Send me new version with who did what (appendix) of template in CV

UNIVERSITAT DE BARCELONA

Sprint 2: Project Progress

Procedurally Generated 2D RPG with AI-Driven Narrative

Group 3

pag 23-25 without appendix

Oriol MIRÓ, Jean DIÉ, Bruno SÁNCHEZ, Dániel MÁCSAI

Master in Artificial Intelligence

Normative and Dynamic Virtual Worlds
Sprint 2: Project Progress

November 17th, 2025

1. Introduction

1.1. General Overview

The project involves developing a simple 2D top-down game in which a player navigates a procedurally generated dungeon filled with enemies, obstacles, and various NPCs. Our goal is to generate almost every element of the game procedurally, from the layout of the dungeon, to the story – within some bounds, of course – and even the ambient music. The main challenge will be to make all this generated content consistent, which requires designing a generation pipeline. Moreover, we would like the story itself to explain *why* so many things change across runs. There will be three parts to the story, each with a boss at the end. Defeating each boss will also serve as a checkpoint, and the generated story and dungeon will be frozen on that point. See Figure 1 for a schematic representation of this.

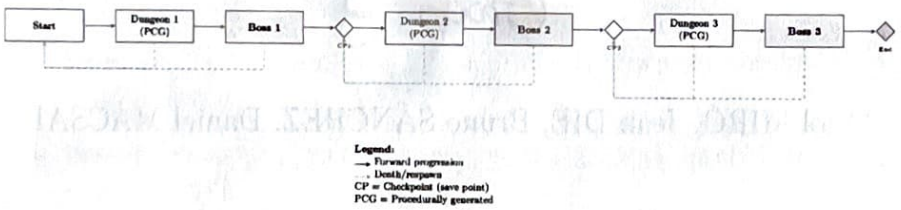


Figure 1: The player's journey through three procedurally generated dungeon sections, each culminating in a boss fight that serves as a checkpoint. Death sends the player back to their last checkpoint, where both the dungeon and narrative regenerate.

1.2. Tutorial we start from

To accelerate development of the core mechanics, we base the initial phase of our project on the online tutorial course "Unity 2D RPG: Complete Combat System" (Unity 2D RPG: Complete Combat System) from Udemy. All the information can be found in this link. Figure 2 shows an example scene from the tutorial. This tutorial guides us through creating a classic 2D top-down RPG in Unity using C#. It covers topics such as player movement, tilemap and rule tile usage, weapon systems, and basic combat. By using this foundation, we can focus our efforts on the AI side of the game rather than reinventing standard mechanics from scratch.

Specifically, the tutorial includes:

- Basic 2D top-down player movement and animation (using tilemaps and rule tiles)
- Implementation of a weapon-based combat system in Unity (including multiple weapons, attack animations, and hit detection)
- Setting up scene workflow and tiled environments using Unity's tilemap features
- Some C# fundamentals tied to game logic in Unity (moving from beginner to intermediate level)

What the tutorial does **not** cover (and which our project will therefore extend) includes:

- Procedural content generation of dungeon layouts (we will implement a BSP-based generator)
- Narrative generation tied to player runs and persistent checkpointing of story chapters
- Procedural ambient music generation and dynamic layering based on gameplay state
- Reinforcement learning-based boss behaviour trained through self-play
- The novel "checkpoint/hall of fixed chapters" mechanic, where defeating bosses 1 and 2 freezes that portion of the dungeon and story

All scenes in the tutorial are manually designed, and all the enemies' movement and attack patterns are random, not tied to any player input.

1.3. Game Agents

In this section, we describe the design and behavior of the various agents within the game. These agents, which include the player character, common enemies, and bosses, are crucial for creating a dynamic and engaging player experience. We will detail their underlying AI, movement patterns, and combat mechanics.

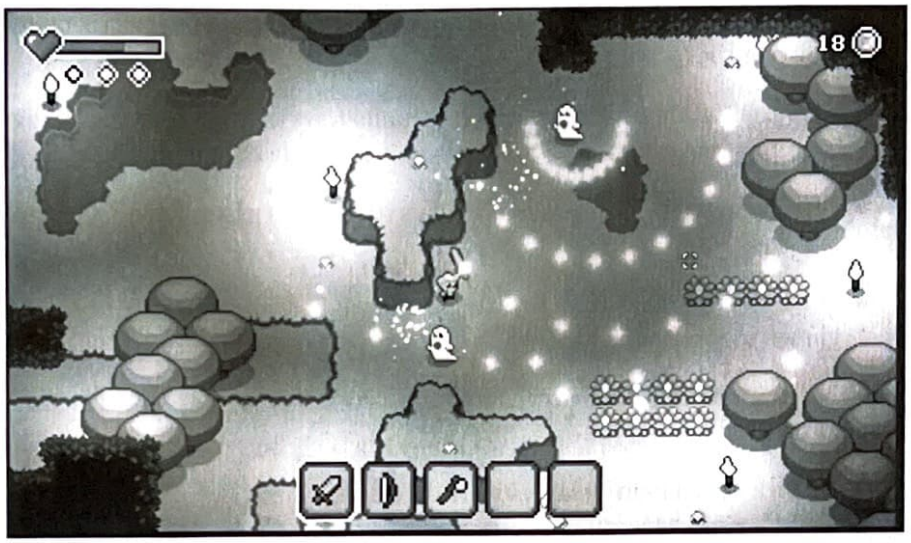


Figure 2: Example screenshot from the tutorial

1.3.1. Player Agent

The player controls a knight character, navigating the dungeon and engaging in combat. The agent's mechanics (implemented following the tutorial mentioned in Section 1.2) are designed to be intuitive and offer a variety of tactical options.

Movement and Abilities. The player uses the standard WASD keys for top-down movement. Aiming is controlled by the mouse cursor, allowing for 360-degree targeting independent of movement direction. A key defensive ability is the dash, triggered by the space bar, which provides a short burst of invulnerability and speed. Dashing consumes stamina from a dedicated bar, which regenerates automatically over time, requiring players to manage its use strategically.

Combat and Weapons. Attacks are initiated with a left-click. The player has access to three distinct weapons, which can be switched using the number keys (1-3):

- **Sword (Key 1):** A melee weapon that performs a sweeping attack in a short arc in front of the player. It is ideal for close-quarters combat against multiple weak enemies.

- **Bow (Key 2):** A long-range weapon that shoots a single arrow. The arrow is destroyed upon hitting an enemy, making it suited for picking off targets from a distance.
- **Magic Staff (Key 3):** A medium-range weapon that fires a piercing beam of magic. The beam travels through all enemies in its path, making it effective for dealing with lined-up groups.

All successful attacks apply a knockback effect to enemies, providing a brief moment of crowd control and repositioning opportunity.

Health and Resources. The player has a health bar that decreases upon taking damage from enemy attacks. Health can be replenished by collecting hearts, which have a chance to drop from defeated enemies. Additionally, fallen enemies drop coins that can be collected. While not yet implemented, the plan is for these coins to be used as currency in a shop system, should time permit its development.

1.3.2. Basic Monsters Agents

The behavior of common enemies is governed by a Finite State Machine (FSM) that dictates their actions based on the player's proximity. Each enemy agent operates in one of two states: `Roaming` or `Attacking`. When in the `Roaming` state, the enemy moves to randomly generated points within the dungeon. If the player enters a predefined `attackRange`, the enemy transitions to the `Attacking` state. While attacking, each enemy is subject to an `attackCooldown` timer, which prevents it from attacking continuously and enforces a minimum delay between consecutive attacks. The enemy will remain in the `Attacking` state until the player moves out of range, at which point it returns to `Roaming`. Figure 3 illustrates the finite state machine (FSM) governing enemy behavior. The diagram shows the two main states (`Roaming` and `Attacking`) and the transitions triggered by the player's proximity.

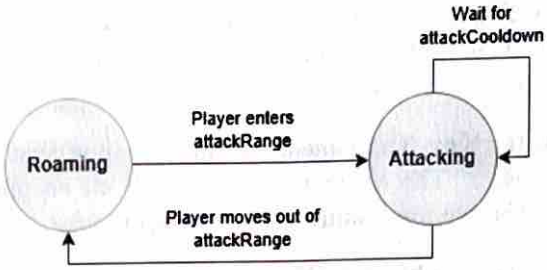


Figure 3: Finite State Machine (FSM) for enemy agents. Enemies switch between `Roaming` and `Attacking` states based on the player's distance.

Currently, there are two types of enemies implemented, as shown in Figure 4:

- **Slimes:** These enemies represent the simplest form of the FSM. They do not have an explicit attack action. Instead, their "attack" is passive; they damage the player upon contact. Their `attackRange` is very small, meaning they only transition to the `Attacking` state when the player is nearly touching them, at which point contact damage is applied.
- **Ghosts:** These enemies are shooters that engage the player from a distance. When they enter the `Attacking` state, they trigger a shooting behavior. They can be configured to fire a burst of projectiles in a wide arc or in an oscillating, wave-like pattern. This allows for varied combat encounters depending on the specific instance of the Ghost enemy.

Figure 4: Sprites for the Slime (left) and Ghost (right) enemies.

1.3.3. Boss Agents

The three main bosses in the game are designed to be significant challenges that test the player's skill. Their behavior is controlled by agents trained using Reinforcement Learning (RL), allowing them to react to the player's actions and create dynamic, unscripted encounters.

Our primary approach is to model the boss agents as enhanced, hostile versions of the player character. At their core, these bosses possess the same set of abilities as the player's knight, but they are controlled by a trained RL agent. To make them more menacing and visually distinct, they may be scaled up in size, given a different color palette, and their attacks may inflict more damage.

The three bosses will represent a progression in difficulty. If time allows it, this will be achieved by using three different checkpoints of the training process of the agent, corresponding to Easy, Medium, and Hard difficulty levels. However, our ideal plan is to diversify the encounters. In this scenario, the first two bosses could be RL-powered versions of the standard Slime and Ghost enemies, featuring more complex and aggressive behaviors. This would make the final confrontation against the final boss (an evil replica of the player character) a unique and climactic event.

The technical specifics of the self-play training process, reward functions, and agent architecture for these bosses are detailed further in Section 3.3.

1.4. Scenario

We now present the story of our game. We tried to create a story that is coherent and consistent with the game mechanics.

Sir Cael of the Shattered Vow is trapped inside the Chronicle of Light, a living book that reshapes itself whenever he dies. Once a knight of the fallen Kingdom of Virel, he tried to destroy the godlike Chronicler who feeds on stories. As punishment, his essence was bound within the Chronicle, forced to relive his tale endlessly. Each death causes the world to be rewritten: dungeons shift, faces change, and the same myths return in new forms. Yet not everything is lost to the quill's rewriting.

Within the ever-changing dungeon lie three great beings that anchor the story: the Sentinel, the Herald, and the Chronicler. These are not just enemies, but keepers of the book's three chapters. When Cael defeats the Sentinel or the Herald, that chapter becomes fixed in ink. Its rooms, people, and memories stop changing, as if the Chronicle itself acknowledges those victories as canon. Death still sends him back to the beginning of the next unwritten chapter, but the world behind him stays intact, a part of the story that can no longer be erased.

Only the final chapter remains mutable. Beyond its shifting corridors waits the Chronicler, the source of the curse and perhaps Cael's reflection. If he defeats it, the book will end, and the world will finally stop rewriting

itself. But to do so, Cael must accept that every sealed chapter, with all its flaws and failures, is truly his – that perfection lies not in rewriting the story again, but in letting it end.

1.4.1. Spoilers!

The twist is that Sir Cael is the Chronicler himself.

Long ago, he wasn't a knight at all but a scholar obsessed with preserving the perfect version of history. When the Kingdom of Virel began to fall, he used forbidden magic to bind reality into the Chronicle of Light so the story could be rewritten until it was flawless. In doing so, he split himself in two: the writer (the Chronicler) and the written (Sir Cael), condemning himself to live and rewrite the same tale forever. Each death, each regeneration of the dungeon, is the Chronicler trying once more to "fix" his own mistakes — through Cael.

The first two bosses, the Sentinel and the Herald, are fragments of his guilt. The Sentinel represents his fear of loss, the Herald his denial of failure. When those chapters are completed, they are "fixed" because Cael has accepted those truths. The final battle, against the Chronicler, is a confrontation with himself: the part of him that refuses to let the story end. To win, Cael must stop fighting – not kill the Chronicler, but accept the story as it is. Only by choosing imperfection and closure does the book stop rewriting, freeing both halves of his soul and ending the curse.

For the game to be consistent with the lore, we suggest that when the Chronicler's health reaches its last sliver during the final battle, the player is prompted to "Strike the final blow", but a second option appears: "Lower your sword". If the player attacks, the loop restarts. If the player lowers the sword, Cael says, "No more rewrites". Thus, the story ends when he stopped trying to perfect it. However, this is an optional feature that we may or may not implement in-game depending on time constraints.

2. Related Work

PCG. Game design increasingly takes advantage of procedural content and AI-driven systems. Several roguelike-inspired action games – notably *The Binding of Isaac*, *Spelunky*, *Dead Cells* and *Hades* – use procedural algorithms such as binary space partitioning or cellular automata to assemble dungeons and item layouts at runtime, to have high replayability. Large

open-world games like *Minecraft* and *No Man's Sky* show how flexible PCG is [1].

Narrative generation. *Middle-earth: Shadow of Mordor/Shadow of War* introduced the "Nemesis" system, in which orc captains are procedurally generated and remember past encounters; they can return with scars or new ranks, creating unique personal stories. Earlier AI-driven titles such as *Façade* and *Versu* started dynamic dialogue and multiple endings, while Ubisoft's *Watch Dogs: Legion* showed how to write a dynamic script: the narrative team wrote twenty variations of every line and tied each version to a procedurally generated persona (e.g., a policeman says "much obliged", a young activist says "appreciate it, fam"). Simulation-heavy games like *Dwarf Fortress* and *RimWorld* are described as story generators because their complex systems of characters, needs and events produce emergent stories rather than fixed plots [2, 3].

Generative music. Dynamic soundtracks have been explored in both indie and AAA games. In *No Man's Sky*, audio director Paul Weir and the band 65daysofstatic created hundreds of musical fragments tagged by key, tempo and mood; the game's "Pulse" tool combines these elements in real time to generate soundscapes that react to what the player is doing. A decade earlier, Brian Eno composed a fully procedural score for *Spore*, making the game's music as generative as its evolving life forms [4].

Reinforcement learning agents. While many shipped games focus on hand-authored AI behaviours or simple difficulty scaling, there is a growing body of work exploring reinforcement learning (RL) agents trained for complex control tasks. AlphaStar, for example, used large-scale self-play to learn high-level strategies and unit micro-management in *StarCraft II*, producing agents that could compete with professional players [5]. Similarly, the Visual Doom AI Competitions challenged participants to train agents directly from raw pixels in *Doom*, where the strongest entries relied on deep RL methods to achieve competitive performance [6].

3. Proposal

The design of our game relies on a combination of procedural content generation (PCG) and adaptive AI systems to ensure that every run feels unique yet narratively coherent. The following subsections describe the main AI

I think having a pseudocode algorithm showing the full process at the start of the section would.

high-level

Maybe, this pseud.-code could refer to others pseudocodes (methods) helpp. BSP, MST, ...

components that create the dungeon structure, generate narrative, generate adaptive music, and control enemy and boss behaviours.

Time-permitting, we will also explore FSMs or other techniques for common enemies.

3.1. Procedural Dungeon Generation

3.1.1. BSP Map Layout Generation

In this section we will explain how the current PCG has been implemented.

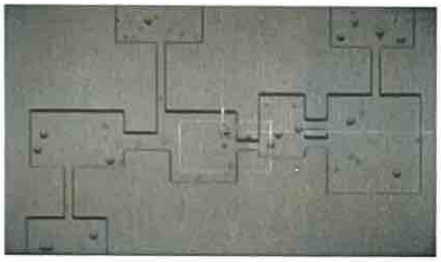


Figure 5: Example BSP geometry

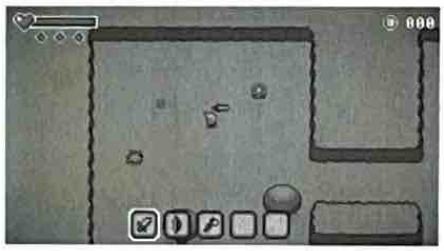


Figure 6: Example player's POV for a PCG dungeon

The dungeon layout is generated procedurally using a Binary Space Partitioning (BSP) algorithm implemented in the `BSPMSTDungeonGenerator` component (a Unity `MonoBehaviour`). Conceptually, the generator operates on a rectangular grid of size `mapWidth` × `mapHeight` and recursively splits this region into smaller "leaves". Each leaf may then host a single rectangular room, and (for now) rooms are connected via corridors to form a single, fully connected dungeon graph.

Leaf splitting. We start from a single root rectangle (optionally centred at the origin) and repeatedly split leaves until either the desired number of rooms is reached or no further valid splits are possible. Each leaf stores integer bounds (`RectInt`) and may be split horizontally or vertically:

- The split orientation is chosen adaptively: if the leaf is much wider than tall, we force a vertical split; if it is much taller than wide, we force a horizontal split; otherwise we flip a biased coin.

- A split is only allowed if both children would still be at least `minLeafSize` tiles in the split dimension, ensuring that later we can carve a room of at least `minRoomSize`.
- The splitting process stops when no leaf is large enough to be split further or when we have enough candidate leaves for the target `roomCount`.

This step yields a BSP tree of rectangular leaves that partition the dungeon space without overlaps or gaps.

Room carving. For each terminal leaf, we optionally instantiate a room whose size and position are sampled inside the leaf:

- We first compute candidate maximum room sizes `maxW` and `maxH` as `bounds.width - 2` and `bounds.height - 2`, and clamp them into the interval [`minRoomSize`, `maxRoomSize`] using `Mathf.Clamp`. If after clamping either `maxW` or `maxH` is still smaller than `minRoomSize`, the leaf cannot host a room and is skipped.
- Otherwise, we sample the actual room width and height uniformly as $w \sim \{\texttt{minRoomSize}, \ldots, \texttt{maxW}\}$ and $h \sim \{\texttt{minRoomSize}, \ldots, \texttt{maxH}\}$, ensuring a one-tile margin between the room and the leaf boundary.
- The room's bottom-left corner is then chosen uniformly at random within the leaf so that the entire room fits inside its bounds while respecting this margin.

Each accepted room is stored as a `Room` with an axis-aligned rectangle and an integer-valued centre (`Center`). The generator then iterates over all room rectangles and fills them with floor tiles via `PaintRect`, sampling from `groundTile` and optional variations to avoid visual repetition. All carved floor cells are also recorded in a `HashSet<Vector3Int>` (`floorCells`), which is later reused for wall placement and camera bounds.

Corridor graph and connectivity. Once we have a list of rooms, we connect them into a graph:

1. We construct a complete graph over room indices, assigning each edge a weight equal to the Euclidean distance between the corresponding room centres.

You could explain why rooms connections that result from BSP are not good enough (corridors follow the split structure) and then you use MST to generate more meaningful corridors, ...ble ....ble. real distances

2. We run Kruskal's algorithm with a Union-Find data structure to compute a Minimum Spanning Tree (MST). This guarantees that all rooms are connected while minimising total corridor length and avoiding cycles.
3. To avoid overly linear layouts, we optionally add up to `extraConnections` edges from the remaining non-MST edges (chosen in order of increasing weight), creating a few loops and alternative paths.

Walls, outside tiles, and camera bounds. After rooms and corridors are carved:

- We iterate over every floor cell and inspect its eight neighbours to decide which wall sprite to place (top, bottom, side, corner, or inner corner). This produces a consistent, fully tiled wall border that adapts to arbitrary shapes while using a small tileset.
- We fill a slightly larger rectangle around the dungeon with "outside" tiles (e.g., grass or void) to avoid having a "blank" outside of the camera

Room population. Once the dungeon's geometry is fixed, we populate each room:

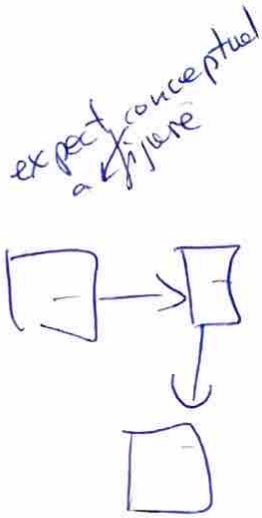
- Populators receive a lightweight `RoomData` struct containing the room index, its rectangle, and its centre, and can spawn arbitrary content (props, enemies, interactables) inside the room.
- To avoid collisions with walls, we sample spawn positions inside the room with a configurable margin, and place all objects under dedicated parent transforms (`Objects`, `Enemies`) for easier scene management.
- Because the RNG is seeded (`seed` field), entire layouts and population patterns are reproducible, which is useful for debugging and testing.

Please note that we must still work on room population and make it story-driven by perhaps designing a set of high-level story points, assigning them to rooms, and conditioning the generation accordingly. We also aim to test substituting corridors for "portals" between connected rooms (like a loading screen between rooms), which, combined with bigger room sizes, should ameliorate the issue we have of camera confines, and would make the game look better.

3.2. Generative AI Pipeline

To create dynamic, context-aware content for each procedurally generated room, we employ a multi-stage generative AI pipeline consisting of three complementary systems: vision-based room analysis, narrative generation, and adaptive music synthesis. These systems work together to transform the procedurally generated dungeon geometry into a coherent, immersive game where all elements – visual, narrative, and auditory – reinforce one another.

The pipeline operates sequentially: a vision model first analyzes a screenshot of the generated room to produce a semantic description of the environment (Section 3.2.2). This description is then fed to a large language model (LLM) that generates contextual narrative content including non-player characters (NPCs), quests, and lore entries (Section 3.2.3). Finally, the emotional tone extracted from the vision analysis drives procedural music generation to match the room's atmosphere (Section 3.2.4). This vision-driven approach should make it so that narrative and audio content semantically align with the visual characteristics of each room, rather than being generated independently.



3.2.1. System Architecture

We implement all three generative AI components as a unified Python backend service, architecturally decoupled from the Unity game engine. This design decision addresses several fundamental constraints inherent to deploying deep learning models within game engines.

Modern generative AI models – particularly transformer-based architectures for vision, language, and audio – rely on the Python ecosystem for inference. Critical dependencies include PyTorch for tensor operations, HuggingFace Transformers for model loading and execution, and specialized inference engines such as Ollama for optimized LLM serving. These libraries are not available in Unity's C# runtime environment, and reimplementing GPU-accelerated inference from scratch would be prohibitively complex. By extracting AI inference into a separate Python process, we gain access to state-of-the-art models with well-tested implementations.

The backend is structured as a FastAPI [7] web service exposing RESTful HTTP endpoints. Each generative component corresponds to a dedicated endpoint:

```
POST   /analyze/vision       Vision analysis of room screenshots
POST   /generate/narrative   Narrative generation (NPCs, quests, lore)
POST   /generate/music       Ambient music synthesis
```

GET	/cache/stats	Cache statistics and monitoring

The Unity client communicates with the backend via asynchronous HTTP requests, serializing parameters as JSON and deserializing responses into C# data structures.

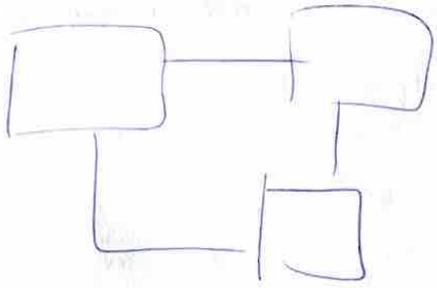
Table 1 summarizes the three AI models employed in our pipeline and their computational requirements. All models were selected based on their open-source availability and lightweightness, ensuring they could run efficiently on consumer-grade hardware (RTX 5070 Ti, 16GB VRAM). The vision and music models were chosen after evaluating multiple alternatives to find the best balance between output quality and inference speed for real-time game integration.

Table 1: AI models used in the generative pipeline

Component	Model	Parameters
Vision Analysis	moondream2 [8]	1.86B
Narrative	llama2 [9] (via Ollama)	7B
Music	MusicGen Medium [10]	1.5B

Figure 7 illustrates the complete system architecture showing the sequential pipeline from room generation through vision analysis, narrative generation, and music synthesis.

more extensive explanation please.



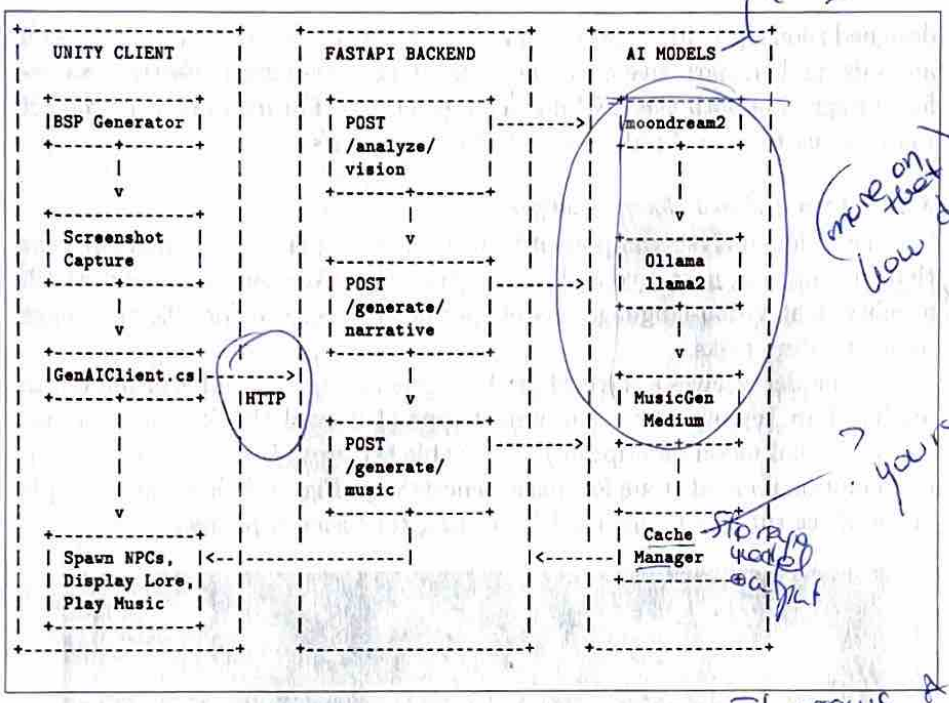


Figure 7: Generative AI pipeline architecture showing the sequential flow from Unity client through FastAPI backend to AI models, with caching layer.

To mitigate inference latency during iterative development and repeated playthroughs, we implement a two-layer caching architecture. The server-side cache indexes generated content by deterministic keys derived from input parameters, ensuring that identical requests retrieve previously generated content rather than invoking the models. The client-side cache stores content in Unity's persistent data path, allowing the game to reload content across application restarts without network communication.

All models run via local inference without external API dependencies. This eliminates runtime API costs, minimizes network latency, and ensures complete data privacy since no game content is transmitted to external servers.

For development and testing, the generative AI pipeline was validated using the three tutorial rooms from the base game (Section 1.2) rather than procedurally generated dungeons – that is left for Sprint 3. These manually

designed rooms provided stable, reproducible test cases to evaluate the vision analysis quality, narrative coherence, and music generation effectiveness before integrating with the BSP dungeon generator. Future work will connect the pipeline to procedurally generated room layouts.

3.2.2. Vision-Based Room Analysis

The vision analysis component transforms gameplay screenshots into text that inform both narrative and music generation. We employ moondream2, a lightweight vision-language model specifically designed for efficient image understanding tasks.

The model receives a screenshot of the generated room and is prompted to analyze four key aspects: environment type (1-2 word classification), atmosphere (visual mood description), observable features (5-7 specific elements), and emotional mood (tone for music generation). Figure 8 shows an example of the three tutorial rooms used for testing the vision pipeline.

Figure 8: Three tutorial rooms from the base game used for testing the vision analysis pipeline.

The prompt structure enforces a consistent JSON output format to ensure reliable parsing and downstream integration:

```
Analyze this 2D roguelike RPG game screenshot (top-down pixel art).

Provide a detailed description including:
- environment_type: what kind of area this is (1-2 words)
- atmosphere: describe the overall feeling and visual mood (2-3 sentences)
- features: list 5-7 specific things you see (terrain, structures, objects, enemies)
- mood: describe the emotional tone for ambient music generation (1-2 sentences)

Respond ONLY in this JSON format:
{"environment_type": "...", "atmosphere": "...", "features": [...], "mood": "..."}
```

The model's output is validated against a Pydantic schema to ensure type safety and correct field structure. An actual generation example for one of the tutorial rooms:

```
{
  "environment_type": "Rural/Forest",
  "atmosphere": "Fantastic, peaceful, serene",
  "features": [
    "Pond",
    "Trees",
    "Flowers",
    "Small structures",
    "Path"
  ],
  "mood": "Calm, peaceful, mysterious"
}
```

This structured output format is temporary and was designed for testing purposes. The schema will likely evolve before final project delivery to better align with the narrative generation requirements and provide more granular control over music generation parameters.

3.2.3. Narrative Generation

The narrative generation component uses the llama2 language model via Ollama to create contextual story content for each room. The system generates three interconnected elements: environmental descriptions, non-player characters (NPCs) with dialogue, quests with objectives, and lore entries that enrich the game world.

The LLM receives a structured prompt that specifies the room's position in the overall dungeon (e.g., "room 3 of 9"), the narrative phase (beginning/middle/end), and optionally, context from previously generated rooms to maintain story continuity. The prompt enforces strict JSON output formatting to ensure reliable parsing:

```
Generate a story segment for room 3 of 9.
This is the middle of the adventure.
CRITICAL INSTRUCTIONS:
1. Output ONLY valid JSON - no explanations, no markdown, no extra text
2. Use proper JSON syntax with double quotes for all strings
3. Ensure all commas are correctly placed
4. The dialogue field MUST be an array with EXACTLY 3 strings
Required JSON structure:
{
  "environment": "atmospheric description here",
  "npc": {
    "name": "character name",
    "dialogue": ["first line", "second line", "third line"]
  },
  "quest": {
    "objective": "quest objective description",
    "type": "DefeatEnemies",
    "count": 3
  },
  "lore": {
    "title": "lore title",
    "content": "lore content and backstory"
  }
}
```

To improve generation reliability, the system implements a retry mechanism with seed perturbation. If the LLM produces malformed JSON (a common issue with unconstrained generation), the request is retried up to three times with different random seeds. As shown in the test output above, the first attempt may fail with JSON parsing errors, but subsequent retries with perturbed seeds typically succeed, achieving reasonable reliability without requiring grammar-constrained decoding or model fine-tuning.

An example generated narrative from actual model output:

```
{
  "environment": "A grassy clearing with scattered ponds and small stone
                  structures. The area has a peaceful yet mysterious
                  atmosphere.",
  "npc": {
    "name": "Wandering Spirit",
    "dialogue": [
      "You have entered the dungeon depths brave one.",
      "Beware of the Ghosts and Slimes that lurk in these rooms.",
      "They guard secrets that have been lost for ages."
    ]
  },
  "quest": {
    "objective": "Clear the room of hostile Ghosts and Slimes",
    "type": "DefeatEnemies",
    "count": 3
  },
  "lore": {
    "title": "The Haunted Depths",
    "content": "These dungeons were once peaceful halls. Now Ghosts drift
               through the corridors while Slimes ooze from the shadows."
  }
}
```

This structured format is experimental and designed for testing the narrative pipeline. The final implementation may evolve to include different content types beyond NPCs, quests, and lore, depending on gameplay requirements and integration with the procedurally generated dungeon layouts. It is also not yet aligned with the story described in Section 1.4, but will be so for Sprint 3.

3.2.4. Music Generation

The music generation component uses MusicGen Medium [10], selected for its simplicity and ability to generate loopable background music from text prompts. Initial experiments explored various generative music approaches, but many produced outputs that either did not sound like coherent background music or lacked musical structure. MusicGen proved to be the most practical solution for procedural ambient music generation.

However, testing revealed that MusicGen's instruction-following capability is quite limited as well as other models. The model responds best to short keyword-based prompts rather than detailed descriptions. Complex or lengthy prompts do not significantly improve output quality and, from our experiments, even degrade coherence. The system constructs prompts from minimal keywords describing the desired mood or setting:

```
def _make_prompt(self, description: str) -> str:
    return f"Atmospheric ambient music for {description},
            slow tempo, immersive, loopable background music"
```

Key generation parameters used in testing:

```
duration = 30.0                 # seconds of audio
guidance_scale = 3.5            # prompt adherence strength
sample_rate = 32000             # Hz
do_sample = True                # enable sampling for variety
```

Example generated music tracks for different game contexts [1]:

- **tavern.mp3**: Prompt "tavern" - peaceful, ambient music for safe areas
- **mysterious.mp3**: Prompt "mysterious" - atmospheric exploration music
- **boss_tense.mp3**: Prompt "boss tense" - intense music for boss encounters

These examples show MusicGen's ability to produce thematically appropriate ambient music from minimal keyword prompts. The generated tracks are loopable and suitable for background music during gameplay, though the limited instruction-following means fine-grained musical control remains challenging even for larger models and a very active research domain.

Future Integration and Potential Extensions. The generative AI pipeline components (vision-based room analysis, narrative generation, and music generation) have been tested individually and show promising potential for procedural content generation. However, for Sprint 3 substantial work remains

[1] To listen to these, please open with Adobe Acroabt; in case it is not possible, we also attached the songs in the delivery.

to integrate these systems with the procedural dungeon generation once the BSP room layouts are finalized. Integration tasks include refining prompts based on actual dungeon characteristics, implementing in-game logic for content delivery, and optimizing the generation flow for runtime performance.

Additionally, we are exploring the possibility of incorporating voice generation models to create a narrative experience. For instance, generated story segments could be converted to speech using text-to-speech models, creating a narrator that reads the lore and dialogue to players. This remains an experimental direction that has not yet been tested, and feasibility will depend on available time and technical constraints.

3.3. Reinforcement Learning for Boss Behavior

Boss behaviours (Sentinel, Herald, and Chronicler) are trained through reinforcement learning (RL) to produce adaptive and challenging encounters. As mentioned in Section 1.3.3, these bosses are currently modelled as replicas of the player character, possessing similar movement and attack capabilities. If time permits, we may diversify the first two bosses to be RL-powered variants of the Slime and Ghost enemies, but for now we focus on the RL training of the player-like boss.

The agents in control of these bosses are designed to mimic the player's capabilities while exhibiting complex human-like strategies. The setup for their training takes place in a custom Arena environment that will allow the agents to learn through self-play. That is, the agent will play against a copy of itself, allowing it to explore a wide range of strategies and counter-strategies. Figure 9 illustrates the Arena training setup.

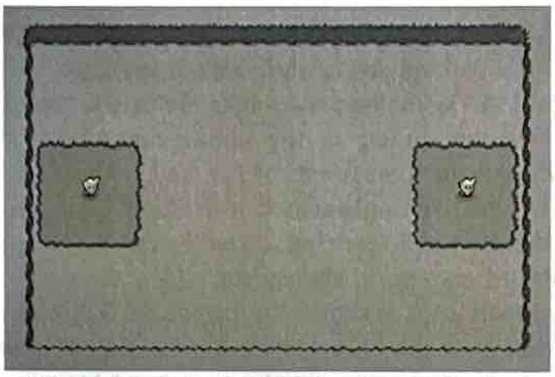


Figure 9: Screenshot of the Arena Unity Scene used for RL training, where the boss agent learns through self-play against a copy of itself.

We consider the possibility of creating a scene with multiple copies of the Arena environment running in parallel, to speed up data collection during training, although this has not yet been implemented.

The RL agents are implemented using the Unity ML-Agents Toolkit, which provides a framework for training intelligent agents in Unity environments. Therefore, the `AgentController` is defined as a subclass of the `Agent` class provided by ML-Agents.

In this class, we define the observation space and action spaces for the boss agents:

- The observation space includes 6 continuous values:
  - The agent's own position (x, y) coordinates.
  - The agent's own health.
  - The opponent's position (x, y) coordinates.
  - The opponent's health.
- The action space consists of 3 discrete action branches:
  - 5 movement actions: stand still, move forward, backward, left, or right.
  - 4 attack actions: do nothing, use sword, bow, or magic staff.
  - 2 dashing actions: no dash, or dash.

What about the reward?

In order for these systems to work under the use of ML-Agents in a self-play scenario (with two agents in the same scene), we had to implement custom `AgentHealth`, `AgentWeapons`, `AgentStamina`, `AgentDamageSource` components, which are similar to the player character's components but adapted to work with the `AgentController` and without singletons.

In addition to this, we implemented a `TrainingManager` component to handle the initialization and resetting of the Arena scene after each episode, as well as the reward system for the agents.

The reward function is designed to encourage aggressive yet strategic behavior, rewarding successful hits on the opponent while penalizing damage taken. The agent also receives a significant reward for winning the match and a penalty for losing. Additionally, small time penalties are applied every timestep to encourage the agent to conclude matches efficiently.

For the training process, we use the Proximal Policy Optimization (PPO) algorithm, for its versatility and effectiveness in control tasks, with the default hyperparameters provided by ML-Agents. If necessary, we will tune these hyperparameters based on initial training results to improve learning stability and performance.

Once all of the components are defined and set-up, the training loop is executed using the `mlagents-learn` command-line tool, which interfaces with the Unity environment to collect experience data and update the agent's policy. For efficiency, we run the training with graphics disabled, allowing for faster simulation speeds. Note that this does not truly disable rendering, but instead allows the training to run unsynchronized from the rendering, speeding up the process.

As a planned extension, the Sentinel and Herald bosses can be implemented as enhanced versions of the Slime and Ghost enemies. This would require designing new agent controllers with action spaces adapted to their specific abilities. Unlike the self-play approach used for the final boss, these agents would be trained by competing against the pre-trained, player-like agent in the Arena environment. This methodology ensures that they learn to counter the player's strategies effectively, providing a varied and escalating challenge.

Next Sprint, Software architecture of the entire project. modules and relation between them.

I interpret that this is not done yet. (In next sprint results and we're into our the decisions taken)

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, Springer, 2016.

[2] A. Liapis, G. N. Yannakakis, J. Togelius, Sentient sketchbook: Computer-aided game level authoring, in: Proceedings of the 8th International Conference on the Foundations of Digital Games.

[3] A. Buongiorno, P. Gervas, R. Hervás, Pangea: Procedural narrative generation via large language models, arXiv preprint arXiv:2404.19721 (2024).

[4] R. Castellon, F. Foscarin, X. Serra, Generative models for symbolic music: A survey, Transactions of the International Society for Music Information Retrieval (2023).

[5] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. Paine, K. Gundogdu, Y. Wang, A. Lerer, R. Nelakanti, B. Marthi, D. Silver, J. Schrittwieser, D. Hassabis, C. Apps, K. Kavukcuoglu, Grandmaster level in starcraft ii using multi-agent reinforcement learning, Nature 575 (2019) 350–354.

[6] M. Wydmuch, M. Kempka, W. Jaskowski, Vizdoom competitions: Playing doom from pixels, IEEE Transactions on Games 11 (2019) 248–259.

[7] S. Ramírez, Fastapi: Modern, fast (high-performance) web framework for building apis, `https://fastapi.tiangolo.com`, 2018. Accessed: 2024-11-17.

[8] VikParuchuri, moondream2: A tiny vision language model, `https://github.com/vikhyat/moondream`, 2024. Accessed: 2024-11-17.

[9] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al., Llama 2: Open foundation and fine-tuned chat models, arXiv preprint arXiv:2307.09288 (2023).

[10] J. Copet, F. Kreuk, I. Gat, T. Remez, D. Kant, G. Synnaeve, Y. Adi, A. Défossez, Simple and controllable music generation, arXiv preprint arXiv:2306.05284 (2023).