Developing an Engaging Metroidvania Game Through Procedural Content Generation

Jasmine Micallef

Supervisor: Dr Keith Bugeja
Co-Supervisor: Dr Sandro Spina

May 2025

Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours) Computing Science and Statistics and Operations Research.

L-Università ta' Malta
Faculty of Information & Communication Technology

Abstract

Procedural content generation (PCG) is widely used in modern game development to enhance replayability and reduce development overhead. However, applying PCG to the Metroidvania genre poses a unique challenge: how can algorithmically generated content preserve the handcrafted feel, spatial logic, and sense of progression that define the genre? This project explores a modular PCG system designed to generate individual Metroidvania rooms with structural coherence, traversal complexity, and gameplay diversity.

Using the Godot Engine, a generation pipeline was implemented based on configurable primitives, zone segmentation, anchor-based pathfinding, and player-aware movement constraints. A custom interestingness scoring function was developed to evaluate each room based on factors such as anchor coverage, goal distribution, difficulty, and the variety of movement and ability-based interactions.

The system was evaluated both quantitatively, via a Monte Carlo simulation analysing over 1,200 generated rooms, and qualitatively, through a visual assessment of selected room screenshots. Results from statistical modelling revealed that room width, door count, and difficulty level significantly influenced interestingness scores. The findings demonstrate that meaningful structure can emerge from procedural methods when guided by a well-defined evaluation framework, offering new avenues for scalable and expressive Metroidvania design.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr Keith Bugeja, for his guidance, patience, and invaluable feedback throughout this project. His support gave me the confidence to explore ambitious ideas and turn them into something meaningful.

I would also like to thank Dr Sandro Spina for his insight and helpful suggestions, which shaped the direction of this work from the very beginning.

To my family, especially my mum — thank you for your unwavering encouragement, love, and for always believing in me, even when I doubted myself. Your support means everything.

A special thank you to my friends who cheered me on, tested my game, or simply kept me sane during late-night debugging sessions. You made this journey not just bearable, but memorable.

Lastly, I'm grateful to the Godot developer community and open-source contributors whose work laid the foundation for many of the tools used in this project.

Contents

List of Figures

1 Introduction

Procedural content generation (PCG) is widely used in game development to automatically create levels, environments, or item placements. It reduces manual workload and increases variability, but raises challenges when applied to Metroidvanias that depend on curated structure and spatial progression.

Metroidvania games are renowned for carefully gated progression systems and tightly woven world designs. Their reliance on spatial memory, backtracking, and gradual unlocks poses unique challenges for procedural generation—challenges this project aims to explore and address.

This project was motivated by discourse surrounding the limitations of procedural generation in Metroidvania games. A widely discussed thread on Reddit's r/metroidvania community revealed scepticism about using PCG. Enthusiasts described how PCG often feels disconnected, lacks meaningful progression, and fails to recreate the "aha" moments typical of handcrafted maps. Titles like Chasm, Sundered, and A Robot Named Fight were cited as examples where procedural generation undermined the intentionality that defines the Metroidvania experience [1].

Some players suggested hybrid approaches—retaining fixed structures with limited procedural variation—while others argued that PCG could never replicate the interconnectedness and emotional rhythm of a well-designed Metroidvania map. These community discussions were a key source of inspiration for this project.

Rather than dismiss the criticisms surrounding PCG in Metroidvanias, this project aims to investigate how procedural generation can be guided by designer-driven principles. The goal is to create a system that procedurally generates rooms with varied layouts and traversal challenges, while ensuring the resulting spaces support coherent movement, exploration, and progression. To do this, a modular architecture was implemented using the Godot engine, with an emphasis on defining new concepts that enable interesting and navigable level designs.

The system does not aim to replicate full Metroidvania maps with item gating and global progression. Instead, it focuses on generating individual rooms that feel complete and intentional. These rooms are structured using spatial segmentation and difficulty scaling, and populated with interactable elements. Special attention is paid to maintaining logical flow and player mobility through the use of anchor-based pathfinding and primitive compatibility rules.

This dissertation presents the literature, rationale, technical architecture, and evaluation of the system, contributing to ongoing discussions in the PCG and Metroidvania communities by offering a concrete step toward bridging the gap between procedural randomness and curated design.

2 Background

This chapter introduces the core concepts that underpin the technical and design decisions made throughout this project. It provides an overview of Metroidvania games, procedural content generation, common game engine architectures, and the algorithms that form the basis of level generation and pathfinding.

2.1 The Metroidvania Genre

The Metroidvania genre, a product of *Metroid* and *Castlevania*, refers to a style of 2D platformer that emphasises non-linear exploration, gradual player progression through ability acquisition, and tightly interconnected worlds. Key characteristics include a large, contiguous world map, backtracking to previously inaccessible areas after gaining new abilities, and a strong emphasis on spatial memory and traversal.

Games such as *Super Metroid*, *Hollow Knight*, and *Ori and the Blind Forest* are quintessential examples of the genre. These games are known for their carefully crafted level design, which balances openness with structure and rewards curiosity with meaningful shortcuts or secrets. As such, their world layouts are typically handcrafted to maintain pacing, challenge balance, and environmental storytelling.

2.2 Procedural Content Generation in Games

Procedural content generation (PCG) refers to the algorithmic creation of game content with limited or indirect human input. This may include levels, textures, quests, enemy placement, and more. PCG has become an increasingly popular technique in modern game development due to its ability to reduce development costs, increase replayability, and introduce variability into game worlds.

However, PCG also comes with significant challenges. Ensuring that generated content remains coherent, engaging, and balanced often requires strict rules or post-processing validation. In genres like Metroidvania, where progression is gated and spatial logic is vital, poorly controlled PCG can result in maps that feel disjointed, directionless, or unfair.

2.3 Game Engines and Development Tools

Game engines provide developers with essential tools for managing graphics, physics, input, and scene organisation. Popular engines for 2D development include Unity, GameMaker Studio 2, and Godot, each offering distinct workflows and capabilities.

2.4 Binary Space Partitioning (BSP)

Binary Space Partitioning (BSP) is a technique used to recursively divide a two-dimensional or three-dimensional space into smaller, non-overlapping regions. This is typically done by selecting an axis (horizontal or vertical) and splitting the space with a straight line or plane. Each resulting subspace is then further subdivided using the same method, creating a binary tree of regions.

BSP is widely used in game development for purposes such as map generation, spatial partitioning, visibility determination, and collision optimisation. The resulting structure allows for efficient traversal, region querying, and organisation of game elements.

2.5 Graph-Based Representations in Game Design

Graphs are a fundamental data structure in game design, especially for modelling progression systems and spatial layouts. In a graph, nodes represent entities such as rooms or zones, and edges represent valid transitions or connections between them.

For Metroidvania games, graphs are commonly used to model item-gated progression. Edges may only be traversed if the player possesses the necessary ability or key, enabling non-linear but constrained exploration.

2.6 Pathfinding Algorithms

Pathfinding is the process of determining a navigable path between two points. It is central to AI navigation, level design, and—in this project—collectible placement. Several common algorithms exist:

- **Breadth-First Search (BFS)** is an uninformed search algorithm that explores a graph level by level. Starting from a source node, it visits all directly connected neighbours before moving to their neighbours, effectively expanding outward in layers. BFS uses a queue data structure to maintain the order of exploration. It guarantees the shortest path in terms of number of edges (assuming all edge costs are equal) and is complete, meaning it will find a solution if one exists. However, it can be memory-intensive and inefficient in large graphs due to its exhaustive nature.

- **A* Search** is an informed search algorithm that combines actual cost from the start node with an estimated cost to the goal, known as the heuristic. It uses a priority queue to explore the most promising path first, balancing cost and heuristic value through the formula $f(n) = g(n) + h(n)$, where $g(n)$ is the cost so far and $h(n)$ is the estimated cost to the goal. A* is both complete and optimal, provided that the heuristic used never overestimates the true cost.
- **Greedy Best-First Search** selects the next node to explore based solely on the heuristic function $h(n)$, aiming to get closer to the goal as quickly as possible. It does not consider the path cost already incurred. While this often results in fast performance, especially in open spaces, Greedy Search does not guarantee the shortest or even a valid path in some cases. It may also get trapped in loops or dead ends if the heuristic is misleading or poorly defined.

2.7 Monte Carlo Simulation in Evaluation

Monte Carlo simulation is a method for analysing systems with inherent randomness by repeatedly sampling outcomes under varying conditions. In game development, it is often used to test algorithm performance and robustness across a wide range of inputs.

In this project, Monte Carlo simulation was used to evaluate how room parameters—such as size, difficulty, and number of doors—affected the system's output. By generating many rooms per configuration, it enabled a reliable statistical analysis of trends, helping to validate the generator's consistency and the effectiveness of the interestingness metric.

2.8 Linear Regression Modelling

Linear regression is a statistical technique for modelling the relationship between a dependent variable and one or more independent variables. It produces coefficients that quantify each variable's effect, along with significance values that assess confidence in those effects.

This project applied linear regression to determine how factors like difficulty, door count, and room dimensions influenced the interestingness score. The model provided a numeric estimate of each parameter's contribution and used R^2 to measure overall explanatory power, offering insights into which features most impacted room quality.

3 Literature Review

3.1 Academic Frameworks and Technical Approaches

Procedural generation in Metroidvania games remains a complex and evolving challenge, as developers strive to maintain the sense of progression, discovery, and spatial intentionality that defines the genre. Stalnaker (2020) approaches this challenge through the use of directed acyclic graphs (DAGs), graph grammars, and constraint satisfaction. His system generates valid 2D Metroidvania layouts by building abstract representations of room connectivity, where edges are locked behind abilities or items that simulate player progression. Crucially, the algorithm guarantees solvability by tracking reachable nodes and accessible items at each construction stage, avoiding dead ends and unreachable zones. The result is a visually coherent and logically consistent prototype developed in Python using Pygame, capable of supporting branching, non-linear maps with clear gating logic. While this current project employs a tile-based, physics-aware placement system rather than high-level graph grammars, Stalnaker's work provides a conceptual foundation for understanding progression logic in procedural map construction [2].

In a similar vein, Xu and Morris (2023) explore the generation and validation of Metroidvania-style layouts through a hybrid system combining Answer Set Programming (ASP) and Perlin noise. ASP provides logical guarantees by solving for valid configurations where keys always precede the gates they unlock, thereby eliminating the risk of soft-locking—a scenario where the player becomes permanently stuck due to poor item placement. To counter ASP's computational limitations, their framework applies a divide-and-conquer strategy that resolves individual rooms iteratively. The comparative analysis between ASP and Perlin noise illustrates the strength of logic-driven systems, as ASP consistently produces traversable levels, while Perlin-generated ones require post-processing to assess reachability. The relevance of this hybrid approach lies in its ability to balance creative freedom with structural validity, aligning closely with the goals of this project [3].

Beyond procedural layout generation, Oliveira et al. (2020) introduce a comprehensive Metroidvania development framework focused on modularity and runtime efficiency. Built in Unity, their architecture features a component-based system that decouples entity behaviour from core game logic through dynamically assigned modules. The system handles complex features such as dynamic level loading, co-op character switching, and memory-efficient scene management. One notable aspect is their use of a pathfinding system based on line tracing, enabling real-time navigation through procedurally generated environments. While the focus here is on spatial content generation rather than full game architecture, Oliveira et al.'s system demonstrates the scale of interconnected systems required to support the Metroidvania experience and provides insights into managing complexity across interconnected mechanics [4].

Machine learning approaches to procedural design are showcased by Gutiérrez Rodríguez, Cotta, and Fernández Leiva (2018), who combine evolutionary algorithms (EAs) with artificial neural networks (ANNs) to simulate the iterative design behaviour of a human level designer. Designers train the ANN using labelled motifs—substructures in level design considered good or bad—allowing the ANN to score and guide the EA's generation process. This feedback loop results in paths that reflect the designer's aesthetic goals and gameplay preferences. Although this project does not involve neural networks, the approach highlights the importance of designer intent and pattern recognition in procedural systems. The motif-based learning echoes this project's use of structured primitives and difficulty scoring to encode qualitative assessments into the generation process [5].

Lastly, Nitsche et al. (2006) examine the philosophical implications of procedural content by focusing on player agency in shaping the game world. Their prototype, Charbitat, generates 3D game environments based on Taoist elemental variables, which evolve dynamically in response to player behaviour. As the player moves through the environment, their actions influence the theme and layout of newly generated areas. This co-creative approach challenges the view of procedural generation as static or purely random, proposing instead that meaningful spaces emerge through the player's influence. While Charbitat targets 3D exploration and symbolic feedback rather than structured level progression, its emphasis on coherence, feedback loops, and spatial meaning aligns with this project's goal of making procedurally generated rooms feel curated rather than random [6].

3.2 Evaluating Player Experience

Evaluating the player experience in games has evolved significantly over the past two decades. What began as a simple extension of traditional Human-Computer Interaction (HCI) practices has grown into a specialised field with its own metrics, heuristics, and philosophical debates. Across this landscape, researchers and designers alike have sought ways to understand not just usability, but the emotional, cognitive, and behavioural dimensions of play. This section explores key contributions in the field, focusing on how they inform the development and evaluation of procedurally generated spaces.

The foundational approach is rooted in adapted HCI methods. As discussed in

Evaluating User Experience in Games: Concepts and Methods, early game usability studies applied familiar techniques like questionnaires, think-aloud protocols, and task completion metrics to identify friction points in gameplay. These adaptations laid the groundwork for more agile evaluation cycles, such as the Rapid Iterative Testing and Evaluation (RITE) method developed by Microsoft Game Studios, which advocated for real-time feedback loops between testers and designers. While these approaches helped refine controls and interfaces, they often fell short of capturing the full richness of gameplay experiences [7].

To address this, researchers began developing game-specific heuristics. The GameFlow framework, for instance, incorporated psychological theories of enjoyment and flow to assess whether a game was not just usable, but engaging. Other models, such as the PIFF (Presence-Involvement-Flow Framework) and CEGE (Core Elements of Game Experience), attempted to quantify aspects of immersion and emotional engagement by linking player feedback to measurable game elements. These frameworks signalled a shift away from viewing players as users, and toward understanding them as emotional agents navigating complex, dynamic systems [7].

This shift is especially important in the context of procedural generation, where traditional usability metrics alone are insufficient. For example, Blašković et al. (2023) proposed a structural model for evaluating platformer games that accounted not only for interface quality and audiovisual feedback, but also for player enjoyment, engagement, and behavioural intention. Their study on *Stranded Away*, a 2D puzzle-combat platformer, used Partial Least Squares Structural Equation Modeling (PLS-SEM) to validate hypotheses about what drives continued play. The results confirmed that mechanics, enjoyment, and engagement were key predictors of a player's willingness to keep playing. This model, while not directly applicable to this project, reinforces the value of treating engagement as a multifaceted construct—something particularly important when designing spaces that feel handcrafted despite being generated [8].

Other voices have challenged the vocabulary of player experience itself. Crawford and Brock (2024), for instance, question the ubiquity of the word "fun" in game design discourse. In their sociological unpacking, they argue that "fun" is often deployed uncritically, masking a range of emotional experiences such as frustration, curiosity, and even boredom—all of which can be meaningful. They propose that designers expand their evaluative language to include not only pleasure and engagement, but also challenge, discomfort, and emotional tension. This is especially relevant for procedurally generated systems, where ensuring player interest may require embracing non-traditional notions of satisfaction [9].

Taken together, these works present a layered understanding of how player experience can be evaluated. From structural models and emotional heuristics to sociocultural critiques, they suggest that effective game evaluation must consider both the psychological and systemic dimensions of play. While this project does not implement a formal UX testing framework, the influence of these perspectives is evident in its design priorities—particularly the use of difficulty scoring, spatial diversity, and intentional pathing to support player engagement within procedurally assembled rooms.

4 Design and Implementation

This chapter describes the full implementation of the procedural generation system developed for this project. The system was designed to generate room layouts for a platformer-style game, using a modular architecture built around the concepts of primitives, atoms, anchors, and procedural pathfinding.

While the original goal was to generate full Metroidvania-style maps with structured progression and ability gating, the final system focuses more specifically on generating playable, interesting, and coherent rooms with varied layouts and movement challenges. Key design decisions were guided by the aim of producing rooms that feel 'hand-crafted,' even when assembled procedurally.

4.1 Game Engine

Implementation was done using the Godot game engine (version 4.4.1), and the scripting was carried out in C#.

Several game engines were considered at the start of the project, including Godot 4, GameMaker Studio 2, and Unity 6. Each engine was briefly tested against core criteria such as tilemap integration, built-in physics support, ease of implementing character controllers, and overall IDE usability. Godot 4 was ultimately chosen due to its accessible node-based architecture, built-in tilemap system, and streamlined scripting workflow. It also offered intuitive support for physics, collisions, and player movement—features essential for iteratively testing room layouts and anchor-based navigation. While Unity offered more extensibility and GameMaker had simpler 2D workflows, Godot struck the best balance between flexibility and ease of use for this particular system.

4.2 System Architecture Overview

The system is structured into several core modules, each responsible for a distinct part of the generation pipeline. These include:

- **Room Generation System** — Responsible for dividing the room into zones, determining overall layout structure, and initiating primitive placement.
- **Primitive and Atom System** — Primitives (e.g. ladders, pits, enemies) are composed of atoms (e.g. individual tiles or entities), each of which manages its own placement validation and behavior.

- **Anchor System and Pathfinding** — Anchors represent potential connection points between primitives. They are used to construct an anchor graph, which enables pathfinding across the room.
- **Environmental and Hazard Systems** — Handles the placement of interactable or animated elements like water, fish, floor blades, and slugs.
- **Path Building System** — Responsible for ensuring that meaningful paths can be constructed between critical locations in the room, such as from one door to another. This includes generating ideal anchor points, evaluating path difficulty, and determining where keys should be placed to align with the intended player journey.
- **Player Feedback and UI** — Integrates player interaction with generated elements, including health loss on contact with hazards and heart UI updates.

This modular approach was chosen to maximize code reusability and isolate logic specific to each primitive or system. Alternatives such as a monolithic generator script or rule-based tilemap generation were considered early in development but discarded due to their lack of flexibility and difficulty handling complex behaviours like vertical traversal or gated key access.

Each module communicates through well-defined methods and relies on the concept of primitives “registering” themselves into a room during generation. This allows for controlled sequencing of generation stages, such as placing core floor tiles before adding hazards or keys.

The full procedural generation pipeline is outlined in Algorithm 1. Each major step in the pipeline corresponds to a dedicated subsection in this chapter.

4.3 Room Structure and Zone Generation

Rooms are defined as 2D rectangular spaces measured in tile units, and the first stage of generation involves dividing this space into smaller rectangular regions known as *zones*. Zones provide structural organisation to the room, allowing local logic to be applied to primitive placement, reachability, and movement pathways.

4.3.1 Zone Representation

Each `Zone` is defined by four parameters: its top-left `(X, Y)` position and its `Width` and `Height`. Zones also contain a boolean `isReachable` flag, which tracks whether the player can logically reach the zone through placed movement primitives (e.g. ladders, slopes, springs).

Algorithm 1 Pseudocode for the Room Generation Pipeline

```
function GenerateRoom(roomWidth, roomHeight)
    zones ← GenerateZonesUsingBSP(roomWidth, roomHeight)
    validZones ← ValidateZoneConnectivity(zones)
    if not validZones then
        return GenerateRoom(roomWidth, roomHeight)
    end if
    for zone in validZones do
        PlaceFloor(zone)
    end for
    ConnectZonesVertically(validZones)
    PlaceEnvironmentals(validZones)
    PlaceHazards(validZones)
    PlaceDoors(validZones)
    anchors ← GenerateAnchorPrimitivesFromMap
    anchorGraph ← BuildAnchorGraph(anchors)
    for targetDoor in doors[1:] do
        keyCandidates ← GenerateKeyCandidates(anchorGraph, targetDoor)
        bestPath ← FindMostDifficultValidPath(doors[0], targetDoor, keyCandidates,
anchorGraph)
        PlaceKeyAlongPath(bestPath)
    end for
end function
```

4.3.2 Zone Creation Using BSP

Zones are generated using a Binary Space Partitioning (BSP) algorithm. A `BSPNode` recursively splits the room horizontally or vertically, with the direction based on the shape of the region being split. Each split uses a random ratio within specified bounds to avoid overly narrow or wide partitions. Splitting continues until a calculated maximum depth is reached, based on the room dimensions and minimum zone sizes.

Leaf nodes in the BSP tree represent the final zones. Each one becomes a candidate for floor and primitive placement. If a cluster of zones cannot form a horizontally connected chain across the room (e.g. due to alignment or overlap issues), the entire set is discarded and generation is retried.

4.3.3 Zone Validation and Floor Placement

After generation, zones are checked for horizontal reachability using the `CheckHorizontallyReachableZones` method. This ensures that zones either touch the room boundary or are adjacent to other zones at similar vertical positions. Each valid zone is then assigned a `Floor` primitive, which acts as the base structure for further placement.

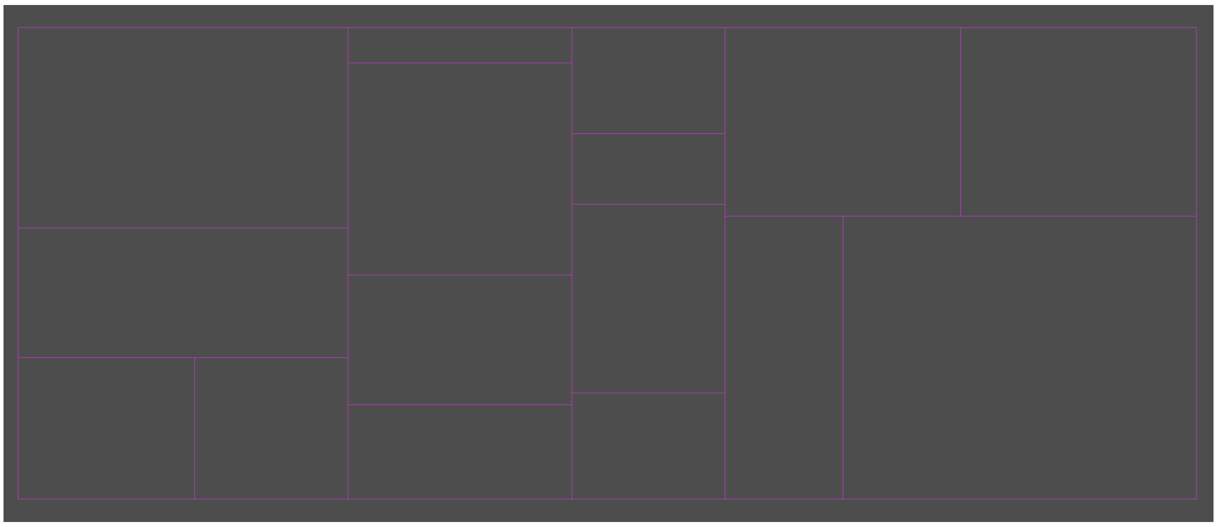
Figure 4.1 Example of Zone Splitting using BSP.

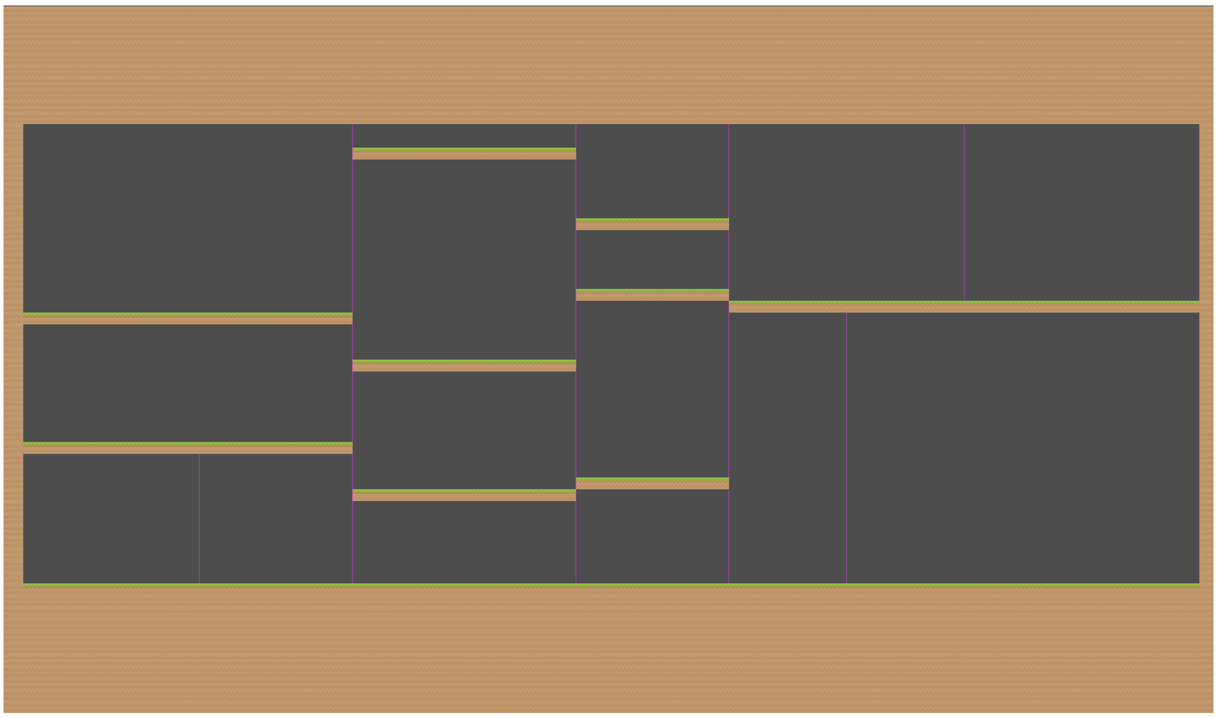
Figure 4.2 Zones with Floors Generated.

4.3.4 Vertical Connectivity Between Zones

Zones that are vertically adjacent but not yet reachable are connected using vertical movement modifiers such as `Ladders`, `Slopes`, or `Springs`. This logic is handled in the `ConnectZonesVertically` method, which detects vertically stacked zones with sufficient horizontal overlap, calculates the vertical gap between floors, and selects the most suitable primitive based on available space and traversal feasibility. Before placing a vertical connector, the system verifies airspace clearance using `canPlaceSpring` or `canPlaceSlope`, with ladders used as a fallback when others are not viable. This guarantees logical vertical traversal without relying on unsupported player abilities.

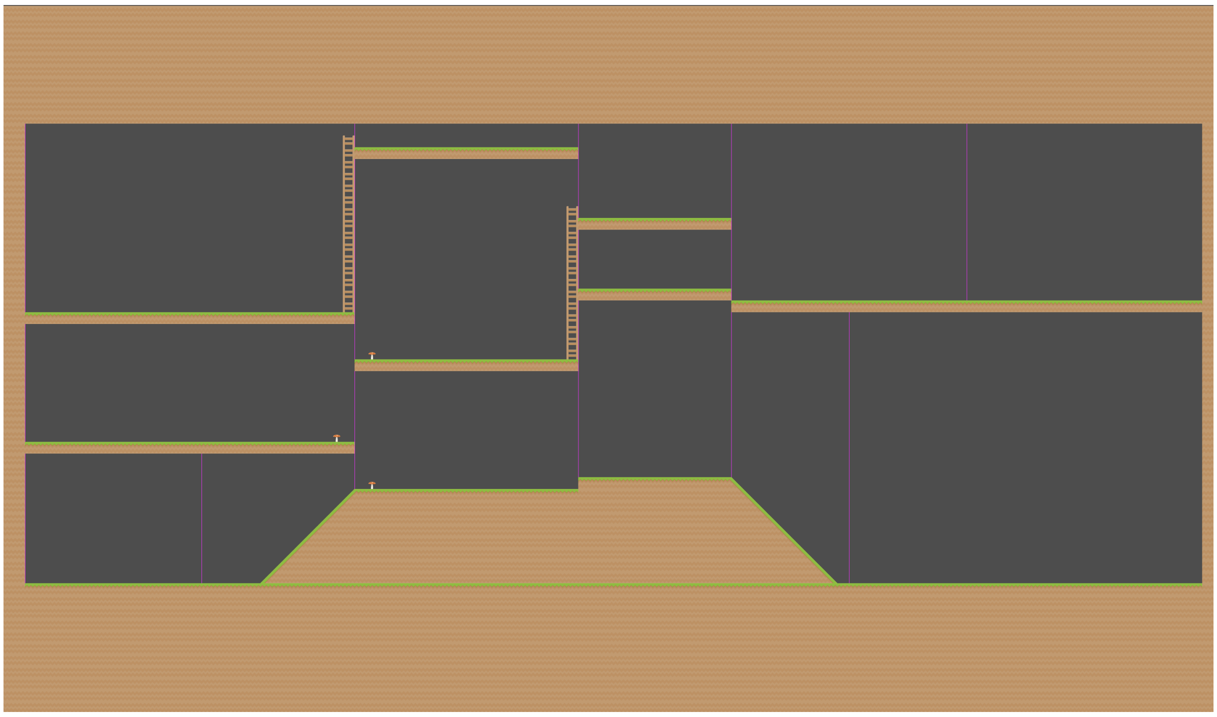

Figure 4.3 Example of Vertical Connectivity Between Zones

4.3.5 Why Zones Were Introduced

Earlier iterations of this project attempted to populate the room without predefined spatial segmentation. Primitives were placed freely across the entire room, and connectivity was handled afterwards using anchor-based pathfinding. However, this approach led to major issues: reachability between doors became unreliable and difficult to enforce; hazards and movement modifiers often overlapped or created unreachable regions; and large rooms lacked structural coherence, resulting in layouts with no clear flow or purpose.

The introduction of a zone-based system resolved these problems by enforcing deterministic spatial segmentation. Each zone could be reasoned about in isolation, guaranteeing that it included a floor, vertical access points, and localised gameplay features. This shift significantly improved the reliability and coherence of room layouts, enabling more deliberate and engaging level structures.

4.4 Primitives and Atoms

The foundation of the level generation system is built upon two key abstractions: *atoms* and *primitives*. These form the basic and composite building blocks of the room and enable structured, rule-driven placement of gameplay elements such as floors, hazards, collectables, and movement modifiers.

4.4.1 Atoms: The Smallest Building Blocks

Atoms are the smallest placeable units in the system. Each atom typically corresponds to a single tile-sized element such as a floor tile, ladder segment, or hazard. The `Atom` class inherits from Godot's `StaticBody2D` and encapsulates a visual sprite, a default tile size (typically 70x70), and optional collision shape. Atoms are responsible for defining their placement behaviour, though in the final system the `ValidatePlacement` method is unused and scheduled for removal.

Atoms are grouped and managed by primitives. They are added to the room using `AddAtom`, which handles scene placement and collision validation. The system prevents atom overlap by checking whether any atoms already occupy the target position before allowing placement. This prevents duplicate primitives from stacking and ensures spatial consistency during generation.

4.4.2 Primitives: Modular Game Elements

Primitives are higher-level constructs composed of one or more atoms. Each primitive represents a meaningful game object or structure, such as a `Floor`, `Ladder`, `Pit`, or `Slug`. These are defined by extending the abstract `Primitive` class, which stores the primitive's position, category (e.g. hazard, movement modifier), difficulty rating, anchor connections, and atoms.

Primitives handle their own placement by implementing the `GenerateInRoom` method. This method defines where atoms are placed and how the primitive ensures its internal layout is valid. For instance, the `Ladder` primitive places multiple vertically stacked `LadderTile` atoms, adjusts the top and bottom tile appearances, and optionally removes the collision from the final tile if it would block movement.

4.5 Environmentals, Hazards, and Door Placement

Once the room's zones, vertical connections, and floor layout are established, the next phase in the generation pipeline involves populating the room with interactable gameplay elements. These include **environmentals** (like pits and water bodies), **hazards** (like blades, slugs, and fish), and **doors and locks**, which mark the entrance and exit points for pathfinding.

4.5.1 Environmentals

Environmental primitives, such as Pit and Water, are large objects that often span multiple tiles and are placed before smaller gameplay elements to avoid overlap. The number of environmentals is dynamically determined based on the size of the room, ensuring proportional density in both small and large spaces. For each selected environmental, a primitive type is randomly chosen, and the system attempts to locate a suitable region within the map for placement. If successful, the list of valid positions is updated to reflect the new terrain. This sequencing ensures that subsequent elements like hazards and doors are not placed on top of pits or water, which have stricter spatial constraints.

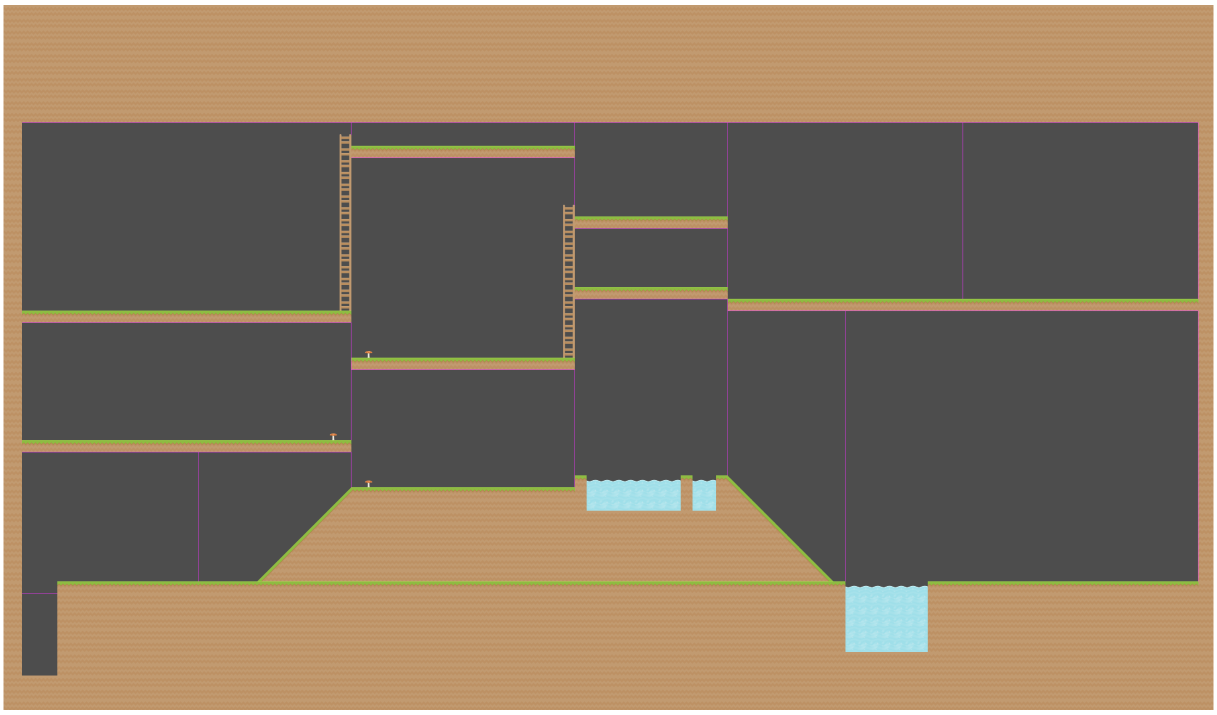

Figure 4.4 Example of Environmentals Generated in the Room

4.5.2 Hazards

Hazards are small, interactable primitives that introduce danger to the player. These include FloorBlade (rotating spikes embedded in the ground), FullBlade (circular saws with radial collision), Fish (aquatic hazards placed inside water), and Slug (patrolling ground enemies). Hazards are placed after the environmental elements, using a similar pool of floor-adjacent positions. Their quantity is determined by the room's difficulty, stored as a percentage and used to compute a minimum and maximum count. Each hazard is instantiated and attempts placement in the room. If it fails due to spatial or overlapping constraints, the generator retries with new positions and/or hazard types until the quota is met or no viable positions remain.

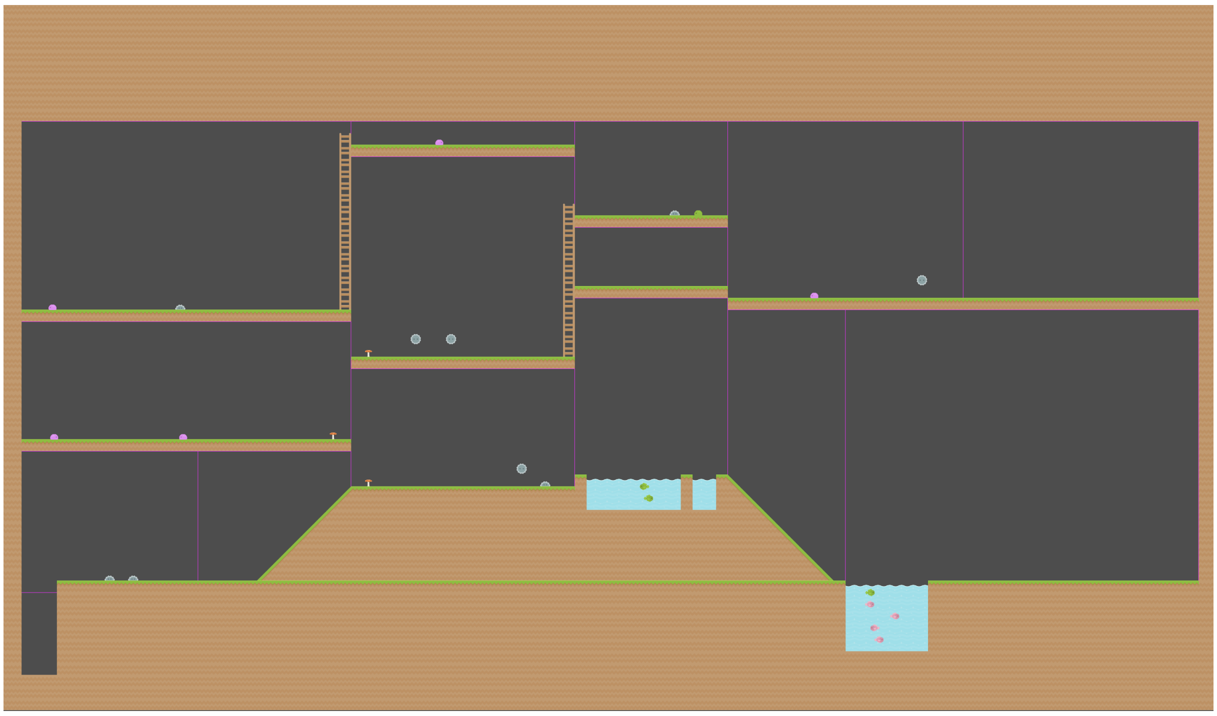

Figure 4.5 Example of Hazards Generated in the Room.

4.5.3 Doors and Locks

Doors act as the entry and exit points for both pathfinding and key logic. Each door is assigned a colour from a shuffled list to ensure visual distinction and is paired with a corresponding DoorLock object, placed one tile to its right. During generation, a number of doors (between two and four) is selected, and candidate positions are sampled above floor tiles. Each door is then paired with a lock and placed at a suitable location. If placement fails, new positions are attempted until successful or all options are exhausted. Only the doors and locks themselves are placed at this stage; the corresponding key placement occurs later in the pipeline, after path building is complete.

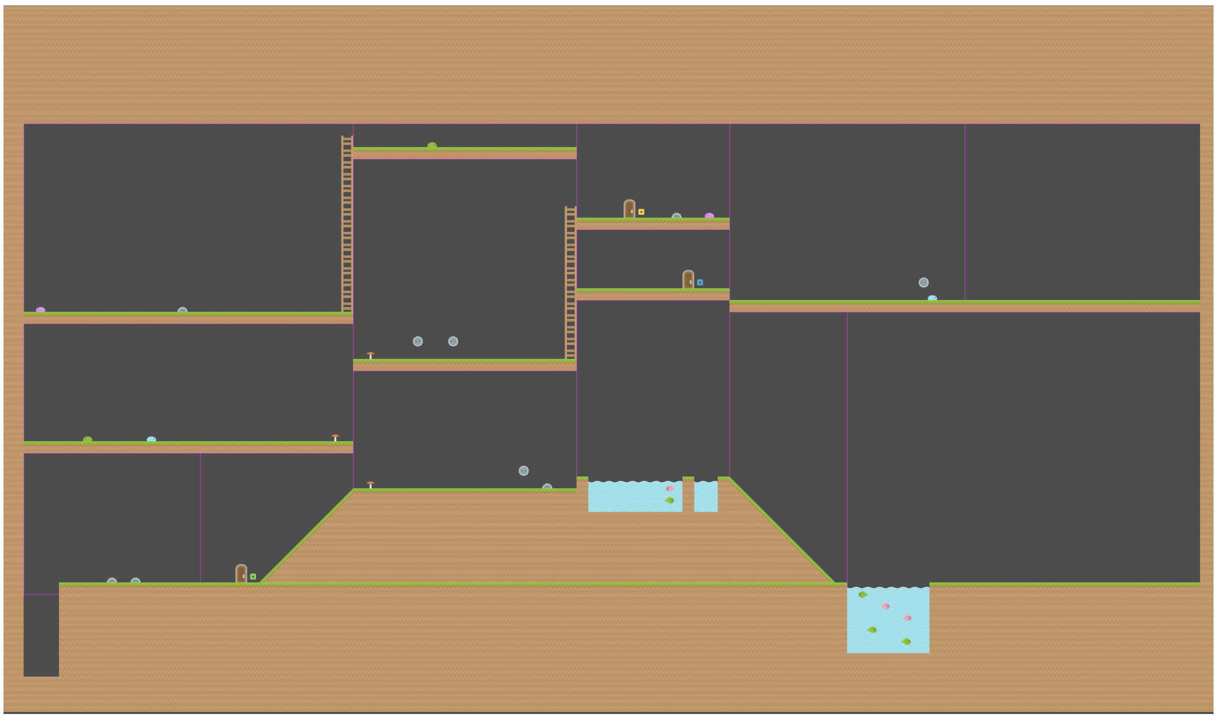

Figure 4.6 Example of Doors Generated in the Room

4.6 Anchors, Connections, and Obstructions

Before constructing navigable paths through the generated room, it is necessary to understand how space and connectivity are represented at a conceptual level. This system introduces a set of abstractions—*anchors*, *anchor connections*, *internal paths*, and *obstruction lines*—to enable precise and flexible control over player movement possibilities.

4.6.1 Anchors

Anchors are spatial points attached to primitives that indicate potential entry or exit locations. Each anchor stores a global position, an orbit radius used to determine whether it overlaps with other anchors, a string label indicating its type (e.g. “topLeft”, “center”) and a reference to the owning primitive.

Anchors enable the system to evaluate whether two primitives can logically connect. For example, the side of a ladder may contain an anchor that connects to the top of a floor, or a jump arc on a mushroom may connect to a platform.

4.6.2 Anchor Connections

An `AnchorConnection` represents a directed or bidirectional link between two anchors, acting as an edge in a navigation graph. They form the foundation of both intra- and inter-primitive traversal logic. Connections are only formed if (i) The orbit ranges of two anchors intersect and (ii) The connection is not obstructed by any solid geometry.

4.6.3 Internal Paths

Internal paths are made up of anchor connections and are used to represent player traversal within a single primitive. For example, a *floor* made up of three tiles defines left-to-right connections between tile anchors and a *ladder* defines a vertical path linking rungs.

These paths are defined inside each primitive's `GenerateAnchors()` method. Without internal paths, primitives such as floors and ladders would consist of isolated anchors, resulting in fragmented graphs during pathfinding.

4.6.4 Obstruction Lines

Obstruction lines are line segments attached to primitives that define impassable regions, such as the tops of pits or walls. They are used to invalidate anchor connections that cross these regions, ensuring paths respect the physical layout of the room.

For example, the `Floor` primitive generates a rectangular boundary around its tiles, and any intersecting anchor connection is removed from the pathfinding graph. Without this mechanism, anchors could inappropriately link across gaps, ceilings, or hazards, undermining both immersion and logic.

Obstruction lines also assist in detecting fully enclosed regions. The generator scans for rectangles bounded on all four sides, and any such area is automatically filled with wall tiles. This prevents the placement of hazards or collectables in unreachable spaces and ensures spatial consistency before final generation steps.

4.6.5 Design Motivation and Evolution

Earlier iterations of this system relied on position-based checks (e.g. tile neighbours or fixed spacing rules), but this approach became unreliable in larger and more complex rooms. Attempts to build tile-level graphs were also abandoned, as they lacked modularity and became too noisy.

By introducing anchors and connecting them through well-defined paths—while checking for visual and spatial obstructions—the system gained the ability to reason

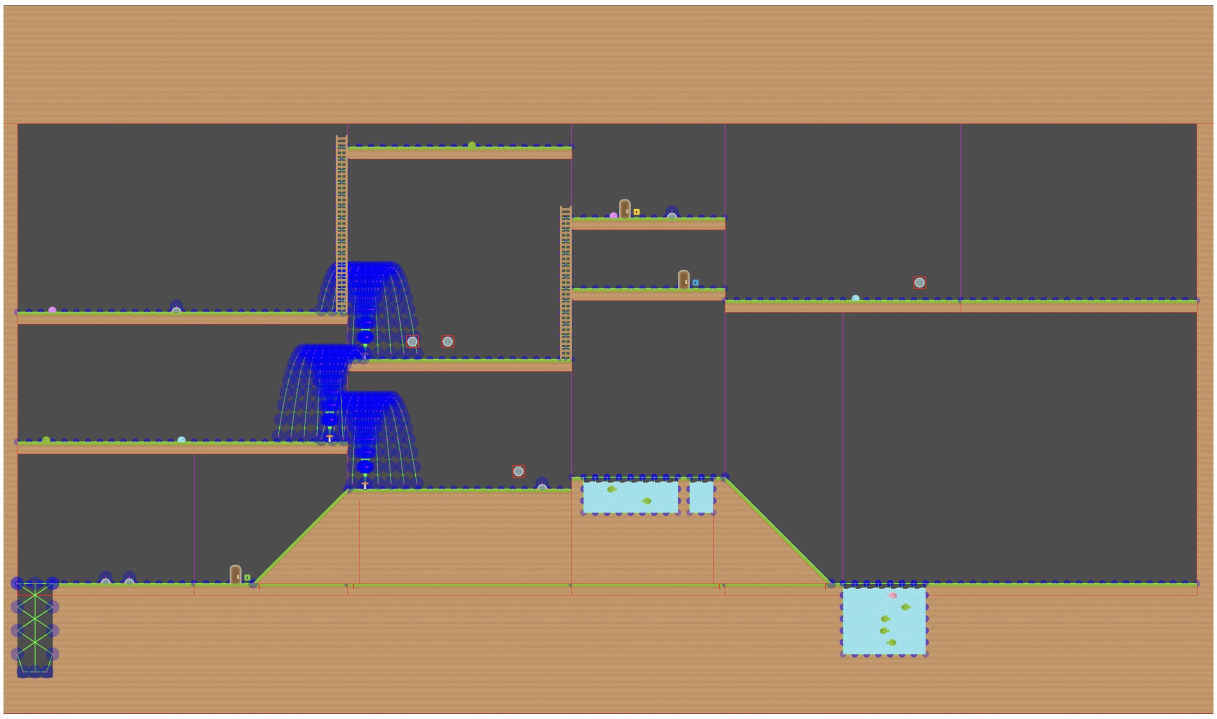

Figure 4.7 Example of Primitives with Anchors, Internal Paths and Obstruction Lines Visible

about space in a modular and scalable way. This foundational structure enables the next stage of the system: path building between entrances, exits, and gameplay-critical locations.

4.7 Difficulty Scaling System

A key feature of this system is its ability to scale room complexity based on a configurable difficulty level. The difficulty is set at room instantiation, defined as an integer between 1 and 5, with 1 being the easiest and 5 the hardest. Internally, this value is converted to a percentage. This normalised value is then used to control multiple aspects of room generation and gameplay behaviour.

4.7.1 Hazard Quantity Scaling

The number of hazards to be placed within the room is computed using the difficulty percentage. A room with low difficulty will contain fewer hazards, while harder rooms will spawn many more. This ensures that the player's challenge increases not only in layout but also in density of threat.

4.7.2 Primitive Difficulty Contribution

Each primitive includes a `Difficulty` field, defaulting to 1. Certain hazard types override this to reflect higher challenge—for example, `FloorBlade` and `FullBlade` are assigned a difficulty of 2, while `Fish` and `Slug` are set to 3. Some primitives also propagate their difficulty to their environment: a `Slug` contributes to the floor it patrols, and a `Fish` to the water it inhabits. This creates a danger profile that is both context-aware and spatially responsive.

4.7.3 Speed-Based Scaling

Dynamic hazards like `Fish` and `Slug` move faster in harder rooms. Their speed is scaled by the difficulty percentage. This makes hazards more aggressive as difficulty increases, encouraging more reactive and deliberate player movement.

4.7.4 Pit Dimension Scaling

The maximum width and depth of pits are calculated based on both the player's movement capabilities and the room difficulty:

```
maxWidth = ceil(maxHorizontalReach * DifficultyPercent / tileSize);
maxDepth = ceil(10 * DifficultyPercent);
```

The final difficulty value for each pit is then computed using the following formula:

```
Difficulty = round(5 * ((Depth / maxDepth) + (Width / maxWidth)) / 2)
```

This ensures that larger pits in harder rooms carry more difficulty weight than smaller ones in easier rooms.

4.7.5 Damage Scaling Based on Difficulty

In addition to controlling the number and behaviour of hazards, difficulty level also affects how much damage the player receives upon contact with an enemy. In rooms of difficulty level 1–3, a hazard collision reduces the player's health by half a heart. In rooms of difficulty level 4 or 5, a full heart is lost. This further emphasises the increased threat posed by harder levels, punishing mistakes more severely and raising the overall stakes of traversal and timing.

4.7.6 Rationale

The scoring system plays a central role in enabling informed decision-making during path evaluation and key placement. By associating difficulty values with primitives and propagating them to connected tiles or structures (such as floors and water bodies), the system constructs a map that embeds challenge not just spatially, but numerically. This quantitative representation of difficulty allows the pathfinding system described in the following section to assess not only the length of a route between doors, but also its gameplay complexity. As a result, keys can be placed along paths that are both spatially extensive and mechanically demanding—offering players a more curated and rewarding traversal experience.

4.8 Path Building and Key Placement

After placing all primitives in the room—such as hazards, environmentals, and doors—the final stage of the generation pipeline constructs traversable paths between doors and decides on ideal locations for keys. This is accomplished through an anchor-based pathfinding system and a strategy for identifying key placements that maximise path length and difficulty.

4.8.1 Selecting a Start Door

The path generation process begins by choosing a start door. The system selects the door positioned furthest to the left in the room layout and uses its central anchor as the origin point for all pathfinding attempts.

This simplifies path generation to a set of one-to-many searches, where each path begins at the start door and ends at one of the remaining doors. For each target door, the goal is not merely to find *a* valid path, but to identify the **most difficult and longest** viable path among a curated set of possible routes.

4.8.2 Ideal Key Candidates

To embed keys meaningfully into the map, the system defines a list of *ideal key positions* for each target door—locations that encourage the player to traverse remote and challenging regions. These include the deepest tile in each pit, the lowest reachable point within water areas, and the four floor tiles that are furthest away (in terms of Euclidean distance) from the target door.

Each of these candidate positions is wrapped inside a temporary `DoorKey` primitive. Anchors are generated for each ideal key, and the pathfinding graph is then constructed including both room primitives and these temporary key objects.

4.8.3 Graph Construction

The anchor-based pathfinding system models the room as a graph where anchors act as nodes and connections—both internal and between intersecting orbits—serve as edges. These edges can be either bidirectional or one-way, depending on the primitive's traversal properties. Graph construction proceeds in three stages:

1. All anchors are added to the graph with empty connection lists
2. Internal paths defined by each primitive are inserted
3. Connections between anchors whose orbits intersect are added

Once built, this graph represents all possible paths the player could take from any anchor to another within the room.

4.8.4 Finding the Best Path

The core logic uses Breadth-First Search (BFS) to find the shortest viable path between the start door and each target door **through** one of the ideal key positions. For each ideal key:

1. A path is generated from the start door to the key anchor
2. A second path is generated from the key to the target door
3. The two paths are concatenated into one complete route

Each of these full paths is then evaluated using the difficulty scoring system (already described in a previous section), and the one with the highest total difficulty is chosen. If multiple paths have equal difficulty, the longest path is preferred.

4.8.5 Final Key Placement

Once the optimal path is chosen for a given door, the key object corresponding to the best anchor is instantiated permanently and placed in the room. This ensures that keys are not only reachable but meaningfully embedded into the level's most complex and remote areas.

4.8.6 Debug Visualisation

All generated paths are drawn onto the map using colour-coded lines that match their corresponding door colours. This assists with debugging and qualitative evaluation by allowing developers to visually confirm the completeness and coherence of the navigation system.

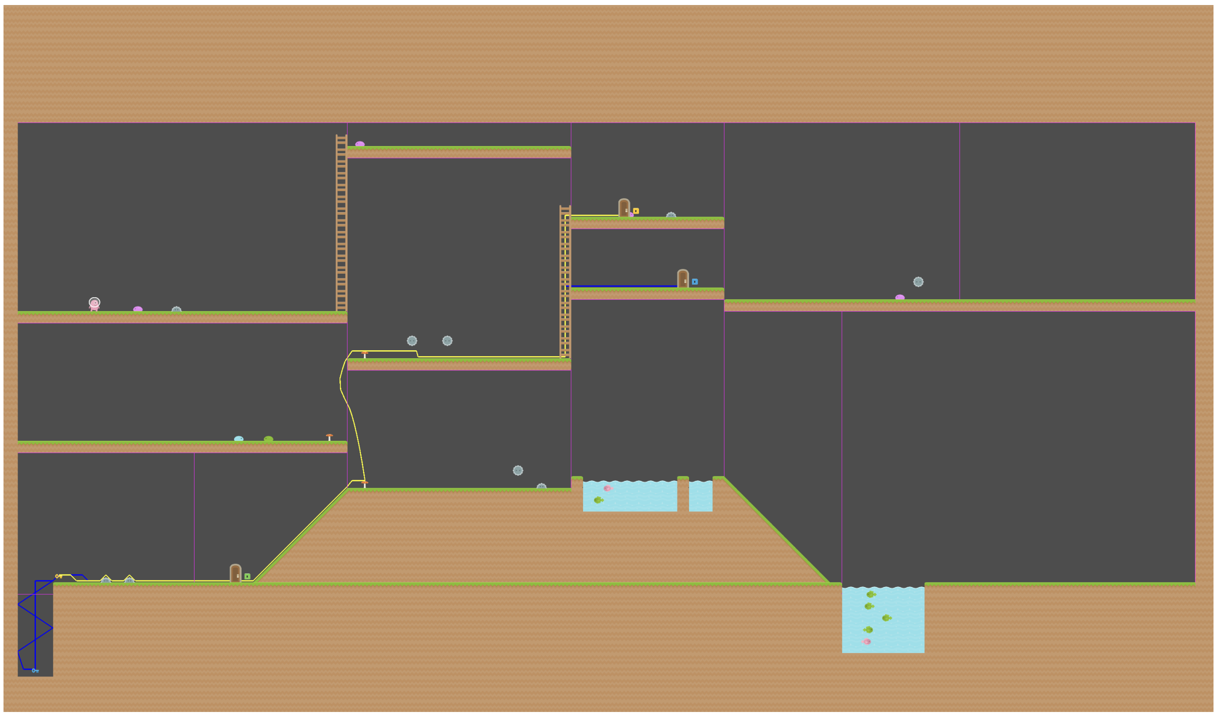

Figure 4.8 Example of Path Generation and Key Placement.

4.8.7 Design Considerations

This approach balances procedural freedom with structure. While anchor graphs allow highly dynamic level layouts, the use of ideal anchor candidates and difficulty-aware path scoring ensures that key placements serve to extend gameplay in meaningful and challenging ways.

Future work may explore the inclusion of gating mechanisms, where keys unlock certain movement abilities rather than doors, aligning the system even more closely with the traditional gating structure of Metroidvania design.

4.8.8 Alternative Approaches Considered

Before arriving at the final anchor-based path evaluation system, multiple alternative approaches to key placement and pathfinding were explored. Each method offered initial promise but ultimately proved inadequate for the level of control, challenge, and coherence required for this project.

Random Placement with Post-Validation

In early implementations, keys were placed randomly on valid tiles above the floor, and the system attempted to find a valid path to and from the key after placement. While simple, this approach resulted in frequent failures and retries. Keys would often end up in trivial or uninteresting locations, and the system had no way of ensuring that the path traversed meaningful portions of the map. This method also lacked any awareness of difficulty or engagement.

A* Search Over Anchor Graph

A more sophisticated method using A* pathfinding over the anchor graph was implemented, combining spatial distance and primitive difficulty into a custom heuristic. However, A* is inherently optimised for finding the shortest path. As a result, it consistently produced efficient but shallow paths that skipped challenging or interesting areas of the room. This behaviour ran counter to the design goal of encouraging exploration and rewarding detours.

Greedy Anchor Expansion Using Primitive Compatibility

Another prototype used a greedy expansion algorithm. Starting from door anchors, the system selected compatible primitives from a matrix and attempted to grow the level outward. While modular in concept, this method lacked global context and often produced disconnected zones or local dead ends. Paths failed to form between distant primitives, especially in vertically segmented maps.

4.8.9 Interestingness Evaluation

To quantify the quality of each generated room, a composite scoring function was implemented to assign an **interestingness score** between 0 and 1. This score evaluates the extent to which a room offers spatial exploration, goal-seeking, mechanical complexity, and ability-driven interaction.

The final score is calculated as a weighted sum of five components:

$$\text{Interestingness} = w_a \cdot s_a + w_g \cdot s_g + w_d \cdot s_d + w_v \cdot s_v + w_p \cdot s_p$$

Where:

- s_a: Anchor coverage — the proportion of unique anchors visited.
- s_g: Goal count — the number of goals (e.g., keys) collected, normalised over a maximum of 3.
- s_d: Average difficulty — the average difficulty score of the paths, normalised over an upper bound of 120.
- s_v: Vertical movement modifier variety — the proportion of unique vertical modifiers used.
- s_p: Player ability usage — the proportion of distinct player abilities required.

The weights applied to each component are empirically chosen to reflect their relative contribution to perceived gameplay complexity and richness:

$$w_a = 0.05, \quad w_g = 0.20, \quad w_d = 0.30, \quad w_v = 0.20, \quad w_p = 0.25$$

Each score component s_x is normalised between 0 and 1 using pre-defined maxima:

$$s_a = \frac{\text{anchorsVisited}}{\text{amax}}, \qquad s_g = \frac{\text{goals}}{3}, \qquad s_d = \frac{\text{avgDifficulty}}{120}$$
$$s_v = \frac{\text{verticalModifiersUsed}}{4}, \qquad s_p = \frac{\text{playerAbilitiesUsed}}{3}$$

Where:

- amax is the total number of anchors in the graph.
- The four vertical movement types considered are `Ladder`, `Mushroom`, `LeftSlope`, and `RightSlope`.
- The three abilities considered are `Swimming (Water)`, `Dodging Pits`, and `Avoiding Full Blades`.

Path difficulty is calculated individually per path and averaged across all valid paths found between door pairs. Vertical modifiers and abilities are inferred from the primitive types associated with the anchors traversed.

This evaluation system was designed to ensure that rooms which offer more structural complexity, path diversity, and mechanical requirements are rewarded with higher scores—providing a quantitative basis for comparison and statistical analysis in the results chapter.

Tile-Based Graph Representation

A low-level approach was considered in which each tile would act as a node in a graph, with traversal occurring from tile to tile. This method was quickly discarded due to its lack of abstraction and high computational overhead. It also ignored the semantic modularity offered by primitives like ladders, slopes, and hazards, making path logic harder to reason about and test.

Justification for Final Approach

The final strategy—evaluating optimal paths through curated anchor connections—offered several advantages. It enabled structured randomness by testing multiple candidate routes, incorporated experiential quality via difficulty scoring, preserved modularity through anchor-based abstraction, and allowed for effective visual debugging and consistent behaviour across varied room designs.

The final method—evaluating the best path through ideal anchor positions—combines several strengths:

- It allows for structured randomness by testing multiple curated candidate routes.
- It evaluates paths not just for validity, but for experiential quality via a difficulty scoring system.
- It preserves modularity and semantic richness through anchor-based abstraction.
- It offers clear, visual debugging and reliable behaviour across different room layouts.

This process of elimination and refinement demonstrates the importance of design iteration in procedural generation. It also reinforces the idea that meaningful content often requires more than simple connectivity—it requires intentionality, pacing, and spatial storytelling.

4.9 Player Integration and Movement Constraints

The player controller is a central component of the game architecture, serving as the basis for validating level navigability and influencing the placement and logic of several primitives. Although procedural room generation occurs independently of the player's immediate actions, the player's movement capabilities — including jump height, wall grip, climb speed, and dash behaviour — directly inform the system's decisions during generation.

4.9.1 Player Implementation Overview

The player is implemented using a `PlayerController` class that inherits from `CharacterBody2D`. It supports various abilities such as jumping, wall-jumping, dashing, climbing, and spring-jumping (via mushroom tiles). Movement is responsive to environmental modifiers like sticky floors, slippery floors, slopes, ladders, and water bodies. The controller also manages animations, health, recoil, and interactions with hazards, doors, and keys.

To maintain clarity and modularity, key movement behaviours like `Jump`, `Dash`, `WallJump`, `Climb`, and `Recoil` are encapsulated in separate classes and instantiated within the player. Environmental detection relies on raycasts and group-based tagging (e.g. "Floor", "Ladder", "Water"), allowing the player to adapt dynamically to various room features during runtime.

4.9.2 Spawn Logic and Scene Reference

When a new room is instantiated, the player is spawned via the `SpawnPlayer()` method in the `Room` class. Once instantiated, the player is assigned a reference to the current room, which is essential for interacting with primitives like hazards, doors and keys.

4.9.3 Influence on Level Generation

The player's physics parameters — particularly jump speed, gravity, and movement speed — play a critical role in guiding level generation. These parameters are used in calculations that determine the maximum width and depth of pits that a player can cross or escape from; the feasibility of placing springs (mushrooms) to allow vertical traversal between zones; the trajectory and reach of jump arcs during anchor generation for mushroom primitives; and the required spacing between vertical movement modifiers (like ladders or slopes) and adjacent zones.

For instance, the `canPlaceSpring()` method evaluates whether a vertical gap can be bridged using a spring jump, factoring in the spring jump height derived from the kinematic equation:

$$h = \frac{v^2}{2g}$$

Where v is the `springJumpSpeed` and g is the player's gravity. Similarly, jump arcs rendered via anchor generation use a parabolic equation informed by both horizontal move speed and vertical jump capacity to produce realistic and reachable paths between platforms or movement modifiers.

By embedding the player's capabilities into the level logic, the system ensures that all generated rooms remain theoretically navigable, assuming correct execution of player mechanics.

4.9.4 Interaction with Generated Elements

The player is also responsible for detecting and picking up keys, which are placed at procedurally evaluated "difficult" locations based on pathfinding results; unlocking doors once the corresponding key has been acquired; reacting to hazards with health loss and knockback (recoil); and navigating modified terrain such as springboards, slopes, water, and sticky or slippery tiles.

By interlinking procedural logic with the physical and interactive constraints of the player character, the system effectively uses the player as a reference model for what counts as reachable, valid, and interesting in the generated space.

4.10 Known Bugs and Limitations

Despite significant effort to ensure the robustness and quality of the system, a number of known bugs and limitations were encountered during the development of the procedural room generation pipeline.

One of the most persistent issues involves the occasional creation of unreachable or unescapable zones. While the zone system, vertical connection logic, and the obstruction-based DetectAndFillEnclosedAreas() method all aim to guarantee global reachability, there remain rare edge cases where inaccessible areas emerge. The most common culprit is the slope primitive. Slopes are movement modifiers that often require a supporting wall to be placed underneath them. In scenarios where one floor is wider than another or where floors are offset in height, the slope's wall may block access beneath or beside it, thereby creating a region that is visually accessible but practically unreachable. For example, if a slope connects a wide upper floor to a narrow lower one, the obstruction lines generated may not form a closed rectangle, preventing the detection and filling of the enclosed area. In such cases, the room's auto-fill logic fails to patch the void, resulting in small, isolated gaps the player cannot traverse. These situations, while rare, expose the challenges of handling spatial edge cases in tile-based PCG systems.

A second limitation lies in the simplified difficulty scoring system. Each primitive is assigned a static integer difficulty value from 1 to 5, with values loosely corresponding to how challenging the primitive is expected to be. While this system successfully allows for dynamic hazard scaling and informs path evaluation during key placement, it lacks the nuance to represent contextual difficulty. For instance, a blade positioned near a ladder or spring might be more challenging than one in isolation, but the system does not account for such interactions. Additionally, the total difficulty of a path is computed by summing the difficulty values of the primitives involved, which is a coarse approximation that does not consider timing, spacing, or player error likelihood.

Another notable constraint involves the static definition of player ability metrics. The player's movement speed, jump height, and spring-assisted jump height are used to determine pit dimensions and spring jump arcs. However, these values are defined at room initialisation and do not dynamically adjust if player stats change mid-game. As a result, the generator cannot yet respond to upgrades or difficulty progression during runtime. This restricts its potential use in a full Metroidvania context, where player abilities typically evolve over time.

Finally, while most placement logic is rule-based and deterministic once random values are resolved, the system is still vulnerable to visual clutter in high-density rooms. This is especially the case when multiple primitives are placed in close proximity without sufficient spatial separation. Although the internal validation logic aims to prevent direct overlaps, there is no formal mechanism to enforce visual harmony or aesthetic spacing. As a result, rooms may occasionally appear chaotic or unintentionally crowded, which can detract from perceived quality.

These limitations do not compromise the fundamental aims of the project, but they point to several promising avenues for future refinement and research. Improvements in difficulty modelling, spatial context awareness, and dynamic system adaptation would all help to evolve the generator toward greater sophistication and broader applicability.

5 Evaluation and Results

This chapter presents the evaluation of the procedural generation system developed in this project, focusing on its ability to generate interesting and diverse Metroidvania-style room layouts. The results were obtained through two primary methods: (1) a Monte Carlo simulation that sampled and statistically analysed room configurations across different parameters, and (2) a set of hand-picked screenshots used to visually assess design quality and structure. This chapter first explains how the simulation was set up and what data was collected, followed by an analysis of the results and the patterns observed. The final section of this chapter presents the qualitative evaluation derived from the screenshot analysis.

5.1 Monte Carlo Simulation

5.1.1 Methodology

To evaluate the statistical behaviour of the generation system, a Monte Carlo simulation was performed. The simulation systematically generated rooms using varying input parameters: *difficulty level* (ranging from 1 to 5), *room size*, and *number of doors*. The difficulty influenced the density and complexity of obstacles, while the number of doors introduced different traversal routes. Four room sizes were considered: 35x30, 70x65, 105x100, and 210x50, representing a mix of square and rectangular layouts. For each combination of difficulty and room size, rooms were generated with two, three, and four doors respectively.

A total of 20 rooms were generated per parameter combination. Each room's layout was evaluated using the custom interestingness formula described in Chapter 3, producing a numerical score that captured the room's engagement value based on factors like exploration, goal diversity, and player ability usage. The final dataset consisted of 1200 samples.

5.1.2 Analysis

Two key types of analysis were performed on the simulation results using RStudio:

- **Visualisation:** Box plots and smoothed trend plots were created to examine how interestingness varied with difficulty, number of doors, and room dimensions. These visualisations provided an intuitive view of performance trends across different configurations. See Appendix 5.1 and Appendix 5.2 for the figures.

- **Statistical modelling:** To quantify the influence of each parameter, a linear regression model was fitted using interestingness as the dependent variable, and difficulty, door count, room width, and room height as predictors. The full output of the regression model is shown in Appendix B.

5.1.3 Results

Boxplot and Smoothed Trend Analysis

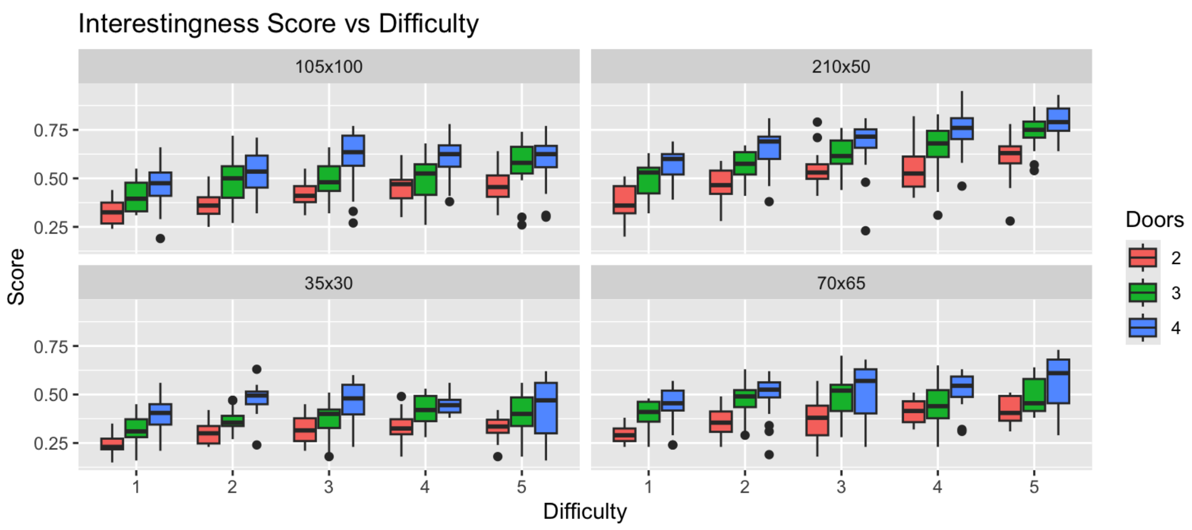

Figure 5.1 Interestingness score vs difficulty, grouped by door count and faceted by room size.

Figure 5.1 reveals that interestingness scores generally increase with difficulty, particularly in larger rooms. The effect of adding more doors is also clearly visible: within each difficulty level, rooms with more doors show a higher median score and wider score distribution, suggesting increased variability and complexity. Notably, rooms sized `210x50` and `105x100` achieved the highest scores overall, particularly with 4 doors and difficulty levels 4–5. Conversely, the smallest room size (`35x30`) consistently scored lower across all configurations, suggesting limited spatial expressiveness.

Figure 5.2 further supports this observation: a clear positive trend is visible between room area and interestingness. While some saturation is seen beyond a certain size (especially at lower difficulties), larger rooms tend to enable more movement diversity, goal placement, and vertical modifiers, all of which are rewarded by the scoring system.

Linear Regression Model

The full results of the regression model are shown in Appendix A. Key findings include:

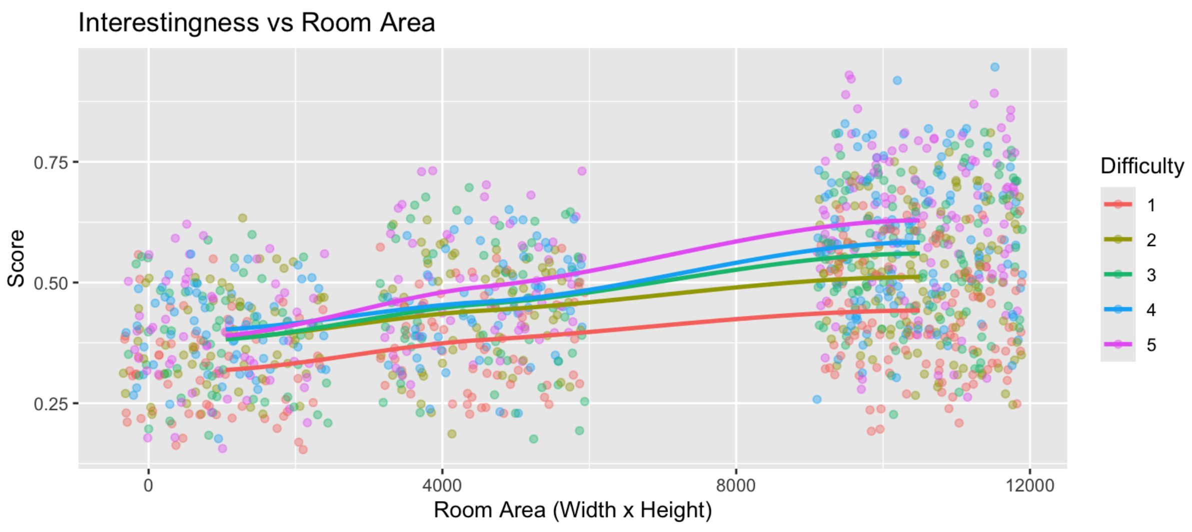

Figure 5.2 Smoothed trend of interestingness scores vs room area, coloured by difficulty.

- The model explains approximately 57% of the variance in interestingness ($R^2 = 0.571$), which is considered reasonably strong for this type of behavioural or simulation-based data.
- All coefficients were statistically significant ($p < 0.01$), indicating strong confidence in their effects.
- The model uses categorical encoding for both difficulty and doors. By default, R uses the first level as the reference. In this case, `Difficulty = 1` and `Doors = 2` are absorbed into the intercept, and do not appear explicitly in the results. The intercept (0.166) thus represents the baseline score when `Difficulty = 1`, `Doors = 2`, and room dimensions are zero — a purely theoretical value.

Interpreting the coefficient estimates:

- **Increasing Difficulty:** The coefficients for Difficulty 2–5 range from 0.066 to 0.138, meaning each step up in difficulty level adds approximately 6–14 percentage points to the interestingness score, all else being equal. This confirms that the scoring system is responsive to increased gameplay challenge.
- **Adding Doors:** Relative to rooms with 2 doors, rooms with 3 doors score 0.088 higher, and 4 doors score 0.15 higher. This suggests that door count is a strong proxy for map complexity and exploration potential.
- **Room Size (Width vs Height):** The effect of room width (0.00125 per unit) is over four times larger than height (0.00031), suggesting that horizontal expansion

contributes more to interestingness. This likely reflects the design's focus on lateral movement, path branching, and horizontal hazards. Height has a weaker effect, possibly because vertical movement modifiers are used less frequently or offer limited diversity compared to horizontal ones.

In summary, the simulation supports the hypothesis that interestingness increases with spatial size, difficulty, and path diversity. The scoring system appears to effectively reward richer gameplay environments and can differentiate between low-effort and highly curated rooms. This quantitative evidence demonstrates the potential of procedural generation techniques to produce engaging level designs when guided by a meaningful evaluation framework.

5.2 Screenshot-Based Qualitative Analysis

In addition to the Monte Carlo simulation, a set of representative room screenshots was selected to qualitatively examine the relationship between room layout and interestingness. Four rooms were chosen, each belonging to a different quadrant of the score-difficulty space: (1) high difficulty and high score, (2) high difficulty and low score, (3) low difficulty and high score, and (4) low difficulty and low score. The corresponding screenshots can be found in Appendix B.

The high difficulty, high score room (Figure B.1) showcases the scoring system's ideal output: a large, vertically and horizontally diverse space with multiple collectibles, rich use of vertical movement primitives (ladders, slopes, water), and high ability usage. The complexity of traversal across many interconnected zones reflects meaningful spatial design enabled by both size and difficulty.

By contrast, the high difficulty, low score room (Figure B.2) highlights how high difficulty alone does not guarantee interestingness. This room is narrower and vertically constrained. It contains very few distinct paths, and the movement primitives used are minimal, despite the difficulty setting. The player likely encounters redundant or overly linear routes, leading to a lower score.

The low difficulty, high score room (Figure B.3) reveals that high scores can still emerge in lower difficulty settings — but only when layout affordances support it. This room features broad exploration space, multiple doors, and well-placed collectibles. It demonstrates that spatial richness and diverse movement opportunities (rather than difficulty alone) are key drivers of perceived interestingness.

Finally, the low difficulty, low score room (Figure B.4) serves as a baseline. The design is overly linear, lacking vertical variation or movement complexity. Key features like hazards, water, or key-lock mechanics are sparse or absent, yielding a flat experience both structurally and mechanically.

This qualitative analysis reinforces the simulation's findings: interestingness emerges not from isolated parameters like difficulty, but from the interplay between layout space, vertical/horizontal expressiveness, and primitive usage. The screenshots illustrate how the system rewards designs that mimic hand-crafted intent — validating the scoring model's ability to distinguish between shallow and engaging level configurations.

6 Conclusion

This project set out to explore how procedural content generation (PCG) can be used to create room layouts in the Metroidvania genre that retain the sense of intentionality and discovery characteristic of hand-designed levels. The central challenge was to balance randomness with structure, ensuring that generated content felt meaningful, navigable, and engaging.

To achieve this, a modular generation pipeline was developed in the Godot Engine. The system was built around concepts of primitives, zones, anchors, and pathfinding, enabling rooms to be assembled from configurable components with spatial logic and player abilities in mind. Each room was evaluated using a custom interestingness metric, incorporating factors such as traversal complexity, vertical modifier diversity, goal density, and ability usage.

The system was evaluated quantitatively using a Monte Carlo simulation and qualitatively through screenshot analysis. The simulation results demonstrated clear patterns: interestingness generally increased with room difficulty, size, and number of doors. A linear regression model confirmed these trends, identifying room width and door count as particularly strong predictors of high interestingness scores.

The screenshot-based analysis provided further insight. Visually comparing rooms from different quadrants of the score-difficulty space illustrated how the best layouts tended to exhibit strong vertical and horizontal connectivity, meaningful branching paths, and thoughtful placement of collectables and movement modifiers. By contrast, low-scoring rooms often lacked spatial diversity or failed to fully utilise the available primitive set.

While the results are encouraging, several limitations remain. The system was evaluated on room-level generation only; future work could extend this to full world generation with inter-room connections. Additionally, primitives like enemies were excluded from this version of the generator and could play a significant role in perceived challenge and interest.

In future iterations, integrating machine learning or player modelling could help adapt generation in real-time based on playstyle or preferences. Another promising direction would be to experiment with hybrid generation techniques — combining authored templates with procedural variation — to further blur the line between human and machine design.

Ultimately, this project shows that procedurally generated Metroidvania content can move beyond randomness to capture aspects of hand-crafted design. Through carefully structured systems and thoughtful evaluation, PCG can support the creation of interesting, explorable, and rewarding spaces — preserving the genre's essence.

References

[1] Reddit users, *Procedural generation in metroidvanias*, `https://www.reddit.com/r/metroidvania/comments/kqpxn6/procedural_generation_in_metroidvanias/`, 2020.

[2] T. Stalnaker, *Procedural generation of metroidvania-style maps using graph grammars*, Undergraduate honors thesis, Washington and Lee University, 2020.

[3] J. Xu and J. Morris, “Procedural generation in 2d metroidvania game with answer set programming and perlin noise,” *International Journal of Artificial Intelligence and Applications*, vol. 14, no. 3, pp. 1–16, 2023.

[4] D. Oliveira, A. R. Tomazzoni, R. M. Bastos, C. F. Alves, and R. Behar, “A framework for metroidvania games,” in *Proceedings of SBGames 2020 — Industry Track*, 2020, pp. 1–10.

[5] A. Gutiérrez Rodríguez, C. Cotta, and A. J. Fernández Leiva, “An evolutionary approach to metroidvania videogame design,” in *Proceedings of the Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, vol. 246, 2018, pp. 1–11.

[6] M. Nitsche, C. Ashmore, A. Hankinson, R. Fitzpatrick, and J. Kelly, “Designing procedural game spaces: A case study,” in *FuturePlay ’06: Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, 2006, pp. 1–8.

[7] G. McAllister and G. R. White, “Evaluating user experience in games: Concepts and methods,” in *Video Game Development and User Experience*, ser. Human-Computer Interaction Series, R. Bernhaupt, Ed., Springer, 2010, pp. 1–19.

[8] V. Blašković, M. Stevanović, J. Gašić, and B. Divjak, “Evaluating a conceptual model for measuring gaming experience: A case study of stranded away platformer game,” *Information*, vol. 14, no. 11, p. 350, 2023.

[9] G. Crawford and T. Brock, “Just fun and games? a sociological consideration of fun in video games,” *Games and Culture*, pp. 1–18, 2024. doi: `10.1177/15554120241254876`.

Appendix A Regression Output

```
Call:
lm(formula = Score ~ Difficulty + Doors + RoomWidth + RoomHeight,
   data = all_data)

Residuals:
     Min       1Q   Median       3Q      Max
-0.45851 -0.05864  0.00940  0.06965  0.26520

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.1663     0.0109   15.31   < 2e-16 ***
Difficulty2  0.0663     0.0090    7.33   4.34e-13 ***
Difficulty3  0.0934     0.0090   10.33   < 2e-16 ***
Difficulty4  0.1112     0.0090   12.30   < 2e-16 ***
Difficulty5  0.1385     0.0090   15.32   < 2e-16 ***
Doors3       0.0878     0.0070   12.53   < 2e-16 ***
Doors4       0.1515     0.0070   21.63   < 2e-16 ***
RoomWidth    0.00125    0.00004  28.30   < 2e-16 ***
RoomHeight   0.00031    0.00011   2.77   0.0057  **
---
Residual standard error: 0.099 on 1191 degrees of freedom
Multiple R-squared: 0.571, Adjusted R-squared: 0.569
F-statistic: 198.5 on 8 and 1191 DF, p-value: < 2.2e-16
```

Listing A.1 Regression Output

Appendix B Screenshot Samples

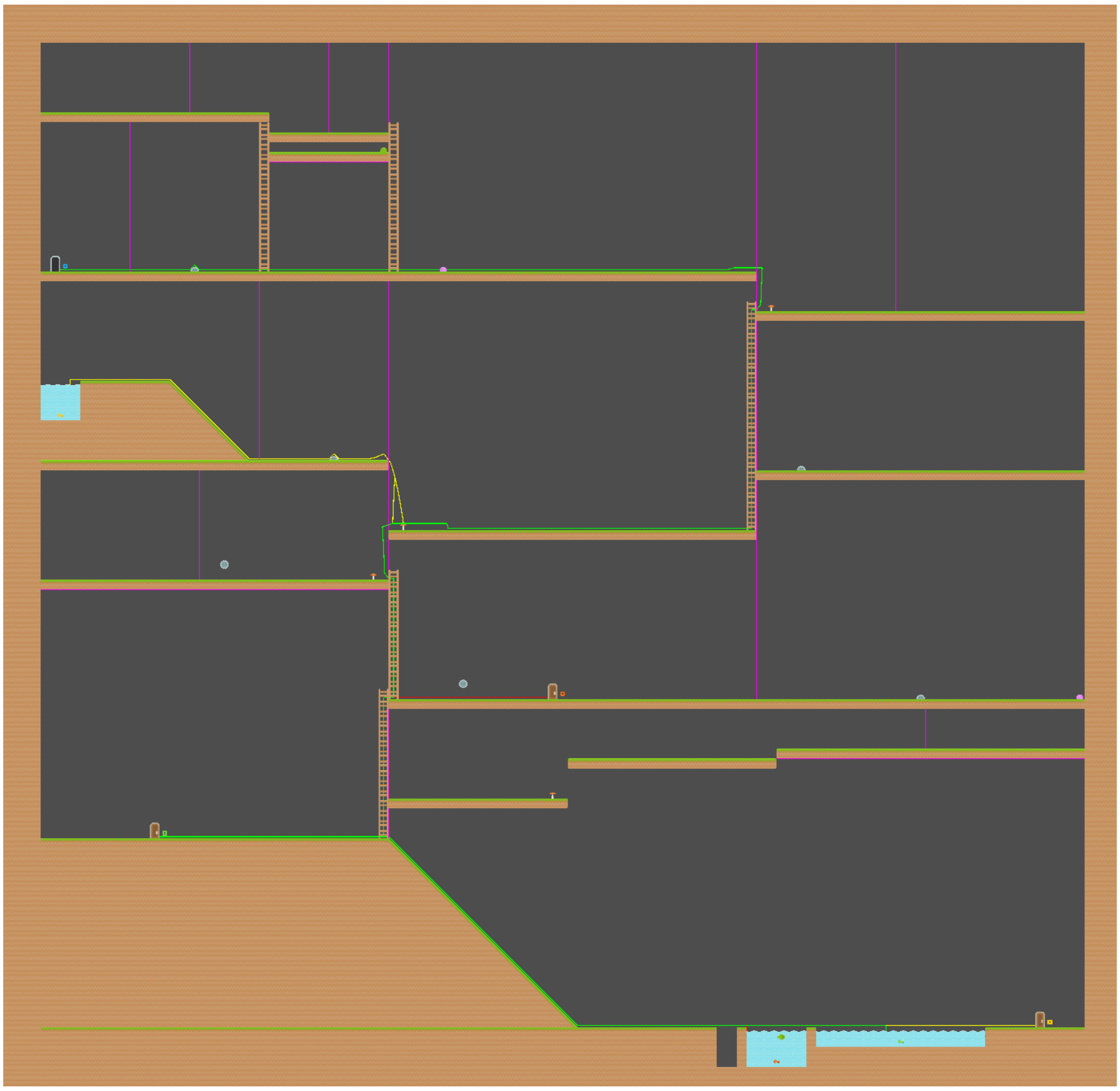

Figure B.1 High difficulty, high score. Room 105×100, Diff 5, Score 0.75

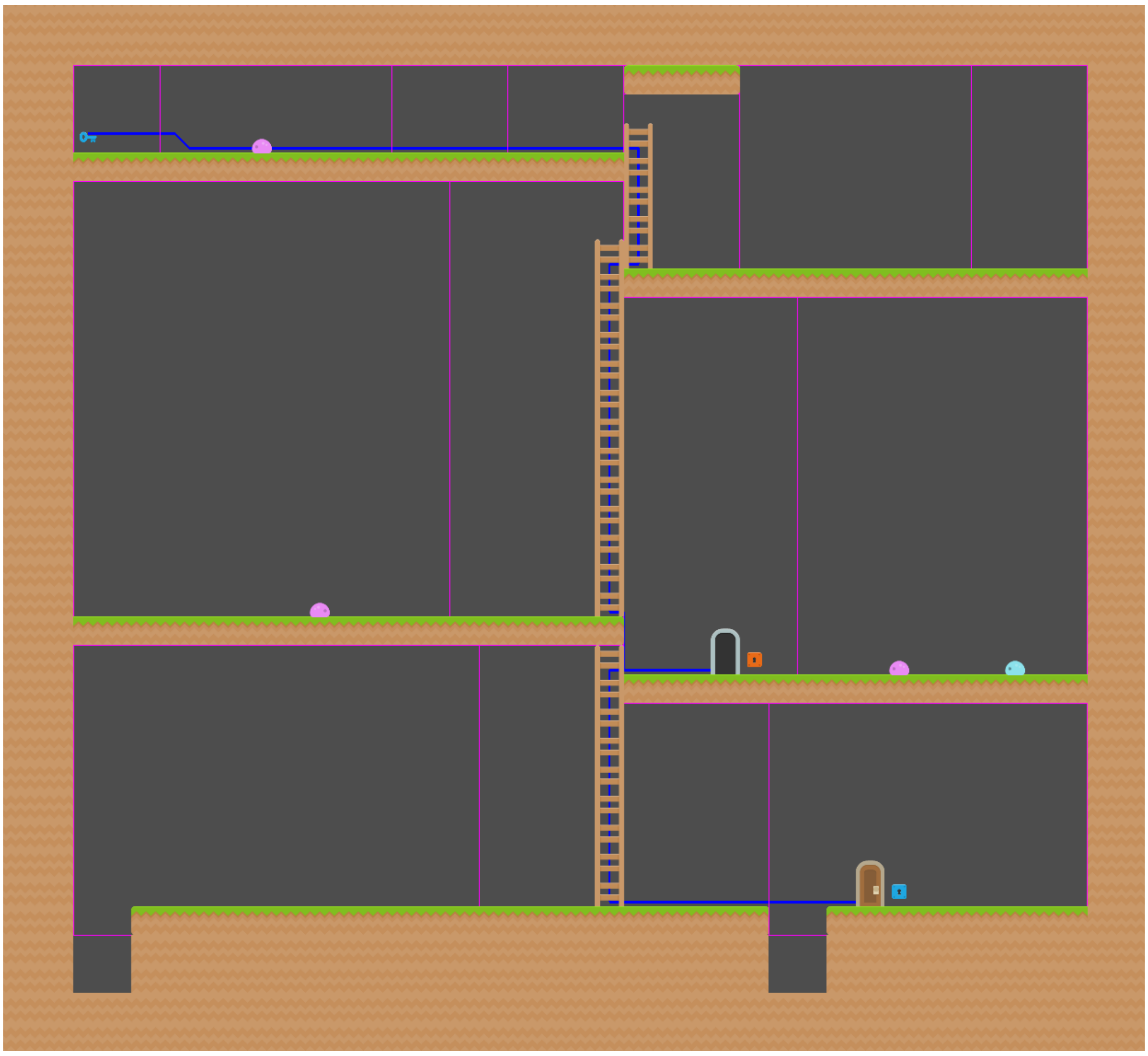

Figure B.2 High difficulty, low score. Room 35×30, Diff 4, Score 0.30

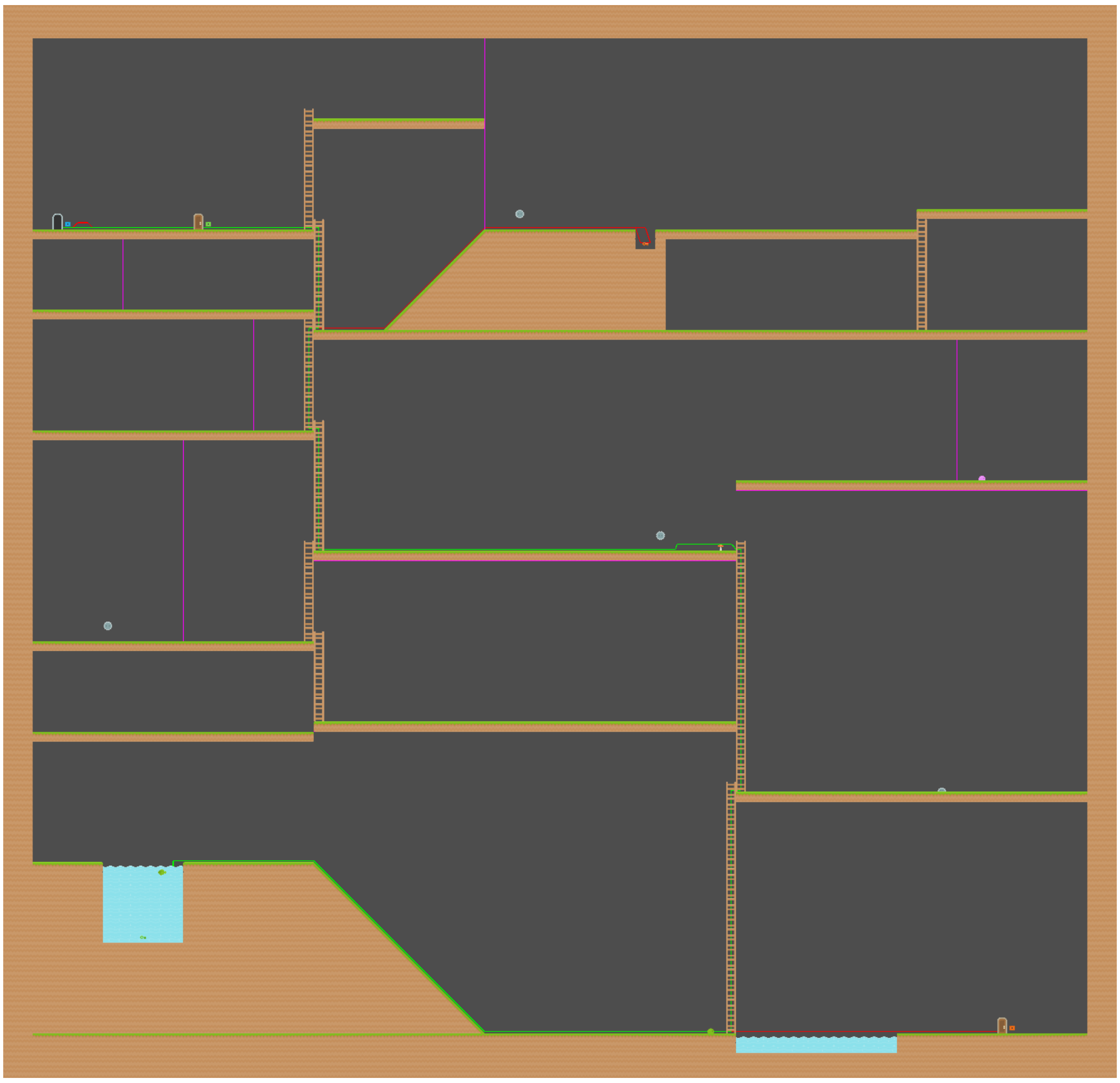

Figure B.3 Low difficulty, high score. Room 105×100, Diff 2, Score 0.71

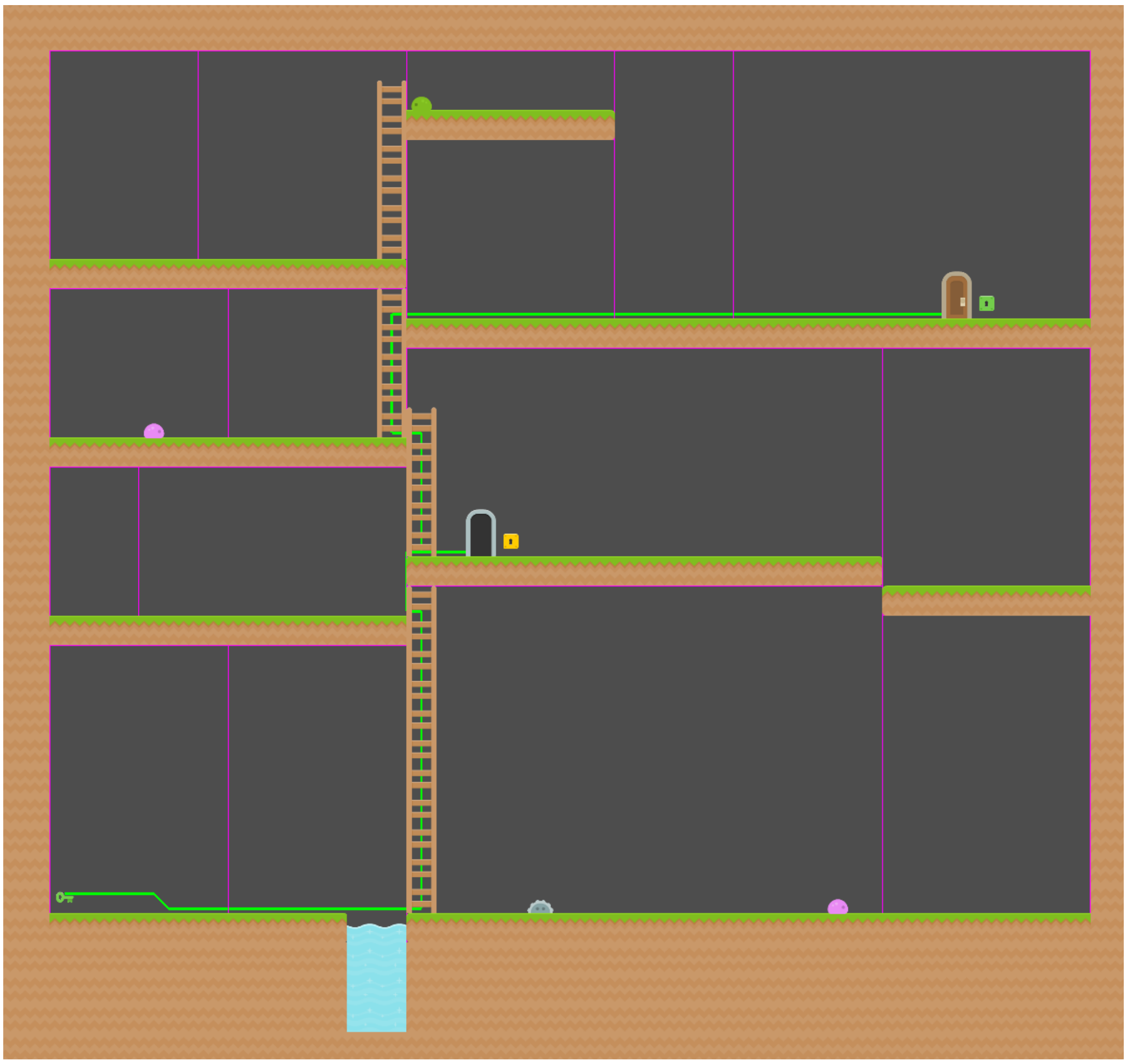

Figure B.4 Low difficulty, low score. Room 35×30, Diff 4, Score 0.18