula and the Emergence ofInnovation from Social Interaction: A Manifesto for Multi-Agent Intelligence Research*", Available At: https://arxiv.org/pdf/1903.00742.pdf

38. Unity Technologies, Unity Documentation (August 2020), "*About ML-Agents package*", Available At: https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/manual/index.html

39. Cristian Coenen and Unity Technologies, ML Agents Documentation (July 2020), "*Training Configuration File*", Available At:

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md

40. Chris Elion and Unity Technologies, ML Agents Documentation (October 2019), "*Environment Design Best Practices*", Available At: https://github.com/Unity-Technologies/ml-agents/blob/release-0.13.1/docs/Learning-Environment-Best-Practices.md

41. Andrew Edelsten and NVIDIA, GDC (May 2017), "*Zoom, Enhance, Synthesize! Magic Upscaling and Material Synthesis using Deep Learning*", Available At: https://on-demand.gputechconf.com/gtc/2017/presentation/s7602-andrew-edelsten-zoom-enhance-synthesize.pdf

42. Lauren Finkle and NVIDIA, NVIDIA Blog (December 2019), "*2D or Not 2D: NVIDIA Researchers Bring Images to Life with AI*", Available At: https://blogs.nvidia.com/blog/2019/12/09/neurips-research-3d/

43. Unity Technologies, Unity.com(Accessed August 2020), "*Unity Shader Graph*", Available At: https://unity.com/shader-graph

44. Unity Technologies, Unity Documentation (August 2020), "*Navigation System in Unity*", Available At: https://docs.unity3d.com/Manual/nav-NavigationSystem.html

45. Guy Hacohen and Daphna Weinshall, 36th International Conference on MachineLearning (May 2019), "*On The Power of Curriculum Learning in Training Deep Networks*", Available At: https://arxiv.org/pdf/1904.03626.pdf

46. Markus "Notch" Persson (2009), "*Minecraft*", [Video game], Microsoft

47. Derek Yu (2008), "*Spelunky*", [Video game]

48. King (2012), "*Candy Crush Saga*", [Video game]

49. Hello Games (2016), "*No Man's Sky*", [Video game]

50. Glenn Wichman and Epyx (1980), "*Rogue*", [Video game]

51. Blizzard Entertainment (2004), "*World of Warcraft*", [Video game]

52. Gearbox Software (2009), "*Borderlands*", [Video game]

53. Subset Games (2012), "*FTL: Faster Than Light*", [Video game]

54. Blizzard Entertainment (1996), "*Diablo*", [Video game]

55. A dog's life software, Unity Asset Store (September 2016), "*Outdoor Ground Textures*", Available At: https://assetstore.unity.com/packages/2d/textures-materials/floors/outdoor-ground-textures-12555

56. Broken Vector, Unity Asset Store (July 2018), "*Low Poly Trees*", Available At: https://assetstore.unity.com/packages/3d/vegetation/trees/low-poly-tree-pack-57866

57. Invector, Unity Asset Store (January 2020), "*Third Person Controller*", Available At: https://assetstore.unity.com/packages/tools/utilities/third-person-controller-basic-locomotion-free-82048

58. Amit Patel, Red blob games (January 2020), "*Making maps with noise functions*", Available At: https://www.redblobgames.com/maps/terrain-from-noise/