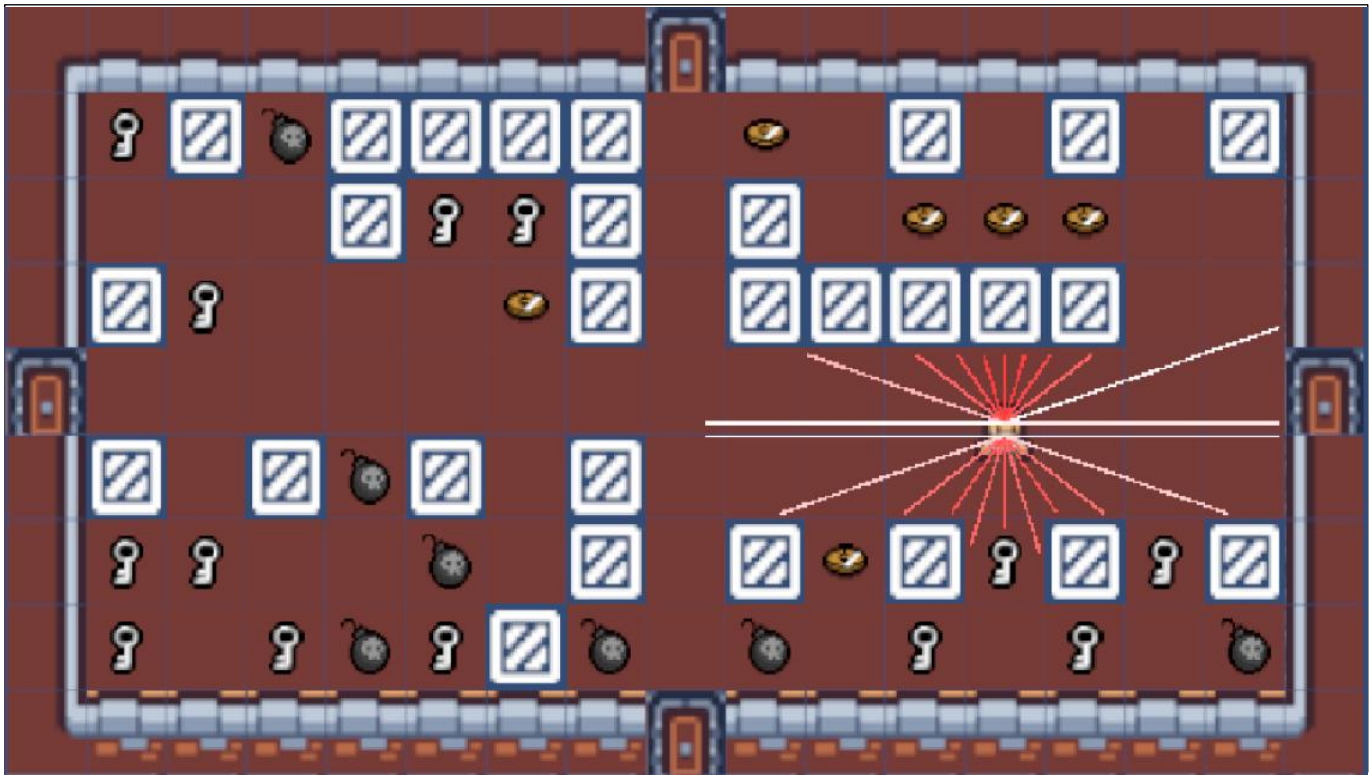
Comprehensive Creative Technologies Project: Machine Learning in Procedural Content Generation

Kieran De Sousa
kieran2.desousa@live.uwe.ac.uk
Supervisor: Louca Coles

Department of Computing and Creative Technology
University of the West of England
Coldharbour Lane
Bristol BS16 1QY



Abstract

Procedural content generation (PCG) is a technique used in video game development to algorithmically create content, and is often used for visual creation. Machine learning (ML) is being explored as a substitute for human testing.

The project explores and evaluates the creation of functional game content with PCG, the training of ML agents in this content, and the evaluation of generated content for engagement through trained ML agents. The project concludes by discussing the strengths and weaknesses of the implementations of PCG and ML, and the effectiveness of engagement data in influencing new PCG outputs.

Keywords: Machine Learning, Procedural Content Generation, ML Agent, Engagement

How to access the project

The project repository can be accessed at: https://github.com/Kieran-De-Sousa/ML-Procedural-Content-Generation

The project viva video can be accessed at: https://youtu.be/ntM_Sgg6gpA?si=Zh6yP5F_6sixF7Ka

1. Introduction

In the context of video game development, procedural content generation (PCG) is often associated with asset creation, such as environments, or non-player characters. Video games like The Binding of Isaac (McMillen, 2011) and Enter the Gungeon (Devolver Digital, 2017) utilise procedural generation to create gameplay variation, by connecting individual rooms created by a level designer to create a level. This project aims to explore the integration of Machine Learning (ML) in PCG to enhance quality and engagement in PCG outputs; these outputs being individual room levels of a top-down, 2D video game created in Unity (Unity Technologies, 2022).

Using PCG to create levels runs into problems for developers, such as resources and time being required to test outputs, and manual editing of PCG methods based on the results of this testing. Using ML, ML agent feedback could be used as a substitute for user testing to improve PCG outputs. The importance of investigating ML in PCG lies in its potential to automate gameplay level creation and assessment, saving developers time and resources which can be allocated elsewhere in the video game deliverable. The project explores beyond the typical use of generating environments for appearance, and investigates content generation focused on gameplay quality.

The project seeks to answer whether ML can effectively assess generated levels for quality (engagement), with its inspiration stemming from the potential streamlining of development for both small-scale and large-scale game studios. The potential benefits of PCG efficiency, gameplay quality, and development cost efficiency, through ML are key motivators for exploring its use in video game development.

1.1 Project objectives

- Develop a PCG tool that creates interactive level rooms in Unity.
- Develop and train a ML agent that replicates typical player behaviour.
- Develop a system for improving PCG outputs based on ML agent engagement data.

1.2 Key deliverables

- A PCG system that creates interactive, top-down, 2D room levels through algorithms such as random generation, and A* pathfinding.
- A trained, ML agent that behaves similarly to a human player, that rewards (engages) itself through room exploration, and item collection.

2. Research questions

The project explores the following research questions, focused on ML, PCG, and the integration of the two:

- How accurately can ML agents simulate human player behaviour?
- What PCG techniques can be used to generate interactive gameplay levels?
- How effective is machine learning (ML) as a method for improving procedural content generation (PCG) outputs?
- How does feedback from ML agents on PCG outputs compare to feedback from human testing?

3. Literature review

3.1 Procedural Content Generation

Procedural Content Generation (PCG) "is the algorithmic creation of game content with limited or indirect user input" (Togelius, 2011). The definition stems from Togelius et al. research into what PCG is, with discussion first starting on what PCG is not. PCG is not "offline" and "online" player-created content, by developer or player, but is closer to being "random", "adaptive", or both, algorithmic generations of game content.

One of PCG's first uses in video games is seen in dungeon crawler Rouge (A.I. Design, 1980), which used dungeon generation as its PCG method. A PCG level was randomised through a seed upon the player leaving the previous level by walking down a set of stairs (Procedural Content Generation Wiki, 2016). Games like The Binding of Isaac (McMillen, 2011) and Enter the Gungeon (Devolver Digital, 2017) similarly utilise dungeon generation to create levels by connecting rooms together through corridors.

Differing from this, platformer video game Spelunky (Mossmouth, 2008) main gameplay loop is centred around its PCG implementation, with generated dungeons being entirely unique. Spelunky's 4x4 grid (16 rooms) generation follows parameters around its four different room types, where a valid solution path to complete the level is generated first, and unoccupied grid

spaces generated with "side room" style rooms (Kazemi).

PCG can be achieved through different methods and techniques depending on suitability, including Cellular Automata and Wave Function Collapse. When developing the artifact, these algorithms were researched for applicability.

3.1.1 Cellular Automata

Cellular Automata (CA) is a method of PCG that uses a grid of cells that each have a finite number of states (Adams, 2017). Over a number of time steps, the state of each cell changes depending on a set of rules which are based on the states of neighbouring cells. Cave generation, the generation of cave systems and underground environments, is one of the most common applications of CA in video game development, as the rule-based states of neighbouring cells are effective at creating natural and complex looking cave structures.

CA might be used over other PCG methods due to being computationally inexpensive, making it valid for real-time applications, such as fluid dynamics in video games. Games like Minecraft (Mojang Studios, 2011) and Dwarf Fortress (Bay 12 Games, 2006) use CA, with the former using CA rules for simulating water flowing, fire spreading, and prior to version 1.17, the "biomes" (style) of each region in the world.

3.1.2 Wave Function Collapse

The Wave Function Collapse (WFC) algorithm is a texture synthesis algorithm initially developed by Maxim Gumin (2016). WFC started seeing use in PCG for its effectiveness in creating complex patterns in generation based on its inputs. WFC follows similar principles to quantum mechanics in the form of superposition, which is when a particle (or tile in WFC) exists in multiple states at the same time. WFC divides a tile-based level into small chunks, with each tile existing in superposition. Constraints reduce the possible states of these tiles until they are eventually in a single state, "collapsing" the tile. The final output of all collapsed tiles results in a pattern, or level generated. Real-time strategy game Bad North (Plausible Content (2018) utilises WFC for generating its islands.

WFC might be used over other PCG methods due to versatility; with open-source examples and versions available for multiple programming languages like C++ and Rust, and video game types such as 2D, and 3D. The artifact is designed around 2D, top-down constraints, which has been shown as working effectively with WFC algorithms.

3.2 Machine Learning

Machine Learning (ML) "is a category of artificial intelligence that enables computers to think and learn on their own" (Alzubi, J, 2018). The term "machine learning" stems from pioneer of artificial intelligence (AI), Arthur Samuel, in 1959 where he developed one of the first self-learning systems through work on computer checkers.

ML continued development into different methods, including reinforcement learning (RL) and supervised learning (SL). ML's most apparent use in video games development was the improvement of game AI systems such as non-player character (NPC) behaviour, seen in games like Creatures (Creature Labs, 1996), and Forza Motorsport 7 (Turn 10 Studios, 2017).

When choosing to implement ML, it is important to consider the different paradigms available, and to select which is most appropriate.

3.2.1 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm that aims to improve agent performance through trial-and-error experiences. An agent's goal is to maximise a reward or to complete an objective by using received feedback to adjust its actions. Reinforcement learning "dates back to the early days of cybernetics and work in statistics… and computer science" (Kaelbling, 1996).

RL differs from other paradigms like supervised learning, for example: In RL, after an action of an agent, the agent is immediately told their reward and the next state they will be in, but are not told what action would have been best in their long-term (future rewards). This makes RL suitable for "search and planning" based-scenarios, which can be commonly found in video games, such as high score maximisation.

RL's first use in video games is often attributed to Creatures (Creature Labs, 1996), an artificial life simulation game created by lead programmer Steve Grand, a computer scientist. The player would raise, and train virtual creatures called "Norns", which would exhibit more complex behaviours and learning patterns due to their utilisation of neural networks, which would over time develop based on interactions from the player and their environment. Creatures, and its series spanning three main games, were financially and critically successful, with all employing RL for AI behaviour.

In applications as a game development tool, Forza Motorsport 7 (Turn 10 Studios, 2017), a racing simulation game, utilised ML and RL to improve their "Drivatar System". Drivatar is Turn 10's advanced AI driving system for the Forza series, and was improved in factors of realism using ML. Agent's were trained to race around

individual racetracks 26,000 times (Esaki, C. 2023) to learn the fastest racing lines, and in gameplay, these agents might factor in human mistakes into their behaviour to improve realism, such as braking too late for corners.

3.2.2 Existing machine learning tools in Unity

In the Unity ecosystem, existing tools for ML are available, the most notable being the Unity Machine Learning Agents Toolkit plugin (ML-Agents) (Unity Technologies, 2017). ML-Agents is an open-source project that gives developers tools to create and train intelligent ML agents using ML algorithms, such as supervised learning (SL) and reinforcement learning (RL). As the algorithms and implementation has been simplified to a downloadable plugin for Unity, development time can be focused and allocated towards the development of expected agent behaviour.

ML-Agents major strength comes from its variety of example projects that showcase different use cases for the tool, along with providing developers useful starting points to streamline development. Examples include "Basic", which uses RL to teach an agent to move towards a reward sphere in a 2D space, and "GridWorld", which uses RL to teach an agent to avoid obstacles to move towards a goal in a grid-world space. Along with this, extensive documentation and courses like Immersive Limit LLC's "ML-Agents: Hummingbirds" course (Immersive Limit LLC, 2020) provided useful information in the development of the artifact. The course discusses implementations of RL, through reward structure creation, and training and re-training, to create intelligent agents in a 3D environment, which found high applicability in the artifacts 2D environment.

3.3 Procedural Content Generation via Machine Learning (PCGML)

Procedural Content Generation via Machine Learning (PCGML) is "the generation of game content by models that have been trained on existing game content" (Summerville, 2018). Summerville's research into PCGML is centred around content being generated "directly" from the ML model, meaning the outputs of a machine-learned model is itself interpreted as content.

Similar to the artifact, the research focuses on the creation of "functional" game content generation as opposed to "cosmetic" game content generation. Functional content generation might include the placement of enemies, the layout of game levels, and the placement of interactable elements (items, buttons), to enhance gameplay experiences. Cosmetic content generation might include visuals (textures, scenery), to enhance game immersion.

Summerville's research into PCGML suggests several use cases and applications in game development. Autonomous generation is the "generation of complete game artifacts without human input at the time of generation"; with PCGML, a designer might create artifacts in the target domain as the model for the generator, and then the chosen PCG algorithm can generate content in this style, avoiding the need of designers to turn their design intentions into code, saving development time and expenses.

PCGML also offers a lower barrier to entry for developers to generate functional game content, as a programming language is not required to specify generation of acceptance criteria. Content design in PCGML is AI-assisted, with a human designer and an algorithm working together to generate content; as the designer is training the ML algorithm through examples in the target domain, the inputs and outputs required of the ML algorithm is given by the designer. This results in no need for a programming language to communicate to the ML algorithm.

4. Research methods and Ethics

4.1 Research methods

The research methodology for this project consists of secondary research, comprising books, journal articles, research papers, and code documentation; on the topics of PCG, ML, and both. The main resources for obtaining research are Google Scholar, and GitHub (GitHub, 2024).

Secondary research was chosen due to the plethora of primary research conducted on PCG and ML. By using this primary research, the artifact can build on established knowledge, and utilise recognised techniques.

Secondary research and application were split into the following:

- Discovery of existing literature and techniques in PCG and ML, which were noted.
- Evaluation of material for relevance to the project, depth, and possible artifact application.
- Application into the artifact, through code, or in evaluation of PCG/ML outputs during development.

4.2 Ethical and professional principles

The project did not include any participant testing or data, and did not require any consent waivers. However, the use of PCG in the project raises concerns indirectly towards the loss of jobs

as a result of automation. As noted in Andrew Doull's "The Death of the Level Designer" (Doull, 2008), PCG continues to make inroads into traditional level designer roles due to it being easier to build and deploy for studios, making these roles potentially "obsolete". To mitigate these concerns, the artifact's scale and focus is around the usage for small developer teams or individuals, which would benefit from the additional time and resources that would be provided and used elsewhere in development.

To adhere to professional guidelines and standards, all code implemented from research and referencing was cited and credited through code comments, and are acknowledged in the report through referencing and bibliography additions. Additionally, all assets sourced externally are licensed for commercial and research use, such as CC0 "No Rights Reserved" (Creative Commons, 2024) licensed material including Tiny Dungeon (Kenney, 2022).

4.3 Research findings

4.3.1 Why use Procedural Content Generation in video game development

A critical element of the research phase was the reason why developers might use PCG in their game development. Research from sources such as "Procedural Content Generation in Games" (Shaker, 2016) highlighted the usefulness of reducing the need for human designers and artists, noted as being slower and more expensive than PCG tool alternatives. Alongside this, PCG enable completely different and unique types of games to be developed; games in the 'Rouge-like' genre commonly use PCG elements, as rogue-likes main gameplay loops rely on randomisation and variability, which PCG effectively enables.

These findings support the project's aim to automate time and cost intensive areas of game development, and aims of creating unique gameplay experiences. Due to PCG being more necessary in rogue-like games, the project's scope was limited to PCG in this genre (top-down, 2D) as opposed to a generalisable PCG tool.

4.3.2 Why use Machine Learning in video game development

Research from G. Skinner et al. (2019) suggests that current implementations of video game AI leave users dissatisfied, where "bad" AI is easy to notice whereas "good" AI is expected of a game. In the application of ML as a tool, it is potentially "limitless", and its use case as a user testing substitute has been explored in research such as Niklas Kühl et al. (Kühl, N. 2022). Kühl's study of pattern recognition in 44 humans compared to three different machine learning algorithms suggests that ML might learn slower, but can reach similar or higher levels of performance than humans.

These findings support the projects proposition of ML being a cost-effective substitute for developer testing of generated content; as after the initial difficulty of learning, their feedback provided might be comparable to humans. Considering this difficulty of initial training based on research, the input parameters of the ML agents were reduced in scope, to allow more time for fine-tuning expected behaviour of agents.

5. Practice

5.1 Artifact scope and delivery

In the initial proposal of the artifact, the project aimed to create a generalised set of PCG development tools that could generate multiple types of PCG, such as both 2D and 3D environments.

After feedback from tutors and peers over scope, the artifact development firstly considered the type of PCG to implement. A 2D, top-down, tile-based system was chosen due to the following:

- Ease of management: Management and generation of content in a 2D space is easier to effectively achieve than in 3D spaces, which allowed greater time investment in refining PCG systems.
- Visual clarity: Results of PCG are easier to visually identify due to camera perspective showing all elements of PCG, which helps in debugging and assessment of PCG effectiveness.
- Suitability in genre: PCG is a common element in rouge-like games, which often utilise 2D perspectives, making the artifact more applicable in professional contexts.

5.2 Development of PCG systems

5.2.1 Evaluating existing Unity systems

To develop the 2D PCG systems, it was essential to expand Unity's implementations of `Tilemap` and `TileBase`, which are components of Unity's Tilemap system (UTS) package. This package was chosen to speed up initial development of the artifact, and proved beneficial as a set of systems to expand and develop from.

Evident problems in the components of UTS were quickly identified in development. Unity `Tilemap` do not inherently support the ability to identify which tile in its grid was collided or triggered, as these collisions/triggers are registered on all tiles as one shared collider, making it unable achieve specific method execution such as deleting items on pickup. Unity `TileBase` do not possess features like identifying their type (e.g. if they

are a wall, door, etc.), and their world position when their owner `Tilemap` is not known.

5.2.2 Tiles

```
/// <summary>
/// Extended implementation of Unity's <c>TileBase</c> class
/// providing additional properties and methods.
/// </summary>
[Serializable]
No asset usages  41 usages  10 inheritors  Kieran De Sousa  1 exposing API
public abstract class Tile : MonoBehaviour
{
    [Header("Base Properties")]
    public TileBase tileBase;  Changed in 0+ assets
    protected TileType tileType = TileType.FLOOR;

    // Tilemap
    protected Tilemap ownerTilemap = null;
    protected Vector3Int tilePosition = new();
    protected Vector3 tileWorldPosition = new();

    // Player
    protected MLAgent player = null;
```

Figure 1. `Tile` abstract class inherited by all custom tile classes.

In response to UTS limitations, the artifact creates custom `Tile` class implementations of `TileBase`. The `Tile` class contains data that is accessed by various systems such as `TriggerTilemap`. Each `Tile` stores its `TileBase` asset, its tile type (e.g. floor), references to its owner tilemap and position data, and the `MLAgent` of its simulation. Following object-oriented programming (OOP) principles of inheritance, this abstract base class (Figure 1.) is inherited and expanded in derived classes, such as `TileDoor`, to provide functionality to various systems.

Following compositional design principles, which complement OOP practices of polymorphism, interfaces like `ICollidable` and `IInteractable` were implemented to modularise components that were used in different `Tile` derived classes. C# not allowing multiple class inheritance but allowing multiple interface implementation, along with C# allowing to query classes if they contain a specified interface, made interface use effective in the artifact.

`TileInteractable` critically implements the `IInteractable` interface, enabling derived classes to override the `Interact` method. With polymorphism, the calling of this method would execute the derived classes implementation of `Interact`, such as in `ItemCoin`, which rewards the `MLAgent` stored in its data.

As the project progressed, the necessity for more expansion of data and methods in `Tile` became evident, and new classes and interfaces were developed as a result.

5.2.3 Tilemap

`Tilemap` was expanded into two main systems, management of all tilemaps in a simulation in `TilemapSystem`, and the handling of collisions and trigger events in `FloorTilemap`, `CollidableTilemap`, and `TriggerTilemap`. `TilemapSystem` holds data of the generated room and tilemaps in a simulation, and critically stores the list of `Tile` classes that can be instantiated by the PCG systems. This was added to provide flexibility to the user over the appearance and functionality of PCG generation. This is seen in Figure 2. below.

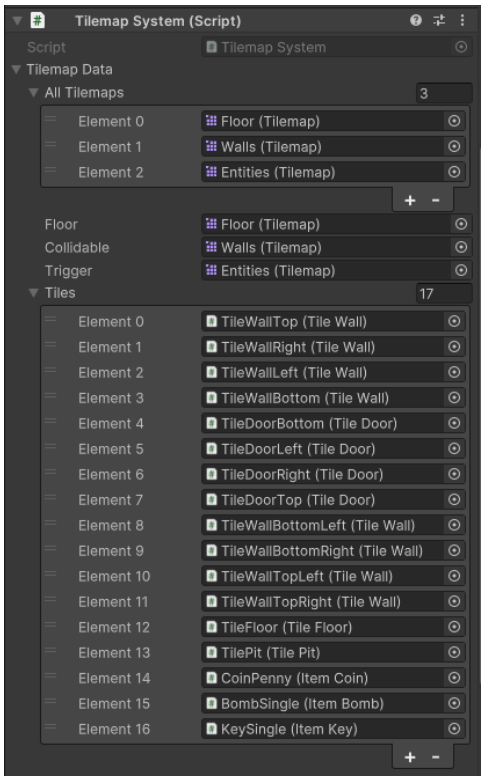


Figure 2. Unity Inspector UI of `TilemapSystem`, showcasing `Tile` list for PCG.

To address weaknesses in UTS with identifying the tile involved in collision/trigger, classes `FloorTilemap`, `CollidableTilemap`, and `TriggerTilemap` resolve the tile by identifying the cell position of the collision and checking the `Tile` 2D array for the `Tile` at this position. Polymorphism then provides the correct method execution from the tile identified at the hit position (`Explore`, `Collide`, or `Interact`).

5.2.4 PCG initial development

After implementation of core functionality required for the PCG tool, development of PCG systems and methods was prioritised. Initial implementation of the two major components, `PCGSystem` and `PCGMethods` was prototyped alongside `Tile` and `Tilemap` system development. As a result, problems were identified in both components which led to eventual refactors.

Notably, PCGSystem contained both user changeable data (width/height of room), and all data related to tilemaps; this was problematic, as the class became bloated with both data and method handling for PCG. In PCGMethods, prototype development utilised only UTS, which the lack of features compared to Tile showed evidently in PCG results and the development of PCG methods.

5.2.5 PCG systems and methods

PCGSystem was refactored (PCGSystemRefactor) to act as an intermediary between the tile and tilemap data in the TilemapSystem, and PCG methods that generate content (PCGMethods). This was done to centralise the user control of PCG into a single system, with user control over tiles used in generation being separated to TilemapSystem. This also made development more manageable due to the more modular approach. A struct data container of the generated room is stored in the PCGSystem, which is a Tile 2D array. A 2D array was chosen for its coordinates layout of tiles matching the grid based tilemap in the artifact.

To give control to users, the PCG generation method can be specified (e.g. A* Pathfinding), along with the room sizing; this alongside tile input control in TilemapSystem gives a comprehensive set of parameters for generation, which are passed to PCGMethods.

PCGMethods (PCGMethodsRefactor) continued a similar approach to its previous iteration in separating generation logic into distinct and reusable methods, however, it now utilised custom Tile classes. Instantiation of various Tile derived classes are provided in methods and used for all PCG generation methods in the class. On a method call for generation (e.g. AStarPathFindingGeneration), room data and tilemap data are passed to the system, along with an optional offset and seed for shifting tile positions and random number generation (RNG) respectively. In methods such as AStarPathFindingGeneration, walls and doors are the first tiles generated and added to the Tile 2D array to ensure that PCG confines to the rules of a room layout, of four doors in the centre position of each room wall, and walls as the outermost tiles in a grid.

The generated room is returned as a RoomData struct, with the appearance on the tilemaps being updated on RenderRoom method call; this was modularised to make room rendering be independent from the type of generation used.

5.2.6 Methods of PCG

PCGMethods access different methods of PCG, which are modularised to different classes to keep the codebase organised. AStarPathfinding is the most functional algorithm method for PCG developed, and was prioritised under consideration of the artifact's timescale. Implementation of this method would be faster due to greater number of resources to reference, along with flexibility to different sized grids.

After being passed the current Tile 2D array, along with a list of door positions, AStarPathfinding ensures a valid path (non-collidable and non-trigger tiles) is created for each door to reach another door. This is done through node representation, where each point in the Tile 2D array is represented as a node storing its position, the sum of GCost (movement cost from start node) and HCost (Heuristic cost to the end node), and the nodes parent node. After node evaluation and neighbour exploration, a path is traced, and after all paths have been traced for each door, a list returning the path grid positions is passed to PCGMethods.

With the generated list, TileFloor tiles are generated at the path positions to ensure a valid (completable) room. For the rest of the grid, RNG, which might be weighted by a trained ML agent passing EngagementMetrics, determines the rest of the tiles generated for the room.

5.3 Development of ML systems

5.3.1 Implementation of Unity ML-Agents

The next stage of the artifact's development focused on ML. After research into possible implementation options, Unity ML-Agents proved most suitable for the artifact. This was due to easy incorporation into the project due to its plugin integration, and the pre-built frameworks for machine learning models, which sped up development significantly compared to alterative options such as bespoke.

After successful isolated testing of movement code, this code was transferred to the developing MLAgent system, in its Heuristic method inherited from ML-Agents Agent. Initial development used a heuristic behaviour type for the ML agent, which provided the valuable ability to control the agent with the keyboard during testing before ML had its implementation.

5.3.2 ML agent

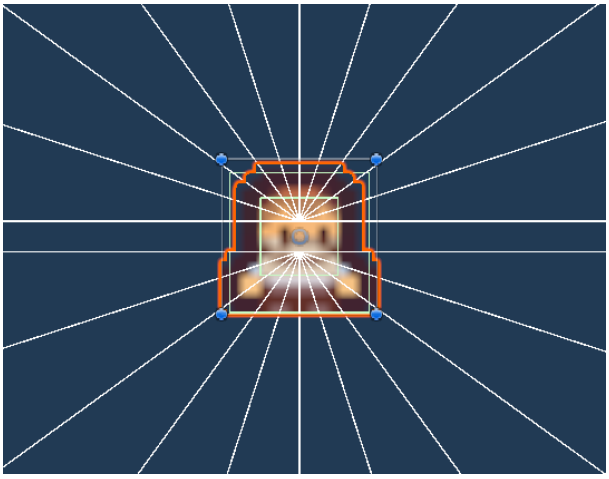


Figure 3. `MLAgent` prefab with 2D ray perception sensors, and multiple colliders.

Figure 3. illustrates the decision to include two colliders on the `MLAgent` prefab, the larger being used for physical collisions, and the smaller trigger collider for trigger collider detection. This was implemented in response to problems with registering trigger collisions in the `TriggerCollider` system, which would identify the wrong tile being triggered in its grid. This change made tile identification more accurate.

2D ray perception sensors were attached to the `MLAgent`, which covered 360 degrees of vision for the agent. These raycast sensors detect and mark hits on the tagged objects relevant to the agent, including obstacles, items, and doors. During development, problems with sensor `Layermasks` were identified, where raycasts would mark hits with the agent they were attached to. This was resolved by creating a new `Layermask` for the agent prefab that was not marked as a hittable layer.

Observations in an agent's environment are collected by these 2D sensors, along with observations added in `CollectObservations`. Critical information is added as observations which are eventually passed to `OnActionRecieved` for decision of actions. This information includes the agent's current local position, the agent's position on the tilemap, the distances between the agent and the nearest item/door/obstacle (tiles), and the tile position of those nearest tiles. These observations provide a wide-ranging list of data sources for the agent's neural network to use in decision-making.

The `OnActionReceived` method determines what action the agent should take based on observations taken of its environment. To speed up development and fine-tuning of ML, the actions available to the player (agent) was scoped to focus on movement. Actions such as combat were beyond the timescale of the project due to increased complexity of reward structures, and higher difficulty in achieving expected behaviours for trained agents.

`EngagementMetrics` of a currently playing room are stored in the form of a data struct owned by an `MLAgent`, which tracks overall engagement through adding exploration float data (floor tiles the agent moved in) and number of items picked-up data. This data is passed to the `PCGSystem` when an `MLAgent` is assigned to use a trained brain model for behaviour.

5.3.3 Training and re-training

With numerous ML methods available in ML-Agents, the artifact chose reinforcement learning as the method for training agents. This was due to the process of RL agents trying to maximise rewards being like how a player might try reward themselves in gameplay. Reward structures of an agent (e.g. item pickup) matched meaningful gameplay interactions that would be expected of a level, and these were passed on from the agent in the form of `EngagementMetrics` to PCG systems for altering of generation results.

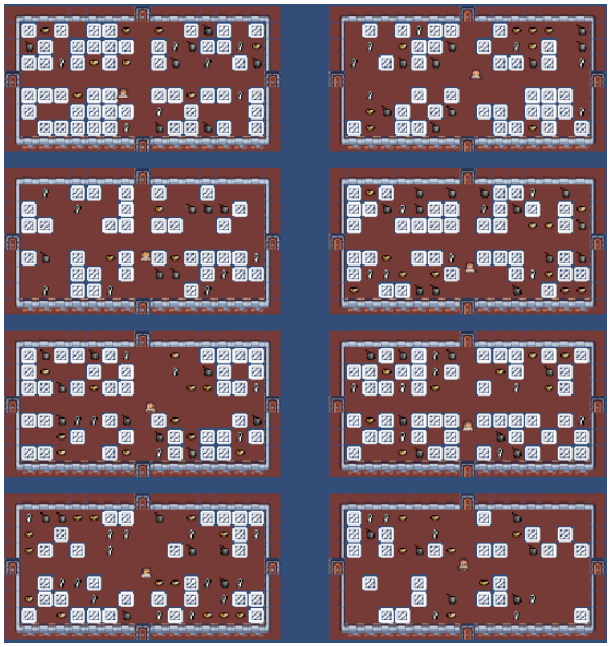


Figure 4. `PCG – Training Environment` training eight agents simultaneously.

Eight `Simulation` prefabs were run simultaneously during the training of agents, reducing the time taken to train agents, as seen in Figure 4. Each `Simulation` ran independently, with agent episodes ending with through an agent reaching its maximum time step of 5000 steps (actions), or through successful completion of a room through the triggering of a `TileDoor`. Allowing the completion of a room ensured agents would have a consistent end-state decision for their actions. Training sessions would

be ended upon average reward results remaining stable and levelled off, as further gains from training would be negligible.

5.3.4 Engagement results

The final element to implement of the artifact was incorporating `EngagementMetrics` into PCG. This required trained ML agents, as they substitute for human-testing in PCG results. `EngagementMetrics` influence the generation of rooms, acting as a "weighting" for random generation elements; for instance, a room will be more likely to generate more items if item pick-up engagement was low.

When a new room is to be generated in `PCGSystems`, a comparison is made to the previous rooms engagement result to the current highest engagement room generated. If the previous room generated higher engagement from an agent, the room's layout (`Simulation`) is saved to a prefab in Unity's asset folder. This room can then be imported into a Unity scene for playtesting.

5.4 Management of systems

To create consistent templates that manage the core systems of the artifact, a `Simulation` script and prefab was created, acting as an overall manager that can be dropped into any Unity scene. A `Simulation` prefab instance holds references to the three major systems of `MLSystem`, `PCGSystem`, and `TilemapSystem`. Simulations can be reset upon two conditions, a player/agent completes a level by triggering a door, or if during agent training, the agent's episode begins (once every 5000 steps). "Engaging" rooms are generated and saved as a `Simulation`.

5.5 Project management, documentation, code

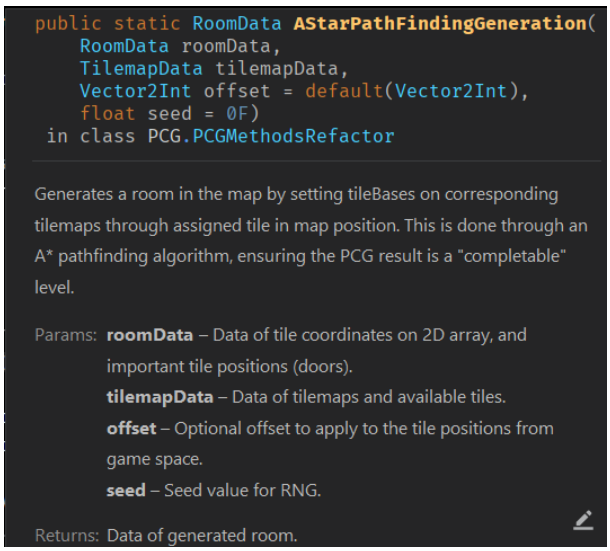


Figure 5. View from JetBrains Rider showing the integration of XML code comments and IDE.

In the development of the artifact, professional software development practices were employed to ensure efficient iterative development. Practices like clear and extensive documentation that integrates with integrated developer environments (IDE's), use of version control, atomic commits, full usage of GitHub project management tools like GitHub Issues and Projects, and adherence to C# standard style guides, ensured effective project development.

To maintain high consistency in code quality and readability, adherence to C# coding conventions like PascalCase for method/function signatures (e.g. `MyFunction`), and camelCase for member variables (e.g. `healthRemaining`) was maintained. These conventions ensured readable code for external readers familiar with C#. Code commenting was extensive, with integration with IDE's through tagging for methods, functions, and classes. Figure 5. provides an example of code commenting integration with JetBrains Rider (JetBrains, 2024).

Git (Linus Torvalds, 2005) was chosen for the version control system as its detailed history tracking and branching features provided necessary rollbacks in case of project development issues. Atomic commits (Hovhannisyan, 2021) were utilised for committing practices, as imperative language and small self-contained commits make tracking changes and issues easier to identify. Git was also chosen due to its use by GitHub, the chosen code repository hosting provider.

GitHub provides useful project management tools that were employed during development, like GitHub Issues, a tracker for bugs, enhancements, and other requests. GitHub Projects displays data related to Issues in common views, such as kanban style boards, Project Roadmaps, and GitHub tables, which made project tracking easier.

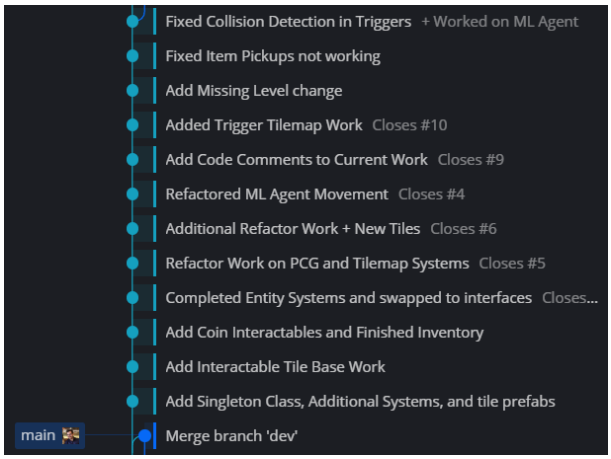


Figure 6. View from GitKraken of artifact commit history, showing the "Closing" feature.

GitHub Issues seamlessly integrates with GitKraken (GitKraken, 2015), the Git GUI client chosen for the project, through displaying currently opened issues, and the ability to reference and close issues directly through Git commit messages, allowing for effective tracking of tasks. Figure 6. illustrates the view of the artefact's repository from GitKraken's user-interface (UI).

6. Discussion of outcomes

To accurately conclude the outcomes of the artifact, the discussion will focus on the three project objectives and assess their level of achievement. The artifact aimed to deliver a PCG tool that generates "functional" (Summerville, 2018) levels that are completable, trained ML agents that replicate player behaviour in the level environment, and a system for improving PCG generated based on trained agent data.

6.1 Evaluation of PCG

6.1.1 PCG tool

In analysis of PCG results, the first element is usability and extendibility of the PCG tool. High levels of modularisation of code and in-depth documentation enhances the practical applicability of the artifact as a development tool to be imported into game development projects. Unity features such as prefabs make for easier importing of simulations in game scenes, and Unity `Header` and `Tooltip` provide clarity to not immediately obvious elements of the Unity Inspector, which were effective in development as a form of documentation.

Developers are provided with an accessible code base that would be easy to expand with further PCG implementations and possible automation of ML testing with unit tests. Unit testing is the process of testing individual parts of a software, often autonomously, to ensure they meet test case expectations (GeeksForGeeks, 2024).

6.1.2 PCG results

Withington (2024) conducted a survey on the most common metrics and features used in the assessment of PCG levels. For the purposes of the evaluation, the relevant metrics that will be used in PCG result assessment are:

- Level Fitness – Single numerical figure to quantify the quality of the level. Factors of consideration are the diversity of items, placement of obstacles, and exploration opportunities in the form of empty tiles. Numerical figures 1 – 10.

- Solvability & Win Rate – A binary flag if the level can be completed or not. Pass or fail.

For the evaluation of ML in PCG, the engagement score was also noted. However, this not be used in the evaluation of the PCG results.

Level	Level Fitness	Solvability & Win Rate	Engagement Score	Fitness vs. Engagement Variance
	6	Pass	2 (1.18)	-4.82
	7	Pass	5.5 (3.24)	-3.76
	4	Pass	10 (5.88)	1.88

	3	Pass	12 (7.06)	4.06
	8	Pass	17 (10)	2

Table 1. Comparison of generated room layouts under Withington's (2024) common PCG assessment metrics.

The results from Table 1. above highlight the success in the implemented PCG methods for generating a full, completable, level with expected interactable elements and obstacles present. All levels generated ensure a pathway to doors for level completion, and this element of PCG objectives was successful. However, limitations can be seen in some generations, for instance, the "cluttered" appearance of engagement score rooms 10 and 12, with some items being inaccessible to a player. Breakable obstacles, which was out of scope for the development of the artifact, would have mitigated this issue in generation.

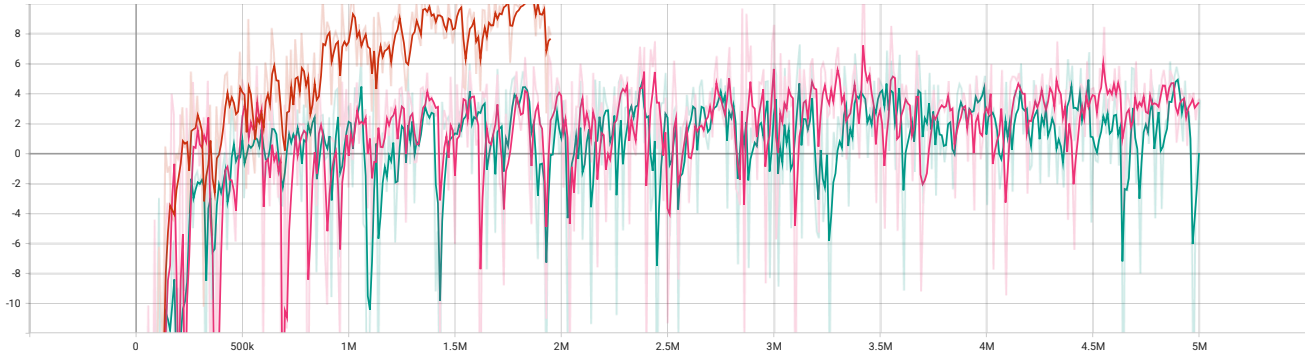
It is worth note, that Withington (2024) themself acknowledge that this data cannot be the sole identifier of the quality of individual generations, and that quantitative evaluation might need a new framework entirely.

6.2 Evaluation of ML

6.2.1 ML agent behaviour

Evaluation of the behaviour of agents during training is crucial to the measurement of success of the artifact. The first successful training session, session 2 (orange) took 1.82 million steps (actions) to achieve desired agent behaviour. During their training, agents were able to identify the flaws of the A* pathfinding generation algorithm. As RL encourages agents to maximise their rewards per episode, agents quickly realised (around 750,000 steps) that a valid path to a highly rewarding door would always be present, and directly in sight of their 2D ray perception sensors. They quickly adopted the behaviour of getting to a level door as quickly as possible, possibly collecting items if nearby on their route. The risk of hitting an obstacle and receiving a negative reward was too great.

To counteract this, new implementations of sessions 5 (pink) and 7 (green) were made that applied exploration rewards for an agent, along with tweaking of reward values for hitting obstacles and picking up items. Observation of their training, along with data from Graph 1., suggested that the previous sessions issue had been mitigated, with agents exploring rooms far more to achieve exploration rewards. However, a new issue was presented, as agents neglected to complete levels by going to a door. Slight trends upwards beyond the 500,000 step range were due to agents being more effective at exploring and picking up items, and was not due to completing levels through door exit.



Graph 1. Agent rewards in training for steps taken. *X axis:* Number of steps (actions). *Y axis:* Reward. Training session 2 (orange), training session 5 (pink), training session 7 (green)

The tweaking of hyperparameters, like `curiosity`, or editing of reward structures, such as gradual rewards for getting closer to a door, could have helped alleviate the issues presented in sessions 5 and 7.

6.2.2 ML training outcomes

In analysis of the effectiveness of ML training and the resulting outputs of the trained agents, visual observation of their behaviour and data on their average rewards was used for evaluation.

As Graph 1. presents, it is evident that training RL agents quickly identify environmental features that positively/negatively reward themselves. The training sessions all shared significantly sharp increases in rewards during the early training steps, during the 0–500,000 steps range. Initially, agents would end episodes with more negative rewards than positive, due to colliding with obstacles often, which was confirmed in visual observation of agents during their training. However, these negative rewards quickly identified to agents the necessity to avoid collision with obstacles, and their behaviour adjusted accordingly. This met an expected outcome for their end behaviour, which is obstacle avoidance.

From the 0-500,000 step range onwards, training sessions saw fluctuation in rewards, most evident in training sessions 5 (pink) and 7 (green). Visual observation in training identified agent's difficulty in being able to observe its whole environment. 2D ray perception sensors would hit the first valid object it encountered, meaning important information such as items behind obstacles tiles would not be known to an agent. Whilst this was considered, and mitigation attempted through the agent tracking its nearest door/item/obstacle position, an observation of the whole environment (e.g. the tilemap grid itself, with each `Tile` position observed) would have led to more effective agent training and behaviour.

From the results, weaknesses of RL were identifiable. The most notable being the extended periods of time required to get expected agent behaviour, as this often requires minor changes to code which would take time to reflect during a new training. In a professional context, this time intensive set-up would need to be evaluated, as the time taken to fine-tune agent behaviour might exceed the time saved from not requiring human testing of PCG results.

6.3 ML in PCG evaluation

To evaluate the effectiveness of ML in influencing PCG outputs to create more engaging levels, the outputs of engaging levels were saved as prefabs for comparative visual and layout analysis. This was done under the same framework as PCG results evaluation, based on Withington (2024).

The highest recorded engagement score by an agent was 17, whereas the highest-level fitness score can be a 10. In consideration of this, variance scores were calculated under the treatment of an engagement score of 17 being the fitness equivalent of a 10. $(\frac{engagement\ score}{highest\ engagement\ score\ (17)} \times 10)$

Using Table 1. for evaluation, human engagement metrics fluctuated between high and low variance compared to agent assessment. Notably, rooms with higher human engagement scores followed layout trends of larger, open-spaces with all items being fully accessible to a player. These preferred rooms would often have higher levels of variance, as the samples from Table 1. highlight.

In contrast, upon visual observation of agent behaviour and influence on PCG generation, higher engagement rooms shared common elements of higher item density rooms, and items not obscured by obstacles. In rooms where the agent was unable to pickup items obscured by obstacles, item generation will increase greatly for the next room generated, which would lead to less obstacles to hinder engagement.

The results evidently show the difficulty in quantifying engagement metrics to a programmable metric of assessment. "Engagement" is inherently a subjective, human concept. Whilst agents might prioritise rewards based on their predefined reward structures, this does not fully translate to human engagement. A broader range of engagement metrics could have been provided to establish a better understanding of what makes a level engaging, for example, exploration variety in the form of branching paths to the same end point.

In hindsight, elements of human testing of ML-influenced PCG levels would have flagged the validity of the results the agent was providing for engagement. If results were inaccurate to what a human might deem engaging (such as the variance score in Table 1.), the ML system could be updated accordingly. Additionally, the feeding of "engaging" and "non-engaging" level data to the ML system (as Withington's (2024) metric of "similarity to Training Data" describes) would have helped to establish the accuracy between `EngagementMetrics` and user-perceived engagement.

7. Conclusion and recommendations

The project aimed to investigate the application of ML in the evaluation of PCG outputs. The findings and results presented showed the challenges of creating "functional" game content with PCG, and highlighted the necessity for more advanced PCG methods such as WFC to generate

this content. The ability to accurately quantify engagement as a programmable metric was a limitation as the findings show, as engagement leans towards subjectivity. Despite this, the findings highlight the strengths of algorithms like A* pathfinding in ensuring completable levels, along with the strengths of ML in being able to learn in its environment from observations and reward structures, and alter its behaviour accordingly.

The next logical steps for the project would involve implementation of more advanced PCG generation methods such as Wave Function Collapse and Cellular Automata. The core functionality developed in the current artifact would provide an effective base for these methods. Development of WFC and CA would enhance the number of PCG options to a user, and provide more sophisticated room layouts. These PCG methods would also allow for comparative analysis of PCG results to identify the effectiveness of methods in different scenarios (e.g. Rooms that are more challenging, rooms that are less rewarding, etc.).

As surmised from Summerville (2018), as the importance of PCG increases in game development, so too does the necessity for higher quality PCG results, with or without human involvement. With ML still being a relatively new paradigm for exploration in incorporation for PCG, the impact of this project would be most applicable as a continued research survey in an academic journal.

After further research and development, either through academic journals or open sourcing the code under a CC0 license, the project's commercial potential is high. Rough-like games would see the most benefit from greater engagement and randomness in their content delivery, whilst other game genres might see applicability in the PCG or ML implementations, and incorporate it into their own code.

The project successfully demonstrated the incorporation of machine learning into the evaluation of output results of procedural content generation, and highlights machine learning's strengths in autonomous replication of human behaviour. The project is viable for open-sourcing, so that users can continue development of the PCG and ML systems present, either for research or commercial purposes.

8. References

A.I. Design (1980) *Rogue*. [Video game]. Epyx. Available from: https://store.steampowered.com/app/1443430/Rogue/ [Accessed 04 July 2024].

Adams, C. and Louis, S. (2017) Procedural maze level generation with evolutionary cellular automata. *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. Honolulu, 2017. pp. 1-8. [Accessed 04 July 2024].

Bay 12 Games (2006) *Dwarf Fortress*. [Video game]. Bay 12 Games. Available from: https://www.bay12games.com/dwarves/ [Accessed 03 July 2024].

Creative Commons (2024) *CC0*. Available from: https://creativecommons.org/public-domain/cc0/ [Accessed 01 July 2024].

Creature Labs (1996) *Creatures*. [Video game]. Warner Interactive Europe. [Accessed 06 July 2024].

Dodge Roll. (2017) *Enter the Gungeon.* [Video game]. Devolver Digital. Available from: https://store.steampowered.com/app/311690/Enter_the_Gungeon/ [Accessed 21 April 2024].

Doull, A. (2008) The Death of the Level Designer: Procedural Content Generation in Games – Part One. *Ascii Dreams* [blog]. 14 January. Available from: https://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html [Accessed 04 July 2024].

Edmund McMillen, EM. (2011) *The Binding of Isaac.* [Video game]. Available from: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/ [Accessed 21 April 2024].

Esaki, C. (2023) *Forza Motorsport's New AI and Physics Make Every Race Competitive*. Available from: https://forza.net/news/forza-motorsport-drivatars-tire-physics [Accessed 03 July 2024].

GeeksforGeeks (2024) *Unit Testing – Software Testing*. Available from: https://www.geeksforgeeks.org/unit-testing-software-testing/ [Accessed 16 July 2024].

GitHub (2024) *GitHub*. Available from: https://github.com/ [Accessed 10 July 2024].

GitKraken (2015) *GitKraken* (10.0.0) [computer program]. Available from: https://www.gitkraken.com/ [Accessed 10 July 2024].

Gumin, M. (2016) *WaveFunctionCollapse* (1.00) [computer program]. Available from: https://github.com/mxgmn/WaveFunctionCollapse [Accessed 06 July 2024].

Hovhannisyan, A. (2021) Make Atomic Git Commits. *Aleksandr Hovhannisyan Blog* [blog]. 15 May 2021. Available from:

https://www.aleksandrhovhannisyan.com/blog/atomic-git-commits/ [Accessed 10 July 2024].

Immersive Limit LLC (2020) *ML-Agents: Hummingbirds*. Available from: https://learn.unity.com/course/ml-agents-hummingbirds?uv=2019.3 [Accessed 18 March 2024].

JetBrains (2024) *Rider* (2024.1) [computer program]. Available from: https://www.jetbrains.com/rider/ [Accessed 10 July 2024].

Kaelbling, L. and Littman, M. (1996) Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* [online]. 4. [Accessed 04 July 2024].

Kazemi, D. *Spelunky Generator Lessons*. Available from: https://tinysubversions.com/spelunkyGen/ [Accessed 04 July 2024].

Kenney (2022) *Tiny Dungeon*. Available from: https://kenney.nl/assets/tiny-dungeon [Accessed 01 July 2024].

Kühl, N. (2022) Human vs. supervised machine learning: Who learns patterns faster?. *Cognitive Systems Research* [online]. 76, pp. 76-92. [Accessed 04 July 2024].

Linus Torvalds (2005) *Git* (2.45.2) [computer program]. Available from: https://git-scm.com/ [Accessed 10 July 2024].

Mojang Studios (2011) *Minecraft*. [Video game]. Mojang Studios. Available from: https://www.minecraft.net/en-us [Accessed 03 July 2024].

Mossmouth. (2008) *Spelunky.* [Video game]. Mossmouth. Available from: https://store.steampowered.com/app/239350/Spelunky/ [Accessed 23 April 2024].

Plausible Content (2018) *Bad North*. [Video game]. Raw Fury. Available from: https://www.badnorth.com/ [Accessed 06 July 2024].

Procedural Content Generation Wiki (2016) *Rogue*. Available from: http://pcg.wikidot.com/pcg-games:rogue [Accessed 04 July 2024].

Shaker, N. and Togelius, J. (2016) *Procedural Content Generation in Games* [online]. Springer Cham. [Accessed 02 July 2024].

Skinner, G. and Walmsley, T. (2019) Artificial Intelligence and Deep Learning in Video Games A Brief Review. *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*. Singapore, 2019. pp. 404-408. [Accessed 04 July 2024].

Summerville, A. and Snodgradd, S. (2018) Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* [online]. 10(3), pp. 257-270. [Accessed 02 July 2024].

Turn 10 Studios (2017) *Forza Motorsport 7*. Standard Edition. [Video game]. Microsoft Studios. Available from: https://www.xbox.com/en-GB/games/store/forza-motorsport-7-standard-edition/9n3nk5ww05ht [Accessed 06 July 2024].

Unity Technologies (2004) *Unity Machine Learning Agents* (Version 3.0.0) [computer program plugin]. Available from: https://unity.com/products/machine-learning-agents [Accessed 18 March 2024].

Unity Technologies (2005) *Unity* (2022.3.6f1) [Computer program]. Available from: https://unity.com/ [Accessed 23 April 2024].

Unity Technologies (2005) *Unity* (2022.3.6f1) [Computer program]. Available from: https://unity.com/ [Accessed 3 March 2024].

Unity Technologies (2017) *Unity Machine Learning Agents* [Computer program]. Available from: https://unity.com/products/machine-learning-agents [Accessed 3 March 2024].

Withington, O. and Cook, M. (2024) On the Evaluation of Procedural Level Generation Systems. *Proceedings of the 19th International Conference on the Foundations of Digital Games*. pp. 1-10. [Accessed 16 July 2024].

9. Bibliography

BUas Games (2018). EPC2018 – Oskar Stalberg – Wave Function Collapse in Bad North. *YouTube* [video]. 11 July. Available from: https://youtu.be/0bcZb-SsnrA?si=iy6Y0E3KNcGJGvx2 [Accessed 04 July 2024].

Gamma, E. and Helm, R. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software* [online]. Addison-Wesley Professional. [Accessed 02 February 2024].

Kim, H. and Lee, S. (2019) Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm. *2019 IEEE Conference on Games (CoG)*. London, 2019. pp. 1-4. [Accessed 04 July 2024].

PavCreations (2020) *Tilemap-based A* algorithm implementation in Unity game*. Available from: https://pavcreations.com/tilemap-based-a-star-algorithm-implementation-in-unity-game/#how-a-star-algorithm-works-the-basics [Accessed 12 July 2024].

Procedural Content Generation Wiki (2017) *Dungeon Generation*. Available from: http://pcg.wikidot.com/pcg-algorithm:dungeon-generation [Accessed 04 July 2024].

Shao, K. and Tang, Z. (2019) A Survey of Deep Reinforcement Learning in Video Games. *ArXiv* [online]. Available from: https://arxiv.org/abs/1912.10944 [Accessed 04 July 2024].

Unity (2020). Kart Racing Game with Machine Learning in Unity! (Tutorial). *YouTube* [video]. 11 January. Available from: https://www.youtube.com/watch?v=i0Vt7l3XrIU [Accessed 18 March 2024].

Zucconi, A. (2022) *Minecraft Biome Generation Explained Using Stochastic Cellular Automata*. Available from: https://www.reddit.com/r/cellular_automata/comments/v36xjp/minecraft_biome_generation_explained_using/ [Accessed 04 July 2024].

Zucconi, A. (2020) *The AI of Creatures*. Available from: https://www.alanzucconi.com/2020/07/27/the-ai-of-creatures/ [Accessed 06 July 2024].

Appendix A: Project Log

Date of entry	Task	Summary	Additional Comments
11/01/2024	1. Add ML agents	Added ML-Agents package to project. Set up Anaconda environments for future training.	
24/01/2024	1. Player Movement 2. PCG Perlin Noise generation method	Player movement added to play game scenes. To test Unity Tilemap System (UTS), perlin noise generation method added. Planned to be scrapped for random (but completable level) algorithm first.	
26/03/2024	1. Tile Custom Classes in PCG Methods 2. Door checking in PCG	Development of custom Tile classes that extend functionality of UTS. Abstract so its not accidently instantiated by itself. Added checks for door locations to fix issue.	
28/03/2024	1. Meeting with Supervisor 2. Custom Tile work continued.	Showed current state of the artifact. Very much behind development and personal goals for the project due to personal circumstances. Current code is well written and implemented, but needs far more done. Requested helping writing for additional time. Added items and inventory systems, and development of interactable tile elements.	Help with email to Tom. Recommendation to start writing report now. Take research from user testing as deliberately limiting the number of variables.
24/05/2024	1. Action plan regarding deadline.	Personal circumstances and high workloads in other modules set back progress in the artifacts development. Established plan of tasks to complete before July.	12/04/2024 meeting had to be cancelled with supervisor due to doctors appointment.
10/06/2024	1. Dissertation Report	Project development continued after burn-out break. Report	

		writing for introduction and research methods. Began collating references in separate document.	
26/06/2024	1. Entity tiles	Entity tiles (items) work completed. Utilise interfaces to allow for multiple inheritenance.	
29/06/2024	1. Refactors of various systems 2. ML agent behaviour tested.	PCGSystems, PCGMethods, and tilemap systems fully refactored. Previous implementations now deprecated. Tested ML agent behaviour in controlling the player. More of the systems surrounding need development before work can be focused on ML.	
01/07/2024	1. Dissertation Report	Report writing continued. Collated references for literature review to begin writing section.	
04/07/2024	1. Dissertation Report	Literature review complete, and non-practice and conclusion sections.	
09/07/2024	1. Trigger tilemap for interactions	Dissertation report writing continued to this date.	
12/07/2024	1. Collision detection fixes 2. Fix critical bugs	Fixed issues in collision detection with trigger items, as coordinates would not be accurate to the collision position translated in cell position. Fixed critical bugs were scripts were being directly modified, and these modifications would be passed permanently to other prefabs and objects.	
14/07/2024	1. ML Agent Training successfully complete	Trained ML agent output achieved. Will walk to doors to complete levels, avoid obstacles, and pickup items. Twitchy behaviour and too focused on beating levels.	
17/07/2024	1. Engagement metrics 2. Reward structure tweaks	Added engagement metrics that assess played level. Used to weight generation of next room. Current metrics of items picked up and exploration. Rewarding mechanisms tweaked for agent to optimise behaviour. Rewarded for exploration.	
18/07/2024	1. Dissertation report 2. Artifact final tweaks	Dissertation conclusion and notes complete. Added final code comments, and saving for high engagement rooms.	
19/07/2024	1. Dissertation report 2. Artifact final tweaks	Dissertation transferred from note layout to required format. Final small bug fixes and project cleanup. Bug found with weightings, and fixed.	

Appendix B: Project Timeline

Title	Number	Status	Assignees	Labels	Author	CreatedAt	ClosedAt	Type	State
Create Tilemap System	1	Done	Kieran De Sousa	feature	Kieran De Sousa	28/03/2024 13:35	27/06/2024 23:25	ISSUE	CLOSED
Create Trigger Collider Tilemap Script	10	Done	Kieran De Sousa	feature	Kieran De Sousa	30/06/2024 13:02	09/07/2024 23:44	ISSUE	CLOSED
Add observations, OnEpisodeBegin TO MLAgent	11	Done	Kieran De Sousa	feature	Kieran De Sousa	12/07/2024 14:48	14/07/2024 20:01	ISSUE	CLOSED
Make system base class	12	Done	Kieran De Sousa	feature	Kieran De Sousa	12/07/2024 15:10	12/07/2024 18:47	ISSUE	CLOSED
Tweak A* Generation	13	Done	Kieran De Sousa	feature	Kieran De Sousa	13/07/2024 22:05	18/07/2024 14:07	ISSUE	CLOSED
Finish reward functionality	14	Done	Kieran De Sousa	feature	Kieran De Sousa	14/07/2024 20:01	17/07/2024 22:29	ISSUE	CLOSED
Fix Tile 2D array not being passed to children	15	Done	Kieran De Sousa	bug	Kieran De Sousa	18/07/2024 11:01	19/07/2024 16:50	ISSUE	CLOSED
Add final code comments	16	Done	Kieran De Sousa	documentation	Kieran De Sousa	18/07/2024 15:39	18/07/2024 17:59	ISSUE	CLOSED
Add Bootstrapper	2	Done	Kieran De Sousa	feature	Kieran De Sousa	28/03/2024 13:41	28/03/2024 22:26	ISSUE	CLOSED
Complete Entity Systems	3	Done	Kieran De Sousa	feature	Kieran De Sousa	25/06/2024 21:21	26/06/2024 17:54	ISSUE	CLOSED
Merge Inventory & Movement System --> ML Agents	4	Done	Kieran De Sousa	feature	Kieran De Sousa	25/06/2024 21:23	30/06/2024 13:03	ISSUE	CLOSED
Add Centre Position, and Door Locations as public variables	5	Done	Kieran De Sousa	feature	Kieran De Sousa	26/06/2024 18:48	27/06/2024 23:24	ISSUE	CLOSED
Complete Generate Doors and Generate Walls methods	6	Done	Kieran De Sousa	feature	Kieran De Sousa	27/06/2024 23:26	29/06/2024 19:44	ISSUE	CLOSED
Swap to Tile Map Implementation in PCG Methods Refactor	7	Done	Kieran De Sousa	feature	Kieran De Sousa	27/06/2024 23:27	12/07/2024 14:44	ISSUE	CLOSED
Implement A* Pathfinding Method	8	Done	Kieran De Sousa	feature	Kieran De Sousa	27/06/2024 23:29	13/07/2024 21:22	ISSUE	CLOSED
Add Code Comments to current work	9	Done	Kieran De Sousa	documentation	Kieran De Sousa	29/06/2024 19:45	30/06/2024 13:49	ISSUE	CLOSED

Appendix C: Assets used in the Project

Tile assets: Kenney (2022) *Tiny Dungeon*. Available from: https://kenney.nl/assets/tiny-dungeon - CC0 license.

Tile assets: Edmund McMillen, EM. (2011) *The Binding of Isaac.* Available from: https://store.steampowered.com/app/113200/The_Binding_of_Isaac/ - CC BY-NC-SA 3.0 license (from *The Binding of Isaac Rebirth Wiki*).