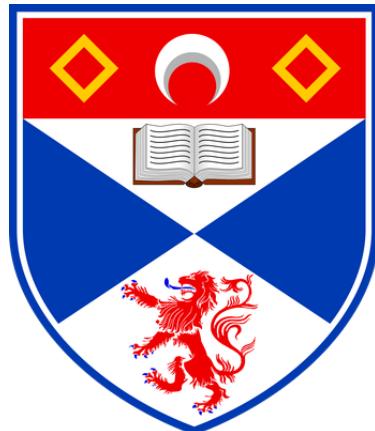


Generating Dynamic Virtual Environments Using Large Language Models

CS5199: Individual Masters Project

Joel Sebastian



Department of Computer Science
University of St Andrews

Supervisor: Dr. Tristan Henderson

Abstract

This research explores the novel application of Large Language Models (LLMs) for generating diverse and engaging terrains within virtual environments, a domain largely unexplored in existing literature. While previous studies have leveraged LLMs for game-related tasks, such as level design and in-game dialogue generation, they primarily focus on discrete, 2-D level creation or specific in-game elements. In contrast, this work delves into the generation of continuous, 3-D terrains, a significantly more complex challenge.

The research addresses the complexities of generating terrains, including simulating natural formations, handling varying elevations, and integrating diverse environmental features. Furthermore, it investigates the interplay between terrain generation and the placement of 3-D structures within the generated environment, extending beyond the scope of existing research that often assumes a pre-defined flat terrain.

This research emphasises the importance of qualitative evaluation, incorporating a user study and subsequent feedback collection to assess the player's enjoyment and engagement with the generated terrains. Additionally, it explores the multimodal capabilities of LLMs, investigating potential approaches for self-assessment and iterative refinement of the generated content.

By pioneering the use of LLMs for 3-D terrain generation, prioritising qualitative evaluation, and developing strategies for mitigating hallucinations, this research contributes significantly to the field of LLM-powered game development, paving the way for more immersive and player-centric virtual environments.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Tristan Henderson, for deciding to take on my project proposal and providing invaluable guidance and support throughout the course of this research. I had many a “Eureka!” moment during our meetings, and I am grateful for the time you took to listen to my ideas and provide constructive feedback.

To my dear family who have supported me my whole life, I am grateful for your unwavering love and encouragement. I would not have made it this far without you.

For my friends who always had a moment to spare for me, let me spitball ideas out of nowhere, and provided me with the much-needed distraction from the stress of this project, I am thankful for your presence in my life.

To the participants of my user study, who took time out of their busy schedules to play the game and provide feedback, I am grateful for your willingness to help me with my research.

To the CS Support Team who put up with many of my requests and provided me with the resources I needed to complete this project, I am thankful for your assistance.

Finally, to the University of St Andrews, for providing me with the opportunity to undertake this research project and for the materials and support that have enabled me to complete it.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is around 21,000 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

Abstract	1
Acknowledgements	2
Declaration	3
Introduction	6
Context Survey	8
AI and Content Generation	8
AI and Content Generation in Games	9
Evaluation	10
Summary	11
Requirements Specification	12
Objectives	12
<u>Primary</u>	12
<u>Secondary</u>	12
<u>Tertiary</u>	13
Requirements	14
Functional Requirements	14
Non-Functional Requirements	14
Software Engineering Process	16
Phases of Development	16
Agile in Action	17
Evidence of Agile Success	18
Ethics	19
Design	20
Choosing the LLM	20
Prompt Engineering	22
First Attempt (Low-Level Approach)	24
Second Attempt (High-Level Approach)	26
Dynamic Content Generation	28
Implementation	30
Mod Structure	31
Core Item Functionality	31
Prompting	35
Iterations and Hallucinations	35
Automation in the Low-Level Approach	40
Tuning the Model	41
Dynamic Content Generation	42
Taxonomy of Hallucinations	45
Hallucination Handling	47
Terrain Generation Algorithms	48
Evaluation	54
Qualitative Evaluation	54

Experiment Design	54
Results	59
Discussion	64
Quantitative Evaluation	68
Results	68
Discussion	77
Achievement of Objectives and Requirements	78
Objectives	78
Requirements	79
Critical Appraisal	81
Conclusions	82
Key Achievements	82
Significant Drawbacks	82
Future Work	83
Appendix	88
Appendix A - Low-Level Prompting	88
Appendix B - High-Level Prompting	90
B.1 - Iterative Terrain Generation	90
B.2 - Terrain Extension	101
Appendix C - Qualitative Evaluation	103
C.1 - Larger Feedback Form Questions	103
C.2 - Example User Data	107
Appendix D - Quantitative Evaluation	108
D.1 - Block Frequency Data For Each Terrain	108

Introduction

Since the emergence of Large Language Models (LLMs), popularised by OpenAI’s ChatGPT (GPT-3) [1], there has been a surge in research dedicated to their utilisation in virtual environments — digital spaces the likes of simulations and video games. This project explores the application of LLMs in generating game content within the game Minecraft (as a proof of concept). The aim is to develop a framework that bespeaks its broader application in other virtual environments.

Current AI-driven game generation methods are limited in scope; beyond generating content statically (before the player steps foot in the game), this framework demonstrates dynamic content generation, i.e., as the player explores the game world, any further generation is based on the existing world context. Previous Artificial Intelligence (AI) solutions in this space have been limited to creating “one-off” 2-D levels for games like Super Mario Bros [2], or even for Minecraft, generating buildings and structures off of textual descriptions [3, 4]. Using precise, structured prompt engineering we can leverage the context caching and understanding capabilities of these models to extend entire worlds by generating outputs that can be parsed and rendered in-game.

The primary aims of this project are:

1. To create a reusable framework for dynamic content generation in Minecraft, employing LLMs to generate and extend terrain as players explore.
2. To tune the models with information about Minecraft and how it originally structures its terrains.
3. To evaluate the effectiveness of this approach through gameplay sessions, capturing player feedback on enjoyment and perceived consistency.
4. To identify and mitigate common hallucination patterns in LLM outputs, particularly as content generation scales.

Inspired by media such as the *Sword Art Online* series [5] and recent papers on game generation via LLMs, this research aims to push the boundaries of procedural generation by integrating the nuance and recall possible with LLMs. *Sword Art Online* places a heavy emphasis on the concept of a virtual world, where an AI is able to generate content on-the-fly based on the player’s actions and the existing world’s state, rules, and history. The ultimate goal is to develop a method for creating expansive, consistent game worlds that could revolutionise open-world game development and other virtual environment applications.

The initial phase focused on recreating and understanding existing content by developing a framework for input prompts to Google Gemini [6], as well as parsing block data within Minecraft into a format that the chatbot could interpret, and then produce coherent output within the game. From this, a terrain reminiscent of the original game’s generation, filled with the same blocks, structures and features, could be shown in-game with some level of “thought” having been applied by the model as to their general placement, in accordance with the rules and design patterns understood by the model from tuning on Minecraft data. This output was refined iteratively using feedback from both a human-in-the-loop [7] (at least initially) and the model’s self-assessment.

Even from this phase, we can start forming a taxonomy of hallucinations, referring to the possible erroneous results that the model may produce. In the field of LLMs, this is the most predominant and inevitable issue [8], and was almost certain to haunt this project. A hurdle magnified in the next stage where extensions to the terrain were requested — the likelihood of hallucination naturally increased, the further we extended the terrain.

I evaluated the generated worlds by combining quantitative metrics with qualitative player feedback to determine whether they aligned with the game's original design principles and provide an enjoyable experience. This research demonstrates how LLMs can serve as tools for scalable, immersive, and player-adaptive content generation in virtual environments, paving the way for their broader adoption in game development.

Context Survey

There is a lack of research on using Large Language Models (LLMs) in the context of generating content such as landscapes/terrains within virtual environments, the likes of which are seen in video games and simulations. Recent research into LLMs has been focused on use cases such as the handling of complex scientific experiments [7], or more pertinently, within a gaming context [9] for enriching the player experience by providing more engaging and dynamic dialogue, creating new games & levels based on templates [2, 10–15], even replacing the player altogether [16].

AI and Content Generation

There is ample debate on the definition of Artificial Intelligence (AI) [17], commonly understood as the simulation of human intelligence in machines that are programmed to think, reason, and learn. Traditional AI typically involves rule-based systems or algorithms designed for specific tasks, while modern approaches leverage machine learning, particularly deep learning, to enable models to learn from data and adapt to new tasks [18]. This distinction is central to the progression from early systems like ELIZA [19] to contemporary Generative Pre-Trained Transformers (GPTs) [20], which form the foundation of modern large language models (LLMs).

The term ‘GenAI’ (Generative Artificial Intelligence) coined in 2023 [21] refers to applying unsupervised or self-supervised machine learning to a data set in order to generate potentially multiple forms of content, e.g., text, images, and audio. This is not completely representative, however, of the distinct other ways AI can be used for this purpose — “traditional” AI (without the learning aspect) has been used for content generation for decades, and well-known examples include: ELIZA [19], a natural language processing conversation simulator; AARON [22], a drawing program; and Ken Perlin’s Improv [23], a system for animating actors in real-time with behaviours.

Most recently, since the introduction of Generative Pre-Trained Transformers (GPTs) [20], which are trained on a large dataset of many, often unrelated, topics in order to form a general understanding, there has been a surge in the research and usage of LLMs for content generation, with LLMs being GPTs that have been fed on billions of data points, and have seemingly limitless potential when it comes to multimodal input and output. Radford *et al.* [20] demonstrated that task-agnostic, unsupervised pre-training allows for the model to be very general-purpose, and can then be fine-tuned for specific tasks; making use of a multi-layer Transformer Decoder architecture allowed the model to be strong on tasks such as syntactic parsing, and there was relatively more structured memory than other architectures for handling long-range dependencies. Results from their evaluation showed that the model was able to outperform then specifically-trained state-of-the-art models on a variety of language tasks, even in zero-shot settings (no prior knowledge), as the models have a greater understanding of higher-level semantics rather than just word-level information.

Despite the promise (and delivering) of increased adaptability, generalisation, and emergent behaviour, current LLMs suffer from ‘hallucinations’ [24] due to a lack of grounding — instances where the model produces an output that is blatantly incorrect or not according to the information they were provided and have access to. Boiko *et al.* [7] addressed this by having the LLM agent (the “Planner”) have access to and consult a database of documentation as well as search the Web, in order to increase its accuracy of responses; they also recommend a human-in-the-loop approach, which would ensure greater safety and reliability, as well as an opportunity to provide feedback to the model.

AI and Content Generation in Games

Currently, the most common methods for generating terrains in games [25] are through the use of Procedural Content Generation algorithms (PCG) [26, 27], and to a lesser extent, through the use of machine learning techniques such as Generative Adversarial Networks (GANs) [28], Variational Autoencoders (VAEs) and Convolutional Neural Networks (CNNs).

PCG has been used extensively in games for all sorts of purposes [26] since their inception, whether it be for providing infinite replayability, reducing the amount of manual work required, or creating content that is tailored to the player’s preferences. Specific examples include: ‘RadiantAI’ by Bethesda Softworks, introduced in The Elder Scrolls IV: Oblivion for creating dynamic schedules and behaviours for NPCs, and then extended in The Elder Scrolls V: Skyrim to include limitless quest creation [29]; the Borderlands series, which uses PCG to create billions (in Borderlands 3) of weapons and items [30, 31] that are unique in their stats, appearance, and abilities; and the case of AI Dungeon, which uses PCG to create text-based adventures that are unique to each player [32].

PCG can drastically reduce the strain on developers and provide a more engaging experience for the player — adding replayability from players wanting to see the different outcomes, or the excitement of finding a rare item. The most relevant usage of PCG to my research is in the creation of terrains [27], this also being the most common application in games, one of the largest examples being No Man’s Sky, an action-adventure game with a procedurally-generated universe encompassing over 18 quintillion unique planets, which makes use of a mix of deterministic, parameterised equations, and algorithms [33] in order to develop such a vast simulation. Another example is Dwarf Fortress, a construction and management game, which uses PCG algorithms to create a world with a history, geography, and inhabitants [34]. For a developer to manually create such gargantuan worlds would be infeasible, PCG allows for the creation of these worlds in a fraction of the time and with a fraction of the effort.

Similarly, Minecraft is no stranger to PCG, with its “biomes” and “chunks” [35, 36] being generated mostly through scaled 3-D Perlin noise [37] with Linear Interpolation, and then further modified by custom algorithms to add features such as caves, trees, and villages. Literature on employing LLMs in Minecraft [3, 4], display that existing models can already convert high-level textual descriptions into 3-D structures with a reasonable degree of accuracy, two teams: Madge and Poesio [4], and Hu *et al.* [3] demonstrating around 38% in their respective metrics (completeness and satisfaction for the latter, accuracy against the Interactive Grounded Language Understanding data collection system’s evaluation baseline for the former) — Madge and Poesio did so by making a ‘Minecraft-like’ building game, their approach abstracting the natural generation of Minecraft terrains by having the agents generate buildings in a flat plane; Hu *et al.* implemented their version within the actual Minecraft game, but again, without regard to the terrain upon which they are building (they assume it to be flat) — the hidden complexity lies within the terrain’s structure not being smooth, and accounting for the various features that may already be present throughout the world, dynamically modifying the terrain to accommodate changes in a believable manner.

As is the case with their other uses, LLMs are still prone to hallucinations when generating content for games. A clear instance of this was in the work of Hu *et al.*, where they used LLMs to create entire games (with levels and rules) using Video Game Design Language (VGDL) [10], where they discovered the LLM producing logical errors in its output, even after informing it of the mistakes it had made — ‘confidently’ being wrong as Gallotta *et al.* describe it [9]. So far, the best way to mitigate this has been to incorporate multiple “rounds” to iterate on the output, Word2World by Nasir *et al.* [13] being a prime example; they showed a significant increase in playability (100%) from just having one round of feedback, further increasing by another 50% when they had 10 rounds altogether, their feedback being an analysis by the LLM of a

number of quantitative metrics. An alternative approach is to reduce the model’s ‘temperature’ post-training, as is the method Jin *et al.* executed for WorldWeaver [15], which minimised deviations from the training input, temperature referring to how “creative” or “random” the model is allowed to be. Finally, the most effective method is to have a human-in-the-loop, particularly pronounced in ‘Practical PCG Through Large Language Models’ [12], before their work on Word2World: taking a two-stage approach, wherein the first stage consists of GPT-3 generating game levels, a human then refining the output by fixing the non-playable ones, the second stage being where this is handled by analysing quantitative metrics instead of a human. In my own work, remedying these hallucinations is a key concern, which is why establishing a taxonomy of hallucination levels is crucial to push the boundaries of what is possible with LLMs in this context.

Evaluation

In the majority of LLM applications, they are primarily evaluated on quantitative metrics such as “accuracy,” “efficiency,” or “knowledge retrieval” [2–4, 12, 13, 16]. Even when they are utilised within a gaming context, they are appraised as such, in spite of the very nature of games being an art form, and thus subjective. A “playable” level does not necessarily imply that it is fun, and vice versa — often the most fun or memorable moments in gaming are those that are not the most polished or well-designed, the very existence of ‘speedrunning’ being one of the many examples attesting to this [38], where veteran players exploit bugs and glitches within games to complete all or part of the game in the shortest time possible.

This modus operandi of only quantitatively evaluating LLM applications may stem from the novelty of the field. Early research has focused on whether LLMs can objectively accomplish the tasks they are designed for, which is appropriate in cases where human intent or emotion is not a factor. However, in the context of games, it is crucial to consider these aspects with equal or greater weight than quantitative metrics.

Zhu *et al.* used LLMs for Multilingual Machine Translation (MMT) to assess their performance [39]. Like other LLM application studies, they empirically compared the results of their fine-tuning on several popular models, including ChatGPT (OpenAI) and LLaMA (Meta AI), benchmarking against the commercial translation solution of Google Translate. While language translation can be objectively measured using metrics like Bilingual Evaluation Understudy (BLEU) [40], which compares the output against a reference human translation, a purely quantitative approach does not fully capture the subtleties of language—particularly fluency, contextual accuracy, and nuanced meanings.

In the same vein, the evaluation of games, especially those with an AI content generation component, there is often too little attention paid to the qualitative aspects of the generated content, Awiszus *et al.* [28] developed a GAN for Minecraft worlds, and although they did provide some qualitative ‘examples’, it was just the researchers themselves looking for variability in the samples generated, and not a formal evaluation.

Critically, there is a lack of player feedback in the literature — the most important stakeholder in the gaming industry, and the one who will be most affected by the generated content. By running qualitative experiments, having players play through the generated terrains and providing feedback, better insights can be gained into the utility of LLMs in this context, and how they can be improved to provide more fun. The Triangular Test from the domain of Food Technology [41] is one example of a test that could be made use of to compare the generated terrains against a baseline, observing player preference, as Henderson did in their evaluation of the effect of network delay on online game enjoyment [42].

Summary

My research is centred on what Galotta *et al.* describe as utilising LLMs as ‘automated designers’, creating playable content, but unlike the majority of existing papers that work on the generation of fixed 2-D game levels [2, 10–14] — or those that do work with 3-D, also using Minecraft, but with their concern on generating singular buildings [3, 4], I am interested in the generation of larger-scale ‘terrains’ that may also contain these structures as well as features such as rivers, forests, hills, etc. Existing literature does not address the technical and design challenges involved with this increase in complexity, and this is the gap that my research aims to fill — along with qualitatively evaluating the effectiveness of the solution, since it is within a gaming context, the result should be engaging and enjoyable for the player.

Requirements Specification

Objectives

Primary

Develop a Framework for Prompt Engineering

- Create a system for generating precise and effective prompts to guide LLMs in understanding structured game world data.
 - Should result in a reusable structure for input prompts that the model can parse that produces a structured output to be used in content generation.

Generate Content Same as Existing Generation

- Within Minecraft, there is already a system in place for generating terrains, so to some extent, the model should be able to produce a comparably similar result.
 - We can quantitatively measure the block similarity here by seeing the frequency of blocks placed at each layer from “sea level.”
 - If it does end up being the case that we can’t exactly reproduce the existing generation at a certain level, the triangular test [41] will highlight how apparent the differences are to players.

Secondary

Generate New Content Based on Model’s Understanding of Context

- When the player accesses a previously uncharted region, the model should be able to produce a completely new output that aligns with the existing game world — particularly the region the player was just in before trying to access a new chunk.
- Visually, it should be ‘obvious’ that the result is coherent with the previously established regions of the world, i.e., if only one half of a hill has been loaded, upon using the new generator, the remainder should be made in a way that makes sense from a design perspective.

Create a Taxonomy of Hallucinations & Produce Strategies for Mitigation

- There is a significant chance that, especially when there is a large amount of data and examples in the model’s ‘memory,’ the LLM will incur ‘hallucinations.’
 - Despite whatever level of prompt engineering we perform, it may be unavoidable that there are some unintended results.
- It may be necessary to have the generation step be part of a pipeline [13] with the produced output being checked by a human or another model to ensure that the output is coherent.

Tertiary

Generate Content For 2-D Tile-Based Games

- To prove that the techniques employed in this project are not just limited to Minecraft, we can try to generate content for 2-D tile-based games like Minesweeper.
- With a game like Minesweeper, it is much easier to validate the model's output as we could see if the game is valid — that the rules have been adhered to.
- Like what we are doing with Minecraft, we would first test that the model can generate content that is the same as the existing game generation and then move on to extend the game.
 - With Minesweeper, we could potentially have buttons on each side of the grid that would add a new grid to the game on that side.

Explore Scalability of Content Generation

- We should be able to measure the relationship between the requested output size and the accuracy of the model's output.
 - This will be important for understanding the limitations of the model in use and to what extent we can rely on it for generating content.
 - It may also provide better insight into how much context the model can hold in memory at any given time without any performance degradation.
- If it ends up being that there is a certain threshold where the model's performance is maximised, we may be able to have it produce content in smaller chunks of this optimal size and then stitch them together to form a larger output.

Investigate Prompting The Model for Specific Content To Generate

- This would possibly entail the player or 'game master' providing the model with a prompt that requests something specific to be generated.
 - For example, the model could be prompted to generate the block data for a castle, and this castle could then be added to the world in a requested location.
 - We could also just have this castle appear as a transparent 'ghost' structure that the player can then choose to 'materialise' in the game world — acting as a build assistant.
 - This would be a more interactive way of using the model to generate content and could be a more engaging way of using the model.

Requirements

These following requirements are based on the primary and secondary objectives outlined above, and can be considered as the success criteria for the project. They are sorted from top to bottom in order of importance (top being the most important).

Functional Requirements

1. **Terrain Generation:** Without explicit user input, the system should be able to produce singular non-extending playable terrains in Minecraft, at least 256x64x256 blocks in size and complete with features and structures a player would expect to find in the game (forests, villages, lakes, etc.).
2. **Dynamic Terrain Extension:** Without human intervention, the system should be able to dynamically extend terrains in Minecraft as the player explores so that they blend visually and structurally with adjacent regions and maintain world consistency.
3. **Basic Hallucination Handling:** The system should be able to automatically handle the lowest level of hallucinations, i.e., those that are the quickest to identify and repair. An example of this would be an error in the structure of the model output that is easily fixed by a simple rule.
4. **Event-Driven Content Generation:** The system should be able to generate content based on in-game events, such as player movement, and operate in real-time while ensuring continuity in gameplay.
5. **Generated Content Memory:** The system should maintain a persistent record of all generated regions (“tiles”) and provide this context to the model for new tile generation.
6. **Model Output Logging:** The system should be able to log the output of the model for each generation step, including the input prompt and the output generated. This auditing will be useful for debugging and understanding the model’s behaviour.

Non-Functional Requirements

1. **Fun:** The system should be able to generate content that is enjoyable to play and explore.
2. **Consistency:** The system should be able to generate content that is consistent with the existing game world.
3. **Visuals:** The system should be able to generate terrains that are visually reminiscent of the existing generation in Minecraft.
4. **LLM Selection Metrics:**
 - **Context Window:** Capable of handling large input and output contexts for maintaining region history.
 - **Reasoning Ability:** Demonstrates high logical consistency for understanding and generating coherent terrain features.
 - **Multimodal Capability:** Supports integration of data beyond text, such as images or structured data.
 - **Cost:** The system should be able to generate content while incurring as little monetary cost as possible.

5. **System Modularity and Extensibility:** The system should be designed in a modular way that allows for easy extension and modification of its components. Adding new terrain features or structures should require minimal codebase changes.
6. **Performance:** The system should be able to generate content in a reasonable amount of time, i.e., not take more than a minute for a 256x64x256 block region to appear in-game.
7. **Scalability:** The system should be able to keep generating content without any noticeable degradation in performance as the player explores further. A generation request rate of at least 3 per minute should be manageable.

Software Engineering Process

The artefacts for this project were developed using an Agile methodology [43]. Agile was chosen for its iterative and flexible nature, which is particularly well-suited to research and development projects involving experimental techniques like prompt engineering for Large Language Models (LLMs). Unlike rigid methodologies, Agile allows for continuous refinement of requirements and designs, enabling the project to adapt to discoveries and setbacks as they arise.

Phases of Development

The project was divided into three main phases, each with concrete goals, deliverables, and iterative cycles:

1. Initial Phase:

- **Objective:** Establish a baseline by understanding and replicating Minecraft's existing content generation.
- **Key Activities:**
 - Parsing Minecraft block data and identifying structural patterns (e.g., Run-Length Encoded block data).
 - Prototyping and testing foundational prompt engineering approaches, with changes tracked in GitLab.
- **Artefacts Produced:**
 - A Python script for finding the frequencies of blocks at each layer from sea level (*block_data_compression.py*).
 - A Python script for filtering bad screenshots from the dataset (*screenshot_filtering.py*).
 - A Node.js script for automating the compression of block data and formatting of JSON files (*JSON_automation.mjs*).
 - A Node.js script for prompting the LLM (*model.js*) with Google Cloud API.
 - A keyboard automation script for taking screenshots in Minecraft (*screenshot_automation.ahk*).
 - Modded Minecraft items for analysing Minecraft terrains, taking screenshots, and generating new terrains from model outputs (*AreaAnalyserItem.java*, *BiomeSamplerItem.java*, *LargeRegionSaverItem.java*, *ScreenshotCameraItem.java*, *TerrainGeneratorItem.java*, *AlternateTerrainGeneratorItem.java*).
 - Early prototype prompts that tested block-level generation, revealing limitations such as incoherent outputs.

2. Extension Phase:

- **Objective:** Dynamically extend terrains by refining prompts and integrating results into Minecraft.
- **Key Activities:**
 - Iterative adjustments to prompts based on feedback from testing cycles.
 - Prompting the model with existing information and leveraging its context caching to extend already generated terrains.

- Rendering generated outputs in Minecraft for visual inspection and player feedback.
- Developing a system for automated hallucination detection and mitigation.
- Automated pipeline of model prompting, hallucination detection, and terrain generation.

- **Artefacts Produced:**

- An updated *model.js* script with support for hallucination detection, mitigation, and an automated terrain generation pipeline.
- Modded Minecraft in-game logic for requesting new terrains and generating them without explicit player input

3. Evaluation Phase:

- **Objective:** Assess the generated content through gameplay sessions, combining quantitative metrics with player feedback.

- **Key Activities:**

- Conducted gameplay sessions where players interacted with generated terrains, providing feedback via surveys.
- Quantitative analysis of generated terrains, comparing block frequencies and structures with existing Minecraft terrains as well as random terrains.

- **Artefacts Produced:**

- A survey template for collecting player feedback on generated terrains.
- Player feedback reports summarising qualitative feedback and suggestions for improvements.
- A Jupyter notebook analysing a number of quantitative metrics (*quantitative_analysis.py*).

Agile in Action

Agile principles guided the project lifecycle, with a strong focus on iteration, collaboration, and adaptability. Each week of the project was, in essence, a “mini-sprint” [44], with clear objectives, deliverables, and feedback loops. Key Agile practices:

1. Version Control with GitLab:

GitLab was used extensively for version control, ensuring all changes to scripts, prompts, and documentation were tracked.

- The GitLab repository provided a comprehensive history of revision, making it easy to trace the evolution of the project and revert changes when necessary.
- Regular commits ensured that changes were well-documented, aiding in the identification of issues and the resolution of conflicts via the use of rollbacks.
- Branching allowed for the development of new features without disrupting the stability of the main codebase, facilitating parallel development.

2. Weekly Progress Reports:

Progress was documented weekly in detailed reports shared with the supervisor. These reports included:

- **Activity Logs:** A summary of tasks completed that week.
- **Goals for Next Week:** Clearly defined objectives for the upcoming week, ensuring alignment with overall project milestones.
- **Items for Discussion:** Highlighting challenges, blockers, or areas requiring input from the supervisor.
- **Reflections:** A brief reflection on the week's progress, identifying successes and areas for improvement. Also includes a summary of feedback and decisions made during weekly supervisor meetings, which were crucial for maintaining focus and adapting plans as needed.

3. Iterative Refinements and Flexibility:

Agile's iterative nature enabled rapid prototyping and testing cycles. For example:

- Initial prompt structures produced incoherent terrains that seemed random and lacked structure. Through frequent refinements documented in GitLab, the model evolved to generate high-level feature-based outputs that significantly improved coherence and long-term scalability.
- Automated hallucination detection was introduced after observing common patterns in model outputs. This iterative approach allowed for the identification of common errors and the development of targeted mitigation strategies.

Evidence of Agile Success

1. Artefacts and Logs:

- The project repository contains a comprehensive history of changes, demonstrating the iterative nature of development and the evolution of the dynamic content generation system over time.
- Weekly reports and meeting notes document the alignment of activities with Agile principles, ensuring continuous progress and accountability.

2. Qualitative Feedback:

- Player feedback from gameplay sessions highlighted the fact that frequently the terrains were indistinguishable from those generated by Minecraft itself and were often at least as fun as the original game.
- Iterative adjustments based on player feedback led to improvements in the generated terrains, demonstrating the flexibility and responsiveness of the Agile approach.

3. Adaptability:

- Early testing revealed limitations in block-level terrain generation. Agile's flexibility enabled a timely pivot to feature-based commands and multimodal inputs, improving output quality without disrupting overall progress.
- Automated hallucination detection was introduced mid-project in response to observed patterns, demonstrating the project's adaptability and responsiveness to emerging challenges.

In summary, adopting an Agile methodology, combined with tools like GitLab for version control and structured weekly progress reporting, ensured the project was both adaptive and transparent. These practices allowed me to dynamically address challenges, maintain alignment with goals, and deliver a robust framework for LLM-driven content generation.

Ethics

The data collected as part of the qualitative evaluation is completely anonymised feedback from users testing my artifact, hence the appropriate “CS Artifact Evaluation Form” is seen below.

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
ARTIFACT EVALUATION FORM

Title of project

Generating Dynamic Virtual Environment Using Large Language Models

Name of researcher(s)

Joel Sebastian

Name of supervisor

Tristan Henderson

Self audit has been conducted YES NO

This project is covered by the ethical application CS15727.

Signature Student or Researcher



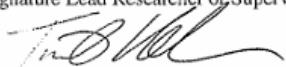
Print Name

Joel Sebastian

Date

16/09/2024

Signature Lead Researcher or Supervisor



Print Name

TRISTAN HENDERSON

Date

17/9/24

Figure 1: CS Artifact Evaluation Form

Design

Designing “seamless” dynamic content generation with LLMs involves three interconnected challenges: structuring prompts to effectively communicate with the model, selecting the most suitable LLM for the task, and developing a robust system that integrates the generated results into the game without requiring human intervention. Each of these components must not only function independently but also scale cohesively to handle the complexity of dynamic content generation.

This design process must account for the significant demands of scalability, including managing extensive cached context for LLMs, handling frequent content-generation triggers, and processing a high volume of prompts and responses efficiently. The overarching goal is to create a system capable of producing coherent, immersive, and believable in-game terrains while overcoming the limitations of current LLM technology.

Choosing the LLM

The key metrics that mattered most when deciding on the LLM to use were: the size of the context window, extent of multimodal capabilities, as well as reasoning ability. To a lesser extent, factors like cost, response time, and ease of Application Programming Interface (API) integration were also considered. The attitude towards cost was one of getting the best “bang for the buck,” so ideally it would be a free service or at least have a free tier that could be used for development and testing; there was a level of uncertainty regarding the timescale involved in how long the model would have to store its conversation history, potentially months, so a service that would charge hourly based on the number of tokens cached was infeasible.

Context window refers to the maximum number of input and output tokens that the model can consider when generating a response — in essence, how much of the conversation history the model can remember; from the onset, some rough calculations were made to estimate the size of the context window needed, i.e., from finding the average number of tokens in just one example of a terrain used to tune the model, which in my first attempt included a number of multimodal inputs (JSON, PNG) as well as the initial prompt text provided to contextualise the training data, along with explaining the model’s role in understanding the examples and generating responses.

The reasoning aptitude of an LLM plays a critical role in determining its ability to produce coherent and logical responses, especially when tasked with complex, multi-step processes like generating terrains. In our case, reasoning is essential for tasks such as understanding the relationships between terrain features, interpreting high-level instructions, and producing outputs that align with Minecraft’s natural terrain generation principles. LLMs with higher reasoning scores demonstrate better performance in scenarios that require analytical thinking, mathematical computations, and logical consistency. This is especially relevant when generating dynamic content based on existing terrain, as the model must account for the context of the player’s surroundings and ensure new terrain integrates seamlessly with what has already been generated. Poor reasoning can lead to incoherent transitions, implausible terrain features, or outright failures in understanding the prompt requirements.

Multimodality allows LLMs to process and understand inputs beyond text (such as images or structured data), another crucial factor in selection. This project relied on multimodal inputs like screenshots of Minecraft terrains and data structures (e.g., JSON or Run-Length Encoded block data) to provide a diverse set of examples for the model to learn from and generate outputs. The ability to interpret visual information is particularly valuable in a game like Minecraft, where spatial awareness and aesthetic consistency are paramount. Multimodal models can extract patterns and features from visual data, enabling them to better understand

the relationship between different terrain elements, such as elevation changes, biome transitions, and feature placement.

Table 1 shows a comparison of the “mainstream” LLMs that were commercially available at the time (late 2024), with the most important metrics discussed above. The multimodal (visual) and reasoning scores are from benchmarks conducted on Large Multimodal Models (LMMs), which are a subset of LLMs that can process both text and other forms of data, such as images [45]. Henceforth, the term “LLM” will refer to LMMs unless otherwise specified as at present, the vast majority of LLMs are now multimodal; the scores are from the Open VLM and Open LMM Reasoning Leaderboards [46] respectively. MMVet is an evaluation benchmark based on the visual reasoning capacity of multimodal models [47], averaging the performance of the model across a number of tasks, including Optical Character Recognition (OCR), spatial awareness and such. “Reasoning Score” is a metric on the Open LMM leaderboard that similarly averages a number of scores in metrics such as MathVista and MATH-Vision [48, 49], involving solving mathematical problems from captioned images.

Model	Context Window (Tokens)	MMVet Score	Reasoning Score	Cost (USD/1M Tokens)	Speed (Tokens/s)
Google Gemini 1.5 Pro	2M	74.6	48.8	\$2.19	61.0
Google Gemini 1.5 Flash	1M	69.7	51.0	\$0.13	183.9
OpenAI ChatGPT 4.0	128K	74.5	50.5	\$4.38	115.1
Anthropic Claude 3.5 Haiku	200K	46.4	45.7	\$1.60	65.2
Anthropic Claude 3.5 Sonnet	200K	70.1	45.7	\$6.00	72.1

Table 1: Comparison of mainstream LLMs [46, 47, 50–54], with the best in each category highlighted in bold.

Some models, like Google Gemini 1.5 Pro, have additional pricing for the number of tokens cached, not just the number of tokens provided in the inputs and outputs. In Gemini’s case, the cost is based on the number of millions of tokens per hour cached, something that cannot be overlooked when considering the potentially enormous context window needed for this project and how long it would have to be stored. Despite this, Google provides a platform (Vertex AI) [55] for using their off-the-shelf models with no limits as well as \$300 in free credits for new customers, a value that was substantially more than the estimated cost of using the model for the duration of the project, especially when taking into consideration the fact that as long as the number of new tokens cached per minute was kept below 1 million and the number of requests per minute was kept below 15, the service is free of charge.

This pricing model, together with significantly larger context windows, faster speeds, and higher multimodal and reasoning scores made using the Google Gemini 1.5 models an attractive proposition. The Pro model and Flash models were both made use of, the former for the initial tuning of the model with multiple data sources in order to gauge the success of the model in understanding the data and generating coherent responses (mostly in the case of generating a singular game region), and the latter was switched to when speed and request frequency became more important — when we needed to generate terrain extensions in real-time as the player explored the game world.

Prompt Engineering

After selecting the LLM, the next step was to design an effective prompting strategy that would enable the model to understand the context of the game world, maximising the number of examples we could provide for tuning, while also balancing the quality of these examples. The level of desired detail from model outputs influences how relevant prompt inputs may be. For instance, if the model is tasked with generating lines of parameterised function calls, a relatively “high-level” approach, it is unnecessary to provide exhaustive details, such as the exact placement of blocks within the game world. Overloading the model with excessive information could convolute its understanding of the task.

Regardless of the approach taken, it is crucial for the model to understand its ultimate purpose. Gemini’s API includes “system instructions,” which is an additional prompt cached and used to provide extra context for model tuning [56]. The system instructions shown in Figure 2, establish a foundation by describing primary task objectives, relevance of the provided data, and what future tasks the model may be expected to perform.

You will be given 300 screenshots of Plains biomes from the game Minecraft (version 1.21.1), my aim is to eventually make use of the code generation abilities of Large Language Models to generate terrains and structures within the game. You should make sense of what features are present in each screenshot to gain an understanding of how this particular biome is usually structured and distributes things like (as examples) hills, buildings, flowers, trees etc. Eventually, I want to be able to give a region that has already been generated, and ask you to generate its surrounding area and features based on the context.

Figure 2: The system instructions provided in the working implementation.

Prompt Chaining and Delegation

Generating terrains dynamically as the player explores the game world involves breaking the task into smaller subtasks, a technique known as prompt chaining [57]. For instance, the model could be prompted to generate terrain tiles one by one, with each tile influenced by the features of its neighbouring regions. This method reduces the complexity of individual prompts and helps maintain consistency across generated regions.

Iterative Design with Feedback

The prompts were iteratively refined using feedback from the LLM itself and ChatGPT. This approach aligns with findings from Zhou *et al.* on Automatic Prompt Engineering (APE) [58], which suggests that involving models in their own prompt optimisation can improve accuracy, often resulting in a higher accuracy than with solely human feedback. The iterative process included adjusting the instructions, rephrasing context explanations, and fine-tuning the balance between example data and task descriptions.

ReAct: Reasoning and Action

To enhance the coherence of outputs, the model was instructed to articulate its thought process both before and after generating its responses. This technique, inspired by the ReAct (Reasoning and Action) framework [59], allowed the model to justify its decisions, such as why it placed a hill or a tree in a specific location. These explanations provided insights into the model’s reasoning capabilities and helped identify areas for improvement.

Reflexion and Self-Assessment

The model’s ability to assess its own outputs was utilised to improve generation quality. Through Reflexion [60], the model provided a self-assessment of the terrains it generated (provided back as screenshots), identifying inconsistencies or areas where its outputs might deviate from the expected biome structure. This self-critical feedback loop was instrumental in refining both the prompt and the model’s outputs over time.

```
Name: Hill,  
Description: Generates a hilly region using Perlin Noise between a set of coordinates (two x and z values), the hills having a maximum amplitude,  
Parameters: Starting X (startX), Starting Z (startZ), Ending X (endX), Ending Z (endZ), Max Amplitude (maxAmplitude),  
Structure: "/generate hill [startX] [startZ] [endX] [endZ] [maxAmplitude]",  
Example: "/generate hill 0 0 100 100 60"
```

Figure 3: Part of the prompt that details how to generate hills.

Few-Shot Prompting

For the lower-level approach, we can say that few-shot prompting was used in the sense that the 5 detailed Plains terrain examples we were giving, acted as the few shots that the model could learn from. This is in contrast to the high-level approach, where the model was given 300 “shots” of Plains biomes to be tuned with, but the function descriptions enumerated some of the possible commands that could be used to generate features (Figure 3), so in a sense the model had few-shot learning in regards to its output commands. This technique enables in-context learning, where the model can adapt to new tasks with minimal examples, providing better performance than just being dependent on zero-shot pre-training data [61, 62].

Meta-Prompting and Function Calling

For the higher-level approach, meta-prompting [63] (MP) was used to define the expected output format explicitly (Figure 2) — this included the specification of functions, their parameters, and their intended use. MP is a method that insists upon the significance of the syntax of information provided to the LLM rather than standalone content. Google Gemini does provide their own native function calling API support; however, this was not put to use as there were limitations in its accepted JSON format, in that it did not have a field for providing examples, which I deemed to be too important to leave out.

Chain-of-Thought Prompting

To improve the model’s logical consistency, Chain of Thought (CoT) prompting [64] was employed. This involved encouraging the model to break down its reasoning process into sequential steps, especially when generating complex or layered terrains. By guiding the model to think through the relationships between features, such as elevation changes and block transitions, CoT prompting contributed to the overall coherence and realism of the generated outputs. More specifically in the case of the second approach, the model was told to first analyse the 300 screenshots provided before performing any code generation, and then only when requested, did it begin terrain generation.

First Attempt (Low-Level Approach)

The fundamental idea behind the low-level approach was to provide the model with as much detail as possible, block-by-block, about Plains biomes in the game, intending for it to then have a deep enough understanding such that it could essentially craft a terrain block-by-block. The four main components of each example in the prompt were:

- **A JSON file containing block frequency data for each layer above sea level**
 - Gives the model an idea of the distribution of blocks in the biome, more specifically the three most occurring block types at each y-level and their respective frequencies.
- **A JSON file containing run-length encoded (RLE) block data**
 - Stores all the block data in the region in a highly compressed format, with the intention of the model developing recognition of patterns in the grids that represented all of the blocks at each y-level (256x256 blocks in each grid)
- **A PNG screenshot of the region**
 - This was to give the model a visual representation of the biome so that it could understand the aesthetic of the biome and how the blocks were distributed in the region. As well as this, the hope was that the model could attribute different patterns in other data sources to the visual data.
- **A PNG heightmap of the region**
 - This was to give the model an idea of the elevation changes in the region, something that is not easy to capture in a screenshot due to the angle at which the screenshot was taken, obscuring the height of blocks.

The example regions were sampled randomly in order to avoid any bias in the training data, and despite having a relatively small number of examples (5), having each example encompass a large area of the map, 256x64x256 blocks, meant that a large quantity of data was still being provided to the model, reducing the bias caused by the small number of examples. The model was then asked to generate a region of the same size as the examples in the same RLE format as part of the input prompt, with the frequency of blocks at each y-level being similar to that of the original game.

Unfortunately, as Figure 3 shows, with this approach, the model struggled even for smaller regions of 16x16x16 blocks, producing outputs that were not only inconsistent with the original game but also incoherent and nonsensical. The “staircase” structure present in some of them is a result of my explicit intervention, where I showed example patterns in RLE data that I wanted the model to learn from, in order to create a hill shape. Not only were there hallucinations present in the look of small regions like Figure 3a, 3b and 3d, but the model also struggled to keep within the constraints of the 16x16x16 region requested as seen in Figure 3c, 3e and 3f. This may partly be a result of Gemini pre-emptively ensuring that the size of its total output is not beyond the limit, meaning that there may be instances where the model had to cut off its output early, resulting in the incomplete regions seen in the images.

There were not hallucinations in the form of the JSON format being incorrect, but the content tended to have inconsistent numbers of blocks on each axis except for the y-axis, which was ensured to always be the desired number of blocks high due to the prompting in this approach done in a chunked, sequential format where the start of the JSON and a number of y-levels from sea level were requested first, then the next N y-levels were requested in batches until the entire region was generated. This was necessary due to the output limits of the model, the number of

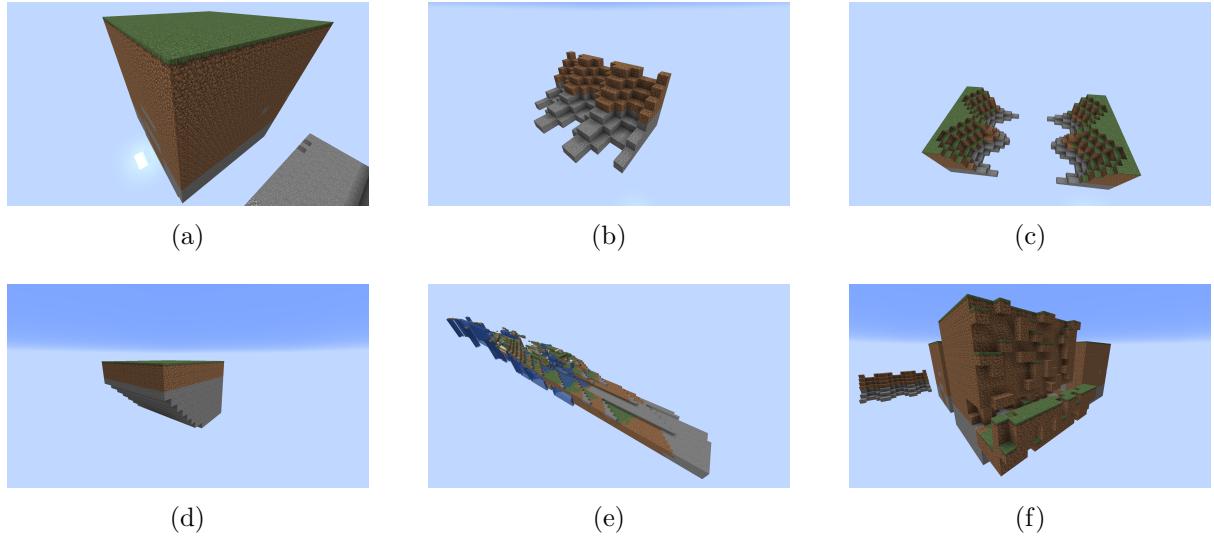


Figure 4: Example outputs produced using a low-level approach.

tokens required for a complete RLE JSON even for a small region ($16 \times 16 \times 16$) was too high to be requested in one go.

One side-effect of having to constantly request the next batch of y-levels was that the request limit of 15 per minute would be reached very rapidly, and another is that the remaining amount of “free” context cache left for the model after the tuning examples were provided was too low to be able to generate more than a few small regions, and so any further generation would start to cost money.

Overall, the major issues with this approach could be summarised as follows:

- **Incoherent Outputs:** The model struggled to generate consistent and coherent terrains, often producing nonsensical structures and block placements. Despite giving a large amount of data with ample room for pattern recognition, the model was unable to generate outputs that aligned with the original game’s terrain structure.
- **Output Size Limitations:** The model had difficulty generating outputs that adhered to the requested region size, often producing incomplete or truncated terrains.
- **Request Frequency:** The need for frequent requests to generate each batch of y-levels led to rapid exhaustion of the request limit, hindering the generation process.
- **Context Cache Limit:** The limited context cache available for the model restricted the number of regions that could be generated, potentially incurring additional costs. As we want to move towards real-time dynamic content generation, this approach was not feasible.

Second Attempt (High-Level Approach)

The lack of promise from the low-level approach led to a pivot towards a higher-level approach, where the model was given a more abstract view of the terrain, just screenshots of Plains biomes, and was asked to generate a list of commands that could then be parsed to run algorithms that would handle their own bespoke purpose; this delegation of the actual feature generation to the algorithms was a way to ensure that the model was not overwhelmed with the task of generating the terrain block-by-block, but rather from a higher-level perspective of what the general structure of the terrain should be and its contents.

Reducing the “workload” of the model will also reduce the room for hallucination, the more constraints that are placed on the model, the less room there is for it to deviate from the expected output. The model was given 300 screenshots of random Plains biomes and asked to analyse them before generating any code — a CoT approach that almost certainly improved the level of understanding the model had of the biome type, and hence the quality of the generated commands.

After its terrain analysis, it was given a list of commands (an example of one is seen in Figure 2), featuring details on the parameters of the command, its structure, and at least one example of how it should be used. It outputs a list of commands that are then parsed and run in the game, and for five iterations, the model is given human feedback on the generated terrain, a screenshot of the region, and is asked to self-assess its output; after receiving this feedback, it is then requested to make an improved terrain. This feedback loop (Figure 5) is imperative in refining the model’s understanding of the biome and its ability to generate coherent and consistent terrains, additionally catching any hallucinations without the need for human intervention.

Caching the system instructions and 300 screenshots requires around 80,000 tokens, leaving more than 900,000 tokens for Gemini 1.5 Flash and 1.9 million tokens for Gemini 1.5 Pro, which is more than enough to generate a large number of regions, even if the model is generating a large number of commands for each region. An average command is around 20 tokens, so with an output token limit of 8192, this is over 400 commands that can be generated in one go, and with the request limit of 15 per minute, this is 6000 commands that can be generated in one minute, which is more than enough for real-time generation.

GenerationCommand Class

The Command Behavioural Design Pattern [65] is here to encapsulate the logic for various terrain generation tasks (e.g., creating forests, flowers, rivers) into discrete, reusable objects. This allows for greater modularity, flexibility, and maintainability in the codebase. Each command represents a self-contained instruction with defined parameters, isolating the implementation details and enhancing the separation of concerns. The parser (Figure 6) refers to the ‘switch’ statement in the logic that is responsible for dynamically determining which generation method to invoke based on the first argument of the command — invoking done at the loop that iterates through the list of commands generated by the model, by calling the ‘execute()’ method.

Some of the advantages of this design pattern include:

- **Scalability:** New terrain features can be added by implementing additional methods within the *GenerationCommand* class, requiring minimal changes elsewhere in the system.
- **Reusability:** Commands can be reused across different contexts (regions), enabling the model to generate a wide variety of consistent terrains without duplicating code.
- **Decoupling:** The separation of command logic from the main game logic enhances code maintainability and readability, making it easier to debug and extend the system.

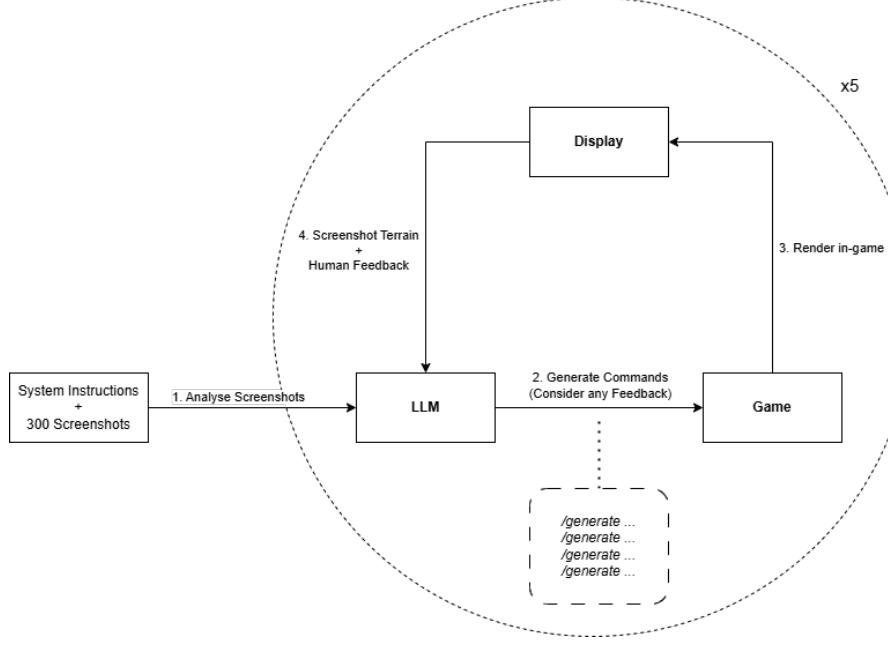


Figure 5: Diagram depicting iterative prompt engineering and feedback loop in the high-level approach to generate a terrain.

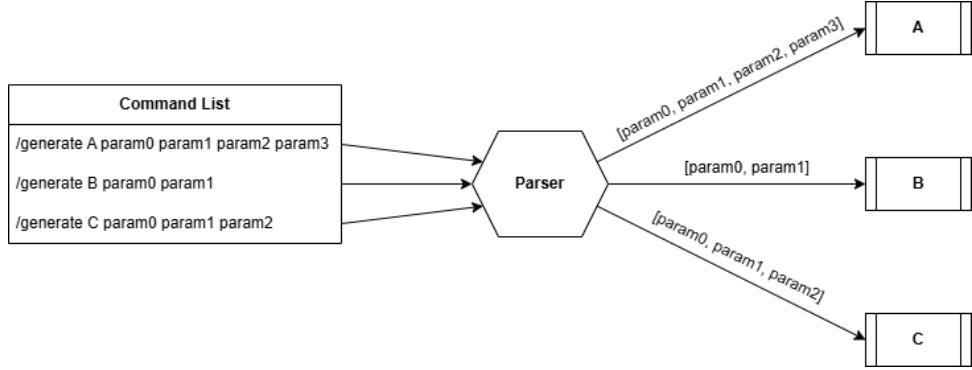


Figure 6: Abstract diagram of the parser in the game logic that determines what subroutine to call (and generate a feature/structure).

- **Support for Feedback Loops:** By encapsulating commands, the system facilitates iterative feedback and refinement cycles. For instance, after each terrain generation attempt, the feedback loop can identify and tweak specific parameters for improved outcomes.

In the high-level approach, the *GenerationCommand* class plays a crucial role in translating high-level model outputs (i.e., a list of terrain features) into actionable game commands. The iterative feedback loop enhances this process by providing the model with visual and contextual feedback from the generated terrain, enabling fine-tuning of commands for better results.

Dynamic Content Generation

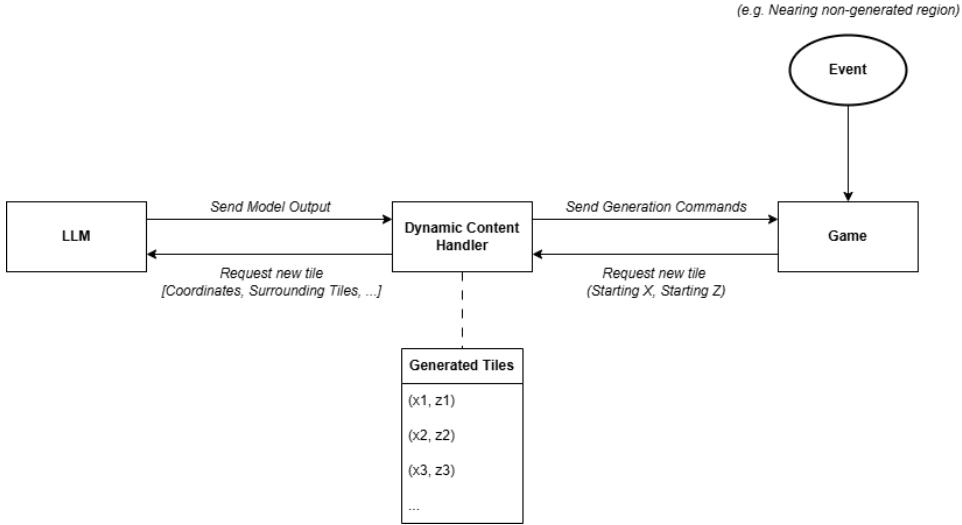


Figure 7: The event-driven architecture behind dynamic content generation in the game.

Dynamic content generation in the game is facilitated by an event-driven architecture (Figure 7) that coordinates the interaction between the player, the model, and the game world. The system is designed to handle real-time triggers, such as a player moving within a certain proximity of the world border — a distance where they are far enough away that they cannot see the lack of terrain, or it being generated but close enough that they may see the region soon if they continue moving in the same direction.

There are three main components to the system: the LLM, an intermediary server, and the game. When a generation event is triggered, the game is notified and sends a request to the intermediary server, which then forwards the request to the LLM after adding the necessary context; the server stores a list of the “tiles” (256x64x256 block regions) that have already been generated, and will be able to determine which of those tiles are adjacent to the new tile we wish to generate (Figure 8). After determining these surrounding tiles, the server will send both the request that contains the new region we wish to generate, as well as the surrounding regions, to the LLM. If any supplementary instructions were provided as part of the request from the game, such as the biome type of the new region, these would also be included in the request to the LLM.

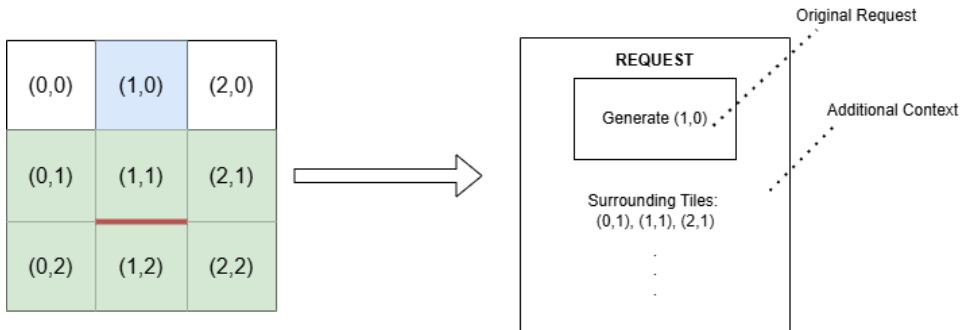


Figure 8: The blue tile is the new region to be generated and green corresponds to tiles that have already been generated. The red line represents the invisible boundary, which when crossed triggers the tile in blue to have its generation requested. The server will send the adjacent green tiles as context to the LLM.

We want to send the surrounding tile context to almost ‘remind’ the model of what to look out for when generating the new region — if there is half of a hill at the end of one tile, we want the model to generate the other half of the hill in its correspondingly adjacent tile. In order to retain the information on which tiles have been generated so far even if the game or handler server were to restart, the list of generated tiles is stored in a local file, loaded upon server start-up, and updated whenever a new tile is generated; this also serves as an audit trail for the generated regions, allowing for easier debugging and monitoring of the system.

All requests to and from the handler are asynchronous, meaning that the game can continue to run without waiting for a response from the server, and the server can continue to handle requests from the game without waiting for a response from the LLM. This is important as the model can take a significant amount of time to generate a region, especially if it is generating a large number of commands for the region, and we do not want the game to be unresponsive during this time.

Implementation

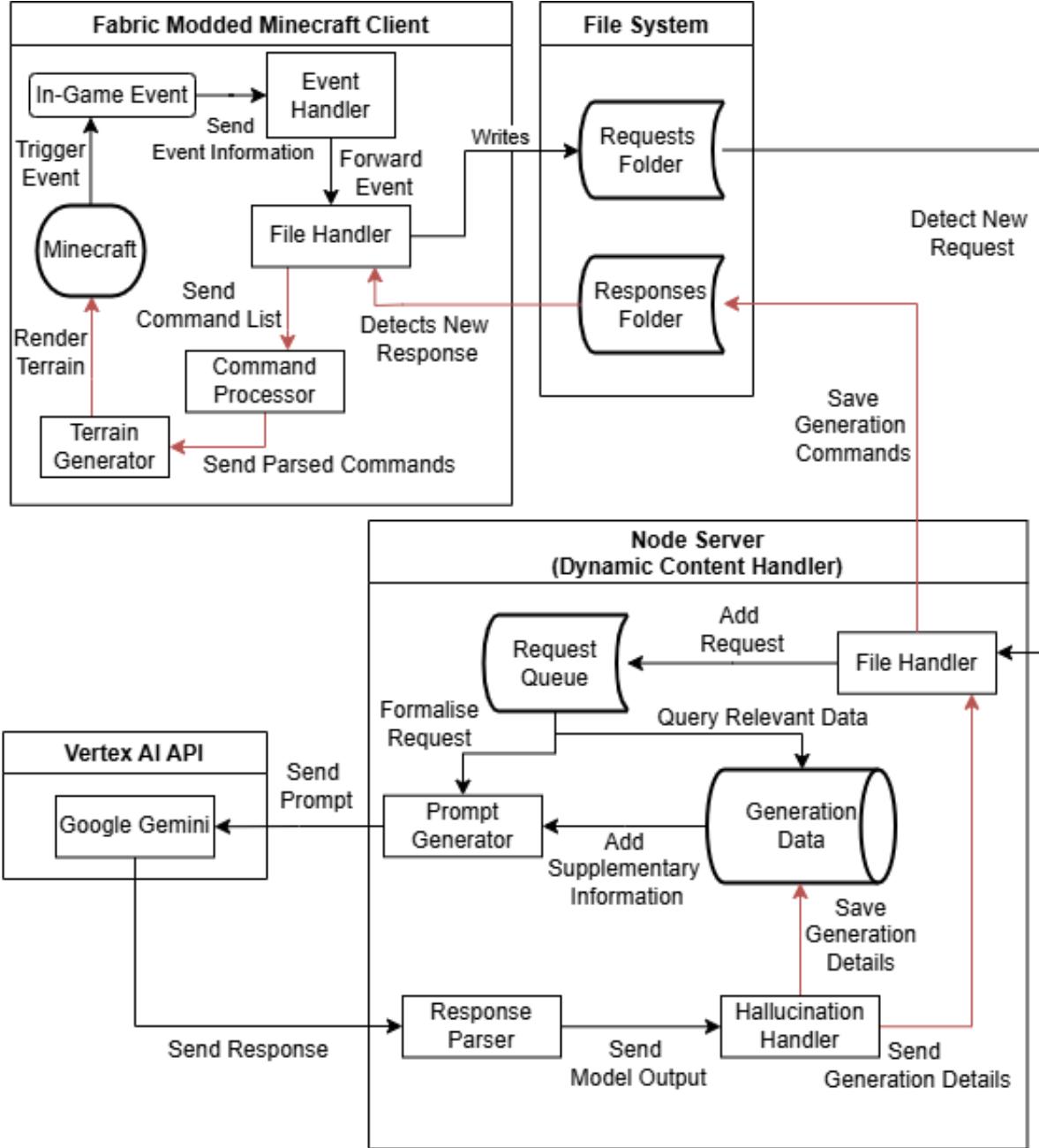


Figure 9: Overall System Diagram: The red arrows indicate the flow of data after the model output has had its hallucinations corrected.

Minecraft [66] is a game written in Java, and as such, the modifications I wrote to the game were also written with an object-oriented approach in mind. The modifications were written using the Fabric Ecosystem [67], a collection of interconnected tools and libraries that simplify mod development, while offering a high degree of flexibility and customisation:

- **Fabric Loader:** The lightweight runtime environment that loads Fabric mods into the game. Its modular nature allows it to integrate seamlessly with Minecraft without adding unnecessary overhead [68].
- **Fabric API:** A set of prebuilt utilities and abstractions for common modding tasks, such as accessing Minecraft’s internal structures, adding new items or blocks, and managing

events [69].

- **Loom:** A build system plugin for Gradle that simplifies project setup and dependency management. Loom ensures compatibility with Minecraft’s obfuscated codebase and makes it easier to compile and test mods [70].
- **Yarn:** A community-driven set of mappings that translate Minecraft’s obfuscated code into human-readable names. Yarn provides an essential layer of clarity for navigating Minecraft’s codebase [71].

The two predominant modding platforms for Minecraft are Fabric and Forge. While Forge is more established and has a larger community, Fabric is more lightweight and has a more modern codebase, and these two reasons were enough for me to choose Fabric over Forge. Fabric allows for Kotlin code to be written, but I chose to write the mod in Java out of familiarity and from the number of resources available online for Java modding.

Mod Structure

The structure of the mod is based on the generic Fabric mod template [72], which provides a basic project layout and build configuration (using Gradle, a build automation tool). There are two entrypoints *LLMTerrainGeneration* and *LLMTerrainGenerationClient*, the former used for the majority of common (server-side and shared) code, and the latter for client-specific code. These implement the *ModInitializer* and *ClientModInitializer* interfaces respectively, and are invoked by the Fabric Loader during the game startup. Both entry points are responsible for tasks such as registering new items and commands, setting up event handlers, and initialising global singletons like the *LOGGER*.

There are two packages used for the utility code within the mod, *java.joel.llm.terrains.utils* and *java.joel.llm.terrains.misc*; the *utils* package is reserved for the client-side utility code, such as the handling of when a request has been made to the client for a screenshot to be taken, and *misc* is used for the rest of the utility code, with classes such as *Coordinate*, *IScreenshotHandler*, and *TerrainData*.

The majority of additions can be found in the *java.joel.llm.terrains.items* package, where the likes of *LargeRegionSaver*, *TerrainGenerator*, *ModItems*, and *ScreenshotCamera* can be found. They contain the logic that allows for manual testing of functionality such as saving and loading terrain data, generating regions from a list of commands, and taking screenshots of what is on the screen. For the parts of the game logic where we would have to automate the process (such as in the case of terrain generation), we can write the logic within *ModItems* and call it from a new command registered in *LLMTerrainGeneration* — these commands able to be called by “command blocks” in the game that can be triggered by an event.

Core Item Functionality

The *ModItems* class serves as a central hub for managing all custom items in the mod. It is responsible for both item registration and for encapsulating shared functionality that is reused across the codebase. This design ensures a modular and maintainable code structure.

One of the key reusable methods within *ModItems* is ‘*saveBlockDataCSV()*’, which plays a pivotal role in saving block data from the game world into a CSV format. This data can then be used for further processing, such as analysing terrain features or generating prompts for the language model. The modularity of this method allows it to be utilised across different items and commands without duplicating code.

The items *AreaAnalyser*, *BiomeSampler*, and *LargeRegionSaver* are almost identical in functionality as all three save some form of data about the game world, the major difference being the volume of blocks scanned. The four methods they can use to save data are as follows:

- **saveHeightmap()**: Saves the heightmap of the scanned area as a greyscale PNG
- **saveHeightmapAsArray()**: Saves the heightmap of the scanned area as a 2-D array in a TXT file (Algorithm 1)
- **saveBlockDataCSV()**: Saves every block in the scanned area into a CSV file. The coordinates and block type are saved (Algorithm 2).
- **saveBlockDataRLE()**: Saves a run-length encoded version of all the blocks in the scanned area into a JSON format (Algorithm 4).

Algorithm 1: saveHeightmapAsArray()

Input: *path* (file path), *startX*, *startZ*, *endX*, *endZ* (region boundaries), *world* (Minecraft world object)

Output: Heightmap saved as a 2-D array in a comma-separated format within a TXT file

Initialize *lines* $\leftarrow \{\}$ // List to hold rows of heightmap data

Initialize *minHeight* $\leftarrow \infty$

Find Minimum Height (Sea Level) in the Region:

```
foreach chunkX  $\in [startX/16], \dots, [endX/16]$  do
    foreach chunkZ  $\in [startZ/16], \dots, [endZ/16]$  do
        world.getChunk(chunkX, chunkZ);
        foreach x  $\in \max(startX, chunkX \times 16), \dots, \min(endX, chunkX \times 16 + 15)$  do
            foreach z  $\in \max(startZ, chunkZ \times 16), \dots, \min(endZ, chunkZ \times 16 + 15)$  do
                y  $\leftarrow$  world.getTopY(x, z);
                minHeight  $\leftarrow \min(minHeight, y);$ 
```

Generate Heightmap Data:

```
foreach z  $\in startZ, \dots, endZ$  do
    line  $\leftarrow \text{"";}$ 
    foreach x  $\in startX, \dots, endX$  do
        world.getChunk(x/16, z/16);
        y  $\leftarrow$  world.getTopY(x, z);
        line  $\leftarrow$  line + (y - minHeight) + ",";
    lines.add(line);
```

Files.write(path, lines);

Unlike the other two methods, the heightmap saving methods have to load the chunk at which we are finding the highest block of, since areas beyond the player's render distance centred on where the method was called are not loaded. A "chunk" in Minecraft refers to a 16x16 block area (in the x and z axes) [36], so we can simply find the integer result of dividing the current block's x and z coordinates by 16 to find the chunk it is in. We want to store the displacement of each block in the region from sea level, which is why we find the minimum height in the region first and then iterate through every block at the topmost y-coordinate in the region, subtracting this minimum height. '*saveHeightmap()*' is almost the same as '*saveHeightmapAsArray()*', but it also takes the maximum height of the region, which gives us the range of values to scale the heightmap to (normalisation); after normalising to the range 0-255, we can get greyscale values for each top block.

Algorithm 2: saveBlockDataCSV()

Input: *path* (file path), *regionCoordinates* (pair of coordinates for the region), *world* (Minecraft world object)

Output: Block data saved as a CSV file

```
Initialize startCoordinate ← regionCoordinates.startCoordinate();
Initialize endCoordinate ← regionCoordinates.endCoordinate();

// List to hold rows of data in the format (Y, Z, X, BlockState)
Initialize lines ← ∅;

// Stream through all block positions in the region and collect data
foreach
    blockPos ∈ BlockPos.stream(startCoordinate.x(), startCoordinate.y(), startCoordinate.z(),
                                endCoordinate.x(), endCoordinate.y(), endCoordinate.z()) do
        lines.add(blockPos.getY() + "," + blockPos.getZ() + "," + blockPos.getX() + "," +
                  world.getBlockState(blockPos));

// Write the data to the file
Files.write(path, lines);
```

‘*saveBlockDataCSV()*’ is a simple method that iterates through every block in the region and saves the block’s coordinates and block state (the type of block) into a CSV file. We store them in Y, Z, X order to make it easier to parse the data later when we store the frequencies of blocks at each Y layer.

The structure of the JSON file produced by ‘*saveBlockDataRLE()*’ is as follows:

- **start:** The starting XYZ coordinates of the region.
- **mappings:** A list of mappings from block names to unique symbols.
- **grids:** An array of 2-D grids, each representing a Y layer of the region. Each grid is a list of strings, where each string is a run-length encoded row of blocks.

Algorithm 3: generateUniqueSymbol()

Input: *blockName*: Name of the block (e.g. “minecraft:stone”), *usedSymbols*: Set of used symbols (e.g. [“S”, “ST”, “G”])

Output: *symbol*: A unique symbol for the block name

Function generateUniqueSymbol(*blockName*, *usedSymbols*):

```
Data: baseName: Extract the block name after “minecraft:”
baseName ← blockNamecontains” : ” ? blockName.split(” : ”)[1] : blockName;

for length ← 1 to baseName.length() do
    symbol ← baseName.substring(0, length).toUpperCase();
    if symbol not in usedSymbols then
        return symbol;

foreach specialChar in {!, @, #, $, %, &, *, (, ), -, _, =, +} do
    symbol ← baseName.substring(0, 1).toUpperCase() + specialChar;
    if symbol not in usedSymbols then
        return symbol;

return “X” // Fallback symbol if all combinations are used
```

Algorithm 4: saveBlockDataRLE()

Input: path: Path to save data, regionCoordinates: Coordinates of region, world:

Minecraft world object

Output: Saves block data in RLE format to the given path

Function saveBlockDataRLE(path, regionCoordinates, world):

```
    Data: startCoordinate, endCoordinate: Coordinates from regionCoordinates
    Data: data: A map to store the data
    Data: start: A map for start coordinates
    Data: mappings: A list to store block mappings
    Data: usedSymbols: A set to track used symbols
    Data: grids: A list to store the encoded grids for each Y layer

    start["x"] ← startCoordinate.x();
    start["y"] ← startCoordinate.y();
    start["z"] ← startCoordinate.z();
    data["start"] ← start;

    for y ← startCoordinate.y() to endCoordinate.y() do
        grid ← List of Lists;
        for z ← startCoordinate.z() to endCoordinate.z() do
            encodedRow ← StringBuilder;
            runLength ← 1;
            lastSymbol ← null;
            for x ← startCoordinate.x() to endCoordinate.x() do
                blockState ← world.getBlockState(x, y, z);
                blockName ← blockState.getBlock().toString();
                if blockName starts with "Block" and blockName ends with "" then
                    blockName ← blockName.substring(6, blockName.length() - 1);
                symbol ← symbolMapping.computeIfAbsent(
                    blockName,
                    λ : generateUniqueSymbol(blockName, usedSymbols));
                if symbol = lastSymbol then
                    runLength ← runLength + 1;
                else
                    if lastSymbol ≠ null then
                        encodedRow.append(runLength + lastSymbol);
                    runLength ← 1;
                    lastSymbol ← symbol;
                if lastSymbol ≠ null then
                    encodedRow.append(runLength + lastSymbol);
                grid.add(encodedRow.toString());
            grids.add(Map.of("y", y, "grid", grid));

            data.put("mappings", mappings);
            data.put("grids", grids);
            Files.writeString(path, toJson(data));
```

Algorithm 3 shows how we generate a unique symbol for each block name when saving the block data in a run-length encoded format. The block name is used to generate a unique symbol for it, and if the symbol is already in use, we try a different combination of characters.

Prompting

A Node.js [73] server was used to send requests to the Gemini API through Google Cloud’s Vertex AI [55] platform. The model was initialised with its system instructions (Figure 2) and default parameters of 8192 max output tokens, a temperature of 1, and a top-p of 0.95. Temperature is a hyperparameter that controls the randomness of the output [74], with a higher temperature leading to more randomness. Top-p [75] is a hyperparameter that controls the nucleus sampling, which is a method of sampling from the model’s output distribution by only considering the most likely tokens whose cumulative probability exceeds a certain threshold. I kept the three parameters constant for all requests to the model, but they could be changed to see how they affect the model’s output — one method prompt engineers use is having a higher temperature for the initial tuning of the model, then lowering it to get more rational outputs and for testing.

All requests are asynchronous and handled by a function ‘*sendMessage()*’ which is responsible for waiting for a response from the API, accessing the relevant ‘text’ field within the JSON response, and then printing and saving this output. To send 300 screenshots to the model, we would call ‘*sendMessage()*’ once with one array that stores the text of ‘Here are the 300 screenshots:’ and 300 JSON objects that state the type of multimodal input (png) and the image data encoded in base64. From the API perspective, it would receive this image data, decode it, and then pass it on to its Convolutional Neural Network (CNN) which would in turn pass the extracted features to the model. The model would then generate a prompt based on the extracted features and the system instructions.

Iterations and Hallucinations

For a full list of the first 5 inputs to the model after it analysed the 300 screenshots, as well as the model’s outputs (with images of the generated terrain), see Appendix B.1. Here are snippets from the first two iterations of inputs and outputs, in conjunction with screenshots of the generated terrains:

From Figure 11, we can see that the terrain generated from this approach is already much more closer to the original game than the low-level attempts (Figure 4). The terrain has hills, a river, lakes, forests, grass, flowers, and a village, all of which are features of a plains biome in Minecraft. The terrain is also more varied than the low-level attempts, with the hills having a more natural shape and the river winding through the terrain. The only potential hallucination-like issue is that the parameters for the commands to spawn grass and flowers had them with centres at (0, 0) (the bottom-left corner of the region), and counts and radii of 255 — the same as the width/length of the regions. This means that the grass and flowers will not spawn in areas such as the top-right corner of the region, as the distance from (0, 0) to (255, 255) is greater than the radius of 255.

Input

User: Now that you have understood the general structure of Plains biomes in Minecraft, what features, structures, vegetation and terrain appear, I would like you to produce a list of functions with parameters to build such a biome in-game on top of a flat terrain that has XYZ coordinates from (0, 64, 0) to (255, 64, 255), where north is in the direction of negative Z and east is in the direction of positive X. Please separate these functions line-by-line starting them with "/generate" (without speech marks), including what to generate and its specific parameters. The possible things you can generate are listed below in the order you should write them (we prioritise terrain shape, then water features, vegetation and finally structures):

[Function definitions have been redacted for brevity]

Please use any number of any of the commands available, making sure to use their names and parameters correctly (and in order) after the "/generate" text. Think about how the terrain will develop after these commands are ran and how they all interplay with each other to provide a fun experience for players of a "quintessential" plains biome in Minecraft.

Output

Model: Here's a possible set of commands to generate a quintessential Plains biome in Minecraft within the specified area, prioritizing terrain, water, vegetation, and then structures:

```
/generate hill 0 0 255 255 15 // Gentle rolling hills, max height variation of 15 blocks  
/generate river 10 20 245 230 5 // A winding river across the landscape  
/generate lake 60 100 circle 20 // A medium-sized circular lake  
/generate lake 180 50 circle 12 // A smaller circular lake  
/generate forest 50 50 30 low // A small, sparse forest  
/generate forest 200 200 40 med // A larger, denser forest  
/generate grass 0 0 255 255 med // Medium density grass across the entire plains  
/generate flowers 0 0 255 255 low // Sparse flower distribution  
/generate village 128 128 // A village near the center
```

Explanation and Design Choices:

[Explanation and design choices have been redacted for brevity]

Considerations for Improved Generation:

[Considerations for improved generation have been redacted for brevity]

This improved set of commands, coupled with your original terrain analysis, provides a solid framework for procedurally generating more dynamic and realistic Plains biomes in Minecraft.

Figure 10: First input and output from the model



Figure 11: First generated terrain

Input

User: Here is a screenshot of the generated terrain from these commands
[Image of the generated terrain has been redacted for brevity]

Based on this screenshot, consider what to improve on when generating another terrain. My own feedback about this generated code is that potentially more code could be written for things like grass and flowers, rather than having radii and counts the same as the length of the region; as well as this, centering at (0, 0) means that there will be one corner that isn't covered if the aim is to cover the whole region. After understanding how to improve, please generate another terrain with the commands.

Output

Model: You're right, the previous generation had some weaknesses. Generating grass and flowers with such a large radius wasn't efficient, and the centering meant incomplete coverage. I also think the forests were too close to the village, which is less common in natural generation. The terrain itself could also use a bit more variation.

Here's a revised set of commands aiming for a more natural and interesting Plains biome, addressing those points:

```
/generate hill 0 0 255 255 20 42 // Slightly higher hills for more
variation, using seed 42
/generate river 10 20 245 230 4 87 // Winding river, slightly narrower,
using seed 87
/generate lake 70 120 circle 15 123 // Shifted lake position, using seed 123
/generate lake 190 60 circle 10 456 // Shifted lake position, using seed 456
/generate forest 40 40 25 low 789 // Smaller, sparse forest, moved away from
center, using seed 789
/generate forest 210 210 30 med 101 // Larger, denser forest, moved away
from center, using seed 101
/generate grass 64 64 64 med // Grass around the central area
/generate grass 192 64 64 med // Grass around the second area
/generate grass 64 192 64 med // Grass around the third area
/generate grass 192 192 64 med // Grass around the fourth area
/generate flowers 0 0 255 255 low 202 // Sparse flowers everywhere
/generate village 128 128 303 // Village near the center
```

Changes and Reasoning:

[Some changes and reasoning have been redacted for brevity]

* **Multiple Grass Patches:** Used multiple grass commands to cover each quarter of the map individually. The radius 64 covers a quarter of the 255 block length from the center of each quadrant, which fully covers the area without leaving gaps at the corners of each grass-generating call. This approach is more efficient and ensures complete coverage.

* **Consistent Flower Coverage:** The flowers now cover the entire area consistently with the original single command due to low density.

* **Seed Values:** Added seed values to all commands for reproducible results and better control over the randomness.

This revised generation strategy addresses the previous shortcomings and aims to create a more dynamic and "quintessential" Minecraft Plains biome. The inclusion of seed values allows for greater control and repeatability.

Figure 12: Second input and output from the model, here we are refining the output from the last iteration.



Figure 13: Second generated terrain (without seeds)



Figure 14: Second generated terrain (with seeds)

Figures 13 and 14 show two terrains generated from the same generation commands. The model seemed to have hallucinated additional seed parameters to each command — most likely a result of its self-assessment on the first iteration, where it stated the benefits of using seeds for reproducibility and control over randomness. It decided itself to include a new parameter for each command without my input, which could lead to potential issues in the future if it determines that it is able to write commands for parameters that don't exist or even whole new commands. This seed was added to every command, despite some, like villages, not being impacted from such a parameter (the randomisation of the village generation is handled natively by the game).

This led to me deciding to “go along” with the hallucination and include parsing of seeds within the mod, so that the model could generate seeds for each command and be able to pick and choose what aspects of terrain generation it wanted to keep or change. This approach is also taken by Hu *et al.* [10] in their work on generating entire games from Video Game Description Language (VGDL), where in their case the model hallucinated the existence of a function with a similar name to that of an existing one, with the only difference being that it swapped the order of the parameters. They were not able to correct the model, so instead they decided to ‘follow’ its syntax and implement the new function. For the instances where seeds make no impact, I corrected the model, and it followed these instructions from the next iteration.

One other hallucination in this iteration was the flower generation command having an extra parameter, it is clear that it must have believed that its first four parameters were starting and ending X and Z coordinates, despite the command only needing a centre X and Z coordinate and a radius. This hallucination was pointed out and promptly corrected in the next iteration.

The model is able to adapt to changes in both its main instructions as well as the definitions of the functions it generates, as evidenced by the prompt used to generate the fifth iteration, where I had asked the model to change the priority of structure (village) generation to be relatively higher than vegetation due to glitches such as village paths spawning on top of grass and flowers. The model was able to understand this change, and for all following outputs, it had ‘*/generate village*’ commands higher than the ‘*/generate grass*’, ‘*/generate forest*’ and ‘*/generate flowers*’ commands. In the same prompt, the model was told that the forest generation function parameters had changed, and even in this case, the model was able to understand the change and generate the correct functions.

Automation in the Low-Level Approach

My original approach still followed an event-driven architecture (Figure 7) for the automation of terrain generation, with the only difference being the context provided to the model. Ideally, if there were not as constraining resource limits, the RLE data, block frequencies in each y-level and heightmap data of the region surrounding the new area to create would be sent to the model.

Sending the raw data would have always been a big ask, as each file can take over 1GB of space, and even if this much context could be provided in a request, the speed at which a response would be received would be extremely slow. This is why I created a script *block_data_compression.py* to extract the top 3 most common blocks in each y-level of the region (from sea level to the maximum height of the region) along with their frequencies as percentages. To handle parsing over 1GB of data, I used the *pandas* library to read the CSV file in chunks and then used the *collections* library to count the frequencies of each block in each y-level. The result is a JSON with the structure:

- **layers:** An array of up to 64 objects that store:
 - **y:** The y-level of the layer
 - **types:** An array of up to 3 objects that store:
 - * **block:** The block type
 - * **frequency:** The frequency of the block type in the layer

Both the JSON files (RLE data and Block-Frequency data) are indented in such a format that they are either too indented or not at all ('pretty' or 'compact'); both of these formats are not ideal for the model to parse in terms of token count — despite having the exact same information, the model parses them differently; compact format is very hard to read and understand, while the pretty format has a lot of unnecessary tokens. To solve this, I made use of a node module named “*json-stringify-pretty-compact*” [76], which stringifies a JSON by attempting to put each item of arrays and objects on a single line, and then if it exceeds a certain length, it will put each item on a new line. This way, the JSON is still readable and has a low token count.

In an automated version of this low-level approach, the intermediary dynamic content handler is given extra responsibilities, in the form of watching for new raw data files to process, and JSON files to stringify into the pretty compact format.

Tuning the Model

To obtain the 300 screenshots of Plains biomes used to tune the model and initiate its Chain of Thought reasoning, to improve accuracy in the generation stage, we need a systematic approach that ensures that the screenshots are sampled randomly and automatically without human input. To achieve this, I created an item in the game (*ScreenshotCameraItem.java*) that would save a screenshot of the player’s current view when right-clicked, saving to a custom directory, as opposed to the game’s default screenshot directory.

Screenshot saving can only be done on the client-side component of Minecraft, as the integrated server does not have a rendering engine, and as such is not capable of passing the current framebuffer on the player’s screen to be saved. The hidden complexity lies in the fact that the client-side logic that handles the actual saving of the screenshot cannot be written in the *items* and *misc* packages of the mod, as these are server-side packages. Instead, an interface, *IScreenshotHandler.java* was created in the *misc* package and implemented in the *ClientUtils.java* class in the *utils* package (on the client-side). Upon the *LLMTerrainGenerationClient* entrypoint being loaded, it “injects” the *ClientUtils* class into the *screenshotHandler* field of the *ModItems* class, the field being a singleton instance of the *IScreenshotHandler* interface. This way, the *saveScreenshot()* method of the *ClientUtils* class can be called on usage of *ScreenshotCameraItem*.

To automatically take screenshots, I created an automation script (*screenshot_automation.ahk*) using a software called AutoHotkey [77]. The script presses the keys that teleport the player to a random location in the world, waits for 20 seconds for the world to load, and then right-clicks the *ScreenshotCameraItem* to save a screenshot. This process is repeated 300 times, with the script waiting for 2 seconds between each screenshot to ensure that the game has saved the screenshot before moving the player to a new location.

Dynamic Content Generation

Working under the event-driven architecture (Figure 7), the dynamic content handler stores a collection of the terrain tiles that have already been generated, and when a new request is made, it gets a list of the coordinates of the surrounding tiles and then sends a request to the model with the context of the surrounding tiles and the new tile to generate.

In my implementation, the handler is a Node.js server that:

1. Watches for new terrain request files created by the Fabric mod.
 - We want the game or its integrated server to slow down as little as possible by not stopping the game till there is a response to this content request, instead checking for a response in the main game loop every N seconds (30 in this case) since the request was sent, until a response file is found containing the generation commands.
2. Stores any terrain requests in a queue, and when is its turn to be sent to the model, the server searches to see which of its surrounding tiles have already been generated and sends the context of these tiles along with the new tile to the model.
 - It will check to see if any of the 8 possible surrounding tiles of the new tile are in the generated tile collection (Figure 15).
3. Sends a request to the model with the context of the surrounding tiles and the new tile to generate.
 - Practically, there is no real need for the request to be asynchronous, we could wait for each request in the queue to be processed one at a time, since the likelihood is that the proceeding requests will be very close to the tile at the front of the queue (most likely the player crosses two boundaries that are next to each other).
 - Waiting may mean that we gain additional context for the next tile to generate, potentially improving the quality of the generation.
 - As a bonus, if we wait for each request to be processed one-at-a-time, we can also ensure that the model is not overwhelmed with requests, and that the game is not slowed down by the server.
4. Parse the response from the model and save it as a file that the Fabric mod can read and generate the terrain from.
 - We can strip the response of all but the necessary generation commands.
 - It is also at this stage where we can run a testing pipeline on the response to ensure that the model has not hallucinated in a way that would cause invalid or undefined behaviour in the game.

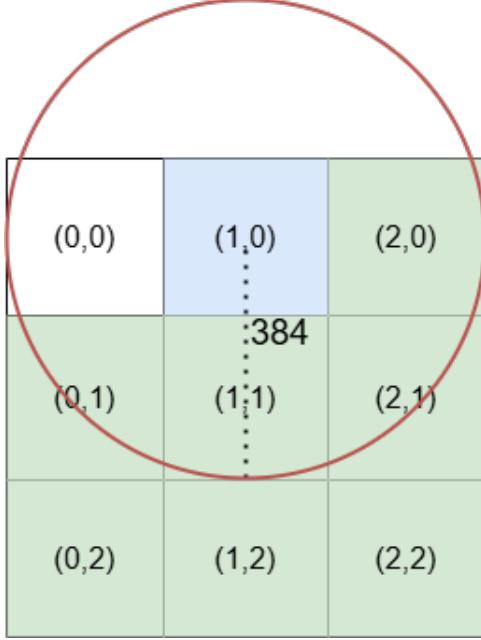


Figure 15: If the player crosses the red boundary, a circle centred on the middle of the blue tile with a radius of 1.5 tile widths (384 blocks), the blue tile is triggered to be generated. We can also see that if a player has explore each of the regions on the outside of the circle, then the blue tile will have had all of its surrounding tiles generated, so we could pass all 8 tiles as context for the model to generate the blue tile.

Ticks are a unit of time in Minecraft, with 20 ticks being equal to 1 second — each tick represents one whole iteration of the main game loop. We would store a hashset of previously explored tiles, and then every tick, we check to see if the player is in such a tile, and if they are not, we can request for its ungenerated adjacent tiles to be generated (Algorithm 5). There will be up to three adjacent tiles that need to be generated, and we can determine which three these are by understanding which direction the player crossed the boundary of the tile in. For example, if the player crossed the boundary of the previously explored tile in the positive X direction, then the tile to the east of the tile the player is in will need to be generated; we would then also check for the diagonals (north-east and south-east) to see if they need to be generated. This solution reduces the time complexity involved in checking for the surrounding tiles to generate from potentially $O(n)$ to $O(1)$, where n is the number of ungenerated tiles adjacent to the border of what has already been generated — this is a significant improvement in performance, especially as we want our generation system to scale to potentially infinite worlds.

Algorithm 5: Check and Generate Adjacent Tiles Based on Player Movement

```
Function checkAndGenerateTiles(playerX, playerZ, exploredTiles, generatedTiles):
    tileSize ← 256;
    currentTile ← ( $\lfloor playerX / tileSize \rfloor$ ,  $\lfloor playerZ / tileSize \rfloor$ );
    if currentTile  $\notin$  exploredTiles and currentTile  $\in$  generatedTiles then
        exploredTiles.add(currentTile);
        // Determine edge crossed
        previousTile ← getPreviousTile();
        dx ← currentTile.x - previousTile.x;
        dz ← currentTile.z - previousTile.z;
        tilesToGenerate ← [];
        // Check direction of movement and add adjacent ungenerated tiles
        if dx = 1 then
            Add tile east of currentTile to tilesToGenerate;
            // Check diagonals
            if (currentTile.x + 1, currentTile.z + 1)  $\notin$  generatedTiles then
                Add tile north-east of currentTile to tilesToGenerate;
            if (currentTile.x + 1, currentTile.z - 1)  $\notin$  generatedTiles then
                Add tile south-east of currentTile to tilesToGenerate;
        else if dx = -1 then
            Add tile west of currentTile to tilesToGenerate;
            // Check diagonals
            if (currentTile.x - 1, currentTile.z + 1)  $\notin$  generatedTiles then
                Add tile north-west of currentTile to tilesToGenerate;
            if (currentTile.x - 1, currentTile.z - 1)  $\notin$  generatedTiles then
                Add tile south-west of currentTile to tilesToGenerate;
        if dz = 1 then
            Add tile south of currentTile to tilesToGenerate;
            // Check diagonals
            if (currentTile.x + 1, currentTile.z + 1)  $\notin$  generatedTiles then
                Add tile south-west of currentTile to tilesToGenerate;
            if (currentTile.x - 1, currentTile.z + 1)  $\notin$  generatedTiles then
                Add tile south-east of currentTile to tilesToGenerate;
        else if dz = -1 then
            Add tile north of currentTile to tilesToGenerate;
            // Check diagonals
            if (currentTile.x + 1, currentTile.z - 1)  $\notin$  generatedTiles then
                Add tile north-west of currentTile to tilesToGenerate;
            if (currentTile.x - 1, currentTile.z - 1)  $\notin$  generatedTiles then
                Add tile north-east of currentTile to tilesToGenerate;
    // Generate tiles
    foreach tile  $\in$  tilesToGenerate do
        if tile  $\notin$  generatedTiles then
            generateTile(tile);
```

Taxonomy of Hallucinations

To understand and systematically address the errors that arise during terrain generation, I have classified hallucinations into four main categories based on their impact and origin. Additionally, potential future hallucinations that may occur as the system handles larger contexts and more complex tasks have been listed.

1. **Parameter Hallucinations:** Errors related to the generation or interpretation of parameters within commands.
 - Extra parameters (e.g., adding "seed" to commands where it is unsupported).
 - Missing parameters (e.g., omitting radius for flower generation).
 - Invalid parameter values (e.g., out-of-range numbers, such as a hill height of -10 or a radius larger than the region itself).
 - Misalignment of parameters across interdependent commands (e.g., overlapping features due to inconsistent coordinates).
 - Cross-context parameter conflicts (e.g., a global variable like "seed" being inconsistently reused across regions).
2. **Command Hallucinations:** Errors where the model fabricates or misinterprets commands.
 - Creation of nonexistent commands (e.g., inventing a /generate meadow command when no such command exists).
 - Misspelt commands (e.g., /genarate instead of /generate).
 - Inconsistent command syntax due to misremembering definitions when working with multiple regions or worlds simultaneously.
 - Inventing higher-level meta-commands to control multiple regions (e.g., /generate_biome_grid), which may not align with the defined system.
3. **Logical Hallucinations:** Errors in reasoning or understanding instructions, leading to illogical outputs.
 - Poor prioritisation of commands (e.g., generating vegetation before a village, causing paths to be built on top of flowers and grass).
 - Overlapping or redundant commands (e.g., generating two forests in the same area).
 - Region edge mismatches, where commands for adjacent regions fail to align properly (e.g., rivers or roads abruptly stopping at region boundaries).
 - Misunderstanding global constraints or goals (e.g., failing to maintain biome diversity across a larger map).
4. **Semantic Hallucinations:** Errors in understanding how parameters, commands, or features interact, leading to unintended behaviour.
 - Misinterpreting coordinate systems (e.g., assuming radii extend from the bottom-left instead of the centre).
 - Generating commands that contradict biome logic (e.g., adding a desert feature in a plains biome).
 - Contradictions across contexts (e.g., generating features that do not match neighboring regions' biomes, such as a forest abruptly bordering an ocean without a beach).

- Semantic drift over iterations, where incremental changes subtly diverge from the intended biome characteristics.

Anticipating Future Hallucinations

As the system evolves to handle larger contexts and more complex scenarios, the following additional hallucinations may emerge:

1. Cross-Contextual Hallucinations:

- **Error:** Mismanaging relationships between regions (e.g., failing to continue a river or road across boundaries).
- **Impact:** Breaks the cohesiveness of the terrain or gameplay experience across regions.
- **Mitigation:** Implement cross-region consistency checks and shared context management to ensure seamless transitions.

2. Memory-Based Hallucinations:

- **Error:** Fabricating commands or features based on outdated contexts or incorrectly storing previously used commands.
- **Impact:** Inconsistent outputs due to reliance on stale or misinterpreted memory.
- **Mitigation:** Implement a memory management system that tracks context changes and updates stored information accordingly.

3. Scaling Hallucinations:

- **Error:** Generating outputs that work well for small regions but fail to scale correctly (e.g., using fixed-size commands for features that should scale with region size).
- **Impact:** Uneven terrain distribution or lack of biome realism in larger maps.
- **Mitigation:** Implement scalable commands and context-aware generation algorithms that adapt to region size and complexity.

We can assign a severity level to each hallucination type based on its impact on the generated terrain and gameplay experience. This is also linked with how difficult it is to spot and correct the hallucination. As LLMs are in essence black boxes, it is challenging to predict the exact nature of hallucinations that may arise, or what these hallucinations are a result of, which is why it is important to account for as much as possible in the system to ensure that model outputs are as good as they can be. Figure 16 shows a taxonomy of possible hallucinations in the terrain generation system, highlighting the relation between longer-term and larger-scale hallucinations/errors alongside the system's ability to handle them; for the highest level, it is arduous even with a human-in-the-loop to correct these errors from collating model output audits.

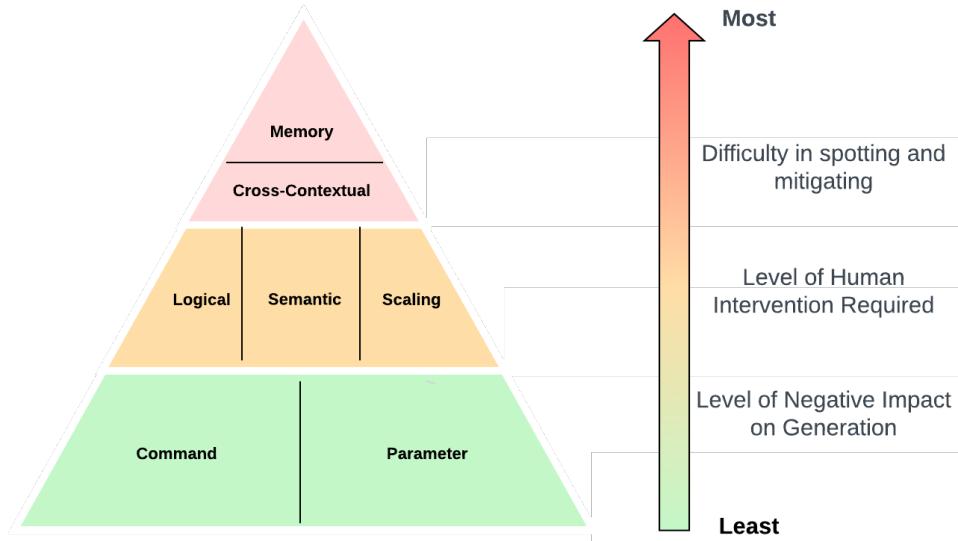


Figure 16: Taxonomy of hallucinations in the terrain generation system

Hallucination Handling

After the model’s output is received and parsed (just the text is saved and all comments are removed), the command list is checked for any hallucinations. The specific hallucinations checked for and how they are rectified are as follows:

- **Number of Parameters:** If the model hallucinates an extra parameter for a command, the parameter is removed from the command. In the case that the hallucination is a missing parameter, the command is corrected by adding the default value for the missing parameter.
- **Command Name:** If the model hallucinates a command name that does not exist, the command is removed from the list. If the command was close in spelling (based on the Levenshtein distance [78] between the hallucinated command and a list of valid commands), the hallucinated command is replaced with the closest valid command.
- **Command Parameters:** If the model hallucinates a parameter value that is out of the range of valid values for that parameter, the parameter is replaced with the closest valid value. If the hallucinated value is a string, the closest valid string is used.
- **Command Prioritisation:** If the model hallucinates the order of commands, the commands are reordered based on the priority of the features they generate. For example, a village command should be placed before a flowers command to ensure that paths are not generated on top of flowers.

For auditing purposes, we save the original model output text in addition to the corrected text. This way, we can see what the model hallucinated and how it was corrected.

Terrain Generation Algorithms

Arguably, the aspect of the mod that has the most impact on the final result on-screen is the terrain generation algorithms. Wherever I could, I used a native solution to spawning a feature or structure; otherwise, a custom algorithm was used with its behaviour as similar to how Minecraft originally generated that feature as possible.

Hills

The hills are generated using a custom algorithm (Algorithm 6) that is similar to the one used by Minecraft. Perlin Noise [79] is used within regular Minecraft to create consistent yet random terrain, and this is what I also decided to use for hill generation. Fabric's API comes with a Perlin Noise generator, for which I chose three octave values and a "squash factor." The octave values determine the number of layers of noise that are added together to create the final noise, and the squash factor determines how flat the terrain is. Using a sigmoid function on the noise value allows us to create a smooth transition from the bottom of the region to the top, and the noise value is compared to this transition to decide whether to place a block or not.

The block placed is a stone block, which is the most common block in the game and is used as the base for all other blocks. The top layers of the hill are then modified to include grass blocks, and the top surface Y is used to determine where to place the grass blocks. At least two grass blocks are placed from the top, with the depth of placement having a 33% chance of increasing to three, and a 33% chance to be four. As seen in the logic of Algorithm 6, iterating deeper from the last grass block placed at each x, z coordinate, there is an 8% chance of placing a coal ore block and a 2% chance of placing an iron ore block. This is to simulate the natural generation of ores in the game, where they are more common at lower y-levels.

The parameters that the LLM has control over are: the bottom-left and top-right coordinates of the hill area, the maximum amplitude of a hill, and the seed for the random number generator.

Algorithm 6: generateHill()

Input: *parameters* (coordinates, amplitude, seed)

Output: Boolean indicating success of hill generation.

Parse Input Parameters:

```
bottomLeftX ← Integer.parseInt(parameters[0])
bottomLeftZ ← Integer.parseInt(parameters[1])
topRightX ← Integer.parseInt(parameters[2])
topRightZ ← Integer.parseInt(parameters[3])
maxAmplitude ← Integer.parseInt(parameters[4])
seed ← If parameters[5] is empty, use 0 else parse as Long;
```

Initialize:

```
random ← Create random generator with seed;
bottomY ← Get top surface Y at (bottomLeftX, bottomLeftZ);
squashFactor ← 0.1;
```

Step 1: Generate Base Terrain

Call

```
placeTerrain(bottomLeftX, bottomLeftZ, topRightX, topRightZ, bottomY, bottomY +
maxAmplitude, squashFactor, random);
```

Step 2: Modify Top Layers

```
foreach xCoord ∈ [bottomLeftX, topRightX] do
```

```
    foreach zCoord ∈ [bottomLeftZ, topRightZ] do
        y ← Get top surface Y at (xCoord, zCoord) - 1;
        randomResult ← random.nextInt(0, 100);
```

Place 2–4 Grass Blocks:

```
SetBlockState((xCoord, y --, zCoord), GrassBlock);
SetBlockState((xCoord, y --, zCoord), GrassBlock);
if randomResult > 33 then
    SetBlockState((xCoord, y --, zCoord), GrassBlock);
end
if randomResult > 66 then
    SetBlockState((xCoord, y --, zCoord), GrassBlock);
end
```

Random Ore Placement:

```
foreach yCoord ∈ [y, bottomY] do
    if random.nextInt(0, 100) < 8 then
        SetBlockState((xCoord, yCoord, zCoord), CoalOre);
    end
    if random.nextInt(0, 100) < 2 then
        SetBlockState((xCoord, yCoord, zCoord), IronOre);
    end
end
end
```

```
return true;
```

Algorithm 7: placeTerrain()

Input: $bottomLeftX, bottomLeftZ, topRightX, topRightZ, bottomY, topY,$
 $squashFactor, random$

Output: Terrain blocks set in the specified region.

Initialize $sampler \leftarrow \text{CreatePerlinNoiseSampler}(random, 0, 3, 0.1, 0.05, 0.01);$

foreach pos **in**

```
BlockPos.stream( $bottomLeftX, bottomY, bottomLeftZ, topRightX, topY, topRightZ$ ) do
    noise  $\leftarrow sampler.sample(pos.X/50, 0, pos.Z/50);$ 
    val  $\leftarrow 2 \times \text{sigmoid}((pos.Y - bottomY) \times squashFactor) - 1;$ 
    if  $noise > val$  then
        | SetBlockState( $pos$ , StoneBlock);
    end
    else
        | SetBlockState( $pos$ , AirBlock);
    end
end
```

Lakes and Rivers

Water feature generation in Minecraft is due to a number of factors, but the most normal way in world generation is from all the elevation in the terrain having been determined, and then any air blocks at $y=62$ (sea level) or lower are replaced with water blocks. This is not the approach I took, however, as this type of generation doesn't include any involvement from the LLM. Instead, I decided to make algorithms for generating square or circular lakes (Algorithm 8), and rivers (Algorithm 9).

For circular lakes, the LLM has control over the centre of the lake and its radius, and if it is a rectangular lake, it decides the length and width of the lake. There is no seed parameter for lakes as there is no randomness to their generation. To avoid the lake generating in a way that is floating above the terrain, the algorithm finds the lowest block in the lake area and then fills the lake at that height.

Rivers are generated by determining a start and end point, a width, and a seed. The algorithm then generates a path from the start to the end point, with the path being either vertical or horizontal, and then at each point on the path, it generates a river of the specified width. The river generation is done by checking the blocks around the current block and then filling in the blocks that are within the width of the river. The algorithm also has a 40% chance of changing the direction of the river at each point, to make the river more winding and natural.

Algorithm 8: generateLake()

Input: *parameters* (coordinates, shape type, size, optional width)

Output: Boolean indicating success of lake generation.

Parse Input Parameters:

```
x, z ← Integer.parseInt(parameters[0]), Integer.parseInt(parameters[1]);
isCircle ← (parameters[2] = "circle");
lengthOrRadius ← Double.parseDouble(parameters[3]);
if isCircle ∧ (lengthOrRadius%1 = 0) then
    lengthOrRadius ← lengthOrRadius + 0.5 // Circles look better with 0.5
    radii.
end
width ← If isCircle use lengthOrRadius else Integer.parseInt(parameters[4]);
lowestY ← Get top surface Y at (x, z) - 1;
```

Step 1: Circle Lake Generation

if *isCircle* **then**

```
    top ← [z - lengthOrRadius], bottom ← [z + lengthOrRadius];
    left ← [x - lengthOrRadius], right ← [x + lengthOrRadius];
```

Find Lowest Y Level:

```
foreach i ∈ [top, bottom], j ∈ [left, right] do
    | lowestY ← min(lowestY, Get top surface Y at (i, j) - 1);
end
```

Place Water Blocks:

```
foreach i ∈ [top, bottom], j ∈ [left, right] do
    | if (j - x)2 + (i - z)2 ≤ lengthOrRadius2 then
        | SetBlockState((j, lowestY, i), WaterBlock);
    end
end
return true;
```

end

Step 2: Square Lake Generation

left ← $x - width/2$, *right* ← $x + width/2$;

top ← $z - lengthOrRadius/2$, *bottom* ← $z + lengthOrRadius/2$;

Find Lowest Y Level:

```
foreach i ∈ [left, right], j ∈ [top, bottom] do
    | lowestY ← min(lowestY, Get top surface Y at (i, j) - 1);
end
```

Place Water Blocks with Fill Command:

ExecuteCommand("/fill" (left, lowestY, top) "to" (right, lowestY, bottom)

"minecraft:water");

return true;

Algorithm 9: generateRiver()

Input: *parameters* (start and end coordinates, river width, seed)

Output: Boolean indicating success of river generation.

Parse Input Parameters:

```
startX, startZ ← Integer.parseInt(parameters[0]), Integer.parseInt(parameters[1]);
endX, endZ ← Integer.parseInt(parameters[2]), Integer.parseInt(parameters[3]);
width ← Ensure odd Integer.parseInt(parameters[4]);
seed ← Long.parseLong(parameters[5]);
random ← Random(seed);
```

Initialization:

```
currentX, currentZ ← startX, startZ;
vertical ← random.nextBoolean();
positions ← ∅;
```

River Path Generation:

```
while currentX ≠ endX or currentZ ≠ endZ do
    foreach i ∈ [0, width/2] do
        if vertical then
            | Update lowestY and add (currentX ± i, currentZ) to positions;
        end
        else
            | Update lowestY and add (currentX, currentZ ± i) to positions;
        end
    end
```

Change Direction (40% chance):

```
if random.nextInt(0, 100) < 40 then
    if currentX = endX then
        | vertical ← true;
    end
    else if currentZ = endZ then
        | vertical ← false;
    end
    else
        | vertical ← random.nextBoolean();
    end
end
```

Move in Current Direction:

```
if vertical then
    | currentZ ← currentZ + sign(endZ - startZ);
end
else
    | currentX ← currentX + sign(endX - startX);
end
```

end

Place Water Blocks:

```
foreach (x, z) ∈ positions do
    | Set block at (x, lowestY, z) to WaterBlock;
end
return true;
```

Grass, Forest, Flowers, Village

Minecraft comes with a command called '*/place*' which allows for the placement of features and structures such as villages, grass, trees, and flowers. The LLM has control over the density of the grass, flowers, and trees, as well as the centre and radius of the placement area. The village is placed centred at the X and Z coordinates given, but the LLM has no control over anything else about the village, as the village generation in Minecraft is already randomised.

For densities of grass, flowers, and trees, I have defined 'low,' 'medium,' and 'high' options for the LLM to choose from, where they each act as a multiplier when deciding how many grass, flowers or trees to place. Additionally, there is a seed parameter available for these commands.

Evaluation

Qualitative Evaluation

Due to the basis of the project being the generation of terrains in games, I believe that the most important factor in the success of such a system is the level of enjoyment that the player gets from playing in the generated terrains. If we can replicate the visuals and gameplay of the original game and its terrains, then the player should be able to enjoy the game at least as much as they would have if they were playing the original game. Even if the player is able to tell that the terrain was generated by a computer, the enjoyment that they get from playing the game should not be diminished.

The two major questions I set out to answer were:

1. Can players tell the difference between naturally-generated terrains and LLM-generated terrains?
2. Do players enjoy playing in LLM-generated terrains as much as they do in naturally-generated terrains?

Inspired by the usage of the Triangular Test [41] by Henderson in their evaluation of player's ability to distinguish between levels of network delay [42], I decided that a similar approach would be the best way to evaluate the effectiveness of the system. The Triangular Test is a method of comparing two products, A and B, by presenting the user with three samples, two of which are the same and one of which is different. The user is then asked to identify the odd one out. This method is particularly useful when the difference between the two products is subtle, as it allows the user to focus on the specific difference between the two products. In the case of this project, the two "products" are naturally-generated Minecraft terrains and LLM-generated terrains.

Experiment Design

To find participants, I sent out an email to all students studying Computer Science at the university, only revealing the premise of the experiment to be related to the enjoyment of various tasks in Minecraft. This was to avoid any bias that might be introduced by participants who would look more closely at the terrains in order to try and identify the differences between them. The email contained a link to a scheduling application online where participants could sign up for a time slot to participate in the experiment. All participant data was anonymised after the experiment was completed, in accordance with the CS Artifact Evaluation Form 1. A total of 13 people agreed to take part in the experiment, with 10 of them completing all three sets, and 3 of them completing only the first two, stopping early for various reasons.

The experiment consists of each participant playing a tutorial that would teach them the basics of the game, followed by small tutorials of the specific tasks that they would be given over the course of the experiment. This was to ensure that beginners could spend more time taking part in the tasks, than learning how to play the game during them. The participants would then be given three sets of three tasks, with each task having a 4-minute time limit; the tasks are not completable, and participants are informed of this before they begin — this is so that they do not feel pressured to "complete" it in any way. In between each set, the short form shown in Figure 19 would be shown to the participants, asking them to rate their enjoyment of the tasks, their difficulty with them, as well as if there was a specific scene in a set that stood out from the other two.



(a) Villager Trading Tutorial. (b) Iron Mining Tutorial. (c) Flag Example in Tutorial.

Figure 17: Screenshots from the tutorial tasks.

Control Variables

- The specific machine that the participants would be playing on.
- The version of Minecraft that the participants would be playing in.
- The settings of the game, such as the render distance and other graphical settings.
- The terrains (“scenes”) that the participants would be playing in.
- The points at which the participants spawn in each scene.
- The instructions given to the participants.
- The tasks that the participants would be given.
- The set of items provided to the participants for each task.
- The time limit for the participants to play in each scene.
- The number of task sets to complete (3)
- The number of scenes shown per set (3)

Tasks

1. Iron Ingot Farming:

- The participants would be given a pickaxe, a shovel, and a blast furnace. These are items that would speed up the process of mining iron ore and smelting it into iron ingots. The information book provided in-game for the task would explain where to find ore and fuel (coal) in the scene, as well as how to use the blast furnace to smelt the ore into ingots.
- The tutorial has players mining inside of a “mini-hill” to obtain coal and iron, which is then smelted in a blast furnace.
- Mining is a common task in Minecraft, and the aim was to see if the participants would be able to tell the difference when inside of a terrain, rather than just from exploring overground.

2. Flag Building:

- The participants would be given an axe and shears. These are items that would speed up the process of mining wood and shearing sheep. The information book provided in-game for the task would tell players to place a flag at the highest point they can.
- The tutorial has players shearing sheep and being shown an example of what a flag consists of — fences and wool blocks.

- The aim of this task was to see if the participants would be able to tell the difference between the terrains from a high vantage point, as they would be able to see a large portion of the terrain with the chosen render distance setting.

3. Emerald Trading:

- The participants would be given 5 emeralds, which could be helpful as there may be villagers (the entities traded with in the scene) that would provide some resource or tool which could then be used to gain more emeralds. The information book would tell players to get as many emeralds as they could in the time limit.
- In the tutorial, players are presented with two villagers that allow them to trade iron ingots and wool they collected earlier for emeralds.
- This task allows players to show their resourcefulness in obtaining emeralds, as well as their ability to navigate the terrain to find the villagers, as well as find or acquire the items that the villagers would trade for emeralds.

Designing the experiment within the game involved using a tool called Amulet Map Editor [80] to extract terrains from different worlds in the game and then place them all within one world. There were two regular terrains along with 2 LLM-generated worlds — the reason for only having these four terrains being that the primary focus was on detecting differences in the two generation methods, and as such, it is best to keep as much control over randomness in terrain. An added benefit of this approach was that it simplified the process of choosing two of the same terrain type and one different, as we could just pick any random three out of the four. The terrains were placed in such a way that the participants would not be able to see the other terrains from the one they were playing in, and the spawn points were placed manually in order to maximise the time spent completing the task (and not just walking around the terrain).

The maximum time allowed to be spent with a participant is up to 1 hour, which meant that I had to split the gameplay in a way that they could quickly engage in relatively simple tasks within a time period that was not too long that they forgot their feelings in each scene, but not too short that they could not complete the tasks. The time limit for each task was set to 4 minutes which gave ample time to interpret the scene in front of them, as well as carry out the task to some extent. I believed that players would enjoy having three sets of three tasks with 4 minutes each, as opposed to four sets of three with 3 minutes each, as it would allow them to spend more time in each scene, and thus enjoy themselves more, and feel less frustration at not being able to complete the tasks in time. Both these setups were pilot tested by myself in order to determine which would be the best for the participants, and the former was chosen.



(a) One of the LLM-generated terrains shown.



(b) One of the regular terrains shown.

Figure 18: Screenshots from the different terrains in the experiment.

All the logging of task results, teleportation to terrains with correct inventories, and returning to the “hub” after the 4 minutes was automated with the use of custom commands and command blocks that are constantly checking for different triggers each game tick in order to activate. This hub is unlocked after completion of the tutorial and acts as a central point for the participants to return to after each task set; they can take a break before starting the next set of tasks if they wish, and this is also an opportunity for them to fill out the short form (Figure 19). To move on to the next set of tasks, the player simply has to stand on top of the teleportation block in the hub, and the timer starts for each task automatically.

Set 1

Which scene did you think was the odd one out?

1	2	3
---	---	---

How sure are you?

Not sure at all  Very sure

How fun was Scene 1?

Not fun at all  Very fun

How difficult was the task on Scene 1?

Very easy  Very difficult

How fun was Scene 2?

Not fun at all  Very fun

How difficult was the task on Scene 2?

Very easy  Very difficult

How fun was Scene 3?

Not fun at all  Very fun

How difficult was the task on Scene 3?

Very easy  Very difficult

Figure 19: Short Form given to participants after each set of tasks. The form was designed in such a way that the participants would be able to easily mark how they felt about the tasks they had just completed immediately after completing them. This specific form was given to the participants after the first set of tasks, the others only differ in the number attributed to the set. Each line seen was ten centimetres long when printed — this meant for an easier process of collecting the data.

Results

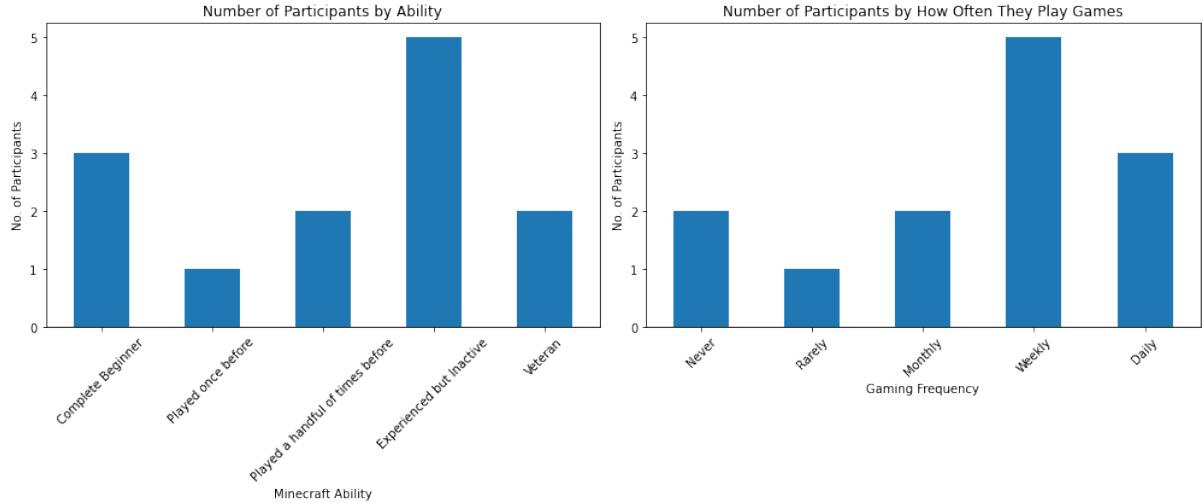


Figure 20: Bar charts showing distribution of participant levels of Minecraft proficiency and how often they play games in general.

After completion of the experiment, participants were asked to fill in a larger feedback form which had: a section dedicated to getting general feedback on the session, how much fun they had overall and such; a section devoted to gaining context on them, asking questions like, “How often do you play video games?” or “How would you rate your Minecraft ability before today’s session?” and a section on their opinions about Generative AI, especially within games. Figure 20 shows that there were only 3 “Complete Beginners,” with the remaining 10 participants having some experience with the game. Appendix C.1 shows the data collected from the larger feedback form, and Appendix C.2 shows an example of a file that aggregates the data of one participant. The value of “Success Rate” is determined by the number of odd ones out that the participant was able to correctly identify, the expected rate being 1 out of 3. “Culled Success Rate” is the same metric, but we disregard correct answers if the “Sureness” value for that answer is less than 0.5. After the user was asked to identify the odd one out, they were asked to rate how sure they were of their answer, with 0 being not sure at all and 1 being completely sure — it is from this question we obtain the Sureness value.

Factor	F-value	p-value
Ability	2.982749	0.266519
How Often	5.248350	0.166397
Gender	15.995201	0.057207
Platform	1.333333	0.428571

Table 2: ANOVA results for success rate.

The ANOVA tables 2 and 3 show that, at a 5% significance level, the only potentially significant factor (i.e., with a p -value < 0.05) was Gender for Culled Success Rate. The problem with this is that the sample size is too small in the female group (4 women participated altogether) for this to be a reliable result. The box plots in Figure 21 do seem to indicate, however, that the more frequently a participant plays games and/or the more proficient they are at Minecraft, the higher the likelihood that they will be able to tell the difference between the terrains. The box plots indicate that the players who had only played Minecraft up to a maximum of one time

Factor	F-value	p-value
Ability	5.925204	0.149581
How Often	10.699226	0.087299
Gender	35.985604	0.026682
Platform	10.329335	0.088266

Table 3: ANOVA results for culled success rate.

and either never played or rarely played video games never guessed correctly. Despite this rate being below the expected 33%, with the small sample size of 3 of the former condition and 4 of the latter, this is still potentially indicative of the fact that to new players, the terrains are indistinguishable.

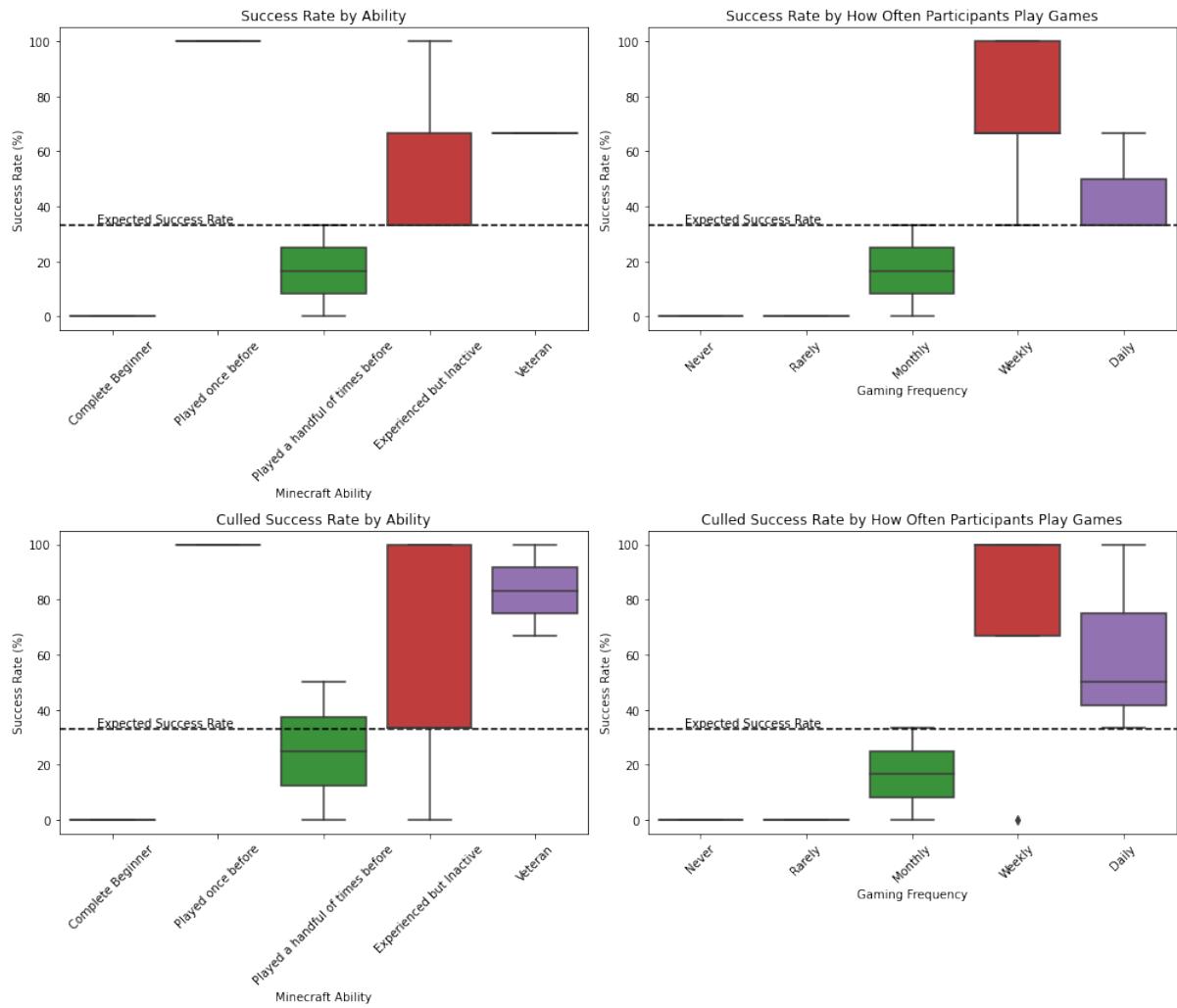


Figure 21: Box plots showing how Success Rate and Culled Success Rate were affected by the Minecraft proficiency and gaming frequency of the participants.

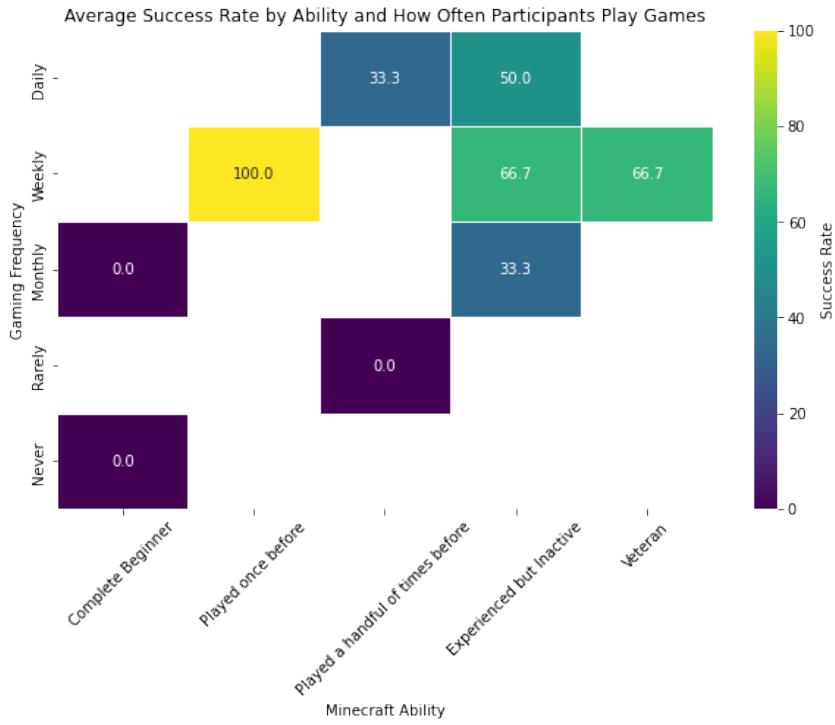


Figure 22: Heat map showing the relationship between the Success Rate of a participant and a combination of their Minecraft proficiency and gaming frequency. Empty squares indicate that there were no participants with that combination of Minecraft proficiency and gaming frequency.

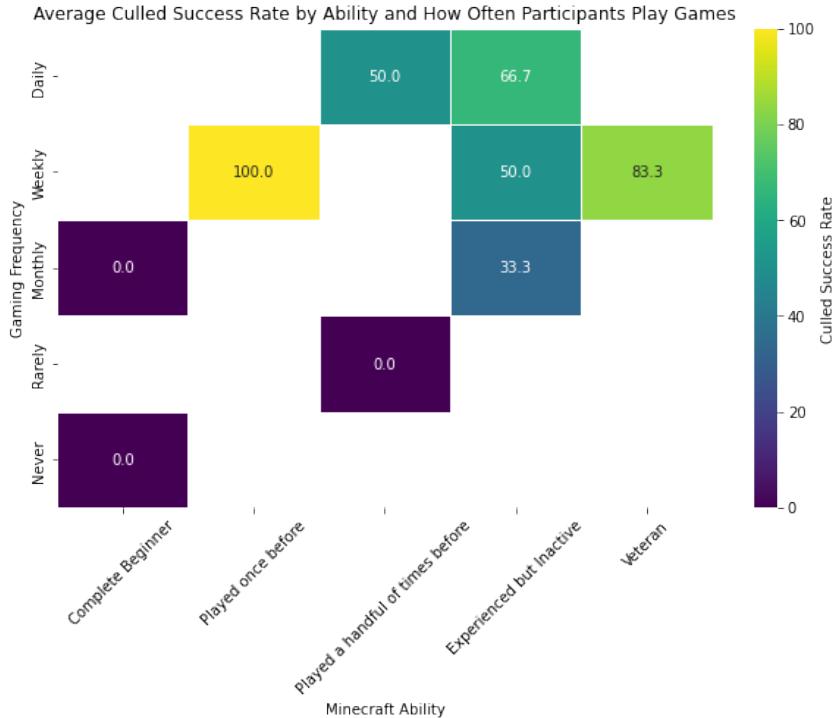


Figure 23: Heat map showing the relationship between the Culled Success Rate of a participant and a combination of their Minecraft proficiency and gaming frequency. Empty squares indicate that there were no participants with that combination of Minecraft proficiency and gaming frequency.

To further delve into the relationship between success rate, Minecraft ability, and gaming frequency, heat maps were created to show the relationship between the three variables. The heat maps in Figures 22 and 23 show that, except in the case of the one participant who correctly guessed all three odd ones out and had only played Minecraft once, the more proficient a player is at Minecraft, the more likely they were to guess the odd one out. The one participant who had only played Minecraft once and guessed all three correctly did also play games on a weekly basis, which could have contributed to their ability to tell the difference between the terrains.

Based on these results, there was also a potential that the success rate may have been tied to the specific tasks that the participants were given. I had initially suspected that tasks that required participants to spend longer on the surface of the terrain, would mean they see more of the terrain and thus be more likely to guess the odd one out — due to the different shapes of hills, valleys, as well as the frequency and placement of structures/features. Figure 24 shows that this was not the case, with the success rate highest in the Iron Ingot Farming task, the task that had players underground for the majority of it. From participant feedback, they remarked that it was hard to find iron in the regular terrains, and the task results seemed to also indicate that LLM-generated terrains were easier to find iron in. This may be a symptom of the fact that the way that the terrains were copied into the world meant that only the blocks from sea level and above were copied, and in regular terrains, players would expect to find exposed iron ore on the surface, which was not the case in the terrains used in the experiment.

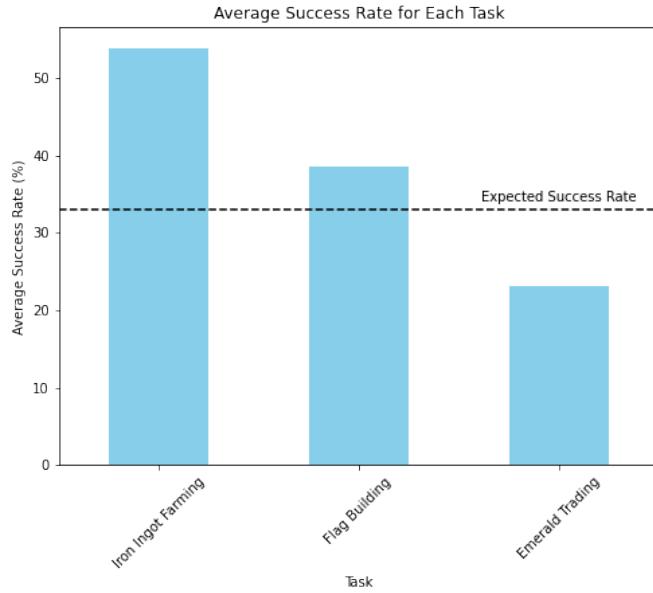


Figure 24: Bar chart showing the distribution of Success Rate for each task.

While there were a lot of correct guesses in general, players also stated after the fact that they were basing this off of how difficult the task was in that scene as opposed to the other two. This means that a lot more people did not notice the difference between the look of the terrains than the data shows, but that the tasks were easier to complete in either the regular or LLM-generated terrains. Figures 25 and 26 show that the average participant found regular terrains more difficult to complete tasks than LLM-generated terrains, and that they enjoyed playing in LLM-generated terrains more than regular terrains; this is reinforced by the fact that participants found all tasks equally difficult on average (Figure 27). This is a good sign for the system, as it shows that the terrains are not only indistinguishable to most players, particularly new players, but also that they are more enjoyable to play in.

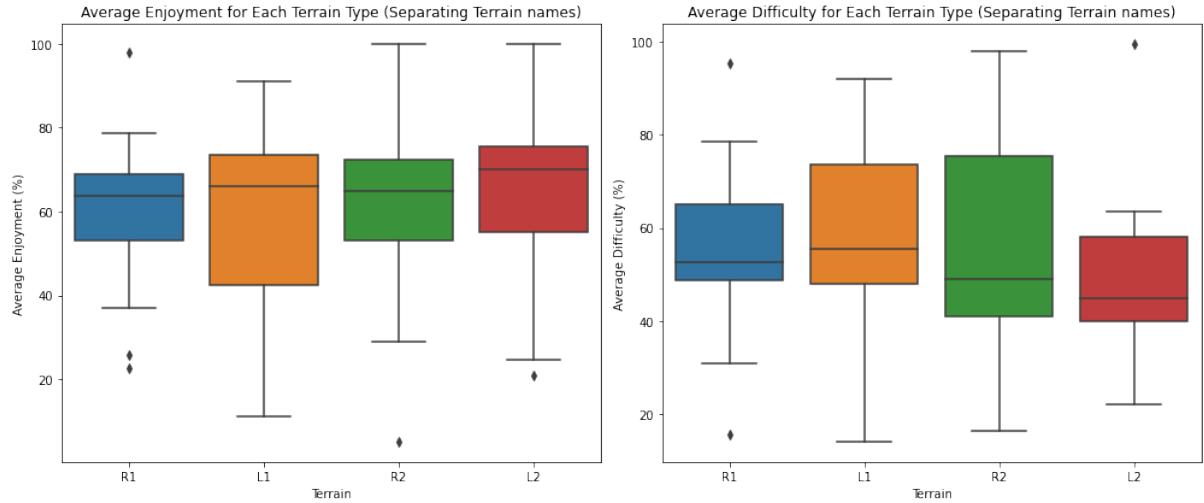


Figure 25: Box plots showing the distribution of the average enjoyment and difficulty for each terrain.

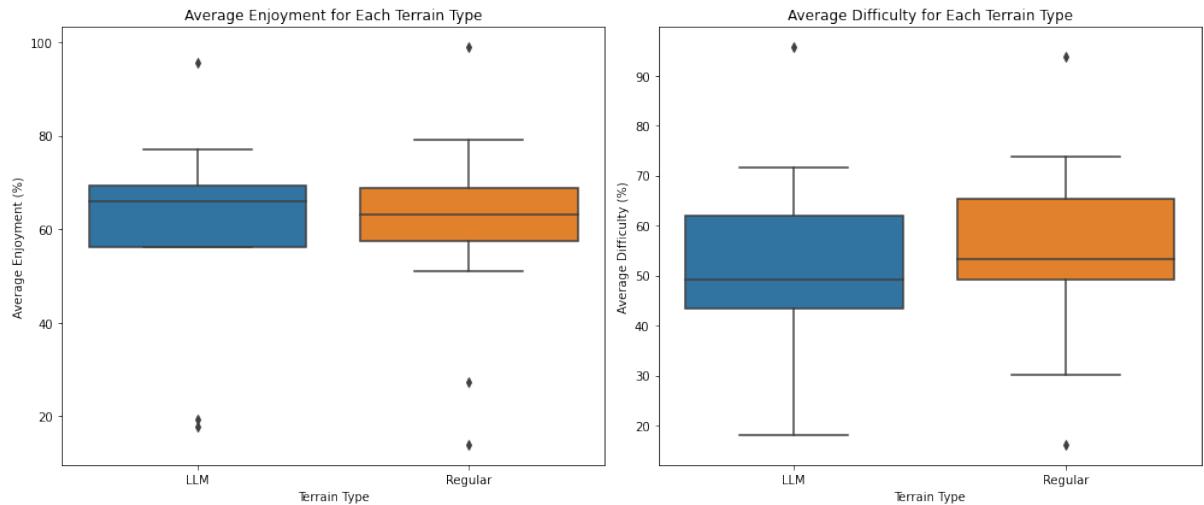


Figure 26: Box plots showing the distribution of the average enjoyment and difficulty for the two types of terrain.

Investigating the link between enjoyment and difficulty (Figure 28) showed that there was no correlation at all between the two variables, which suggests that players enjoying the LLM terrains more were not as a result of tasks being easier in them.

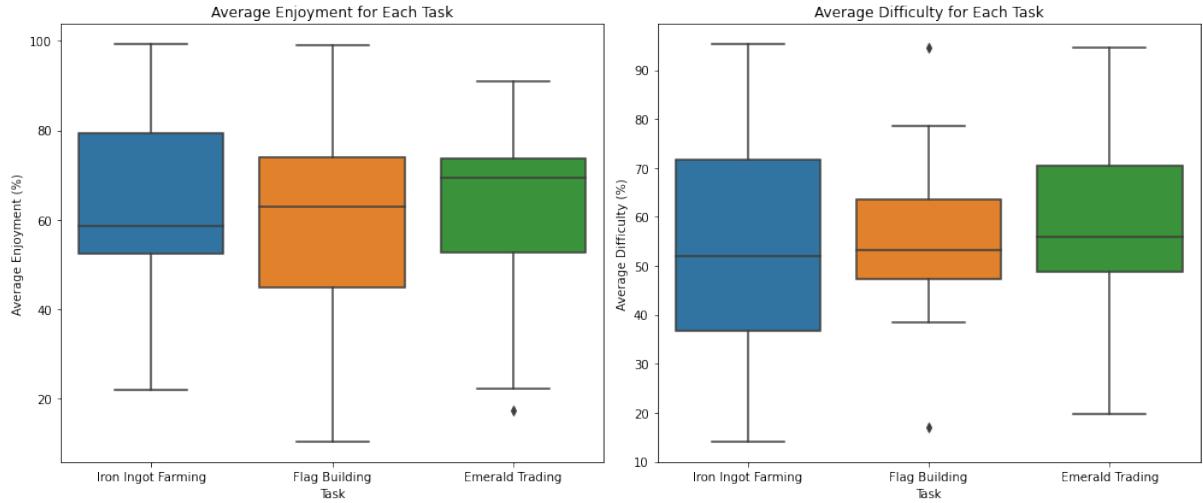


Figure 27: Box plots showing the distribution of the average enjoyment and difficulty for the three tasks.

Discussion

The qualitative feedback gathered from participants provides key insights into their gameplay experience and perceptions of AI in game design. This discussion synthesises participant responses and reflects on their implications for the study’s research questions and broader themes.

Overall Enjoyment

Participants rated their average enjoyment of the session at 4/5, suggesting a generally positive experience. This indicates that the tasks and terrains were engaging, despite potential issues identified in specific areas, and the fun score they gave during the session after each task was 60.1% on average (without ignoring outliers) — which one would expect to have an overall rating of 3/5 on average. As indicated by Figure 26, participants found LLM-generated terrains more enjoyable than regular terrains, which is a promising result for the system. This suggests that the system is capable of generating terrains that are as enjoyable to play in as naturally-generated terrains, which is a key requirement for the system to be successful.

Task-Specific Observations

The Trading task was most commonly cited as the participant’s favourite moment in the session. Interestingly, this task was also the one that had the lowest success rate (Figure 24). Figure 27 also shows they found this task the most fun at around a 70% enjoyment score on average which is in spite of the fact that all tasks had a similar average difficulty score. The fun may come from the creativity enabled by the nature of the task, with players not being handheld on how they could earn emeralds within the 4-minute time limit, unlike the other tasks where they had a much more linear route to task “completion.” This inverse relationship between enjoyment and success rate might indicate that participants paid less attention to the visuals of the scene they were playing in, and were more occupied on the task at hand. Due to the number of villagers present in each scene, there was a lot of flexibility in how the task could be completed, terrain generation playing less of a role in this task than the others.

When asked what moments in the session they ‘didn’t enjoy,’ participants most commonly cited the Iron Ingot Farming task. This task was also the one that had the highest success rate (Figure 24), which suggests that the difficulty of the task almost certainly impacted how much they enjoyed it. Notably, this task had the highest successful odd one out guess rate, most likely a result of its very linear nature (quoted as ‘boring’ by some) and being most impacted by the

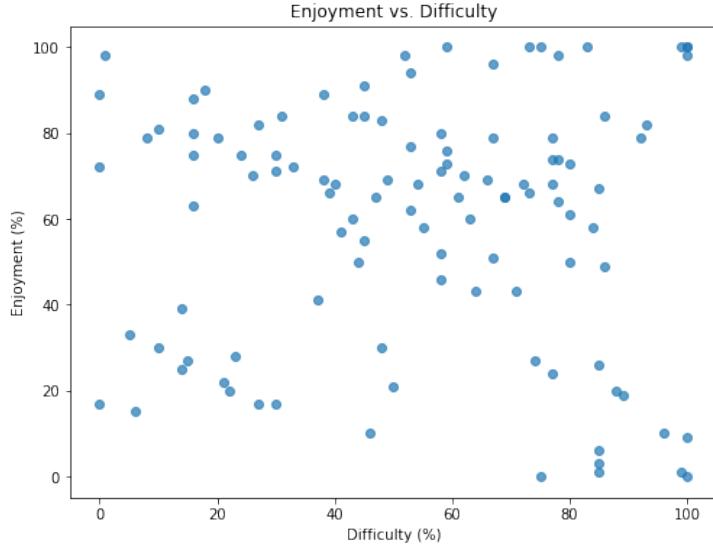


Figure 28: Scatter plot showing the relationship between the average enjoyment and difficulty of the tasks.

randomness of ore generation in terrains. With less thinking required for the more experienced players in this task, they were able to focus more on the terrain they were in, and thus this may have increased the likelihood of guessing the odd one out. One commented in a later question that iron ore blocks spawning singularly in the terrain and not in a “vein” of multiple blocks was a dead giveaway for them.

Perceptions of Terrain Features

When asked about what features stood out to them in the scenes shown, they noted:

- High frequency of flowers and grass in some scenes.
- Hills with flat tops.
- Isolated single iron blocks instead of clusters.

These comments were clearly referencing the LLM-generated terrains and highlights issues stemming from the actual algorithms used in generation after a list of functions is outputted by the LLM. Amending the frequency of foliage is simply a matter of changing the multiplier that is applied the area to determine how many grass/flower blocks to place and the flat tops of hills are a result of how the *placeTerrain()* function’s Perlin Noise generator works (Algorithm 7); the *maxAmplitude* parameter set by the model means that the hill tends to be cut off at that height. This is unnatural, and experienced players would be able to spot this difference (only one participant commented on this, however), the actual parameters used for Minecraft’s Perlin Noise generator are unknown, and so it is difficult to determine what the best values for these are. The isolated iron blocks are a result of the last part of *generateHill()* (Algorithm 6), which randomly loops through all blocks inside the hill and has a 2% chance of placing a singular iron ore; an improved algorithm would be to have a chance of placing a randomly shaped vein of iron ore blocks instead. The LLM cannot be blamed for these issues; however, these comments are a potential indication of how there are some issues with generation that can only be seen after the fact.

Most feedback on “strangeness” in the session was related to the tasks rather than the terrains. This suggests that participants’ immersion in completing the tasks might have reduced their focus on noticing terrain differences and/or that the terrains were not different enough to be

noticeable.

Link to Gaming Preferences

Among the participants, there were 8 that played Adventure games, 8 that played RPGs and 7 that played Action; these were the three most common genres. The first two genres particularly emphasise exploration, yet the results show that the participants struggled with distinguishing between terrains in the triangle test. This raises the question of whether terrain differences were too subtle to notice, even for players familiar with exploration-heavy games.

AI in Game Design

The general sentiment was one of enthusiasm when asked about Generative AI's potential to enhance game design, particularly for improving and expanding procedurally generated worlds. However, many felt that AI-generated worlds would not fully replicate the depth and uniqueness of hand-crafted environments.

When asked if they could identify AI-generated terrains, the vast majority stated they could not tell. Some participants noted in retrospect that certain scenes might have been AI-generated, while two participants felt that all scenes (including regularly generated terrains) seemed AI-generated. This suggests that the terrains were sufficiently convincing to blur distinctions.

For the question of, 'If you could use AI to design your own game world, what features or elements would you prioritise?', participants mentioned terrain generation (3 responses) and adaptive content creation (3 responses) as potential applications of AI in games. This aligns with the focus of this study and highlights a perceived strength of AI in automating world-building tasks and creating dynamic, responsive game environments.

Finally, participants were asked for potential benefits and drawbacks they could think of in regards to AI in game design. They acknowledged AI's ability to speed up development but raised concerns about the potential for:

- Boring or low-quality content.
- Legal issues, such as unintentional use of copyrighted material.
- Inclusion of inappropriate content, like profanity or nudity, if AI is not carefully monitored.

Reflection

More than half of the participants were Computer Science students (8), the rest recruited via word of mouth, but this may have influenced responses to the AI-related questions. Their familiarity with AI and its applications might have led to more optimistic perspectives about AI's role in game creation. While it may be inevitable that in the future, AI will play a larger role in game design [9], it is important to consider the potential risks and limitations of AI-generated content, as highlighted by participants, but from these findings, it is an encouraging sign that AI-generated terrains can be as enjoyable to play in as naturally-generated terrains and people are not so averse to the idea of AI in game design.

For this study where LLMs were used to replicate the generation of an existing game, the level to which this could be achieved in the end came down to the quality of the algorithms called by the model when it composed its list of functions. For entirely new games, there is no such "basis" to compare the generated content to, and since players were not able to tell the difference between the terrains in an already-existing game, especially one that they had played before, it is likely that they would not be able to tell that an LLM generated the terrains in a new game — it may be assumed to be PCG.

The first of the original two research questions was, “*Can players tell the difference between naturally-generated terrains and LLM-generated terrains?*” the data suggests that the more experienced the player was with the original “natural” generation, the more likely they were to be able to tell the difference between the LLM-generated terrains and the original game generation. Despite this, some of the participants who could tell the difference stated that the terrains that were the odd ones out were still within the realm of possibility for the original game, and that the only determining factor was the difficulty of tasks in the scenes. So to some extent, the answer to this question is “no,” as the terrains were not so visually different, but the answer is also “yes,” as the terrain generation affected how tasks were completed in the scenes.

As for the second research question, “*Do players enjoy playing in LLM-generated terrains as much as they do in naturally-generated terrains?*” on average, players had slightly more fun in LLM-generated worlds than regular ones, and found the tasks a little easier in them. That LLM-generated terrains were at least as enjoyable to play in as regular terrains is a success for the system, as it shows that the terrains are not only visually similar to naturally-generated terrains, but they are also just as fun to play in, and sometimes more. This latter question was the more critical of the two, as a game “lives and dies” by how much fun it is to play, and if the terrains were not gratifying to play in, then the system would not be a success.

Quantitative Evaluation

The aim of the quantitative evaluation was multi-faceted. Firstly, it was to determine how similar the terrains generated by the LLM were to the original terrains in the game, and if not, how different they were (and where these differences are). As the LLM outputs a list of functions for generation, there was a question of whether the LLM actually had proper reasoning behind its choices, and whether if we had randomly chosen parameters and commands instead, we would have gotten the same results; to this end, I created a Jupyter notebook (*worldgen.ipynb*) that could create these random terrain generation instructions.

These random terrains had their own constraints, which I deemed to be “realistic” for the game, such as the maximum height of a hill or the number of villages in a tile. 4 of these terrains were generated, so the results shown will include these 4 as well as the 4 terrains used in the qualitative evaluation (2 LLM-generated terrains and 2 regular terrains). These terrains were called: *llm_1*, *llm_2*, *reg_1*, *reg_2*, *random_1*, *random_2*, *random_3*, *random_4*.

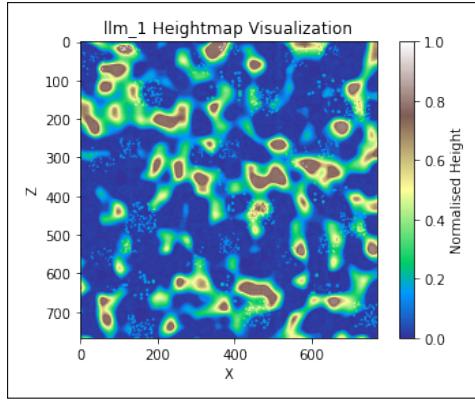
The data was obtained through the use of various modded item methods, many repurposed from their original use in the low-level prompting approach — namely, *saveBlockDataCSV()*, *saveHeightmap()* and *saveBlockDataRLE()*, as well as some new methods such as *saveWatermap()* and *saveHeightmapAsArray()*. The data was then analysed using a Jupyter notebook (*quantitative_analysis.ipynb*). Data such as the number of trees were based on generation logs or counted manually in the case of regular terrains (the numbers were low enough that this was feasible).

Results

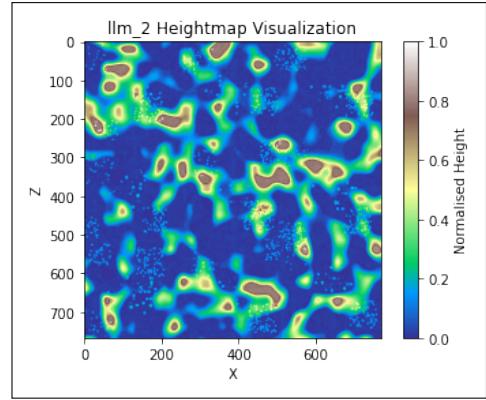
Figure 29 shows the normalised heightmaps of the 8 terrains. It is clear that they are a lot more frequent, smaller-width hills in the LLM-generated terrains, with most of the terrain flat at sea-level. The regular terrains have larger regions of high elevation, with the hills having much more variety in shape and size. The random terrains have a mix of both, with some having more small hills (Figures 29f, 29g, and 29h) and one having a larger hill (Figure 29e) with a unique shape. For the non-regular terrains, there is a lot more flat area at sea-level which is not water, whereas for the regular terrains, this y-level is almost exclusively water (Figure 31, Tables 5, 22, and 23).

Delving into the steepness of the terrains, Figure 30 shows gradient maps for each terrain, with brighter colours indicating steeper regions. For regular terrains, the steepest regions are around the edges of the large landmasses, with the steepness of the terrain staying relatively consistent throughout them. LLM-generated terrains are slightly more chaotic with the hills being steeper than the original game’s, and in areas where the terrain is flat, the steepness value is non-zero but this is most likely a result of the high frequency of foliage such as grass and flowers (Tables 8 and 9). Disregarding this factor, the LLM-generated terrains are much flatter than the regular terrains with the most chaotic being the random terrains. As we would expect, there is almost no consistency in general terrain shape, with not much of their components coherently grouped together, and almost no gradual elevation changes (Figures 29e, 29f, 29g, and 29h).

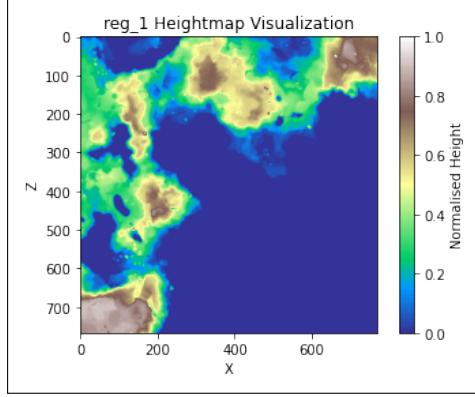
The LLM made decisions such as placing a village near a lake, river, or forest as it reasoned that this would be more engaging for the player than spreading features far apart. It is clear that random generation does not have such context or reasoning behind its “decisions” (random number generation), and as such, the features tend to be spread out more and have no guardrails from placing features on top of each other. If a human was in control of writing a list of generation functions and parameters, they would most likely have similar reasoning to the LLM, which suggests that this is a task that could be delegated to the LLM without much issue (save for the hallucinations mentioned in Figure 16).



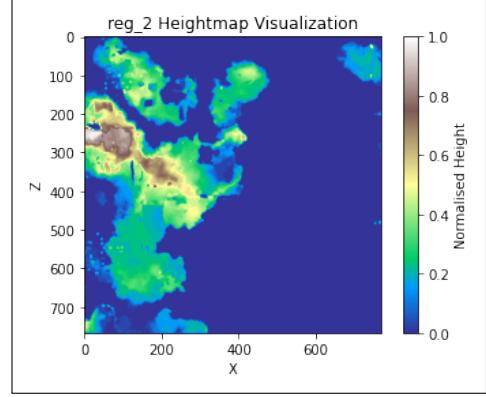
(a) *llm_1* Heightmap.



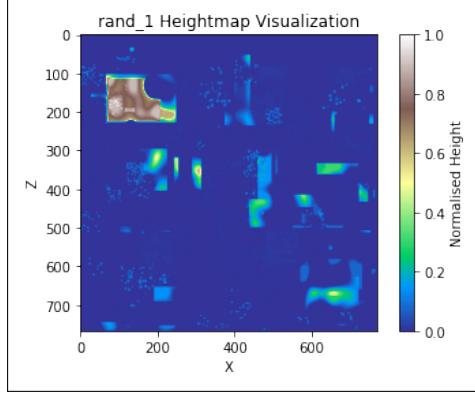
(b) *llm_2* Heightmap.



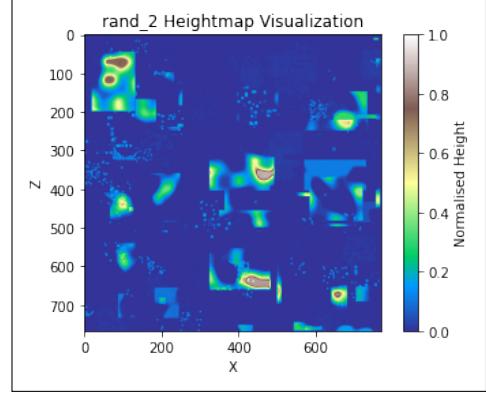
(c) *reg_1* Heightmap.



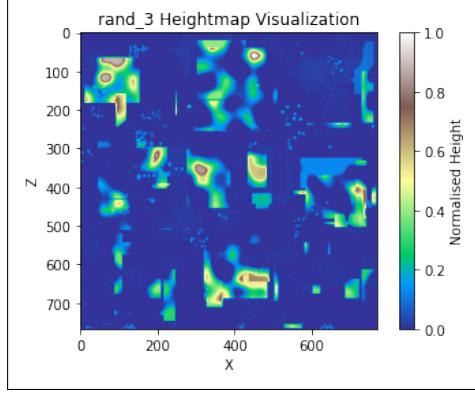
(d) *reg_2* Heightmap.



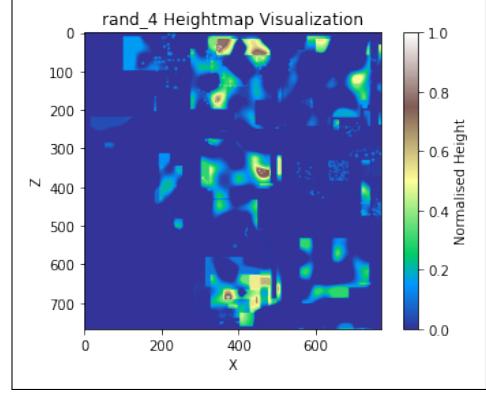
(e) *random_1* Heightmap.



(f) *random_2* Heightmap.

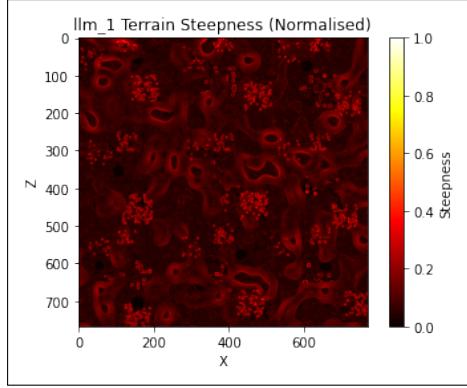


(g) *random_3* Heightmap.

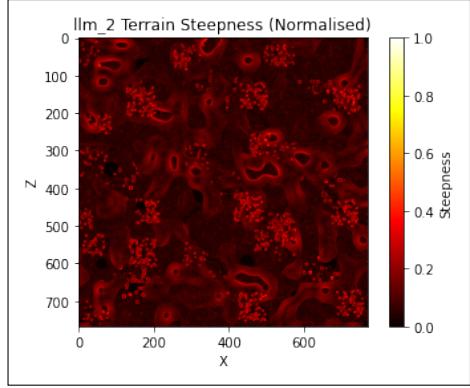


(h) *random_4* Heightmap.

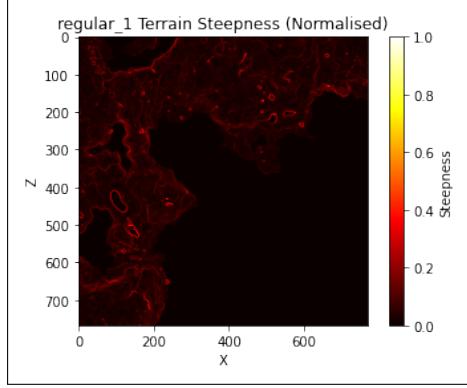
Figure 29: Heightmaps of the terrains used in the quantitative evaluation. The heights have been normalised to a range of 0 to 1, with 1 being the highest point in the map.



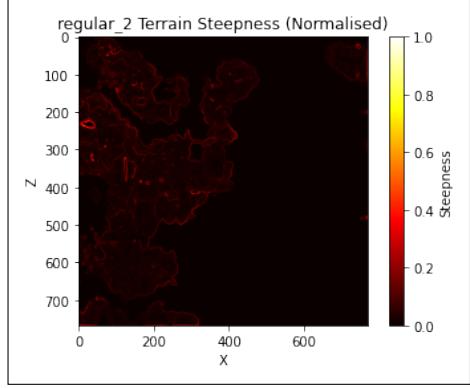
(a) *llm_1* Terrain Steepness Map.



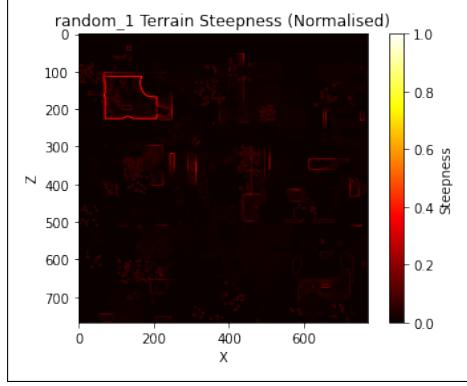
(b) *llm_2* Terrain Steepness Map.



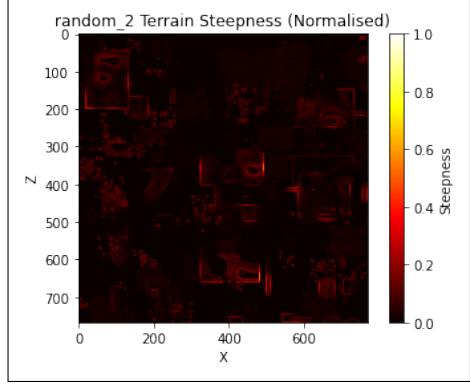
(c) *reg_1* Terrain Steepness Map.



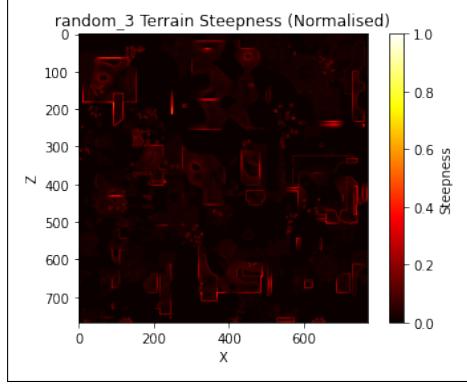
(d) *reg_2* Terrain Steepness Map.



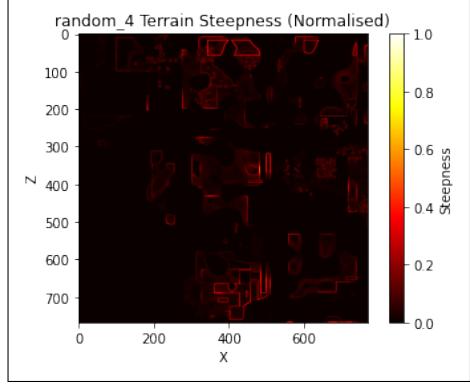
(e) *random_1* Normalised Terrain Steepness Map.



(f) *random_2* Normalised Terrain Steepness Map.



(g) *random_3* Normalised Terrain Steepness Map.



(h) *random_4* Normalised Terrain Steepness Map.

Figure 30: Terrain Steepness Maps of the terrains used in the quantitative evaluation. The steepness values have been normalised to a range of 0 to 1, with 1 being the steepest point in the map.

Terrain Name	Average Gradient Magnitude	Variance
LLM 1	0.7361	0.3986
LLM 2	0.7545	0.4383
Regular 1	0.3645	0.4800
Regular 2	0.2818	0.5322
Random 1	0.3329	1.2964
Random 2	0.3366	0.6910
Random 3	0.3869	0.9062
Random 4	0.2844	0.9817

Table 4: Average gradient magnitude and variance for different terrains.

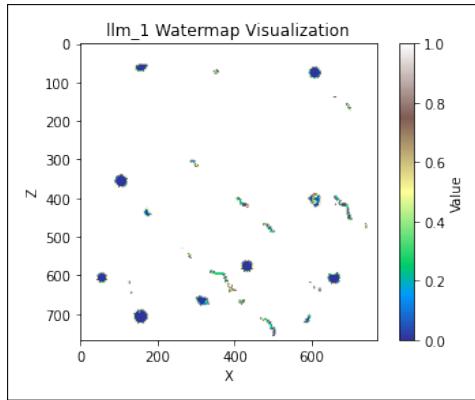
Terrain Name	Water Percentage
LLM 1	1.34%
LLM 2	1.67%
Regular 1	52.56%
Regular 2	65.76%
Random 1	10.59%
Random 2	14.94%
Random 3	32.98%
Random 4	67.72%

Table 5: Water percentage for different terrains.

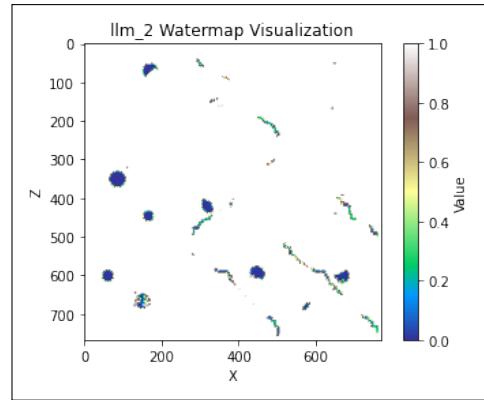
We can see the average gradient magnitude, which is the average steepness of the terrain, for the LLM terrains is more than double that of the regular terrains, with less variance, but the random terrains have their gradient magnitudes and variances resemble close to that of the regular terrains (Table 4). *rand_4* has its water percentage very close to that of the regular terrains (Table 5 and Figure 31h), but on inspection, due to the lack of regularity in terrain shape, there is not a larger area of landmass (“playable area”) — instead the terrain feels very noisy.

Block Type	Count	Percentage on Playable Area
Short Grass	43,587	15.58%
Lilac	0	0.00%
Rose Bush	0	0.00%
Lily of the Valley	0	0.00%
Peony	0	0.00%

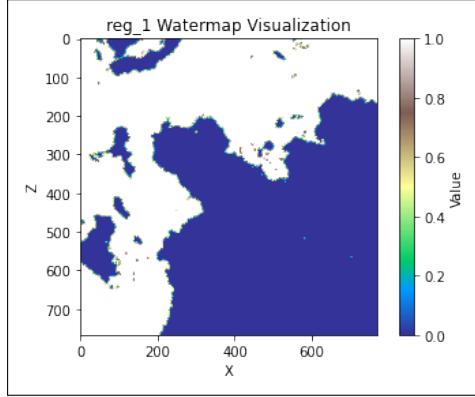
Table 6: Foliage Counts for *reg_1*.



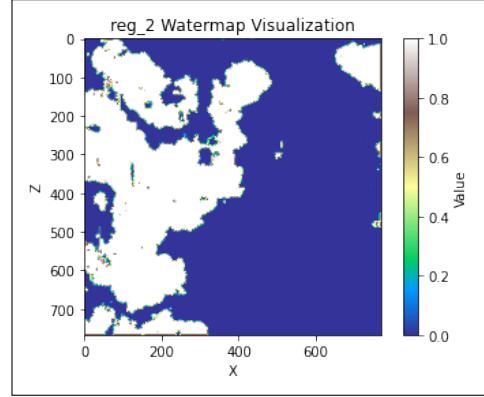
(a) *llm_1* Watermap.



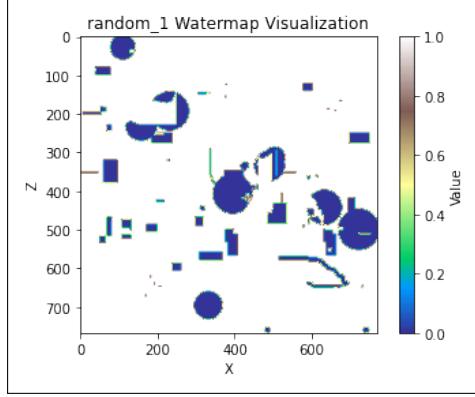
(b) *llm_2* Watermap.



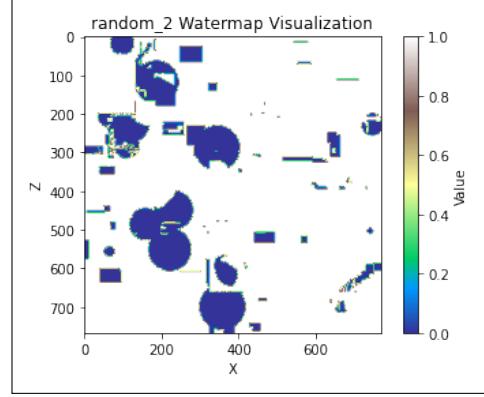
(c) *reg_1* Watermap.



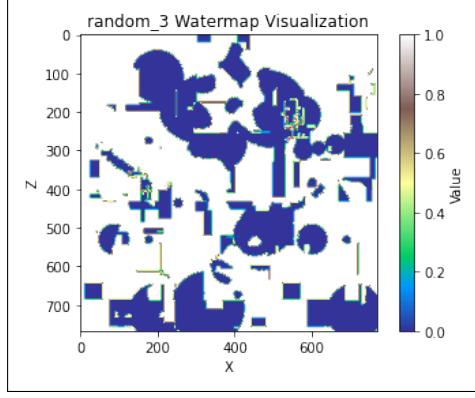
(d) *reg_2* Watermap.



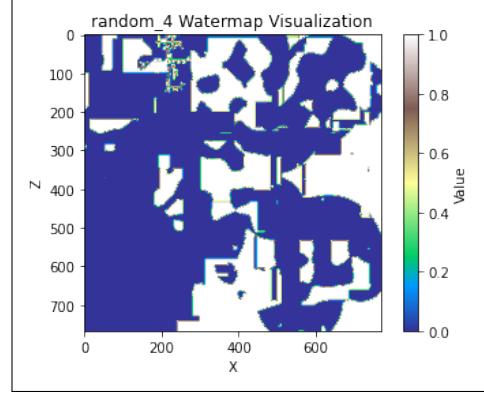
(e) *random_1* Watermap.



(f) *random_2* Watermap.



(g) *random_3* Watermap.



(h) *random_4* Watermap.

Figure 31: Watermaps of the terrains used in the quantitative evaluation. The blue represents where there is water and white where there is land.

Block Type	Count	Percentage on Playable Area
Short Grass	23,331	11.55%
Lilac	0	0.00%
Rose Bush	0	0.00%
Lily of the Valley	0	0.00%
Peony	0	0.00%

Table 7: Foliage Counts for *reg_2*.

Block Type	Count	Percentage on Playable Area
Short Grass	134,697	23.15%
Lilac	6,987	1.20%
Rose Bush	6,854	0.59%
Lily of the Valley	3,499	0.60%
Peony	7,991	0.69%

Table 8: Foliage Counts for *llm_1*.

Block Type	Count	Percentage on Playable Area
Short Grass	130,905	22.57%
Lilac	7,287	1.26%
Rose Bush	6,972	0.60%
Lily of the Valley	3,376	0.58%
Peony	7,234	0.62%

Table 9: Foliage Counts for *llm_2*.

Block Type	Count	Percentage on Playable Area
Short Grass	34,204	6.49%
Lilac	1,913	0.36%
Rose Bush	1,691	0.16%
Lily of the Valley	890	0.17%
Peony	2,016	0.19%

Table 10: Foliage Counts for *random_1*.

Block Type	Count	Percentage on Playable Area
Short Grass	29,924	5.96%
Lilac	2,607	0.52%
Rose Bush	2,423	0.24%
Lily of the Valley	1,215	0.24%
Peony	2,651	0.26%

Table 11: Foliage Counts for *random_2*.

Block Type	Count	Percentage on Playable Area
Short Grass	43,993	11.13%
Lilac	2,628	0.66%
Rose Bush	2,485	0.31%
Lily of the Valley	1,384	0.35%
Peony	2,811	0.36%

Table 12: Foliage Counts for *random_3*.

Block Type	Count	Percentage on Playable Area
Short Grass	16,699	8.77%
Lilac	528	0.28%
Rose Bush	428	0.11%
Lily of the Valley	439	0.23%
Peony	838	0.22%

Table 13: Foliage Counts for *random_4*.

Metric	Value
Trees	780
Villages	8
Playable area (blocks)	581,940
Playable blocks per tree	746
Playable blocks per village	72,742

Table 14: Playable Area Statistics for *llm_1* on Trees and Villages.

Metric	Value
Trees	855
Villages	9
Playable area (blocks)	579,959
Playable blocks per tree	678
Playable blocks per village	64,440

Table 15: Playable Area Statistics for *llm_2* on Trees and Villages.

Metric	Value
Trees	57
Villages	2
Playable area (blocks)	279,801
Playable blocks per tree	4,909
Playable blocks per village	139,900

Table 16: Playable Area Statistics for *regular_1* on Trees and Villages.

Metric	Value
Trees	37
Villages	2
Playable area (blocks)	201,927
Playable blocks per tree	5,457
Playable blocks per village	100,964

Table 17: Playable Area Statistics for *regular_2* on Trees and Villages.

Metric	Value
Trees	444
Villages	4
Playable area (blocks)	527,342
Playable blocks per tree	1,188
Playable blocks per village	131,836

Table 18: Playable Area Statistics for *random_1* on Trees and Villages.

Metric	Value
Trees	467
Villages	5
Playable area (blocks)	501,686
Playable blocks per tree	1,074
Playable blocks per village	100,337

Table 19: Playable Area Statistics for *random_2* on Trees and Villages.

Metric	Value
Trees	219
Villages	3
Playable area (blocks)	395,326
Playable blocks per tree	1,805
Playable blocks per village	131,775

Table 20: Playable Area Statistics for *random_3* on Trees and Villages.

Metric	Value
Trees	259
Villages	2
Playable area (blocks)	190,381
Playable blocks per tree	735
Playable blocks per village	95,190

Table 21: Playable Area Statistics for *random_4* on Trees and Villages.

“Playable Area” is defined by the number of blocks where the highest block at each (X, Z) coordinate is not water. The reason for calculating this is since the regions where players are most likely to spend their time are on land, it would be interesting to see how much of that is covered by flowers, grass, trees, and villages (terrain features able to be generated by the LLM). Even if some terrains have less area available for feature placement, there is a chance that the distribution of them may be relatable. The Foliage Count tables (6 – 13) show that the regular terrains have very little grass, only around 10–15%, both the regular terrains sampled also having none of the flowers that the other terrains have; *random_3* and *random_4* are the closest in percentages on playable area with short grass to the regular terrains.

It is a similar story in the case of playable blocks per village, yet another instance where the random terrains are much more in line with the statistics of the regular terrains than their LLM-generated cousins. The LLM-generated terrains have a much higher density of both trees and villages, with the other terrains having an order of magnitude more playable blocks per tree and over double the amount per village. The model tended to output one village and two forests per tile during the dynamic content generation phase, which is what led to these results. A sufficient correction to this behaviour could be achieved by asking the model to generate villages and forests more sparingly.

Discussion

The overall idea I am trying to present with this quantitative evaluation is that we could easily use random number generation to determine our parameters for terrain generation, and this may be successful in producing results that align closely with the original game; however, the terrain will still lack the coherence and attention paid to it from being reasoned about by a human or an LLM. The LLM-generated terrains produced results that did not match the original game when looked at from purely a quantitative standpoint, except computer games are an art form, one which is not easy to decompose into quantifiable metrics. We cannot write off LLM terrain generation simply as a result of it not matching the original game in terms of arbitrary quantitative values, as the qualitative evaluation showed that the terrains were still enjoyable to play in, and the tasks were still completable — often more easily than in the original game.

Achievement of Objectives and Requirements

Objectives

Primary

Develop a Framework for Prompt Engineering

- This objective was achieved after attempting a low-level approach, realising its limitations and moving to a high-level approach. The working framework involves giving the Large Language Model context of the larger task at hand (generating game terrains), informing it of the long-term goals of model usage, providing it relevant examples of existing content that we wish to replicate, and then prompting it to understand the examples before defining functions/commands that it can use to generate new content.

Generate Content Same as Existing Generation

- Our quantitative evaluation showed that the terrains generated by the LLM were not composed exactly the same as the original game, but the qualitative evaluation indicated that even experienced players were unable to tell the difference between the terrains.
- On top of this, the terrains were just as, if not more, enjoyable to play in than the original terrains.

Secondary

Generate New Content Based on Model’s Understanding of Context

- The LLM was able to generate new “tiles” that snap on to the already generated tiles of the game’s terrain — Appendix B.2 shows an example of this where we ask the model to generate a new tile on the East and West of a central tile.
- Not only was the model able to generate new content on the fly, but this content fit in with the existing terrain, examples including instances where rivers would flow into the new tile, incomplete hills would be completed, and villages would be placed at a reasonable distance from each other.

Create a Taxonomy of Hallucinations & Produce Strategies for Mitigation

- 7 potential hallucination types were identified and ranked in terms of severity (Figure 16). These were based on hallucinations experienced first-hand during the development of the system along with some that were hypothesised.
- The hallucinations at the lowest tier in the pyramid could be easily mitigated via simple validation of model outputs, with the approach for the higher tiers being more complex and requiring human intervention. For these types, the logging done as part of the system would be invaluable in identifying and correcting them.

None of the Tertiary objectives were achieved due to a lack of time for the latter two objectives (**Explore Scalability of Content Generation** and **Investigate Prompting the Model for Specific Content to Generate**), but in the case of the first (**Generate Content for 2-D Tile-Based Games**), there was also not enough additional work required to be done to build upon the work of existing papers that generate 2-D levels using LLMs [2, 10–13]. It is unclear whether the framework presented in this paper would be a better fit for generating 2-D levels than what has been established in these papers, but it is clear that the framework is a good fit for generating 3-D terrains.

Stress testing the Google Gemini model for exploring scalability may have resulted in the usage of all the free credits that were provided for API access, and as such, this was not attempted.

As will be discussed in the “Future Work” section, generating specific content from prompting the model is definitely an achievable task and would involve building on the work of Madge and Poesio, and Hu *et al.* [3, 4], but it was not attempted due to time constraints.

Requirements

Functional

1. **Terrain Generation:** The system was able to generate 256x64x256 terrain tiles as seen in Figures 10 and 12, complete with the expected structures and features.
2. **Dynamic Terrain Extension:** The Dynamic Content Generation system implemented was able to produce such as the ones seen in Appendix B.2 that blend in with the existing terrain.
3. **Basic Hallucination Handling:** The lowest tier of hallucinations identified are easily mitigated by model output validation.
4. **Event-Driven Content Generation:** Figures 7 and 9 show the architecture behind the event-driven content generation system, which was able to generate new content based on the proximity of a player to the edge of the generated terrain.
5. **Generated Content Memory:** Both the model and the Dynamic Content Handler store their own records of what has already been generated, with the latter using its memory to improve the prompt given to the model.
6. **Model Output Logging:** The system logs the output of the model before and after it has been processed for hallucinations, and the Dynamic Content Handler stores these logs for future reference.

Non-Functional

1. **Fun:** The user study conducted showed that the terrains generated by the LLM were at least as enjoyable to play in as the original game (Figure 26).
2. **Consistency:** The terrains generated by the LLM were consistent with the context provided to it, and the Dynamic Content Handler helps assist the model in maintaining this consistency.
3. **Visuals:** The terrains generated by the LLM were similar enough to the original game that experienced players were unable to tell the difference between them.
4. **LLM Selection Metrics:** The chosen Google Gemini model offered the best balance of being free of charge, while having the largest commercially available context window, high level of reasoning ability and multimodal capabilities that allow it to perform reasoning tasks on images (Table 1).
5. **System Modularity and Extensibility:** The specific implementation of *GenerationCommand* which parsed a list of generation commands produced by the model, allowed for easy modification of existing commands, as well as the addition of new ones due to the use of the Command Behavioural Design Pattern [65].
6. **Performance:** The system was able to generate terrains within a minute, with the biggest bottleneck being the actual ‘/place’ commands waiting to resolve in the game, not the generation of the commands themselves or their processing and subsequent sending to the game.
7. **Scalability:** The most outputs that could be requested from the LLM within a minute is 3 due to the nature of event-based generation loop, and this is 20% of the maximum

request rate of the Google Gemini model, each prompt taking at most 10 seconds to produce a full prompt, complete with reasoning of choices made in generation.

Critical Appraisal

This research explores the novel application of Large Language Models (LLMs) for generating diverse and engaging terrains within virtual environments, a domain largely unexplored in existing literature. While previous studies have leveraged LLMs for game-related tasks, such as level design [2, 10–15], they primarily focus on discrete, 2-D level creation or specific in-game elements. In contrast, this work delves into the generation of continuous, 3-D terrains, a significantly more complex challenge.

For instance, studies like *MarioGPT* [2] and *Level Generation Through Large Language Models* [11] demonstrate LLM capabilities in generating 2-D game levels, often focusing on specific game mechanics and objectives. However, these approaches do not address the complexities of 3-D terrain generation, such as simulating natural formations, handling varying elevations, and integrating diverse environmental features.

Furthermore, existing research within the Minecraft domain, such as *3D Building Generation in Minecraft via Large Language Models* [3] and *Large Language Models as Minecraft Agents* [4], primarily focus on generating individual 3-D structures (buildings) within the game environment. These studies often assume a flat or pre-defined terrain, neglecting the intricate interplay between terrain features and the placement of these structures. In contrast, this research addresses the challenge of generating the terrain itself, including its underlying topography and environmental characteristics, while simultaneously considering the placement of 3-D structures within this dynamically generated environment.

A key distinction lies in the approach to evaluation. Existing research primarily relies on quantitative metrics, often prioritising accuracy and efficiency over player experience [2–4, 12, 13, 16]. For example, Word2World [13] utilises quantitative metrics within its "rounds" of iterative refinement, focusing on objective measures of game quality. This research emphasises the importance of qualitative evaluation, incorporating a Triangle Test-based user study and subsequent feedback collection to assess the player's enjoyment and engagement with the generated terrains. This approach aligns better with the subjective nature of game design and player experience, prioritising the human element in the evaluation process.

Another significant divergence from existing work lies in the proactive approach to mitigating hallucinations. While previous studies acknowledge the limitations of LLMs, they often lack a systematic framework for identifying and addressing different types of hallucinations. This research introduces a novel taxonomy of hallucination levels, including "Parameter," "Command" and "Semantic" hallucinations, and outlines specific strategies to mitigate their occurrence during the terrain generation process, such as incorporating contextual constraints. This structured approach provides a foundation for future research to address hallucinations in LLM-generated content effectively.

Moreover, this research leverages the multimodal capabilities of modern LLMs in a unique way. While some studies utilise multiple LLMs for comparison [2, 10–15], this research focuses on a single, powerful model, exploring its capabilities for both terrain generation and self-assessment through iterative refinement. Unlike Word2World, which relies solely on quantitative metrics for iterative refinement, this research explores the potential of leveraging the LLM's vision capabilities for self-assessment. While direct in-game screenshotting presents challenges, alternative approaches, such as utilising a separate rendering engine or a human-in-the-loop system, are considered to facilitate this innovative approach to model refinement.

Conclusions

Addressing the complex challenge of 3-D terrain generation, this research demonstrates the successful application of LLMs to create dynamic and engaging virtual environments. By overcoming the inherent complexities of generating continuous, 3-D spaces, this project pushes the boundaries of LLM-powered game development and opens new avenues for creative exploration in the field of AI-driven game design.

Key Achievements

Successfully being able to generate 3-D terrains using LLMs is a significant step beyond previous work that primarily focuses on generating individual 3-D structures [3, 4] or 2-D levels [2, 10–13, 15]; LLMs have been shown to be capable of generating larger-scale 3-D terrains that are both continuous and dynamic.

Creating of a taxonomy of hallucinations and mitigation strategies is another key achievement of this research. By identifying the most common types of hallucinations as well as others that may appear in the future, this research provides a ranking of their severity, as well as how involved a person would have to be to resolve them.

Conducting a qualitative evaluation of player experience that indicates the potential of LLM-generated terrains to outperform traditional PCG methods in terms of player enjoyment is a massive step forward in the field of AI-driven game design.

The Dynamic Content Generation system implemented in this research is another key achievement. By developing an intermediary handler that can act as the bridge between the LLM and the game engine, this research has demonstrated the potential for more adaptive and responsive game environments. Within this current system, I have demonstrated an event-based approach to content generation, with new areas of the world being generated as the player nears the border of the currently generated area.

Significant Drawbacks

Despite these achievements, there are several limitations to this research. Particularly when it comes to the qualitative user study, there are multiple factors that could have influenced the results. The small sample size of 13 participants means that the results may not be generalisable to a larger population.

Some participants noted some weird features of LLM terrains such as too many flowers and grass patches, as well as flat tops of hills. These issues come down to the algorithms used to generate those features, not the LLM itself. While it was not noticed by the participants, the LLM-generated terrains also did not have the 60% water coverage that the traditional PCG terrains had, instead having less than 2% for both terrains. Before conducting the experiment, these are all problems that I should have addressed in retrospect by improving the algorithms called by the command parser and prompting the model to generate more water features.

As a result of prompting the model at a “high-level” in order for it to produce a list of commands, which are parsed in the game logic to generate the terrain, the model has lost some aspect of its control over the generation, as it is now delegating that task to a PCG algorithm, rather than deciding itself where each block should be placed. This is a significant drawback, as it means that the model is not generating the terrain itself, but rather providing the instructions for a PCG algorithms to do so. Despite this, the model still has some idea over what the terrain will look like in general, just not at a block-level fidelity.

As a proof-of-concept, I had only tested generation for Plains biomes in Minecraft. If other biomes were implemented and had their own set of commands, the model would have more “work to do” in terms of blending them together, as well as generating the correct features for each biome.

Future Work

There are several avenues for future work that could build upon the findings of this research. One potential area is incorporating the structure generation techniques from previous work [3, 4] into the LLM, allowing it slightly more control over the generation of the terrain, rather than just providing a list of commands and delegating their processing to a PCG algorithm. Following on from this idea, the LLM could use a mix, with more advanced PCG algorithms than the ones used in this research — the LLM determining in what case to use the PCG approach, and in what case to generate the terrain/feature itself. In the future, LLMs could also write the code for the PCG algorithm and then call it in a command list, if it decides a new algorithm is needed.

A more extensive user study with a larger sample size could also be conducted to further validate the findings of this research. Experiments could be done that test different levels of LLM control over the terrain generation, e.g., the LLM generating the terrain and all features itself, the LLM generating the terrain and some features, and the LLM generating the terrain and providing a list of commands for a PCG algorithm to generate the features. These experiments would be invaluable in determining the optimal level of LLM control over the terrain generation process for player enjoyment.

The Dynamic Content Generation system could be further developed to include more advanced features, such as the ability to generate new areas of the world in response to player actions, rather than just when the player nears the border of the currently generated area. This would require the LLM to have a better understanding of the game world and the player’s actions, required an audit log of player actions that will regularly be updated and sent to the LLM for processing. In tandem with specific structure/feature creation, this could open up new possibilities for more adaptive and responsive game environments.

As LLM token limits and reasoning power increase, the lower the level of abstraction the LLM can generate commands at. It would be interesting to see how much stress the system could take before hallucinations reach a point where they are too severe to be mitigated — scalability testing of the system is crucial to understanding the limits of the system with which to work around.

In summary, this research contributes to the field of LLM-powered game development by:

- **Pioneering the use of LLMs for dynamic 3-D terrain generation.**
- **Prioritising qualitative evaluation of player experience.**
- **Developing a taxonomy of hallucinations and a comprehensive framework for mitigating them.**
- **Exploring the multimodal capabilities of LLMs for iterative refinement.**

By addressing these key distinctions and leveraging innovative approaches, this research aims to advance the state-of-the-art in LLM-powered game development and pave the way for more immersive and player-centric virtual environments.

References

- [1] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: <https://arxiv.org/abs/2005.14165>.
- [2] Shyam Sudhakaran et al. *MarioGPT: Open-Ended Text2Level Generation through Large Language Models*. 2023. arXiv: 2302.05981 [cs.AI]. URL: <https://arxiv.org/abs/2302.05981>.
- [3] Shiying Hu et al. *3D Building Generation in Minecraft via Large Language Models*. 2024. arXiv: 2406.08751 [cs.AI]. URL: <https://arxiv.org/abs/2406.08751>.
- [4] Chris Madge and Massimo Poesio. *Large Language Models as Minecraft Agents*. 2024. arXiv: 2402.08392 [cs.CL]. URL: <https://arxiv.org/abs/2402.08392>.
- [5] Kawahara, Reki. *Sword Art Online*. ASCII Media Works. Light novel series. 2009.
- [6] Google AI. *Gemini: A Family of Highly Capable Multimodal Models*. <https://arxiv.org/abs/2312.11805>. arXiv preprint arXiv:2312.11805 [cs.CL]. 2023.
- [7] Daniil A. Boiko, Robert MacKnight, and Gabe Gomes. *Emergent autonomous scientific research capabilities of large language models*. 2023. arXiv: 2304.05332 [physics.chem-ph]. URL: <https://arxiv.org/abs/2304.05332>.
- [8] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. *Hallucination is Inevitable: An Innate Limitation of Large Language Models*. 2024. arXiv: 2401.11817 [cs.CL]. URL: <https://arxiv.org/abs/2401.11817>.
- [9] Roberto Gallotta et al. “Large Language Models and Games: A Survey and Roadmap”. In: *IEEE Transactions on Games* (2024), pp. 1–18. DOI: 10.1109/tg.2024.3461510.
- [10] Chengpeng Hu, Yunlong Zhao, and Jialin Liu. *Game Generation via Large Language Models*. 2024. arXiv: 2404.08706 [cs.AI]. URL: <https://arxiv.org/abs/2404.08706>.
- [11] Graham Todd et al. “Level Generation Through Large Language Models”. In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*. FDG 2023. ACM, Apr. 2023. DOI: 10.1145/3582437.3587211.
- [12] Muhammad U Nasir and Julian Togelius. *Practical PCG Through Large Language Models*. 2023. arXiv: 2305.18243 [cs.CL]. URL: <https://arxiv.org/abs/2305.18243>.
- [13] Muhammad U. Nasir, Steven James, and Julian Togelius. *Word2World: Generating Stories and Worlds through Large Language Models*. 2024. arXiv: 2405.06686 [cs.CL]. URL: <https://arxiv.org/abs/2405.06686>.
- [14] Jake Bruce et al. *Genie: Generative Interactive Environments*. 2024. arXiv: 2402.15391 [cs.LG]. URL: <https://arxiv.org/abs/2402.15391>.
- [15] Meiqing Jin et al. *WorldWeaver: Procedural World Generation for Text Adventure Games using Large Language Models*. 2024. URL: <https://wordplay-workshop.github.io/pdfs/15.pdf> (visited on 10/09/2024).
- [16] Guanzhi Wang et al. *Voyager: An Open-Ended Embodied Agent with Large Language Models*. 2023. arXiv: 2305.16291 [cs.AI]. URL: <https://arxiv.org/abs/2305.16291>.
- [17] Morgan Lewis. “EU AI Act: Institutions Debate Definition of AI”. In: *Morgan Lewis* (2023). <https://www.morganlewis.com/pubs/2023/09/eu-ai-act-institutions-debate-definition-of-ai>.
- [18] Pecan AI. “Rule-Based vs. Machine Learning AI: Which Produces Better Results?” In: *Pecan AI Blog* (2023). <https://www.pecan.ai/blog/rule-based-vs-machine-learning-ai-which-produces-better-results>.

- [19] Joseph Weizenbaum. “ELIZA—a computer program for the study of natural language communication between man and machine”. In: *Commun. ACM* 9.1 (Jan. 1966), pp. 36–45. DOI: 10.1145/365153.365168.
- [20] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018. URL: <https://api.semanticscholar.org/CorpusID:49313245>.
- [21] Gavin Newsom. *Executive Order N-12-23: Artificial Intelligence (AI) Governance*. Accessed: 2024-10-09. 2023. URL: <https://www.gov.ca.gov/wp-content/uploads/2023/09/AI-EO-No.12--GGN-Signed.pdf>.
- [22] Mihály Héder. “AARON”. In: *Encyclopedia of Artificial Intelligence: The Past, Present, and Future of AI*. Ed. by P L Frana and M J Klein. Santa Barbara, CA, USA: ABC-CLIO, 2021, pp. 1–2.
- [23] Ken Perlin and Athomas Goldberg. “Improv: a system for scripting interactive actors in virtual worlds”. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996, pp. 205–216. ISBN: 0897917464. DOI: 10.1145/237170.237258.
- [24] Ziwei Ji et al. “Survey of Hallucination in Natural Language Generation”. In: *ACM Computing Surveys* 55.12 (Mar. 2023), pp. 1–38. DOI: 10.1145/3571730.
- [25] Ishaan Srivastava. “A comparative analysis of generative models for terrain generation in open-world video games”. In: *Journal of High School Science* 8.1 (2024), pp. 120–143. URL: <https://jhss.scholasticahq.com/article/92856-a-comparative-analysis-of-generative-models-for-terrain-generation-in-open-world-video-games>.
- [26] M. Dahrén. “The Usage of PCG Techniques Within Different Game Genres”. Bachelor’s dissertation. Malmö University, 2021. URL: <https://www.diva-portal.org/smash/get/diva2:1604550/FULLTEXT02.pdf>.
- [27] Jonathon Doran and Ian Parberry. “Controlled Procedural Terrain Generation Using Software Agents”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2.2 (2010), pp. 111–119. DOI: 10.1109/TCIAIG.2010.2049020.
- [28] Maren Awiszus, Frederik Schubert, and Bodo Rosenhahn. *World-GAN: a Generative Model for Minecraft Worlds*. 2021. arXiv: 2106.10155 [cs.LG]. URL: <https://arxiv.org/abs/2106.10155>.
- [29] GameSpy Staff. *Is Skyrim’s AI Storytelling the Future of Gaming?* Accessed: 2024-10-09. 2012. URL: <http://uk.pc.gamespy.com/pc/elder-scrolls-v/1217754p1.html>.
- [30] *Borderlands 3 Weapons*. Accessed: 2024-10-09. 2024. URL: https://borderlands.fandom.com/wiki/Borderlands_3_Weapons.
- [31] *Borderlands 3 Gear*. Accessed: 2024-10-09. 2024. URL: https://borderlands.fandom.com/wiki/Borderlands_3_Gear.
- [32] *AI Dungeon*. Accessed: 2024-10-09. 2024. URL: <https://aidungeon.com/>.
- [33] Raffi Khatchadourian. “World Without End”. In: *The New Yorker* (2015). Accessed: 2024-10-09. URL: <https://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian>.
- [34] John Harris. “Interview: The Making of Dwarf Fortress”. In: (Feb. 2008). Accessed: 2024-10-09.
- [35] “Biome Generation”. In: (). Accessed: 2024-10-09. URL: https://minecraft.wiki/w/World_generation#Biomes.

- [36] *Java Edition Chunk Loading*. Minecraft Wiki. Accessed: 2024-10-09. URL: https://minecraft.wiki/Chunk#Chunk_loading.
- [37] Ken Perlin. “An image synthesizer”. In: *SIGGRAPH Comput. Graph.* 19.3 (July 1985), pp. 287–296. DOI: 10.1145/325165.325247.
- [38] *Speedrun Leaderboards, Resources, Forums and more*. <https://www.speedrun.com/>. Accessed: 2024-10-09.
- [39] Wenhao Zhu et al. *Multilingual Machine Translation with Large Language Models: Empirical Results and Analysis*. 2024. arXiv: 2304.04675 [cs.CL]. URL: <https://arxiv.org/abs/2304.04675>.
- [40] Kishore Papineni et al. “BLEU: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. ACL ’02. Philadelphia, Pennsylvania: Association for Computational Linguistics, 2002, pp. 311–318. DOI: 10.3115/1073083.1073135.
- [41] A. J. Byer and D. Abrams. “A comparison of the triangular and two-sample taste-test methods”. In: *Food Technology* 7.4 (Apr. 1953), pp. 185–187.
- [42] Tristan Nicholas Hoang Henderson. *The effects of relative delay in networked games*. 2003. URL: <https://tnhh.org/research/pubs/thesis.pdf>.
- [43] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: <http://www.agilemanifesto.org/>.
- [44] Google Ventures. *The Design Sprint*. URL: <https://www.gv.com/sprint/>.
- [45] Zheyi Chen et al. “Evolution and Prospects of Foundation Models: From Large Language Models to Large Multimodal Models”. In: *Computers, Materials & Continua* 80.2 (2024), pp. 1753–1808. DOI: 10.32604/cmc.2024.052618.
- [46] *Open VLM and Open LLM Reasoning Leaderboards*. URL: <https://huggingface.co/spaces/opencompass>.
- [47] Weihao Yu et al. *MM-Vet: Evaluating Large Multimodal Models for Integrated Capabilities*. 2024. arXiv: 2308.02490 [cs.AI]. URL: <https://arxiv.org/abs/2308.02490>.
- [48] Pan Lu et al. *MathVista: Evaluating Mathematical Reasoning of Foundation Models in Visual Contexts*. 2024. arXiv: 2310.02255 [cs.CV]. URL: <https://arxiv.org/abs/2310.02255>.
- [49] Ke Wang et al. *Measuring Multimodal Mathematical Reasoning with MATH-Vision Dataset*. 2024. arXiv: 2402.14804 [cs.CV]. URL: <https://arxiv.org/abs/2402.14804>.
- [50] George Cameron. *AI model and API providers analysis*. URL: <https://artificialanalysis.ai/>.
- [51] Wei-Lin Chiang et al. *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference*. 2024. arXiv: 2403.04132 [cs.AI].
- [52] *OpenAI ChatGPT Pricing*. URL: <https://openai.com/chatgpt/pricing>.
- [53] *Google Gemini 1.5 Pricing*. URL: <https://ai.google.dev/pricing>.
- [54] *Anthropic Claude 3.5 Pricing*. URL: <https://www.anthropic.com/pricing>.
- [55] *Google Vertex AI*. URL: <https://cloud.google.com/vertex-ai>.
- [56] *Gemini System Instructions*. URL: <https://ai.google.dev/gemini-api/docs/system-instructions>.
- [57] *Prompt Engineering Guide*. URL: <https://www.promptingguide.ai/>.
- [58] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023. arXiv: 2211.01910 [cs.LG]. URL: <https://arxiv.org/abs/2211.01910>.

- [59] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: <https://arxiv.org/abs/2210.03629>.
- [60] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: <https://arxiv.org/abs/2303.11366>.
- [61] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: <https://arxiv.org/abs/2302.13971>.
- [62] Sewon Min et al. *Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?* 2022. arXiv: 2202.12837 [cs.CL]. URL: <https://arxiv.org/abs/2202.12837>.
- [63] Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. *Meta Prompting for AI Systems*. 2024. arXiv: 2311.11482 [cs.AI]. URL: <https://arxiv.org/abs/2311.11482>.
- [64] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: <https://arxiv.org/abs/2201.11903>.
- [65] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.
- [66] Mojang Studios. *Minecraft*. Microsoft Studios. 2011.
- [67] FabricMC. *Fabric for Minecraft*. URL: <https://fabricmc.net/>.
- [68] FabricMC. *Fabric Loader GitHub Repository*. URL: <https://github.com/FabricMC/fabric-loader>.
- [69] FabricMC. *Fabric API GitHub Repository*. URL: <https://github.com/FabricMC/fabric>.
- [70] FabricMC. *Fabric Loom GitHub Repository*. URL: <https://github.com/FabricMC/fabric-loom>.
- [71] FabricMC. *Fabric Yarn GitHub Repository*. URL: <https://github.com/FabricMC/yarn>.
- [72] FabricMC. *Fabric Template Mod Generator*. URL: <https://fabricmc.net/develop/template/>.
- [73] *Node.js*. URL: <https://nodejs.org/>.
- [74] Max Peeperkorn et al. *Is Temperature the Creativity Parameter of Large Language Models?* 2024. arXiv: 2405.00492 [cs.CL]. URL: <https://arxiv.org/abs/2405.00492>.
- [75] *How to Use the Top-P parameter?* URL: <https://www.vellum.ai/llm-parameters/top-p>.
- [76] *json-stringify-pretty-compact npm package*. URL: <https://www.npmjs.com/package/json-stringify-pretty-compact>.
- [77] *AutoHotkey*. URL: <https://www.autohotkey.com/>.
- [78] Vladimir I Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (Feb. 1966), p. 707.
- [79] Ken Perlin. “An image synthesizer”. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 287–296. ISBN: 0897911660. DOI: [10.1145/325334.325247](https://doi.org/10.1145/325334.325247).
- [80] The Amulet Team. *Amulet Map Editor for Minecraft*. URL: <https://www.amuletmc.com/>.

Appendix

Appendix A - Low-Level Prompting

System Instructions for Low-Level Prompting Approach

I aim to generate Minecraft terrains by training a model on existing biome data and then prompting it to generate new terrains. Below, I will provide 5 examples of Minecraft terrain data sampled from Plains biomes. Each example includes:

- A 256x64x256 region heightmap.
- A screenshot (top-down view).
- Run-Length Encoded (RLE) block data in JSON format.
- A block frequency summary JSON detailing the top 3 block types for each y-layer, including their respective percentages.

Your task is to learn from these examples and generate a new terrain structure for a 64x64x64 region. The output must:

- Maintain biome-specific characteristics observed in the input examples.
- Use the RLE JSON format provided below for output.
- Follow the provided block frequency distribution trends for coherence across layers.
- Preserve a plausible height variation as seen in heightmaps.

RLE Block Data Format:

```
{ "start": { "x": INT, "y": INT, "z": INT },
  "mappings": [ { "block": STRING, "symbol": STRING }, ... ],
  "grids": [ { "grid": [ [ ... ], ... ], "y": INT }, ... ] }
```

Block Frequency Summary Format:

```
{ "start": { "x": INT, "y": INT, "z": INT },
  "end": { "x": INT, "y": INT, "z": INT },
  "layers": [ { "y": INT, "types": [ { "block": STRING, "frequency": PERCENTAGE }, ... ] }, ... ] }
```

The screenshot and heightmap are aligned to a yaw of 180° (North) and pitch of 90° (vertical). Use these inputs to understand the spatial relationships and biome patterns.

Your final task is to generate plausible RLE JSON output for a 64x64x64 region, aligned to biome-specific principles derived from the training data.

Note: I will request what layers to generate the JSON for, but it will always be 16 layers. For example, I might request from y=62 to y=77 or from y=110 to y=125.

The system instructions provided for generating Minecraft terrains using low-level biome data prompting.

User Request:

Based on these 5 examples, can you now provide a JSON that produces a similar Plains biome for the layers of y=62 to y=77, ensuring that the output produced is "interesting" with a clear variety in the blocks used and where they are placed.

AI Response:

```
{  
    "start": {"x": 0, "y": 62, "z": 0},  
    "mappings": [  
        {"block": "minecraft:air", "symbol": "A"},  
        {"block": "minecraft:stone", "symbol": "S"},  
        {"block": "minecraft:dirt", "symbol": "D"},  
        {"block": "minecraft:grass_block", "symbol": "G"},  
        {"block": "minecraft:water", "symbol": "W"},  
        {"block": "minecraft:gravel", "symbol": "R"},  
        {"block": "minecraft:sand", "symbol": "N"},  
        {"block": "minecraft:tall_grass", "symbol": "T"},  
        {"block": "minecraft:dandelion", "symbol": "L"},  
        {"block": "minecraft:poppy", "symbol": "P"},  
        {"block": "minecraft:oxeye_daisy", "symbol": "O"}  
    "grids": [  
        {"grid": [  
            "16S12D28G8W12A4S2D1A1G7N13S12D30G1LA6A2S4D2A2G13N15S9D29G2LA",  
            "18S10D30G6W10A6S4D1LA1G1ON12S11D30G1LA5A3S5D1A1G16N16S12D27G3LA",  
            "20S8D30G4W8A8S4D3A2G8N19S9D28G1LA2A4S4D4A1G15N17S10D27G5LA",  
            "...",  
            "...",  
            "..."  
        ], "y": 62},  
        ...  
    ]  
}
```

The user's request and AI's generated JSON output for a varied Plains biome generation.

Appendix B - High-Level Prompting

B.1 - Iterative Terrain Generation

Input

User: Now that you have understood the general structure of Plains biomes in Minecraft, what features, structures, vegetation and terrain appear, I would like you to produce a list of functions with parameters to build such a biome in-game on top of a flat terrain that has XYZ coordinates from (0, 64, 0) to (255, 64, 255), where north is in the direction of negative Z and east is in the direction of positive X. Please separate these functions line-by-line starting them with "/generate" (without speech marks), including what to generate and its specific parameters. The possible things you can generate are listed below in the order you should write them (we prioritise terrain shape, then water features, vegetation and finally structures):

1. Name: **Hill**,

Description: Generates a hilly region using Perlin Noise between a set of coordinates (two x and z values), the hills having a maximum amplitude.

Parameters: Starting X (startX), Starting Z (startZ), Ending X (endX), Ending Z (endZ), Max Amplitude (maxAmplitude).

Structure: "/generate hill [startX] [startZ] [endX] [endZ] [maxAmplitude]".

Example: "/generate hill 0 0 100 100 60"

2. Name: **Lake**,

Description: Generates a circle or rectangular-shaped pool of water at a given center point with a given radius, or if rectangular, width and length.

Parameters: Center X (centerX), Center Z (centerZ), Shape (circle), Length/Radius (length/radius), Width (width).

Structure: "/generate lake [centerX] [centerZ] [shape] [radius]" (for a circular lake) or "/generate lake [centerX] [centerZ] [shape] [length] [width]" for rectangular lakes.

Examples: "generate lake 45 50 circle 10", "generate lake 84 21 12 5"

3. Name: **River**,

Description: Generates a river of a certain width between a start and end point, the river's direction randomly changing at different points.

Parameters: Starting X (startX), Starting Z (startZ), Ending X (endX), Ending Z (endZ), Width (width).

Structure: "/generate river [startX] [startZ] [endX] [endZ] [width]".

Example: "/generate river 20 30 200 230 3"

4. Name: **Grass**,

Description: Generates grass within a circular radius at a low, medium (med) or high density.

Parameters: Center X (centerX), Center Z (centerZ), Radius (radius), Density (density).

Structure: "/generate grass [centerX] [centerZ] [radius] [density]".

Examples: "/generate grass 82 56 10 low", "/generate grass 42 59 14 med", "/generate grass 6 201 3 high"

5. Name: **Flowers**,

Description: Generates flowers within a circular radius at a low, medium (med) or high density.

Parameters: Center X (centerX), Center Z (centerZ), Radius (radius), Density (density).

Structure: "/generate flowers [centerX] [centerZ] [radius] [density]".

Examples: "/generate flowers 72 50 40 low", "/generate flowers 20 200 16 med", "/generate flowers 68 231 9 high"

6. Name: **Forest**,

Description: Generates a forest of oak trees within a circular radius at a low, medium (med) or high density.

Parameters: Center X (centerX), Center Z (centerZ), Radius (radius), Density (density).

Structure: "/generate forest [centerX] [centerZ] [radius] [density]".

Examples: "/generate forest 98 56 42 low", "/generate forest 29 209 23 med", "/generate forest 22 222 6 high"

7. Name: **Village**,

Description: Generates a village centered at a set of coordinates.

Parameters: Center X (centerX), Center Z (centerZ).

Structure: "/generate village [centerX] [centerZ]".

Example: "/generate village 183 35"

Please use any number of any of the commands available, making sure to use their names and parameters correctly (and in order) after the "/generate" text. Think about how the terrain will develop after these commands are ran and how they all interplay with each other to provide a fun experience for players of a "quintessential" plains biome in Minecraft.

Output

Model: Here's a possible set of commands to generate a quintessential Plains biome in Minecraft within the specified area, prioritizing terrain, water, vegetation, and then structures:

```
/generate hill 0 0 255 255 15 // Gentle rolling hills, max height variation of 15 blocks  
/generate river 10 20 245 230 5 // A winding river across the landscape  
/generate lake 60 100 circle 20 // A medium-sized circular lake  
/generate lake 180 50 circle 12 // A smaller circular lake  
/generate forest 50 50 30 low // A small, sparse forest  
/generate forest 200 200 40 med // A larger, denser forest  
/generate grass 0 0 255 255 med // Medium density grass across the entire plains  
/generate flowers 0 0 255 255 low // Sparse flower distribution  
/generate village 128 128 // A village near the center
```

Explanation and Design Choices:

1. **Gentle Hills:** The `hill` command creates the base terrain. A maximum amplitude of 15 ensures gentle slopes, typical of plains.
2. **Winding River:** The `river` command creates a primary water feature. The start and endpoints are near opposite corners to encourage a meandering path. A width of 5 creates a sizable river.
3. **Lakes:** Two `lake` commands add variety with different sizes and positions, simulating natural water bodies. Circular shapes are generally visually appealing in plains.
4. **Varied Forests:** Two `forest` commands generate oak trees with differing densities and sizes. This creates a more natural and less uniform feel than a single large forest.
5. **Grass and Flowers:** The `grass` and `flowers` commands cover the entire plains area with their respective elements. Medium grass density and low flower density create a balanced, realistic look.
6. **Central Village:** The `village` command places a village near the center of the plains, making it a focal point and easily accessible. The proximity to the river increases the realism and provides a water source for villagers.

Considerations for Improved Generation:

* **Seed Values:** Adding seed values to each command (e.g., `/generate hill 0 0 255 255 15 12345`) would allow for reproducible results and control over randomness. * **More Complex Rivers:** The `river` command could be enhanced with parameters to control its curvature or allow for tributaries. * **Tree Variety:** Adding parameters to the `forest` command to specify tree types (e.g., birch, acacia) would create more diverse landscapes. * **Flower Types:** Similar to tree variety, allowing specific flower types in the `flowers` command would add realism. * **Village Placement Logic:** Instead of specifying village coordinates directly, an algorithm could be implemented to find suitable locations based on terrain flatness and proximity to water. * **Biome Transitions:** For future expansion, functions could be added to handle transitions to other biomes at the edges of the plains.

This improved set of commands, coupled with your original terrain analysis, provides a solid framework for procedurally generating more dynamic and realistic Plains biomes in Minecraft.

First iteration input and output from the model, where we start building the terrain for the plains biome.



First generated terrain

Input

User: Here is a screenshot of the generated terrain from these commands

[Image of the generated terrain has been redacted for brevity]

Based on this screenshot, consider what to improve on when generating another terrain. My own feedback about this generated code is that potentially more code could be written for things like grass and flowers, rather than having radii and counts the same as the length of the region; as well as this, centering at (0, 0) means that there will be one corner that isn't covered if the aim is to cover the whole region. After understanding how to improve, please generate another terrain with the commands.

Output

Model: You're right, the previous generation had some weaknesses. Generating grass and flowers with such a large radius wasn't efficient, and the centering meant incomplete coverage. I also think the forests were too close to the village, which is less common in natural generation. The terrain itself could also use a bit more variation.

Here's a revised set of commands aiming for a more natural and interesting Plains biome, addressing those points:

```
/generate hill 0 0 255 255 20 42 // Slightly higher hills for more
variation, using seed 42
/generate river 10 20 245 230 4 87 // Winding river, slightly narrower,
using seed 87
/generate lake 70 120 circle 15 123 // Shifted lake position, using seed 123
/generate lake 190 60 circle 10 456 // Shifted lake position, using seed 456
/generate forest 40 40 25 low 789 // Smaller, sparse forest, moved away from
center, using seed 789
/generate forest 210 210 30 med 101 // Larger, denser forest, moved away
from center, using seed 101
/generate grass 64 64 64 med // Grass around the central area
/generate grass 192 64 64 med // Grass around the second area
/generate grass 64 192 64 med // Grass around the third area
/generate grass 192 192 64 med // Grass around the fourth area
/generate flowers 0 0 255 255 low 202 // Sparse flowers everywhere
/generate village 128 128 303 // Village near the center
```

Changes and Reasoning:

[Some changes and reasoning have been redacted for brevity]

* **Multiple Grass Patches:** Used multiple grass commands to cover each quarter of the map individually. The radius 64 covers a quarter of the 255 block length from the center of each quadrant, which fully covers the area without leaving gaps at the corners of each grass-generating call. This approach is more efficient and ensures complete coverage.

* **Consistent Flower Coverage:** The flowers now cover the entire area consistently with the original single command due to low density.

* **Seed Values:** Added seed values to all commands for reproducible results and better control over the randomness.

This revised generation strategy addresses the previous shortcomings and aims to create a more dynamic and "quintessential" Minecraft Plains biome. The inclusion of seed values allows for greater control and repeatability.

Second input and output from the model, here we are refining the output from the last iteration.



Second generated terrain

Input

User: I have attached a screenshot of the generated terrain from these commands:

[Image of the generated terrain has been redacted for brevity]

Based on this screenshot, please consider what to improve on when generating another terrain. My own feedback about this generated code is that you added a seed value to functions like `/generate village` which only takes in starting X and Z coordinates, the same being true for lakes which have no randomness to them. For `/generate flowers`, you hallucinated and added an extra parameter before density, so make sure you are strictly following the function rules. One other thing is that, if you liked a particular feature in a generated terrain, then it may be worth re-using the seed and other parameters used to generate that feature.

Based on your use of seeds in the second output, here are the function definitions to adhere to when generating a terrain:

[Function definitions have been redacted for brevity] After understanding how to improve, please generate another terrain with the commands.

Output

Model: You are absolutely correct! My apologies for the errors in the previous command set. I hallucinated a seed parameter for `village` and `lake`, and added an erroneous parameter to `flowers`. I've also taken your suggestion about reusing successful seeds for features I liked - specifically, the river from the second generation.

Here's a new set of commands for a Plains biome generation, adhering strictly to the function definitions and incorporating the river from the second attempt:

```
/generate hill 0 0 255 255 18 1738 // Gentle rolling hills, seed 1738 for a new terrain
shape
/generate river 10 20 245 230 4 87 // Winding river, reusing seed 87 from the second
generation
/generate lake 75 115 circle 18 // Slightly shifted lake, medium size
/generate lake 195 55 circle 10 // Slightly shifted smaller lake
/generate forest 50 45 28 low 9643 // Sparse forest, new seed and position
/generate forest 215 205 35 med 2487 // Larger forest, new seed and position
/generate grass 64 64 64 med // Central grass patch
/generate grass 192 64 64 med // Second grass patch
/generate grass 64 192 64 med // Third grass patch
/generate grass 192 192 64 med // Fourth grass patch /generate flowers 64 64 64 low 5512 //
Flowers in first quadrant, seed 5512
/generate flowers 192 64 64 low 6832 // Flowers in second quadrant, seed 6832
/generate flowers 64 192 64 low 7955 // Flowers in third quadrant, seed 7955
/generate flowers 192 192 64 low 8104 // Flowers in fourth quadrant, seed 8104
/generate village 128 128 // Centered village
```

Key Improvements:

* **Strict Adherence to Function Definitions:** All commands now use the correct parameters and no extraneous ones.

* **Reused River Seed:** The successful river generation from the second attempt (seed 87) is reused.

* **Improved Flower Distribution:** Flowers are now generated per quadrant, improving their distribution across the whole area, and allowing for more controlled randomness with individual seed values.

* **New Seeds for Hills and Forests:** New seed values introduce variation in the terrain shape and forest layout.

* **Slightly Adjusted Lakes:** Maintained the circular lakes but shifted their positions slightly for a new look.

This generation should produce a more refined Plains biome. I'm learning and improving with each iteration thanks to your feedback! I am also getting closer to understanding how best to generate such a terrain by learning how each of the parameters affects the biome, and taking note of visually appealing features.

Third input and output from the model, where we are refining the output from the last iteration.



Third generated terrain with improved terrain features and seeds. Hallucination of seed parameter for village commands has been corrected.

Input

User: The generated terrain screenshot is below:

[Image of the generated terrain has been redacted for brevity]

My only feedback about this terrain is that you again provided a seed value for the lake function, when there is no seed parameter needed (there is no randomness for lakes). Please come up with your own feedback of the terrain and generate an improved terrain.

Output

Model: You are absolutely right, I mistakenly added seed values to the `lake` commands again. It seems I'm still struggling a bit with consistently applying the function parameters correctly.

Regarding the terrain itself, I think it's a decent Plains biome, but here's what I'd like to improve:

* **Lake Placement:** While the lakes are fine, they feel a bit too close to the edges. I'd prefer them more integrated within the plains. * **Forest Diversity:** Both forests are using oak trees. While this is common in Plains, a bit of variation would be nice. Perhaps introducing a small birch forest would add visual interest. * **Terrain Smoothness:** The hills look a bit too regular. I'd like to experiment with the hill generation parameters or seed value to create a slightly more natural and less predictable terrain.

Here's my next attempt at generating a Plains biome, taking into account these improvements and correcting the lake command errors:

```
/generate hill 0 0 255 255 22 777 // Slightly increased hill amplitude and a new seed for more varied terrain  
/generate river 10 20 245 230 4 87 // Keeping the successful river from previous generations (seed 87)  
/generate lake 90 140 circle 20 // Relocated lake, further from the edge  
/generate lake 160 90 circle 12 // Relocated lake, further from the edge  
[Some commands have been redacted for brevity]  
/generate flowers 64 192 64 low 7955  
/generate flowers 192 192 64 low 8104  
/generate village 128 128 // Centered village
```

Key Changes:

* **Corrected Lake Commands:** Removed the erroneous seed values from the lake generation.

* **Lake Relocation:** Moved the lakes further inward for better integration with the plains.

* **Added Birch Forest:** Included a small birch forest to diversify the vegetation, assuming a birch variant of the forest function is available. If not, this command could be omitted or replaced with another oak forest.

* **Adjusted Hill Generation:** Used a new seed and increased the amplitude slightly to attempt a less regular, more natural hill distribution.

* **Maintained Successful Elements:** Kept the river (seed 87) and flower distributions (using previous seeds) that I liked from the previous generation.

I believe this generation will result in a more visually appealing and diverse plains biome. I'll continue to refine my understanding of the parameters and their effects to generate even better terrains in the future. I'm also ready to take on the challenge of generating terrain adjacent to an existing region!



Fourth generated terrain with improved terrain features and no lake seed.

Input

User: The generated terrain's screenshot:

[Image of the generated terrain has been redacted for brevity]

I personally have not noticed any hallucinations from the last output, but please generate your own feedback from this last terrain. From now on, let's do structures before vegetation like flowers, grass and trees to avoid weird looking village paths. Please be careful that seed values are integer values; I have also introduced a change to the forest generation function, its new definition is as follows:

Name: Forest

Description: Generates a forest of oak or birch trees within a circular radius at a low, medium (med) or high density.

Parameters: Center X (centerX), Center Z (centerZ), Radius (radius), Density (density), Tree Type (type), Seed (seed),

Structure: "/generate forest [centerX] [centerZ] [radius] [density] [type] [seed]",

Examples: "/generate forest 98 56 42 low oak 77", "/generate forest 29 209 23 med birch 90", "/generate forest 22 222 6 high any 45"

Please generate another terrain based on these changes and your own feedback.

Output

Model: I'm glad you haven't noticed any further hallucinations! I'm working hard to be more precise with the commands.

Looking at the last generated terrain, I think it's an improvement, but here's my feedback:
* **Birch Forest Size:** The birch forest, while a good addition, feels a bit too small. I'd like to increase its size a little to make it a more prominent feature.
* **Village Integration with River:** The river seems to clip through the village a bit awkwardly. It would be nice if the village generation could better adapt to the presence of the river, creating a more natural waterfront. (This would require more sophisticated village generation logic than the current 'village' command allows.)
* **Overall Density:** The terrain still feels a little sparse. While plains are generally open, increasing the density of grass and flowers in some areas might make it feel more lush.

Here's my next attempt, adjusting the birch forest size, trying for a better river-village interaction (within the constraints of the available command), placing the structures before vegetation as you suggested, and adding an option for "any" tree type which generates equal amounts of oak and birch trees:

```
/generate hill 0 0 255 255 25 912 // New seed for hills, slightly higher variation  
/generate river 10 20 245 230 4 87 // Keeping the familiar river  
/generate lake 95 135 circle 17 // Adjusting lake positions slightly  
/generate lake 170 85 circle 10  
[Some commands have been redacted for brevity]  
/generate flowers 192 192 72 low 8104 // Increased radius to 72 for each quadrant to improve flower coverage
```

Changes:

* **Increased Birch Forest Radius:** Increased the radius of the birch forest to make it a more noticeable feature.
* **Adjusted Forest and Lake Positions:** Made subtle adjustments to positions to try and avoid overlap.
* **New Hill Seed:** Trying a new seed (912) for the hill generation to see if it creates a better terrain.
* **Structures Before Vegetation:** Generated structures (village, forests) before vegetation to avoid issues with paths and other village elements.
* **Increased Grass and Flower Coverage:** Slightly increased the radii of grass and flower generation to enhance density and coverage.

This generation should result in a more balanced and visually engaging Plains biome. The changes are subtle, but I believe they will contribute to a more interesting and enjoyable Minecraft experience. I'm looking forward to seeing the result and incorporating your feedback on this iteration. I'm also excited to work on context-based generation next, once we're happy with the overall Plains generation.



Fifth generated terrain with improved terrain features and not giving it any feedback as part of the input. Despite changing the forest generation function, the model still understands and produces correct commands.

B.2 - Terrain Extension

Input

User: The generated terrain's screenshot:

[Image of the generated terrain has been redacted for brevity]

Could you generate the region west of this one, with the coordinates from (-256, 0) to (-1, 255)?

Output

Model: Here's a set of commands to generate the region west of the existing terrain, spanning coordinates (-256, 0) to (-1, 255). I'll try to create a seamless transition, maintaining the style and density of the existing Plains biome:

```
/generate hill -256 0 -1 255 25 1492 // Maintaining hill variation, new seed  
for this region  
/generate river -240 30 -10 220 4 654 // New river flowing roughly parallel  
to the existing one.  
/generate lake -150 150 circle 15 // Lake placed away from the edge, similar  
in size to existing lakes.  
/generate lake -80 70 circle 10 // Another smaller lake.  
/generate forest -200 50 25 low oak 3378 // Oak forest, positioned towards  
the west  
/generate forest -100 200 32 med birch 9152 // Birch forest, further from  
the edge  
/generate forest -128 20 28 med any 156 // Mixed forest with a new seed,  
closer to initial birch forest, increasing density with vegetation  
/generate grass -64 64 68 med // Grass patches adjusted for this region  
/generate grass -192 64 68 med  
/generate grass -64 192 68 med  
/generate grass -192 192 68 med  
/generate flowers -256 0 -1 255 low 4444 // Consistent flower density across  
the region
```

Rationale and Continuation of Patterns:

* **Hills:** Similar hill generation parameters as the previous region to maintain consistency, but with a new seed (1492) to avoid identical terrain. * **River:** Generated a new river roughly parallel to the existing one, with a slightly different seed to ensure it doesn't look like a direct copy. The width is kept at 4 to be consistent. * **Lakes:** Created lakes similar in size and distribution to those in the existing terrain, positioning them towards the center of this new region. * **Forests:** Generated a mix of oak and birch forests with varying sizes and densities. Positioned them to create a balanced and natural feel, not clustering them too close to any single location (except where there is "any" type forest). * **Grass and Flowers:** Maintained the same grass patch strategy and flower density. The coordinates are adjusted for this new western region. I've also experimented with generating flowers across the whole region with one command due to low density. If you prefer more control over flower distribution, you can revert to generating per quadrant using the previous seeds. The goal is for this western region to feel like a natural extension of the existing Plains biome, while still introducing some variety to keep it interesting. I'm eager to see how this generation turns out, and I'm ready to refine it further based on your feedback!

The prompt and output used to generate the terrain west of the initial terrain.



Generated terrain for the region west and east of the initial terrain.

Appendix C - Qualitative Evaluation

C.1 - Larger Feedback Form Questions

The larger feedback form used for the qualitative evaluation.

Minecraft Gameplay Experience Study

* Required

About Today's Session

1. What is your Participant number? *

2. How much did you enjoy today's session?

☆ ☆ ☆ ☆ ☆

3. Where there any moments you found particularly enjoyable or memorable?

4. Was there anything you didn't enjoy about this session?

5. Was there anything that stood out to you about any of the scenes shown?

6. Any other comments about your gameplay experience?

About you

7. How often do you play video games?

- Daily
- Weekly
- Monthly
- Yearly
- Rarely
- Never
- Other

8. What genre(s) of video games do you play the most, if any?

- Action
- Adventure
- RPG
- Simulation
- Strategy
- Puzzle
- Sports
- Music/Rhythm
- Party
- Casual
- VR/AR
- Hybrid/Experimental
- Sandbox
- Other

9. What gender do you identify as?

- Woman
- Man
- Non-binary
- Prefer not to say

10. What platform do you play games on the most, if any?

- Mobile Phone
- PC
- Console
- Cloud
- TV

11. How would you rate your Minecraft ability before today's session?

- Complete Beginner
- Played once before
- Played a handful of times before
- Occasional Player
- Experienced but Inactive
- Regular Player
- Veteran

12. Anything else about you that may be relevant?

Generative AI

13. Do you think AI could enhance the creativity or design of game worlds? Why or why not?

14. Did any of the scenes you played in feel like they might have been designed by AI?

15. How do you feel about AI's role in game creation overall?

16. If you could use AI to design your own game world, what features or elements would you prioritise?

17. What potential benefits or drawbacks do you see with AI-generated game content?

C.2 - Example User Data

Task Order: [2, 1, 3]
Terrain Order: [L1, R1, R2, L2, R1, R2, L1, R1, R2]
Task Results: [31, -16, 137, 8, 0, 0, -1, 15, 0]
True Odd Ones: [L, L, L]
Chosen Odd Ones: [L, L, R]
Sureness: [0.77, 0.34, 0.14]
Enjoyment: [0.79, 0.79, 0.75, 0.57, 0.03, 0.06, 0.76, 0.83, 0.77]
Difficulty: [0.08, 0.2, 0.16, 0.41, 0.85, 0.85, 0.59, 0.48, 0.53]
Total Enjoyment: 4
How Often: Daily
Genre(s): Action; RPG; Strategy; Casual
Gender: M
Platform: Mobile Phone
Ability: Experienced but Inactive

Appendix D - Quantitative Evaluation

D.1 - Block Frequency Data For Each Terrain

Layer	Block	Percentage
Sea Level + 0	Water	53.84%
	Stone	37.17%
	Dirt	5.54%
Sea Level + 1 - Sea Level + 3	Air	56.62%
	Stone	35.00%
	Dirt	4.18%
Sea Level + 4 - Sea Level + 8	Air	62.09%
	Stone	30.55%
	Dirt	4.34%
Sea Level + 9 - Sea Level + 12	Air	67.23%
	Stone	24.78%
	Dirt	4.50%
Sea Level + 13 - Sea Level + 16	Air	73.19%
	Stone	19.03%
	Dirt	4.69%
Sea Level + 17 - Sea Level + 19	Air	78.56%
	Stone	14.75%
	Dirt	3.54%
Sea Level + 20 - Sea Level + 24	Air	83.97%
	Stone	11.07%
	Dirt	3.22%
Sea Level + 25 - Sea Level + 28	Air	88.56%
	Stone	6.71%
	Dirt	2.22%
Sea Level + 29 - Sea Level + 36	Air	94.08%
	Stone	3.39%
	Dirt	1.46%
Sea Level + 37 - Sea Level + 63	Air	99.47%
	Dirt	0.25%
	Stone	0.11%

Table 22: Block frequency data for *reg-1*.

Layer	Block	Percentage
Sea Level + 0	Water	75.06%
	Stone	19.98%
	Dirt	2.54%
Sea Level + 1 - Sea Level + 5	Air	77.41%
	Stone	17.84%
	Dirt	2.77%
Sea Level + 6 - Sea Level + 8	Air	80.62%
	Stone	11.80%
	Dirt	5.31%
Sea Level + 9 - Sea Level + 11	Air	85.85%
	Stone	8.42%
	Dirt	1.79%
Sea Level + 12 - Sea Level + 17	Air	91.32%
	Stone	4.86%
	Dirt	2.27%
Sea Level + 18 - Sea Level + 34	Air	97.67%
	Stone	1.16%
	Dirt	0.70%
Sea Level + 35 - Sea Level + 63	Air	99.97%
	Dirt	0.01%
	Grass Block	0.00%

Table 23: Block frequency data for *reg_2*.

Layer	Block	Percentage
Sea Level + 0	Grass Block	53.85%
	Dirt	42.18%
	Water	2.99%
Sea Level + 1	Air	34.72%
	Stone	31.73%
	Short Grass	19.22%
Sea Level + 2 - Sea Level + 3	Air	51.16%
	Stone	31.50%
	Dirt	7.05%
Sea Level + 4 - Sea Level + 5	Air	58.70%
	Stone	25.21%
	Dirt	6.30%
Sea Level + 6 - Sea Level + 7	Air	66.47%
	Stone	20.14%
	Dirt	5.59%
Sea Level + 8 - Sea Level + 9	Air	73.36%
	Stone	15.71%
	Dirt	4.84%
Sea Level + 10 - Sea Level + 11	Air	79.12%
	Stone	12.27%
	Dirt	3.67%
Sea Level + 12 - Sea Level + 14	Air	84.71%
	Stone	9.12%
	Dirt	2.66%
Sea Level + 15 - Sea Level + 19	Air	90.53%
	Stone	5.71%
	Dirt	1.61%
Sea Level + 20 - Sea Level + 26	Air	95.32%
	Stone	1.93%
	Dirt	1.44%
Sea Level + 27 - Sea Level + 63	Air	99.95%
	Short Grass	0.03%
	Grass Block	0.01%

Table 24: Block frequency data for *llm_1*.

Layer	Block	Percentage
Sea Level + 0	Grass Block	62.53%
	Dirt	32.37%
	Water	4.07%
Sea Level + 1	Stone	38.86%
	Air	27.53%
	Short Grass	18.67%
Sea Level + 2 - Sea Level + 3	Air	51.82%
	Stone	30.48%
	Grass Block	6.36%
Sea Level + 4 - Sea Level + 5	Air	59.37%
	Stone	24.66%
	Grass Block	5.68%
Sea Level + 6 - Sea Level + 7	Air	67.16%
	Stone	19.66%
	Grass Block	4.89%
Sea Level + 8 - Sea Level + 9	Air	73.94%
	Stone	15.34%
	Grass Block	4.29%
Sea Level + 10 - Sea Level + 11	Air	79.12%
	Stone	12.27%
	Grass Block	3.21%
Sea Level + 12 - Sea Level + 14	Air	84.71%
	Stone	9.12%
	Grass Block	2.66%
Sea Level + 15 - Sea Level + 19	Air	90.53%
	Stone	5.71%
	Dirt	1.61%
Sea Level + 20 - Sea Level + 26	Air	95.32%
	Stone	1.93%
	Dirt	1.44%
Sea Level + 27 - Sea Level + 63	Air	99.95%
	Short Grass	0.03%
	Grass Block	0.01%

Table 25: Block frequency data for *llm_2*.

Layer	Block	Percentage
Sea Level + 0	Grass Block	82.24%
	Water	12.71%
	Dirt	4.36%
Sea Level + 1	Air	80.21%
	Stone	8.58%
	Short Grass	8.07%
Sea Level + 2 - Sea Level + 7	Air	90.66%
	Stone	5.63%
	Grass Block	1.43%
Sea Level + 8 - Sea Level + 19	Air	96.76%
	Stone	1.86%
	Grass Block	0.71%
Sea Level + 20 - Sea Level + 63	Air	99.25%
	Stone	0.46%
	Grass Block	0.13%

Table 26: Block frequency data for *random_1*.

Layer	Block	Percentage
Sea Level + 0	Grass Block	78.09%
	Water	19.41%
	Dirt	1.61%
Sea Level + 1	Air	75.28%
	Stone	12.33%
	Short Grass	7.12%
Sea Level + 2 - Sea Level + 5	Air	85.66%
	Stone	8.06%
	Grass Block	3.80%
Sea Level + 6 - Sea Level + 11	Air	92.58%
	Stone	4.35%
	Grass Block	2.00%
Sea Level + 12 - Sea Level + 63	Air	99.26%
	Stone	0.45%
	Grass Block	0.19%

Table 27: Block frequency data for *random_2*.

Layer	Block	Percentage
Sea Level + 0	Grass Block	53.54%
	Water	44.44%
	Dirt	1.57%
Sea Level + 1	Air	71.00%
	Stone	14.47%
	Short Grass	8.20%
Sea Level + 2 - Sea Level + 5	Air	81.94%
	Stone	10.74%
	Grass Block	4.81%
Sea Level + 6 - Sea Level + 9	Air	88.16%
	Stone	7.10%
	Grass Block	3.25%
Sea Level + 10 - Sea Level + 16	Air	93.90%
	Stone	3.51%
	Grass Block	1.88%
Sea Level + 17 - Sea Level + 63	Air	99.52%
	Stone	0.25%
	Grass Block	0.17%

Table 28: Block frequency data for *random_3*.

Layer	Block	Percentage
Sea Level + 0	Water	85.94%
	Grass Block	13.22%
	Dirt	0.44%
Sea Level + 1 - Sea Level + 2	Air	79.66%
	Stone	13.30%
	Grass Block	4.58%
Sea Level + 3 - Sea Level + 6	Air	85.36%
	Stone	8.74%
	Grass Block	3.71%
Sea Level + 7 - Sea Level + 10	Air	91.10%
	Stone	4.78%
	Grass Block	2.84%
Sea Level + 11 - Sea Level + 27	Air	97.04%
	Stone	1.60%
	Grass Block	0.93%
Sea Level + 28 - Sea Level + 63	Air	99.83%
	Stone	0.09%
	Grass Block	0.05%

Table 29: Block frequency data for *random_4*.