Matthew Taylor

Registration number 100151729

2020

Third Year Project: Report

Supervised by Dr Rudy Lapeer

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This report details an implementation of procedural generation in a video game environment.

Acknowledgements

I would like to thank Dr Rudy Lapeer, from University of East Anglia, for his wisdom and guidance.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Aim of the project

The project aimed to implement a first person video game which uses procedural generation to provide a novel and interesting game experience. The project will explore the technical aspects of using PCG as a core feature of a game, specifically, implementation and performance.

1.2. Motivation

Procedural generation is not a new field, especially in video games, but there are interesting areas that are seldom explored. One of these areas is real-time procedural generation.

The motivation behind the project's aims were to explore new and interesting concepts, namely, using PCG as a core feature in a game and using real-time procedural generation to provide an interesting experience.

1.3. Literature

The project also drew inspiration and guidance from existing PCG games, as described in papers by Spufford (2003), Welsh (2016) and Brown (2016).

The project also took into account the wealth of research done on PCG techniques, as described in papers by Perlin (1985), Ebert and Musgrave (2003) and Dormans (2010).

2. Design and planning

2.1. Implementation methodology

The project development operated under an Agile methology. Agile does not require that all requirements (or "user stories") are defined up front, instead it is encouraged that they are developed and refined during the lifecycle of the project.

The project itself was divided into Agile “sprints”. It was decided to deliver a sprint every two weeks, as this fit with the development time given to the project and providing updates to the project supervisor.

While all the stories were not required at the outset, as part of the planning, it was decided to plot rough timescales against Agile “epics”. These epics provide a high-level view of functionality and are detailed in Table 3 in Appendix D.

The epics also contain very high-level user stories, which provide a little more detail. These were decided at the project outset, then refined during development. Before any user story can be developed, it was required that more detailed requirements were written. These were then recorded and tracked on Trello.

2.2. Implementation plan

With the high-level design documented in the epics, planning of these epics against timescales was required. This timeline plan is visible in Figure 3 in Appendix A. Progress was then tracked against this Gantt chart to ensure that progress was being made and that the project could be achieved as planned.

As the project was run using Agile principles, the Gantt chart was revised to reflect changes to the plan. These changes are visible in Figure 4 in Appendix A.

Changes to the Gantt chart include the addition of a Level Generation task - the Level Re-Generator. This reflects a focus of the project on procedurally re-generating levels in real-time, so this has been split out as a seperate task and given a shorter timescale, reflecting the importance of the task to the project.

The gameplay section was split into two tasks, reflecting the focus on implementing core gameplay as a priority, with enhanced gameplay being desirable but not as critical to the evaluation of success of the project.

2.3. UML

As the project began, a UML diagram was designed to provide a high level framework to work towards. Unity does not prescribe much structure, so a UML design to begin with was useful. The initial UML design can be seen in Figure 8 in Appendix B. The iterated design can be seen in Figure 9 in Appendix B, which shows the definition of classes being enhanced with the additional methods and attributes required to support the design, as well as the addition of the **Level Re-Generator** class.

2.4. Requirements and prioritisation

The project defined epics (or, high level requirements) at the outset, with more detailed requirements written at each stage. This allows the project the be flexible, with lessons learned in the prototyping incorporated into future requirements. It also saves time, because requirements do not need to be written in advance when they would likely be changed anyway.

The epics and requirements written for the project so far are available in the appendix in Table 4 on 26.

2.5. Design prototypes

2.5.1. Guaranteed path generation

The first part of the design to be implemented required a guaranteed path through a maze to be generated. This was achieved by using a random walk - a stochastic process implmented in two dimensions to provide a definite route through the maze.

The random walk is implemented using an agent based approach. The agent can choose from three random directions to travel in - left, right or down. These probabilities are detailed in Table **??**. When the agent has chosen a valid direction, it randomly selects a valid tile.

The agent has methods of checking where it has been and where it is going next, so that it picks valid tiles that have openings in the directions it has come from and intends to go. The agent is given bounds and restrictions while it is performing the random walk, which provides a method of parameterising the level and deciding on an endpoint.

2.5.2. Decoration of rooms

Rooms are decorated with interior sub-dividing walls and other elements to provide a convincing and interesting environment to traverse. Decoration is achieved by populating each room with a random pre-defined set of geometry, examples of which can be seen in Figures 10, 11 and 12, in Appendix C.

In the prototype design, a single point is used to populate all geometry from the centre of the room. However, iterations on the design will use several points which an agent will scan through to populate things like sub-rooms, corridors, walls and desks with appropriate decorations, providing considerable variation for little asset generation cost.

2.6. Evaluation of success

A key issue with this project is evaluating if it is successful. If there is no clear notion of what success looks like, the project could progress in unproductive directions. Below are the key measures of success chosen at the start of the project to measure success against.

2.6.1. Convincing levels

The levels must appear convincing. This is difficult to quantify, but nevertheless is an important attribute. The levels (at least in the initial design) will be mazes generated in a grid, with the grid sections subdivided in different ways.

This could easily create a boring level, where the grid pattern is discernable and the rooms are boring and procedural to traverse. Successful generation of levels will avoid these issues.

2.6.2. Convincing decoration

The decoration in each individual room will contribute to how convincing and hand-crafted each level looks. As described in Germer and Schwarz (2009) and Taylor and Parberry (2010), there are several challenges to making room decoration appear to be hand-made when it is procedurally generated. Of particular focus will be generating furniture and objects in positions that feel realistic and unique.

2.6.3. Performance

Performance is another key component of the project succeeding. If the game responds poorly, it will be frustrating to play. Performance will be measured in a few ways:

- **Framerate**: must be at least 30 frames per second
- **Initial loading time**: must take less than 30 seconds
- **Smooth level re-generation**: no dip in framerate below 30 frames per second

2.7. Experimental methodology

The experimental results of the project will focus entirely on performance, as this is objectively measureable, as compared to the more subjective nature of how convincing the levels are. The experiments will focus on two main areas of performance: generation time performance and runtime performance.

2.7.1. Generation time performance

The experiments around the time it takes to procedurally generate levels will be simple timing experiments, using Unity scripting methods to provide timings of the time it takes to generate levels. These experiments will be run multiple times, with averages taken, to provide more accurate results.

The experiments will be run on different parameters, like level size, to test how effective the procedural generation implementation is at scale.

2.7.2. Runtime performance

Runtime performance will be evaluated using the Unity Profiler tool. This (by default) provides a detailed snapshot of system resources and performance metrics from a rolling 300ms window of gameplay. For the experiments, the game will be allowed to generate the level and when gameplay begins, a 5 second period of movement will be allowed to elapse before measurements are taken. This is to allow for the performance to stabilise.

These experiments will be repeated multiple times, with averages taken, to provide more accurate results.

These experiments will be run on several different parameters, comparing:

- Different map sizes
- Static vs real-time map regeneration
- Whether AIs are navigating the level
- Whether map culling is enabled

Performance tests were run using the Unity Editor, as compared to compiled version of the code. This was done for two reasons, the first being that much of the timing experiment code outputs to a debug console. The second reason is that the Unity Editor is required to use the Unity Profiler (see Unity Documentation 2019.3 (2019)) in order to access detailed performance statistics.

2.7.3. Reference hardware

To ensure consistent results, the same hardware was used for all experiments and a consistent test environment was established. The hardware used for testing was considered to be a reasonable reflection of recent PC hardware - it is not sufficiently powerful to make the results unreplicable by other users.

The reference hardware specification was:

- **Operating System**: Windows 10 Home
- **CPU**: Intel Core i5-7200 @ 2.5GHz
- **Graphics**: Intel HD Graphics 620 (integrated chipset)
- **RAM**: 8GB

The test environment involved running Windows 10 with no other applications open, except for the Unity Editor. All tests were run operating on mains power, so no power-saving CPU throttling would occur.

3. Implementation

- Screenshots of game working - Commented code - Technical documentation

4. Evaluation

4.1. Quality of levels

- Write about the quality of my levels

4.2. Level generation performance

Experiments were run to evaluate the performance of the level generation algorithm. The first experiment involved timing code being added to the level generation code, as detailed in Algorithm 1.

4.2.1. Generation code execution testing

Algorithm 1 Level Generation Timing Code

Input: Level Generation Size S

Output: Time Elapsed T

```
 1: $t \leftarrow StartTime$                      ▷ set the time the generation started
 2: Initialise
 3: Place all tile spawn points                   ▷ based on S
 4: Generate Critical Path                        ▷ adds rooms/interiors with guaranteed route to goal
 5: for $i \leftarrow 1$ to $S$ do                ▷ for each tile wide
 6:     for $j \leftarrow 1$ to $S$ do            ▷ for each tile deep
 7:         if $Tilenotgenerated$ then            ▷ check tile not on critical path
 8:             Add filler tiles                  ▷ add random tile
 9:             Add room interiors                ▷ adds random interior
10:             Add doors                         ▷ adds random doors
11:         end if
12:     end for
13: end for
14: return $T \leftarrow timenow - t$             ▷ return time elapsed
```

Varying sizes of level generation grids were used and each experiment run multiple times to provide averaged results. The generation times are displayed in Table 1.

These results demonstrate that the time taken to run the level generation script does not scale significantly with increasing the level size. However, from running the tests it was clear generating a large level (eg, 10x10 or bigger) was taking longer than the reported time of less than a second. It was observed that from starting the game to being able to play it was taking several seconds longer than this.

Level Size (in tiles)	**Number of tiles**	**Generation Time (in seconds)**
3x3	9	0.630
5x5	25	0.790
7x7	49	0.591
9x9	81	0.665
10x10	100	0.937
12x12	144	0.929
15x15	225	0.914
18x18	324	0.921
20x20	400	0.953

Table 1: Timing results from timing the level generation code of varying size of levels

4.2.2. Profiling the generation code

It was clear from the experiments that even though the generation code runs quickly, even with large levels, the time taken to having a playable level took longer. A new experiment was devised, using the Unity Profiler to gather more accurate and usable results.

The Unity Profiler incurs additional overhead from being used, however the overhead is consistent so results are still useful. The Profiler provides very detailed information, but for this experiment only one measure was considered - the time to run the Level-Generator script during the first frame of execution. An screenshot of the Unity Profiler show results from the first frame can be seen in Figure 1.

The Profiler provides more accurate data, because it measures the time taken to load geometry and textures onto the GPU. This explains the gap between the initial experiment and the observed results, because for large levels a significant amount of level geometry is required to be loaded.

The results of the profiling are shown in Table 2. They are much more realistic when compared to anecdotal evidence of observing the performance of the game.

Level Size (in tiles)	**Number of tiles**	**Profiled Generation Time (in seconds)**
3x3	9	0.595
5x5	25	1.974
7x7	49	2.965
9x9	81	5.842
10x10	100	6.754
12x12	144	12.516
15x15	225	25.183
18x18	324	53.932
20x20	400	86.276

Table 2: Timing results from profiling the level generation script execution of varying size of levels

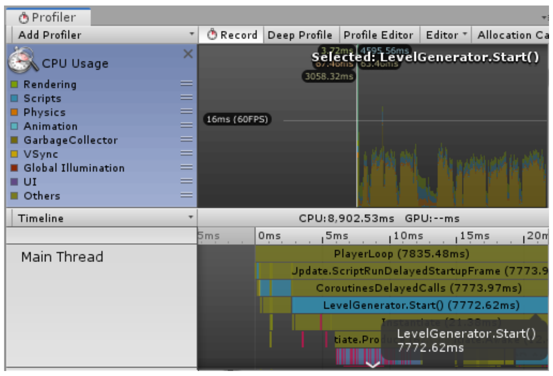

Figure 1: Unity Profiler showing execution time of LevelGenerator script

When plotted on a graph, as shown in Figure 2, the relationship between the number of rooms generated and the generation time can be seen. It was hypothesised that as the number of rooms increases exponentially, because it is based on a grid of *nxn* dimensions, that the generation time would also increase exponentially. The plotted trendline and the high R^2 value demonstrates this.

4.3. In-game performance evaluation

References

Brown, M. (2016). How (and why) spelunky makes its own levels | game maker's toolkit.

Dormans, J. (2010). Adventures in level design: Generating missions and spaces for action adventure games. pages 1:1–1:8.

Ebert, D. S. and Musgrave, F. K. (2003). *Texturing & modeling: a procedural approach*. Morgan Kaufmann.

Germer, T. and Schwarz, M. (2009). Procedural arrangement of furniture for real-time walkthroughs. *Computer Graphics Forum*, 28(8):2068–2078.

Perlin, K. (1985). An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296.

Spufford, F. (2003). *Backroom Boys: The Secret Return of the British Boffin*. Faber and Faber.

Taylor, J. and Parberry, I. (2010). Computerized clutter: How to make a virtual room look lived-in.

Unity Documentation 2019.3, U. (2019). Profiler overview.

Welsh, O. (2016). No man's sky review (eurogamer.net).

A. Gantt charts

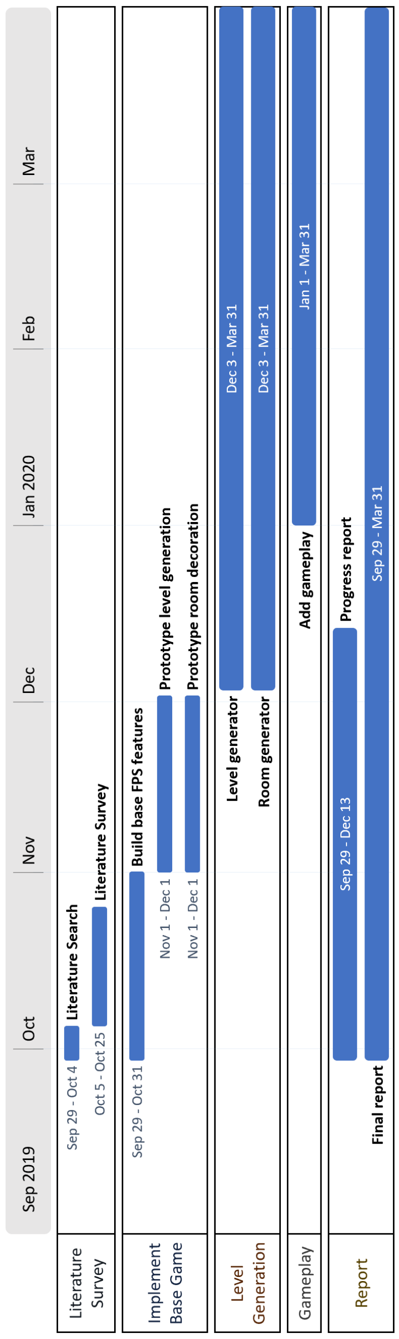

Figure 3: Gantt chart of original plan

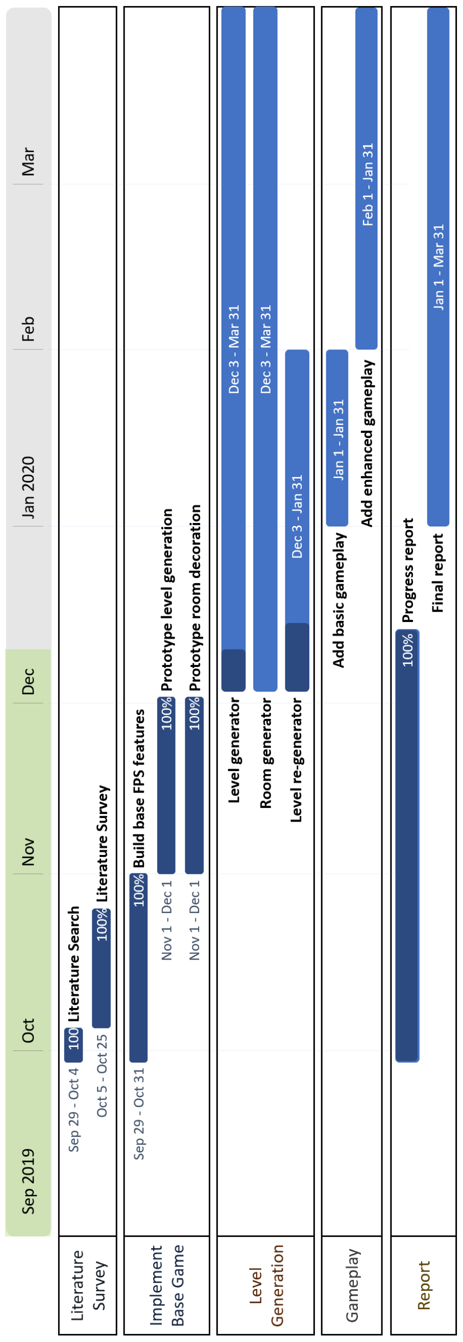

Figure 4: Gantt chart of updated plan

B. Diagrams

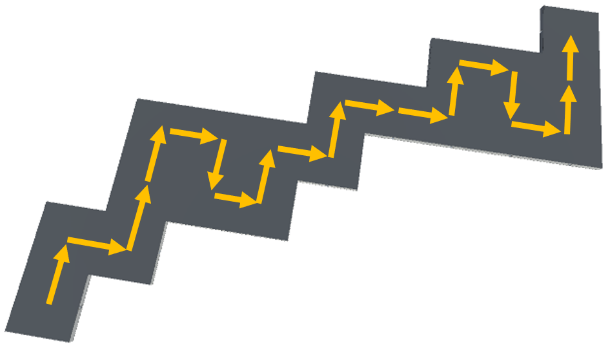

Figure 5: A top-down view of the guaranteed path generation, showing how the PCG algorithm moves in three directions to create a route through a maze

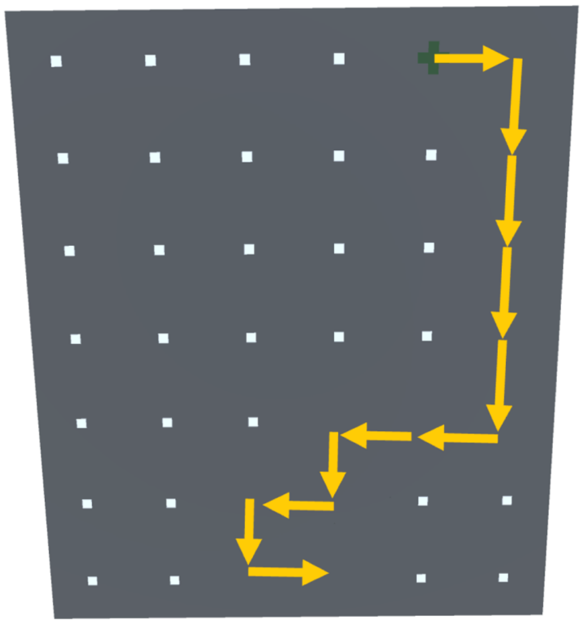

Figure 6: A top-down view of the guaranteed path with remaining tiles filled in, demonstrating how the filling algorithm works

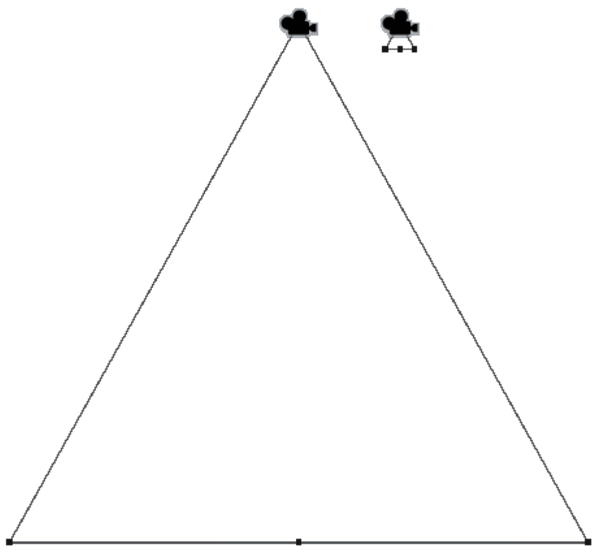

Figure 7: A top-down of the difference in default and optimised viewing frustrum sizes, with Unity's default frustrum size on the left and the optimised size for the generated mazes on the right

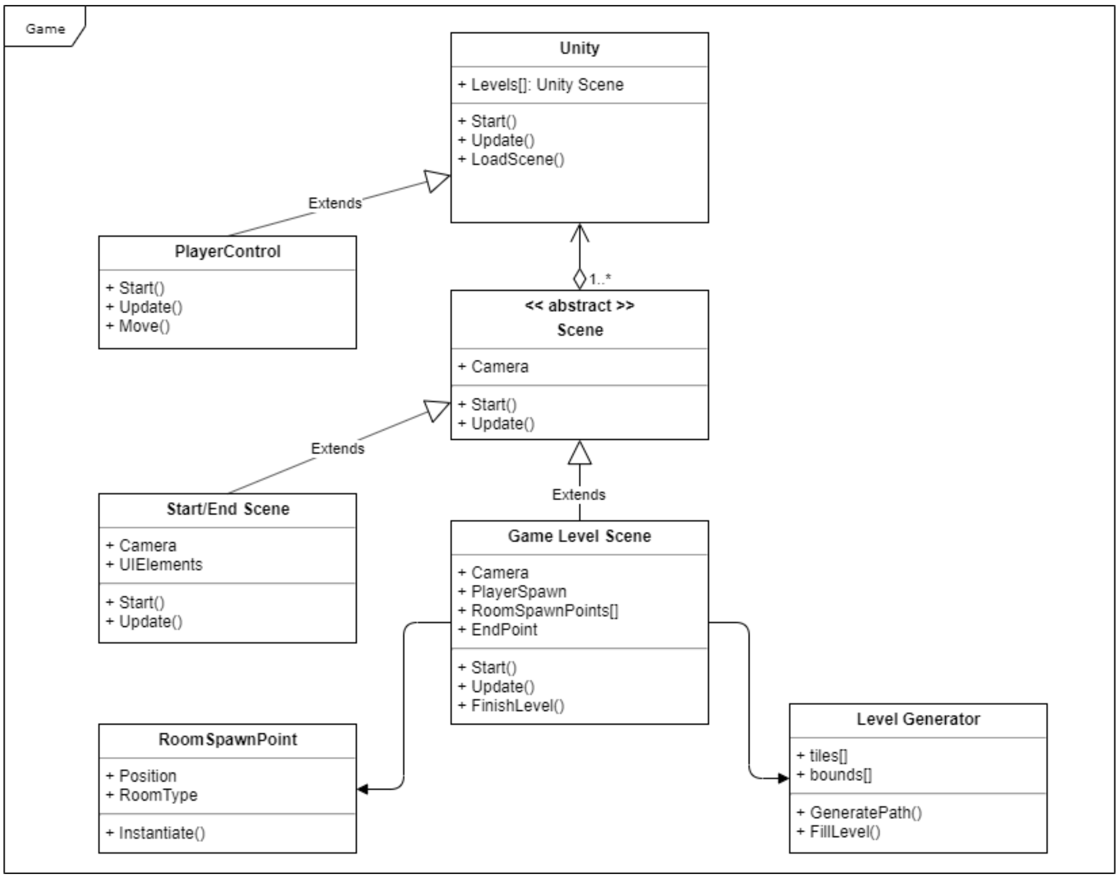

Figure 8: Initial UML diagram

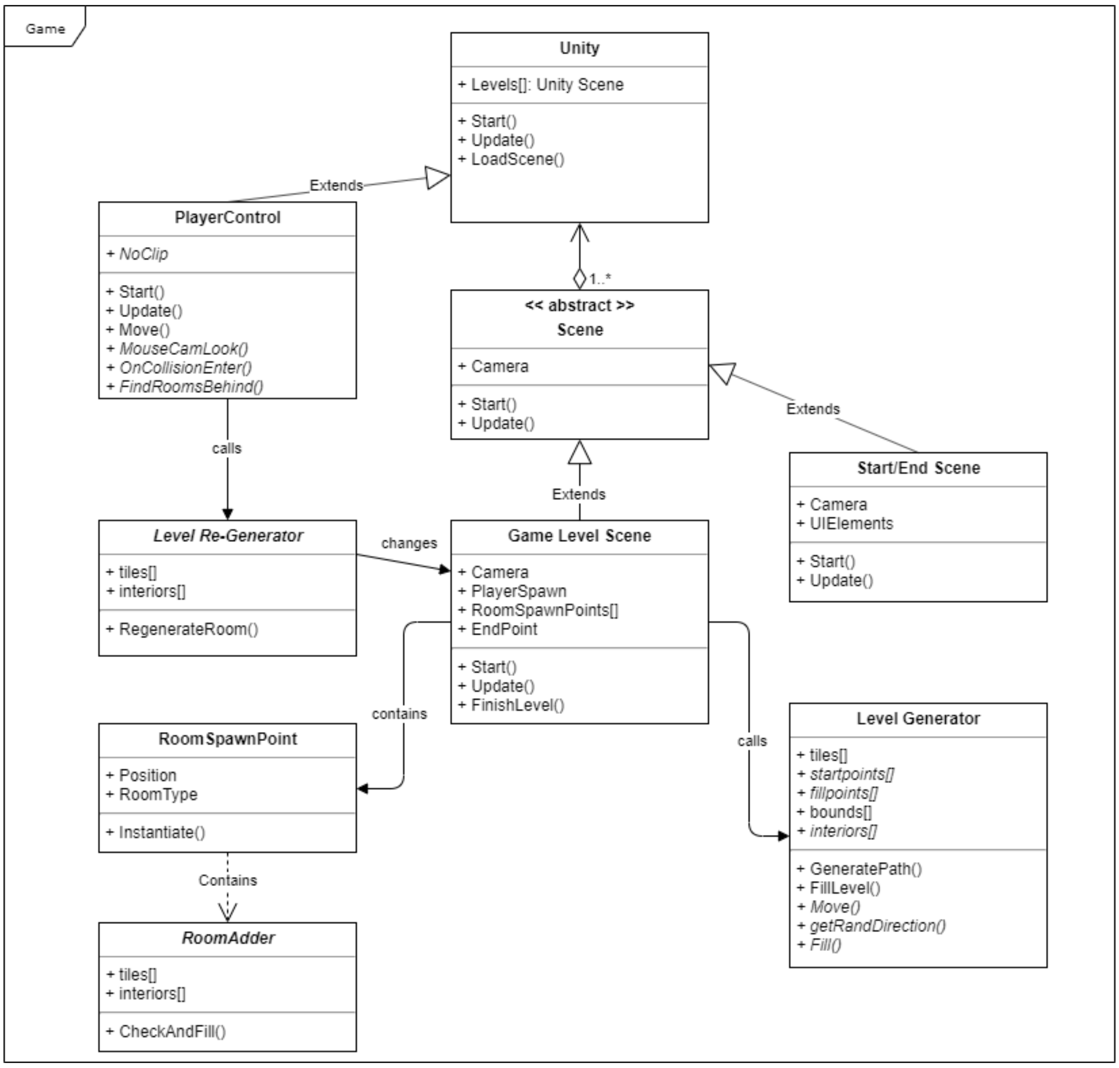

Figure 9: Redesigned UML diagram

C. Screenshots

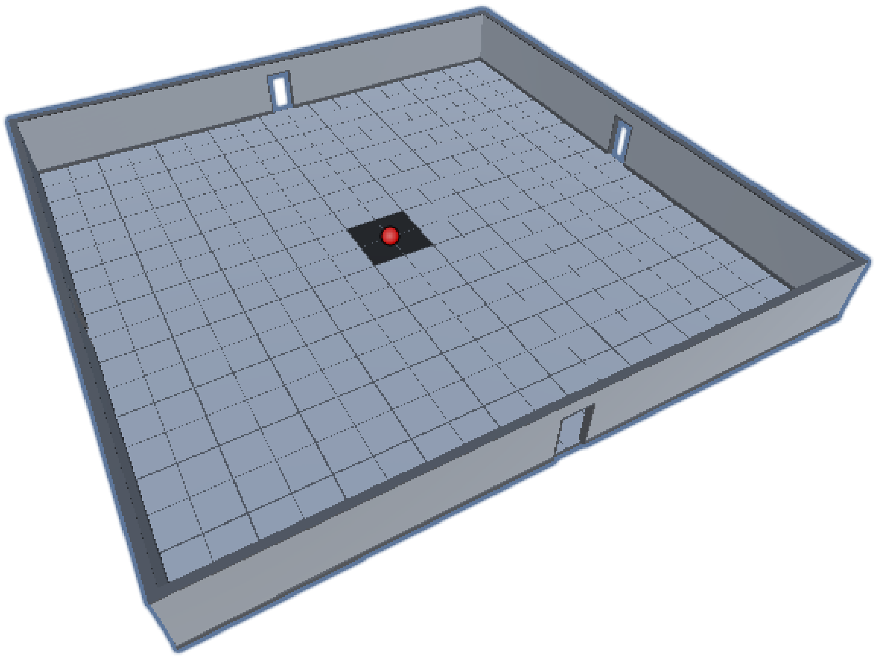

Figure 10: A screenshot of a room tile, showing the walls and exits of a tile (with the ceiling hidden)

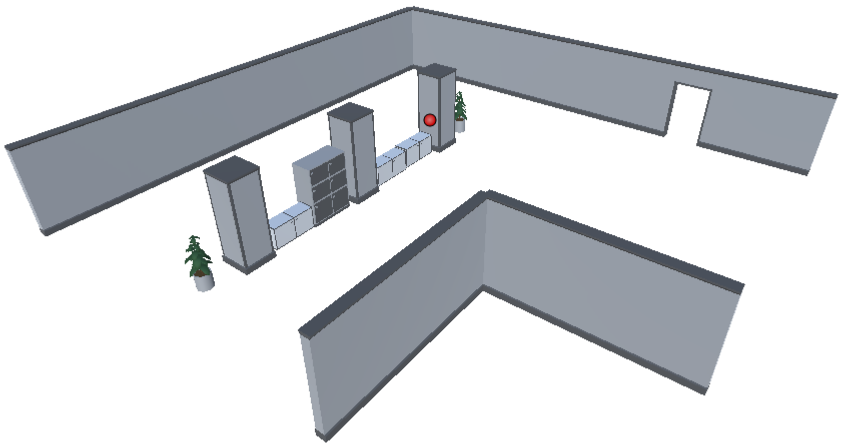

Figure 11: A screenshot of a room interior, designed to fit into any tile with any exit

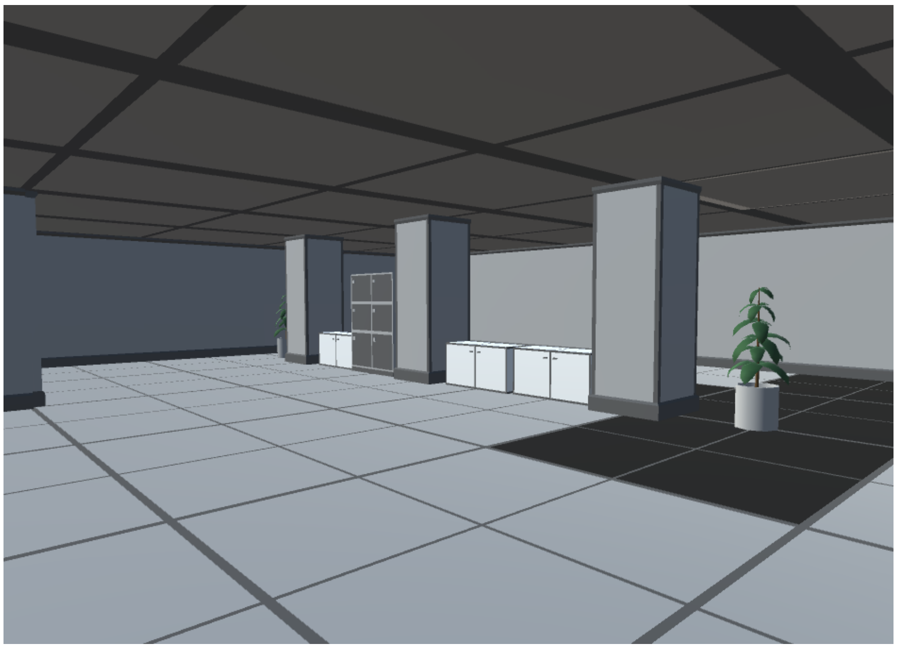

Figure 12: A screenshot of a procedurally generated room, with the tile and interior composited in a real-time environment

D. Tables

Epic	Stories
Build base "first person" 3D game features	
	Provides a basic level design
	Proves the concepts of vision and movement
Prototype level generation	
	Prove a method of generating a guaranteed path through a maze
	Prove a method of filling in non-guaranteed paths through the maze
Prototype room generation	
	Prove a method of generating room interiors
	Room generation must not interfere with guaranteed path
Prototype level re-generation	
	Design method to procedurally re-generate level sections
Add gameplay elements	
	Decide on how level is finished by the player
	Provide some interest and threat when playing
	Provide means of assisting navigation
Implement final designs	
	Use non-prototype textures
	Ensure performance meets goals

Table 3: Agile-style epics and associated user stories

Requirement type	**Requirement**	**Priority**	**Status**
Epic	Write Project Proposal	Should Have	Done
- Requirement	Research PCG	Should Have	Done
- Requirement	Write proposal	Should Have	Done
Epic	Write Literature Review	Should Have	Done
- Requirement	Research/find literature	Should Have	Done
- Requirement	Get feedback	Should Have	Done
- Requirement	Write review	Should Have	Done
Epic	Build base FPS features	Must Have	Done
- Requirement	Create title/game over screens	Should Have	Done
- Requirement	Create end goal, transitions to end screens	Should Have	Done
- Requirement	Implement first person controller	Must Have	Done
Epic	Prototype level generation	Must Have	Done
- Requirement	Design room tile prefabs	Must Have	Done
- Requirement	Implement "random walk" generation	Must Have	Done
- Requirement	Implement maze filling algorithm	Must Have	Done
- Requirement	Implement different sized tiles	Could Have	
Epic	Prototype room decoration	Must Have	Done
- Requirement	Design room decoration prefabs	Must Have	Done
- Requirement	Implement room decoration filling algorithm	Must Have	Done
- Requirement	Design random sub-decorator agents	Could Have	Done
Epic	Progress report	Must Have	Done
- Requirement	Design plan	Should Have	Done
- Requirement	Write first draft	Should Have	Done
- Requirement	Write final draft and submit	Must Have	
Epic	Prototype re-generation	Must Have	
- Requirement	Design approach for re-generating parts of level in real-time	Must Have	
- Requirement	Implement re-generation process	Must Have	
- Requirement	Implement re-generation trigger process	Must Have	
Epic	Level generator	Must Have	
Epic	Room generator	Must Have	
Epic	Level re-generator	Must Have	
Epic	Basic gameplay	Must Have	
Epic	Enhanced gameplay	Should Have	
Epic	Final report	Must Have	

Table 4: A table showing epics and requirements, their priorities and statuses

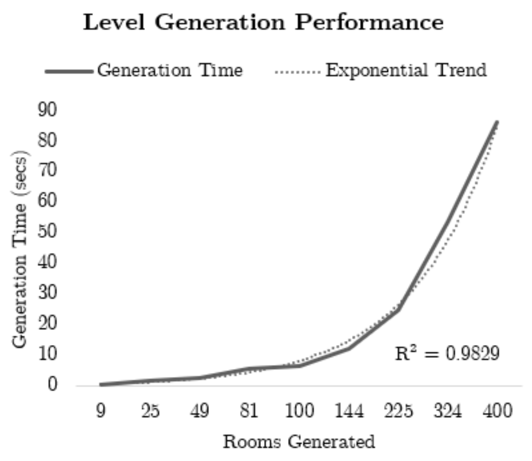

Figure 2: Results of profiled generation performance