Project Greyheaven: Preference-based procedural level generation through nearest-centroid classification in the context of game development

Radoslav M. Kirilchev

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2018

Abstract

Video games are becoming ever more popular and ever more expensive to build. Developers thus turn to procedural algorithms to reduce development time and costs; unfortunately a pseudo-random algorithm cannot replace careful design done by a professional... Or can it?

This dissertation examines some of the field of related literature and details the production and implementation of a game that uses machine learning techniques to augment the generation of its levels, adapting produced content to the preferences of every individual player, with the ultimate purpose of presenting an improved experience.

The proposed method is completely novel, though it draws inspiration from some extant works. Levels in the game – consisting of interconnected rooms – are modelled using two separate models (the *level model* and the *metamodel*), then built on the basis of a graph-structure layout, adhering to the models and a set of rigorous rules to ensure level playability.

Generated levels are then scored by their players, and learning is performed using a nearest-centroid clustering algorithm. Levels are clustered based on their optimal *playstyle*, a proposed feature of a level, varying on a per-player basis, that denotes how players engage with it. The learning system is exceedingly simple conceptually, though its implementation is complex.

The project seeks to prove that building a playable game from scratch, without relying on or modifying any existing title, is a viable approach towards creating an adaptive procedural system.

The initial evaluation of this proposed system is promising, with players giving ”adaptively generated” levels up to 35.4% higher ratings on average, as well as ”misclassifying” such levels (in cases of the player and system disagreeing about the best way to play a level) 7% more rarely. Ultimately, we conclude that despite some shortcomings of the presented implementation, such a procedural system reinforced with learning methods *does* indeed provide an improved player experience in comparison to a generic random system, is likely to be a beneficial addition to real-world projects, and can provide greater player interest and retention.

Finally, we examine what is yet to be done: given the unfortunate time constraints of an Honours project, as well as the gargantuan scope of this specific project, the examined implementation can be regarded as a ”proof of concept”, with a number of suggested improvements and directions for future development.

To summarise, this project consists of:

- *A feat of grand engineering* – creating a fully playable, *real* game from scratch, that utilises adaptive procedural generation techniques;
- *A feat of thorough research* – conducting a proper experiment and determining whether or not adaptive procedural generation performs better than simple randomness.

Acknowledgements

First and foremost, I'd like to thank my supervisor, Dr Taku Komura, for his insight and assistance during the development of the project and the writing of this dissertation.

This project received a bursary from the Royal Society of London for its potential applications in the industry, for which I remain ever grateful.

Finally, I'd like to thank all of the volunteers that participated in the testing phase, as no conclusion about the validity of my hypothesis could have been made without their honesty.

All errors, should there be any, are mine and my own.

Table of Contents

Chapter 1

Introduction: On Randomness and *Greyheaven*

This chapter introduces the reader to some current state of affairs in the video games industry at the time of writing, and discusses what *"procedural content generation"* (PCG) is and why it has been increasing in popularity over the last few years. I discuss how some drawbacks of PCG may be remedied using learning techniques, providing motivation for this project, and provide a brief summary of the *Greyheaven* project and how it fits into the big picture.

Finally, I give a brief summary of *this report*, so as to provide the reader with an immediate understanding of the chapters present.

1.1 The Need for Adaptive PCG

In this section, we'll examine the motivation for the project, relating it to current events and issues in need of solutions. I go into some detail about PCG and its potential benefits and drawbacks, and introduce the concept of "adaptive PCG".

1.1.1 The Ever-Increasing Costs of Modern Games

The video games industry has recently become larger in revenue than films and music combined. Development costs continue to skyrocket as developers create more complex experiences than ever, whilst pressed by their publishers to complete their works in the shortest possible time. At the same time, the oversaturated market and increasing prices of games creates "pickier" users that gravitate towards only playing a few titles instead of many.

Coupled with the remarkable successes of heavily-procedural titles like *"Minecraft"*, more and more developers, especially "indies" ("independent", i.e. not supported by

a publisher), turn to procedural content generation with the hopes of creating a full-featured product faster and cheaper than before. However, procedural generation has its own drawbacks, and research towards resolving them is only now starting to be done.

1.1.2 Procedural Content Generation: Reducing Costs, Reducing Variety

Procedural content generation (PCG) is a term used to describe all mechanisms that create data through algorithms instead of by hand. PCG is well-known in the world of video game design, having been a staple of some genres since 1980 (with the game *"Rogue"*, progenitor of the *rogue-like* genre.) PCG allows designers to rapidly produce content for their games using pseudo-random algorithms: for an example, instead of creating the scene of a forested area by hand, manually placing every single tree, bush, or other object in the level, a procedural system, once developed, could generate an infinite amount of forest scenes at the click of a single button, dramatically reducing subsequent development costs.

NB: The term "PCG" may be applied to every facet of a game that is generated in an algorithmic manner – this may be levels, animations, models and textures, even music or non-player character (NPC) behaviours. For this thesis, we shall focus our attention of procedural *level* generation, i.e. creating environments dynamically.

Entire genres, such as the aforementioned "rogue-likes" (and the more modern *"roguelike-likes"*), revolve around the procedural generation of entire game worlds. PCG techniques have been used in a myriad of games, especially in the recent years, and are likely to become ever more prevalent as development costs continue to increase and development times continue to decrease.

However, procedural generation often comes at a terrible cost: a lack of variety in whatever is generated, among other issues; as players are very prone to notice similarities in the experiences they're presented with. This has become exemplified in titles such as *"No Man's Sky"*, a game that promised an entire procedurally-generated universe, but suffered from an incredible repetition (and, even worse, incredibly *visible* repetition) on nearly every planet of that universe.

The truth is the truth, as painful as it might be: a pseudo-random algorithm is simply unable to replicate what a talented human developer could do. Aside from the high potential for reduced variety in the product, games relying on procedural generation often suffer from other issues. Given the innate difficulty of testing procedural systems (errors in edge-case scenarios may never appear during testing unless a random condition occurs), they may create odd or completely malfunctioning results: For an example, in level generation, a secret room with a treasure chest might be created in such a way that it cannot be accessed by the player; or, worse yet, a room containing the exit for the level might be unreachable. The player may be presented with a room containing an unreasonable number of opponents presenting an impossible challenge; the list goes on.

Additionally, games – not necessarily only those using PCG – may often provide a fixed experience that doesn't take players' individual preferences into account, in some cases eschewing even a basic difficulty setting (such as the famous *Dark Souls* series). This prevents such games, as well as other media, from reaching the largest possible audience they could.

However, PCG still remains useful, and there are ways to remedy some or all of these weaknesses.

1.1.3 The Best of Both Worlds: More Variety at a Lower Cost

One of the issues of PCG stems from the "undirectedness" of the pseudo-random algorithms that all procedural systems use – their innate *randomness*, and the designer's frequent inability to influence their behaviour, resulting in the issues described in the previous section.

What if we were to remedy this through machine learning techniques? If we could still provide enough variety so that players don't feel the experience is repetitive, and if we could make games adapt to players' preferences? Data may be gathered from the player (such as their performance and preferred mode of playing the game) at run-time, and may be used to regulate the output of any procedural system, even such outside of the video game domain.

This is by no means a groundbreaking idea by itself – research on these topics has been done in the recent past (as we'll see in the next section), and appears to be becoming more popular with every passing year. Despite this, I am not aware of *any* outstandingly successful commercial products that implement "smart" procedural techniques in existence at the time of writing. I admit that I can't be *completely* certain why that is, but I would argue that some "adaptive" solutions – whilst performing better than a non-adaptive one – simply don't perform as well as a game publishing company would want, or perhaps some suggested solutions may be impractical to implement in a real, large-scale game. Alternatively, a simpler reason may be rooted in the fact that research in this field is still relatively new, only becoming more prevalent over the last few years [13].

With Project Greyheaven, my goal is to provide a simple, useful, practically-orientated proof-of-concept for such a "smart" procedural system that performs better than a "non-adaptive" game that simply generates levels procedurally.

In the following section, we shall examine, in short, what Project Greyheaven *is*, and how it operates.

The concepts and plans outlined in this paper should be rather applicable to other domains of interest outside the field of video games, such as AI research, security and evacuation training, or VR simulations (discussed in chapter 7).

1.2 Project *Greyheaven*: A Summary

This section is intended to serve as a brief overview of the entire project. (Please refer to the next section for details about the order in which these points are covered in greater detail.)

Greyheaven is a first-person shooter game, built from scratch, that utilises procedural level generation as well as an ”adaptive” level recommendation system that uses labelled, (nearest-centroid) clustering. In short, *Greyheaven*’s objective is to learn what sort of levels the individual players prefer playing, and generate levels more likely to match their preferences.

I generate levels procedurally, using pseudorandom algorithms governed by a *level model* with 19 distinct parameters. Given the incredibly vast space of possible levels allowed under this model, the model has been condensed into the *metamodel*: a set of three distinct parameters (as well as five fixed *constraints*) used to generate the model. This allows me to reduce the space sufficiently to allow for learning to take place, whilst still retaining enough variety.

Players of the game are given levels to complete (i.e. reach a specified level exit whilst defeating enemies encountered through the level). After completing a level, they are asked to give it a rating (on the scale of 1 to 10 inclusive), as well as to specify which *playstyle* the player found optimal for this level. The game presents what it believes is the optimal playstyle as the ”suggested” choice, but only the player’s choice is taken into account.

For this paper, a *playstyle* is a discrete category of player behaviour, and essentially corresponds to the *type* of levels generated. As the game is a first-person shooter, I define three playstyles that largely correspond to the player’s preferred weapon of choice, as well as preferred range of engagement with the enemies.

Completed levels are added to *weighted clusters*, with one cluster maintained per playstyle – the level being added to the cluster of the player’s choice of optimal playstyle. Every set of level (metamodel) parameters thus becomes a node in a cluster, and the weight of this node is the rating given by the player upon level completion.

When the player chooses a new level to play, they pick one of the three playstyles – one that the level should be optimised for. The set of level parameters produced for this level is essentially the *centroid* of the respective playstyle cluster (since the three metamodel parameters can be modelled as three-dimensional vectors), plus a little random variance to ensure the level is ”new”. The playstyle suggested as optimal for this new level is the one the player selected in their request.

Alternatively, the player may pick a completely random level, in which case the system needs to know which playstyle to suggest as the optimal one. The centroids of all three clusters are calculated (their weights being the average weight of all nodes in the respective cluster), and the nearest centroid is used to select the suggested playstyle.

Note that since the clustering algorithm requires at least some levels of the respective playstyle to be completed in order to operate, all players are presented with a set of

”Calibration levels” – three per playstyle – that they are suggested to complete before using the procedural generation system. These Calibration levels are pre-defined, the same for all players, and intended to steer the clustering algorithm into operation.

This adaptive PCG system was evaluated on a sample of 23 volunteers, completing a total of over 600 levels (including the Calibration ones). Testers were split into two groups, one playing the game with the adaptive system present, and another (the control group) – using simple, non-adaptive, pre-set rules for parameter generation and playstyle suggestion. Evaluation suggests that, on average, player ratings improved by over 1 unit (from 4.44 from the control group, to 5.47 for the ”adaptive” group), or by over 1.5 units if we exclude Calibration levels (4.46 average rating from the control group; 6.04 from the ”adaptive” group) – an improvement of 35.4%.

On the basis of this, as well as other data, I conclude that the project is relatively successful, providing an example for a conceptually simple framework that allows for a better player experience in comparison with non-adaptive randomness. Naturally, the project also has its flaws, such as the impossibility of applying learned information for one player to another one; the requirement for players to answer questions at the end of every level; the need for players to complete some levels before the system can adapt. (Those, and others, shall be addressed in greater detail in due course.)

This report shall describe every aspect of the project thoroughly, and a guide for how the information is ordered is presented in the following section.

1.3 This Report: A Summary

This report is intended to serve as a comprehensive guide for the project, ranging from a review of the literature, through an elaboration on all choices made about the design of the game and the adaptive system, through detailed descriptions of the implementation and evaluation of the adaptive PCG level generation system.

In this section, I provide brief summaries of the chapters to come, in the hopes that this may serve as an introduction to the copious amounts of material.

- **Chapter 2** is a short review of some relevant literature. Due to the relative scarcity of material in this specific domain (level generation for FPS games), the chapter isn't the longest.
- **Chapter 3** sets the scene for the project, providing further motivation for its existence, as well as elaborating on *why* certain important decisions were made – such as why *Greyheaven* is a stand-alone game, and why the specific genre and setting were chosen.
- **Chapter 4** consists of an overview of what the initial plans for the implementation and evaluation of the project were, and what had to be changed. Section 4.1 covers the original plans for the human-computer interaction aspect (i.e. the ”core gameplay loop”); section 4.2 presents a pragmatically-motivated description of what exactly PCG systems needs to be present in the game; and section

4.3 details how the plans for evaluating the adaptive systems changed during the development process.

- **Chapter 5** provides highly detailed descriptions of the implementation and functionality of all PCG systems present in the project, as well as information about game mechanics, operation, and a more in-depth look into the clustering algorithm. Section 5.1 describes the operation of the implemented room and level generators, and provides the equations that ”map” level models to metamodels; section 5.2 discusses the facets seen by the players (i.e. the ”game mechanics”) and how they were designed; and section 5.3 describes how the level recommendation and playstyle suggestion ”learning” algorithms operate.
- **Chapter 6** describes the evaluation process and provides an analysis of the collected data, as well as a final summary of the conclusions, and some discussion on the flaws of the presented implementation.
- **Chapter 7** discusses future work, both in terms of gameplay mechanics in need of improvement, as well as potential improvements to the research system.

NB: It is additionally worth noting that this document allows itself some liberties in terms of writing style. This was done due to the rather *large* size of the project, and thus the numerous elaborations it requires, to hopefully allow for more engagement with the reader than would otherwise be possible.

Chapter 2

Literature Review

In this section, we perform an abridged, yet critical examination of the existing literature that focuses on similar issues as those tackled by this project. I discuss the paper that inspired this project, and motivate my decision to engage in a similar, yet different, investigation.

The field of procedural generation in itself is exceptionally broad, with research having been done on the generation of virtually everything one may find in a game – from materials and textures for objects [3] to entire role-playing games [4]. Additionally, research on procedural content generation has become significantly more popular over the last years. Thus, attempting to cover the entirety of the field would be folly – instead, we shall go over several sections of relevant works, that are of interest to this particular project.

2.1 Dynamic Difficulty Adjustment

An obvious use case for procedural content generation arises in the context of *dynamic difficulty adjustment* (DDA) – applying machine learning techniques to regulate a game's difficulty, ensuring that players are presented with sufficiently challenging experiences that are nonetheless manageable and not overwhelming.

Whilst *Greyheaven* doesn't *focus* on DDA per se, it is a stand-alone game, and features an aspect of DDA – a single level parameter determines the toughness of enemies spawned (discussed in several locations, including 5.3.3).

A particular case of DDA that is of interest for this project is the "Hamlet" system presented by Hunicke [5], which adjusts the number of ammunition and health pick-ups provided to the player in a custom first-person shooter game based on the *Source* engine. The paper suggests that using DDA techniques improves players' actual performance, but without improving players' *impression* of performing better.

A small part of the initial plan for this project involved experimenting with this proposed system, but unfortunately, that did not come to pass.

An interesting example of a DDA system at work in the real world can be found in the first-person shooter *SiN Episodes: Emergence* (2006) by Ritual Entertainment, which features dynamic adjustment of the number of spawned enemies as well as player and enemy damage outputs; unfortunately, whilst the game was positively received, the studio was subsequently acquired by another company and nothing else became of it.

Whilst I would regard the DDA term to explicitly denote techniques for adjusting aspects of difficulty unrelated to actual *level* generation, in truth it is a broad term, and any technique that directly or indirectly modifies difficulty can be considered a case of DDA.

With that stated, I shall simply discuss some works relating to level generation without partitioning them into sections.

2.2 Procedural Level Generation

As described in the previous chapter, PCG has been quite widely used in the domain of video games, for all variety of games. Research about adaptive PCG – about procedural level generation in particular – however, isn't as broad.

A vast majority of the explicit level generation investigations address two-dimensional platformer games, creating levels for known titles, usually the classic *Super Mario Bros.* The majority of papers also seem to favour evolutionary techniques.

Pedersen et. al. [9] model *Super Mario Bros.* players' emotional states using neural networks to model three emotional criteria; Shaker et. al. [11] employ grammatical evolution to create levels for the same game, using fitness functions, and optimising results based on the same three criteria of the players' "emotional states", utilising previously-recorded models of player behaviour. Jennings-Teats et. al. [6] present a system for dynamic level generation (for the same game) using statistical modelling of difficulty obtained from small "play-traces" and comparing those to the player's current performance.

Research also focuses on procedural levels for two-dimensional side-scrolling games, specifically designed for mobile devices and more "casual" (not serious) players [8].

Whilst most algorithms revolving around 2D platformer games ultimately intend to be generalisable to other forms of games (and similar human-computer interaction-focused applications with a need for procedural generation), and pose interesting ideas, I have not attempted to replicate any of them directly for *Greyheaven*.

A particularly interesting paper is one by Yannakakis & Togelius [13], in which they put forth the *experience-driven procedural content generation* (EDPCG) framework, suggesting a generic approach to solving problems of a similar nature. Whilst intriguing and thought-provoking as an alternative to what they describe as *search-based PCG* (SBPCG) in an earlier paper [12], the methods utilised for *Greyheaven* ultimately fall within the broader scope of SBPCG.

One ambitious project of relative interest, a relatively recent work by Lopes et. al. [7], proposes a novel method for specifying the design constraints of the procedural framework, by attempting to abstract away as much of the details as possible, leaving high-level tools for concept definition in the hands of the designers. Whilst very exciting and interesting, their framework should be best applicable for larger development teams, and suggested a greater separation between learning and content generation. The focus of *Greyheaven*, at least in its current state of a proof-of-concept, was on a more ”tighly-coupled” solution, and on the mostly unexplored world of PCG for first-person shooters.

There are two more papers that need to be addressed; one of them is responsible for the existence of this project, and the other addresses a point I at one time considered. For this great contribution of theirs to history, and from standing out from the rest, they deserve to be examined more thoroughly.

2.2.1 First-Person Shooter Map Evolution

Of particular interest for *Greyheaven* is another paper as well – again by Yannakakis and Togelius amongst others (Cardamone et. al. [2]), concerning the use of evolutionary algorithms for the purposes of ”evolving” maps for a first-person shooter (FPS).

This paper echoes some of the points about automatic level evaluation that shall be discussed later in this thesis, but I didn’t know of its existence until very late during the project’s development.

Whilst interesting – with the coincidence making it even more interesting – the paper’s goals and the goals of *Greyheaven* are fundamentally different. Cardamone et. al. optimise maps for FPS levels in a ”team deathmatch” contest, where players (and non-player bots) kill each other and respawn until a timer runs out; *Greyheaven* focuses on a more ”directed” approach, with levels having purpose and direction.

Additionally, the approaches undertaken by *Greyheaven*’s level generator are completely different; furthermore, the entire *Greyheaven* game is created from scratch, instead of being a modification of an existing title.

It’s worth noting that the authors of this specific paper state that theirs is the first work ever done for FPS. It seems that research on PCG, focusing on FPS games in particular, is a scarce commodity indeed.

2.2.2 ”The Quake Paper”

The most important paper I have come across – in fact, the paper which inspired this project – is called *”Learning-Based Procedural Content Generation”*, by Roberts & Chen [10]. This paper, that I usually refer to as *”the Quake paper”*, is of extreme interest, and thus it shall be discussed at greater length.

Roberts and Chen introduce an adaptive level generation framework, building a modified version of the classic FPS game *Quake*. Unlike the majority of other works (including the EDPCG briefly discussed earlier), this framework doesn't rely on *search-based* procedural generation, but is instead called *"Learning-Based Procedural Content Generation"*, and seeks to provide an alternative approach to the issues faced by adaptive PCG systems.

I should note that whilst this paper inspired my project, I have not attempted to replicate the framework described within. *Greyheaven* focuses on a different approach for adaptive PCG.

Roberts and Chen correctly make the point that player *styles/types* need to be identified for the successful generation of suitable levels, and that these styles ought to capture all possible player behaviours; this idea gave rise to my own concept of *playstyles*. However, they make the point that asking players for feedback directly should be avoided, given that this is detrimental to the "flow" of the game for players. Whilst this is a point I agree with in principle, *Greyheaven*, as essentially a proof-of-concept, is not afraid to ask its players what their opinions are. (This is addressed as a point to be considered for future development in chapter 7.)

The authors then provide a detailed description of the intriguing process their framework entails for building and evaluating procedural levels in several stages, a process not followed at all by the implementation outlined in this thesis.

They proceed to utilise a variety of learning techniques to extract information about games, in a manner far more complex than what I strove to achieve. I shan't describe the methods, as they are quite specific and are of no relevance to this project. It's worth noting that the authors use clustering at one point – and clustering is what this project utilises – but that is not a great surprise, considering the variety of methods they employ, and the different use they have for their clustering technique.

An interesting point of the paper, which I considered during the evaluation of my own system, was the fact that positive level ratings and difficulty were apparently correlated, at least in their experiment.

Despite the semi-deliberate lack of much congruence between my work and the one in "the Quake paper", I still regard it as highly important – as without it this project would have most likely not existed at all, and believe it to be an interesting read.

Chapter 3

Project Greyheaven: The Concept

This section serves as a high-level introduction to the game (the titular *Greyheaven*), as well as a description of what it's intended to be, how it's supposed to work, and how one would evaluate its hypothetical benefits over a fully-random system. I also include brief discussions of the choices made when selecting genre, mechanics, and gameplay elements.

3.1 The Goal: ”Smart” Procedural Level Generation

The goal of Project Greyheaven, as mentioned, is to provide a working implementation of a game with an adaptive (or ”smart”; I shall use these terms interchangeably) procedural level generation system that would, should all go well, provide more enjoyment to players than a generic PCG system would.

To make that happen, we need to have a game. For this project, I've decided to create a game of my own from scratch; an argumentation of why I made that choice may be found in the next section of this chapter.

We thus turn our gaze upon *Greyheaven*, the game I created to facilitate the completion of the project's goal. We shall firstly explore the concept of the game; the next two chapters (chapters 4 and 5 respectively) shall guide the reader through more in-depth information about the game's design, creation, and changes with time, and shall present thorough information about the operation of all systems found within the game.

But for now, let us discuss the *concept*: The core concept behind *Greyheaven* is simple. In itself, *Greyheaven* is a *first-person shooter* (FPS) game, inspired by the ”old-school” style of FPS games characteristic of the 1990s and early 2000s. (A brief introduction to the genre can be found later in this chapter, in section 3.3.1.)

In unison with games of that era, and contrary to the increasing popularity of ”open-world” games, *Greayheaven* has its players be presented with separate levels. Players navigate these levels, potentially completing some sort of objective, then locate the ”exit” and proceed to the next levels. All levels are procedurally generated.

Whilst the players are interacting with the game, completing objectives and levels and defeating the enemies that block their path, the game gathers data about their behaviours, and uses it to formulate a model of what sort of levels they prefer.

Finally, the game generates those levels that it has determined the player would prefer, adjusting various parameters of the procedural level generation algorithm, e.g. level size, variety of enemies, etc. Should the learning process have gone successfully, this is bound to improve the satisfaction of the player with the product, and thus improve user retention, and, most likely, sales (in a real-world commercial context).

3.2 A Game from Scratch: More Work, yet More Flexibility

As visible from most related work cited in the previous chapter, the usual *modus operandi* when performing this sort of research is to build a ”smart” system on top of an existing game, ”modding” (modifying) it in some way. However, *Greyheaven* was planned as its own game from the very beginning, and has been created as such for a number of reasons: despite being *exquisitely* more demanding from a development standpoint, creating a game from scratch offers much greater control and flexibility over all aspects of development, and over all facets of the finished product. I shall elaborate on these points shortly.

Before that, however, I’d like to address the significant potential shortcoming of a ”modded” system (i.e. one built over an existing game): it imposes arbitrary constraints on the development of the adaptive system. The researcher has to comply with various restrictions and specifics imposed by the nature of the game modified, and by the ”application programming interface” (API) that she uses to interact with the existing game, no matter what they may be. For an example, an adaptive level generator for an adaptive system built on top of *”Super Mario Bros.”* could need to generate its levels in difficult ways, perhaps even through third-party ”level-creation” applications, or inelegant practices such as overwriting game files in ways the original developers never indended for. (**NB:** This is a hypothetical example; I have no knowledge about the mechanics of implementing a learning level generation system for *”Super Mario Bros.”*) Furthermore, a researcher may find himself constrained in the data he may collect, if the game was not created with the intent of collecting specific data about the player and their performance.

Additionally, existing games are, a lot more often than not, the intellectual property of their respective developers or publishers, which may render the practical use of a proposed implementation quite impossible, if, for an example, it would be illegal to distribute the modification to a large amount of users. Naturally, this varies on a per-application basis.

Many researchers use open-source titles and extend them to serve their purposes; whilst this would remedy any potential legal issues, it doesn’t help with the other potential issue, and instead raises the question *why concern oneself with a particular game’s*

engine limitations and built-in quirks instead of creating one from scratch?

Though the two points above may not seem to pose much of a problem, I would certainly disagree. Indeed, modifying an existing game is significantly easier and faster, but I would dread having to cope with arbitrary restrictions and potential (though perhaps unlikely) legal issues. It is mainly for those reasons that I chose to develop *Greyheaven* from scratch, despite this having required circa two and a half thousand workhours more than it otherwise would have. That is certainly more than the allotted for an Honours project, isn't it...?

At any rate, I believe that creating a game from scratch, despite introducing the significant problem of *creating a game from scratch*, is a better approach. It is understandable that researchers in general may perhaps feel intimidated by the substantial amount of work needed to produce a playable game; and would prefer to instead focus on solving the problem at hand by working with what they can. However, I believe that this may be remedied if the researchers have experience in game development, and are willing to go the extra miles to make it happen. Fortunately, yours truly *has* such experience, which allowed me to approach the idea the way I did.

Creating a truly custom game, "tailor-made" for the problem at hand, allows the researcher full and complete access to any and all aspects of the game, of the procedural system, and of the machine learning methods used for the adaptation. Furthermore, it allows for the potential future adaptation of the system, with arguably greater relative ease, to other applications that might be of interest, perhaps even ones not related to video games. (Some suggestions will be explored in section 7.)

3.3 The Case for Careful Constraints: Genre, Mechanics, Theme

Creating a game is no easy feat. There's an infinitesimal variety of kinds and types of games to choose from, in terms of genre, mechanics, and setting (theme). *Greyheaven* was conceived as a first-person shooter with a science-fiction aesthetic; in this section we'll examine *why* that is.

3.3.1 The First-Person Shooter: Everyone's Favourite

The first-person shooter is one of the most popular genres of video game in the world. Every year, multi-million dollar franchises such as *Battlefield* or *Call of Duty* release newer and newer titles in their series; and series such as *Doom*, *Quake*, *Unreal Tournament*, *Half-Life*, or *Counter-Strike* have been familiar to players for a long time, in some cases since the last two decades of the previous century.

FPS games are played by millions of people on a daily basis, and are – at least in my own opinion – so popular that any person that has a vague notion of what video games are knows, or at least has seen, what a first-person shooter is. For this very reason I am

tempted to eschew providing a description of the genre itself; but for utmost clarity, or perhaps a reminder to the reader, I shall do so.

As is evident by the name, a first-person shooter game gives ”first-person”-type control to the player, i.e. the player is in direct control of the camera, as if it were his or her respective head. This is illustrated in figure 3.1. As can be noticed from the images, FPS games can offer a multitude of different themes and settings. However, as long as most of the combat – another heavily-featured element in FPS games – mostly occurs at range (which may arguably not be the case for the pictured *Dark Messiah*), they may be considered a *shooter*.

Additionally, most, if not all FPS games, share the same core mechanics: the player is given a variety of weapons, and is tasked to eliminate a number of foes whilst traversing a three-dimensional environment, and the weapons usually consist of multiple different projectile-based configurations with an overall similar behaviour (terms known to players include *recoil, spread, aim down sights, hip-fire*, and so on). This allows players, once they’ve played *any* FPS game, to quickly and easily become familiar (if not proficient) with another game of the same genre.

The aforementioned popularity, combined with the inherent mechanical ”similarity” of FPS games, outlined just above, renders them a very suitable choice for building *Greyheaven* as. Theoretically, finding players shan’t be a difficult task (given the popularity of the genre), and interested players would most likely have experience with any other FPS game, this knowing what they’re supposed to do almost immediately. This constitutes the reason for my choice of creating *Greyheaven* as a first-person shooter.

3.3.2 Theme and Setting

As seen in figure 3.1, FPS games – like most other games, really – can be set in a plethora of settings, and may have any of a plethora of themes. The choice of setting mostly depends on what the game’s designers have in mind. The most popular settings for FPS games appear to be historical (usually World War II and more modern), futuristic (”near-future” or fully science-fiction), or modern (present-day combat).

For *Greyheaven*, I chose a science-fiction themed setting that would allow me some flexibility when creating levels, choosing enemy types, and creating weapons. The specifics of these shall be discussed in later chapters; I believe that the general choice of theme is not an important facet of the project, and that any theme would have been of similar use. An example of the visual style of a room in *Greyheaven* is pictured in figure 3.2.

However, there is an aspect of the theme and setting which I consider vital: the specific choice of environments, i.e. *what exactly* the procedural level generator is supposed to create. Every individual theme provides a likely infinite amount of variety in itself, as should be painfully obvious – ”science-fiction” is too broad a term to describe the specifics of a game. Is it set in space or on a planet? On Earth or somewhere else? Perhaps on spaceships or space stations? Both? Are there aliens? Will they be the

(a) *Battlefield 1* (2016): The player, as an Ottoman sniper armed with a German *Gewehr 98* rifle, overlooks a courtyard of the Ottoman fortress at Al-Faw. *Battlefield 1* is a modern FPS with a historical World War I theme (yet is historically inaccurate). (The protanomaly colour deficiency filter is in use in this screenshot.)

(b) *Dark Messiah of Might and Magic* (2005): The player, as a young man on a mission to retrieve an ancient artifact, aims a bow at a hostile orc archer, overlooking the ruins of a temple of Asha, the Spider Goddess. *Dark Messiah* is a fantasy-themed FPS/first-person hack-and-slash that incorporates a variety of intriguing mechanics.

Figure 3.1: An example of two FPS games with vastly different themes and settings.

player's enemies? What exactly is science-fiction-esque in the whole ordeal? And so on.

For *Greyheaven*, once again mostly due to the constraints of its development – remarkably limited time-frame, no other developers, no budget – and since creating a game is a monumental task, though it doesn't appear so for the casual observer, I decided to keep it simplistic: an indoors setting with no outdoors segments. In other words, one may regards the entire game as if taking place in a building. Limiting the level generator to indoors levels (i.e. a series of interconnected rooms) allowed me to achieve all I wanted to do within the allotted time-frame, even though the final product isn't the most inspiring of settings.

(The level- and room-generation systems themselves shall be discussed in detail in the next chapter, namely in section 4.2.)

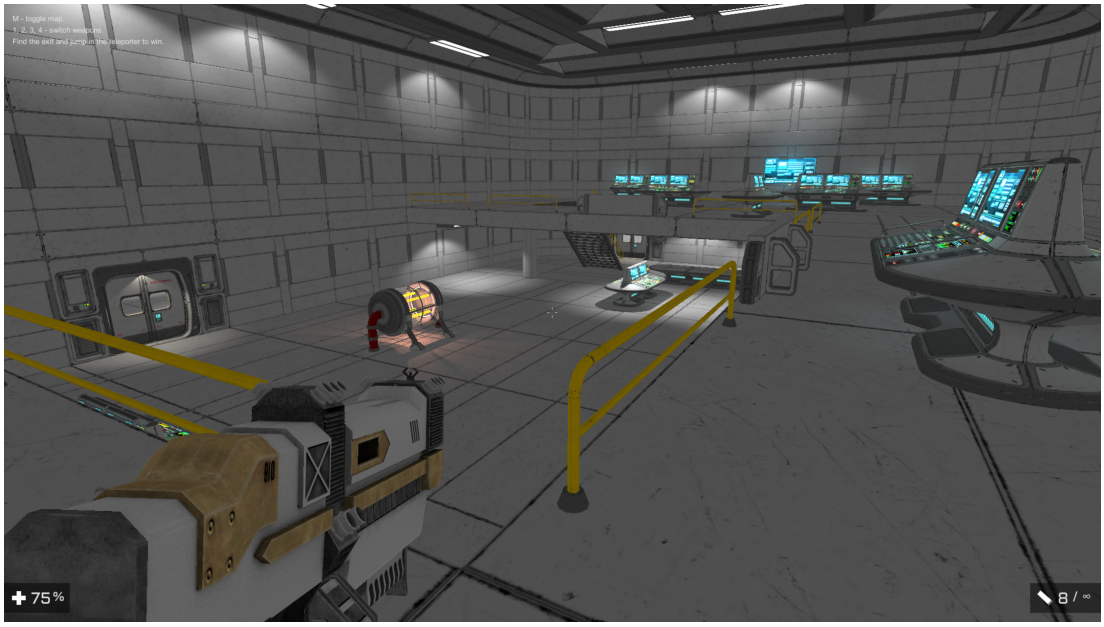


Figure 3.2: *Greyheaven*: The player, holding a bulky shotgun, overlooks a typical, moderately-sized, procedurally-generated room. The sci-fi setting is reflected in the use of futuristic weapons and props (objects, i.e. computer terminals, modern lights, automatic doors, and mysterious reactors). The process that allows us to generate rooms like this will be examined in the next chapters – theoretically in section 4.2; and the implementation in detail in section 5.1

Chapter 4

Initial Design; Subsequent Revisions

In this section, we take a critical look at how the level generation system was originally envisioned, and how it would operate to provide a vast array of possible results. Afterwards, we discuss the issues I encountered with this design, and how I had to resort to simplifying some concepts.

4.1 An Overview of the Design Decisions: Weapons, Enemies, Playstyles

As discussed in the previous chapter (namely in section 3.1), we need the output of our procedural level generation (PLG) system to be modified by the behaviour of an individual player. For this to happen, we need (a PLG system, discussed in the next section of this chapter, and also to) to come up with a way to determine what the given player's preferences are. For this, I defined the concept of discrete, mutually-exclusive *"playstyles"*. In this section, we shall examine what they are and how they tie the mechanics of the FPS game together.

NB: This section is theoretical and attempts to convey my thought processes and explain what (and why) certain decisions were made. An in-depth examination of the *implemented* operation of the facets described here, including visual examples, can be found in the next chapter.

The word "playstyle" is generally accepted to mean "the manner in which somebody plays". In the world of video games, this meaning is retained. Additionally, it is usually used to denote the specific behaviours that a player may naturally gravitate towards. However, I have decided to take this word and give it more concrete meanings within the context of *Greyheaven*.

For *Greyheaven*, a "playstyle" largely correlates with the player's preferred weapon, and determines what sort of foes the level spawns for the player. Those three aspects are tightly connected, so we'll discuss them in order in this section. We shall firstly

examine the weapons, then the enemies players may encounter, and finally list the playstyles and what they represent.

As discussed in the previous chapter, *Greyheaven* is an FPS game; and one of the most important aspects of an FPS game is the weapon selection. Weapons have existed since time immemorial, and there is an infinitesimal amount of weapons a designer of an FPS may come up with and give to his players. However, since this project suffers from a *very* limited development time, I opted to stick to the classics that all players have used time and time again, and should thus be immediately familiar with, instead of devising creative and more unique weapons. (Naturally, a game that provides *both* creative and fun weapons, *and* an adaptive PLG system is likely to be an even greater hit among players.)

At the very start of a *Greyheaven* level, players are given *all* of the following weapons:

- a **sabre** – deals high damage to enemies and requires no ammunition, but serves no purpose outside of mêlée range; (initially conceived to be able to deflect rifle shots, giving it additional defensive properties, but this was never implemented;)
- a **shotgun** – rather potent at close range, but rapidly loses its usefulness at longer distances;
- an **assault rifle** – with a quick rate of fire, but poor hip-fire (unaimed) accuracy, as well as a lack of aid whilst aiming at long-range, plus a rather specific damage profile (technical details are discussed in chapter 5), the rifle is designed to perform optimally at medium range.
- a **sniper rifle** – providing extremely high damage per shot, especially at longer ranges, coupled with a slow fire rate, abhorrent hip-fire accuracy, and a zooming scope, every FPS player (and likely every other person in existence) knows that the role of the sniper rifle is to be unparalleled at long range combat.

An aspect of FPS mechanics that I was confident I'd like to implement was the presence of the "aim down sights" functionality – more characteristic of modern FPS games than classical ones. In games that allow for aiming down sights (ADS), the player may hold down a key to *aim down the sights* of their weapon, usually improving weapon accuracy at the cost of movement speed and peripheral awareness. "Hip-fire" accuracy – mentioned above – thus refers to weapons being fired without ADS.

As will likely have become obvious, all these weapons (with the exception of the sabre and the shotgun) fill a substantially different niche in the player's arsenal. It is thus easy, though arguably not terribly clever, to equate potential player "playstyles" to their preferred weapon, i.e. to their preferred range of engagement. In turn – and in combination with a specific choice of foes, as we'll see in the next paragraph – it becomes easy to portray what levels might be adequately "optimised for" a given playstyle (and what levels we ought to generate, should that be the player's favourite playstyle).

An integral part of any FPS game is the roster of enemies the player faces. Whilst this is a field with a potential almost as vast as that of weapons, I've proceeded the same way as outlined above for the weapons: defining a small list of enemies fulfilling

different roles, and with different strengths and weaknesses. The list of ”niches” one may need, given the set of weapons defined above, can be the following:

- **Mêlée enemies (Enforcers)**: Ones that engage at close range, charging directly towards the player, and are optimally defeated by the sabre or the shotgun;
- **Medium-range enemies (Riflemen)**: Ones that keep to medium range, stick to cover, and are best defeated by the assault rifle;
- **Long-range enemies (Snipers)**: Ones that keep as far away from the player as possible, are often taking cover in hard-to-reach places, and are thus best defeated using the sniper rifle;
- **Impervious to bullets (Shieldbearers)**: The initial idea was to have an enemy type that is very well defended against bullets, requiring the use of the sabre to dispatch. A slow-moving, mêlée-attacking enemy with a lot of hitpoints (durability), bearing a massive shield, came to mind. For all their protection from ranged weapons, Shieldbearers would be vulnerable to sabre strikes. **(This enemy type had to be revised dramatically in the final implementation; more details in section 5.2.2.)**

Those are the four different types of foes that may be found in *Greyheaven* at the present time. We shall examine the specifics of their behaviours, statistics, and implementations in chapter 5.

With our weapons and our foes already in place, we may proceed to examining the playstyles. They, along with what sort of levels one would presume would be more naturally ”fit” for them, are listed below:

- **Close Quarters**: Optimal for shotgun (and sabre) usage. Focusing on short-range engagements. Enemies consist of Enforcers and Riflemen, as well as Shieldbearers on higher difficulties. Levels optimised for this playstyle would have smaller rooms with more props (objects) to be used as cover by the player;
- **Assault Rifle:** Optimal for assault rifle usage, as evident by the name. The focus is on medium-range engagements with an enemy selection consisting of mostly Riflemen and perhaps a small amount of Snipers. Rooms in levels corresponding to this playstyle would be moderately large, with longer sight-lines than Close Quarters levels, and a moderate amount of cover for both enemies and the player;
- **Sniper Rifle:** Focusing on long-range engagements versus a majority of hostile Snipers. Rooms are very large, providing long sight-lines and improving the benefits of sniper scope usage. The amount of props and cover is low, making levels more dangerous to traverse.

It’s worth noting that I initially planned to have a fourth, ”mêlée” playstyle, corresponding to heavy use of the sabre, whilst relegating the shotgun alone to the ”Close Quarters” playstyle. However, it was not entirely clear how one would distinguish between the two, and how a player would do the same, thus that idea was dropped, and the number of playstyles was reduced to three.

Those three playstyles, i.e. the player’s preferences for each of them, would determine

how levels are generated for the player. In other words, our only task – with this system in place – would be to determine what is the player's favourite playstyle – whether they prefer the risky, yet exciting tactic of "getting up, close, and personal" with enemies using the shotgun or sabre, whether they prefer to keep at range, relying on the versatility of the assault rifle, or prefer standing back, picking foes off at long range using the sniper rifle.

Note that players have all four weapons available at all times. This increases the difficulty of the problem (if the level is considered optimised for assault rifles, yet, due to the randomness of the procedural algorithm, the player is having great success with the shotgun, do we recommend "close quarters" or "assault rifle" levels to the player afterwards?), but restricting the arsenal would prove immensely frustrating for players; I can state this from experience. Furthermore, giving players the entire arsenal allows them to be properly capable of tackling unintended "edge cases" (such as the aforementioned example), which sometimes occur when procedural algorithms are concerned.

4.2 Level Generation and its (Hyper)Parameters

An (arguable, subjective) beauty of procedural algorithms is that it's easy to get simple results with them, yet the more complexity one adds, the more fascinating their outputs become. For the purposes of *Greyheaven* (generating interesting, varied levels), I had to design a rather complex level generator that is capable of producing a dramatic variety of different levels. In this section, we shall examine the basic design considerations that a procedural level ought to adhere to, and I shall describe the level generation models I produced.

NB: This section is theoretical and attempts to convey my thought processes and explain what (and why) certain decisions were made. An in-depth examination of the *implemented* operation of the level generator, including visual examples, can be found in the next chapter.

As established in the previous chapter, *Greyheaven* is a first-person shooter taking place entirely indoors, with levels comprised of interconnecting rooms. Since the game is three-dimensional, it needs to model vaguely realistic environments.

4.2.1 On Rooms

As with many things in life, adherence to the KISS ("Keep It Simple, Stupid!") principle helps us retain our sanity when designing something as massive as this. I shall start from the simplest possible configuration of a "room", then iteratively add elements I'd like my levels to contain.

What *is* a room? A space, usually in the form of a rectangular parallelepiped, bounded by walls, a floor, and a ceiling. We can thus represent a room in terms of a three-dimensional matrix of `boolean` values – a `True` value represents the presence of a

wall, floor, or ceiling, and a `False` represents empty space. (In fact, we can represent a complete level using a sufficiently large matrix!)

An individual ”value” in this hypothetical matrix shall be henceforth called a **cell**.

However, we need to differentiate between floors, walls, and ceilings. There are also different types of cells with walls (depending on the rotation of the wall). Rooms also need doors. We thus replace the notion of a `boolean` matrix with a matrix that can contain arbitrary values, where different values correspond to different types of cells. **(This is not a completely truthful representation of the algorithm I used, but the specifics of that shall be discussed in the next chapter, in section 5.1.1.)**

Since the rooms are three-dimensional, they can have different ”floors”. The player needs to access these floors, so we also introduce staircases to rooms. Furthermore, rooms need lights, especially if they won’t have windows (and ours won’t).

What we’ve got so far can describe rooms relatively well, save for one exquisitely important detail – these rooms would be completely empty. In the previous section, I discussed the differences in *cover* that different levels need to provide. Ergo, we need to add some *props* (a game developer’s term for any ”loose” objects found in a scene) to our rooms.

And whilst we’re talking about props, let’s consider what props we want to have. Since it’s a science-fiction setting, we could have computer terminals, flat-screen monitors on walls, generic tables, beds, and storage lockers, and perhaps more ”industrial” objects such as pipes and power generators. Naturally, the potential for prop variety is close to unlimited, however those props listed are the ones found in the presented implementation of *Greyheaven*.

Additionally, it doesn’t make much sense to have a room full of beds, tables, *and* industrial pipes all the same time. We can thus define the concept of a ”room theme”, whereby each room has a ”theme”, and every theme includes only sensible props. Exempli gratia: a room denoted as a ”living space” may contain beds, tables, lockers, and a small amount of computers or monitors, whereas a room marked as ”labs” may contain multiple computers, ”server cabinets”, power generators, and pipes.

(If you ever thought creating games was easy, I’d humbly ask you to reconsider.)

Long story short, I created and implemented a complex algorithm that takes all these variables (as well as multiple others) into account, generating a single room: the *room generator*. The total amount of parameters this generator requires to generate a room is equal to **24**. (We shall examine the operation of all systems in detail in the next chapter.)

The parameters for these rooms, however, are determined on a room-per-room basis by the *level generator*. Whilst the room generator is responsible for creating one individual room, the level generator bears the burden of deciding what the level will look like and what rooms it’ll consist of. For this initial exploration of how the level generator came to be, we’ll follow the same procedure of considering what a level is.

4.2.2 On Levels

So what *is* a level then? A series of interconnected rooms, as previously discussed. The first one is the ”entrance”, where the player is spawned at level start; the final one is the ”exit” that the player must navigate to.

But how many rooms are there in the level? We need to know. We also want to know how large individual rooms are, and how many routes there are to the exit, i.e. how open the level is. Are all rooms going to be positioned on a route to the exit, or are there going to be any ”extra” rooms – ”dead ends” that serve no purpose in the player's journey towards the end, except perhaps containing some amount of health or ammunition for them?

How vertical are levels going to be? Will they ascend or descend in space, requiring the player to effectively ”climb” or ”descend”, or maintain a relatively horizontal state? And in terms of a horizontal direction, are the level's rooms mostly arrayed along a straight path, being fairly easy to navigate, or do they wind around each other in various directions, creating something more akin to a maze?

How are the props going to be distributed among a level's rooms, and how many of them are we going to have? What enemies will be placed in this level, and how difficult are they going to be? Finally, are levels *always* going to be the same, or may we allow a smidgen of chaos to keep things varied?...

In the end, I devised the so-called *level model*, consisting of **19** effective parameters that determine a level's unique characteristics. Most of these parameters are `float` values ranging from 0 to 1 (i.e. percentages), but some take on discrete values, and others range from -1 to 1 instead. **(We shall examine the operation of the level generation algorithm in detail in the next chapter: in section 5.1.2.)** This model was able to provide quite sufficient complexity and variety of outcome. All that was left was to learn how the levels produced by these models correspond to our established playstyles.

However, it quickly became clear that attempting to do learning on the parameters of this model may be *a little* too much in terms of complexity. As per the advice of my supervisor and project coordinator, who suggested that this aspect of the task I've set for myself may indeed be too ambitious given the constraints of the project, I decided to reduce the complexity of the issue to a manageable level.

4.2.3 The Metamodel: An Abstraction for Level Models

For that purpose – reducing the vast space of level variance allowed by the level models – I created the *metamodel* – a structure that dramatically reduces the number of available parameters (and, unfortunately, the amount of potential variety in the spectrum of levels generated by it) by essentially placing multiple level model parameters under the control of a single metamodel hyperparameter, and ”extracting” some parameters away from the task as ”constraints”. A visual representation of the one-to-many mapping from the metamodel to the model can be found in figure 4.1.

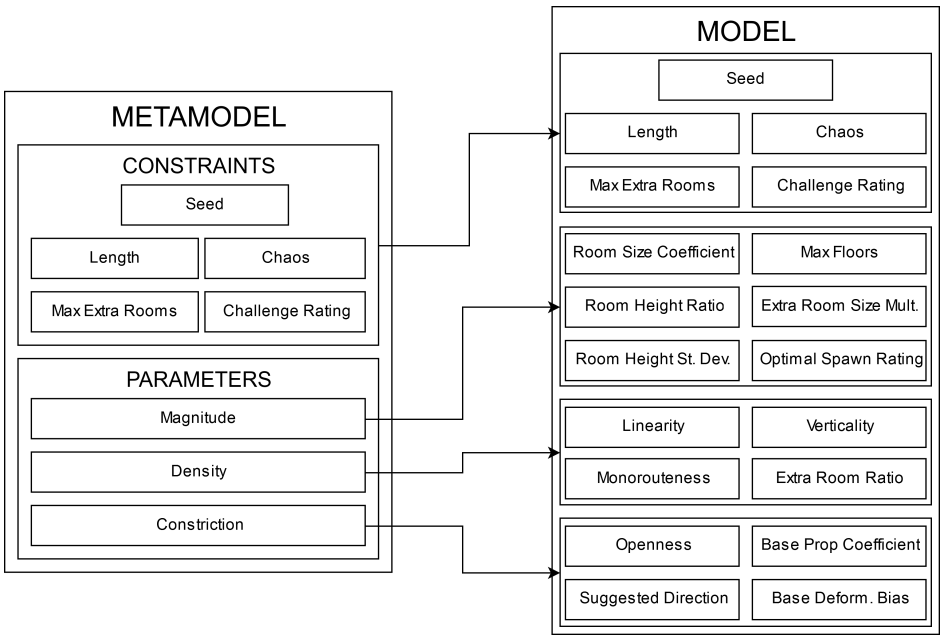


Figure 4.1: A visual representation of the relationship between the metamodel and the level model it generates.

Levels are still generated using a level model, but the model itself is created using the handful of parameters present in the metamodel. This reduces the complexity of our learning by a significant margin, by essentially leaving only three (or four) parameters whose correspondence to a playstyle needs to be learned.

However, before we discuss learning techniques, let me elaborate on the specifics of the metamodel, and the differentiation between constraints and parameters.

There are some parameters of a PCG algorithm that are best kept the same throughout all configurations generated with it. Those have been retained as the ***constraints* of the metamodel**:

- **Seed**: A seed is a value that is used to initialise a pseudo-random number generator. Use of the same seed in multiple experiments ensures that the output sequence of the generator will be the same. For an example, the first five numbers of the output of the default C# random generator (`System.Random`, numbers from 0 to 9 inclusive) will *always* be the sequence of $\{0, 5, 5, 4, 9\}$ if seeded with the value of 1911, and $\{0, 8, 5, 3, 0\}$ if seeded with the number 416. Since we aim to generate levels based on different parameters, and we would like to allow for the possibility of replaying previous levels, it is wisest to keep the seed a constant value.
- **Length**: The number of rooms between the starting point and the exit room. The level generator is complex enough that keeping this value constant continues to generate vastly different levels; moreover, setting the length of the level as a constraint is sensible considering we want to evaluate the performance of the system – which implies generated levels should be roughly ”similar”, and there would be a *dramatic* difference between a level with a length of 3 and one with

length equal to 12.

- **Chaos**: In both the room and level generator, *chaos* is a variable used to introduce randomness independent of the (constant) seed of the pseudo-random generator. As a floating-point value between 0 and 1, I made the choice to keep it a constant across all generated levels, fixed at 0.1. This was done for the same reasons level length would be kept constant, and its effects are negligible.
- **Max Extra Rooms**: As discussed briefly in an earlier section, levels may generate a different number of routes to the exit (up to three, all of which of length equal to the *length* constraint). However, there may be some other rooms not on a route to the exit, i.e. ”extra rooms”, which are usually a ”dead end”, with just a single door, and don’t serve much purpose for the player. The maximum number of these rooms is limited accordingly, mostly to prevent player frustration.
- **Challenge Rating**: How difficult the spawned enemies in the level should be. Note that whilst considered a constraint, this one is special, as it is adjusted according to the player’s preferences and feedback.

The three ***parameters* of the metamodel**, all of which are floating-point values ranging from 0 to 1, in turn, are the following:

- **Magnitude**: Generally, how large the rooms of the level are; higher values correspond to larger individual rooms. Also the most important factor in determining what enemies should be spawned in the level.
- **Density**: In general, how ”directed” a level is. Higher values produce less routes to the exit, less vertical displacement, and less extra rooms, i.e. a ”denser” level.
- **Constriction**: How ”constricted” spaces in the level are, with higher values producing levels with more props and less open floors. Also governs directional deviance of rooms for no particular reason.

Again, levels in the game are generated using a full ”level model” as usual, but the level model itself is created through the ”learnable” metamodel. The specifics of this process are described in the next chapter.

Leveraging the simplified power of the metamodel, and thus having obtained the means of creating a reasonable yet powerful PLG system, we may now focus on the aspects of *learning* – the way to satisfy the actual research goals of this project:

4.3 Evolution is NOT a Speedy Process

”Search-Based Procedural Level Generation”, as it’s frequently named [12], and that the principles of which *Greyheaven* ultimately follows, relies on techniques for searching through the possible content space provided by the PCG systems. Evolutionary algorithms are a frequently-chosen candidate for this sort of work, in this domain of research.

I must confess that I am fascinated by genetic algorithms. My (not quite) initial idea was to automatically assess the fitness of all possible levels for all possible playstyles, then use evolutionary strategies (based on a work of Thomas Bäck's [1]), to produce optimised level configurations for the playstyles in order to present them to the players, essentially *learning what the parameters do...*

But that was folly. In this section, we shall examine *why* that was, and I shall provide some initial information about what type of learning technique I opted for instead.

So far in this chapter, I've more or less described the design decisions made for my procedural level generation system, and have discussed my choice of weapons and enemies. However, two most important questions remain to be solved: *How do we learn what the player likes? How do we learn what levels fit which playstyle?*

Initially, I decided to focus on the second question, believing that the first one is less relevant if we're able to always know which levels "fit" which playstyle.

4.3.1 Automatic Level Evaluation Using NPCs

Allow me to first explain how one may automatically evaluate the "fitness" of a level for a given playstyle. There is a relatively simple answer, provided by a wonderful idea I had: use a "bot", an "AI agent", to do the job instead of a player.

In the world of game development, "artificial intelligence" (AI) is a term used to refer to the procedures utilised by *non-player characters* (NPCs), i.e. the enemies, in our case, though the term "NPC" encompasses any friendly characters as well. (The term "bot" may also be used interchangeably, especially if regarding a hostile NPC.) NPCs behave as ordered by whatever the developer creates; they usually don't *learn* anything, at least not in the context that a research may be prompted to think about upon encountering the term "AI". This is an important distinction to make!

For clarity's sake, I shall only refer to such computer-controlled characters as NPCs.

Back to automatic evaluation: We may use an NPC agent in the player's stead, scripted to behave similarly to a player – that is to say, the agent spawns at the start of the level, where the player is spawned; then proceeds to navigate through the rooms, defeating all enemies present before proceeding to the next room, with the ultimate goal of reaching the exit. Unlike a player, however, this agent is given a single weapon (corresponding to the playstyle we're evaluating: shotgun for Close Quarters, assault rifle for the style with the same name, and the same deal with the sniper rifle), and his performance in the level is given a score according to the following equation:

$$FS = T + D - (K \cdot 50) - (L \cdot 1000) \tag{4.1}$$

where FS is the final score of the NPC agent for the given level, T is the elapsed time between level start and finish (in seconds), D is the amount of damage taken by the agent, K is the amount of foes killed by the agent, and L is a boolean variable, equal to 1 if the agent has successfully reached the level exit (and equal to 0 otherwise).

The result of this equation is calculated either upon the agent's successful completion of a level, or upon his unfortunate death. Lower scores are better.

In all other aspects save for the arsenal, the NPC agent traversing the level was made to be the same as the player. This scoring metric appeared to work reasonably well, given the clear distinction between playstyles, and the inadequacy of some weapons at some ranges, i.e. the shotgun in combat versus enemy snipers.

Unfortunately, other aspects of the "automated evaluation" strategy were deemed inadequate, as shall be discussed below.

4.3.2 The Infeasibility of Grid Search

My initial supposition was the following: The PLG can produce different levels by varying the metamodel parameters; parameters which range from 0 to 1. It is not difficult to see this as a parameter optimisation problem and consider that a potential strategy could be to perform an exhaustive "grid search" through all parameters, perhaps at intervals of 0.05, and evaluate the "fitness" of every produced level (using the technique detailed above). This would allow us to effectively "map" the three-dimensional space produced by "every" possible configuration of our three metamodel parameters.

That way we would begin to know which levels are best suited to which playstyle, and would be able to recommend them to players without the need for any learning to be done on the players' own machines (i.e. would be a form of "off-line" learning).

However, even with the aforementioned 5% intervals (stepping through every parameter by starting at 0 and incrementing it with 0.05 at a time), we obtain a total of $20 \cdot 20 \cdot 20 = 8000$ possible level configurations. Every level needs to be evaluated for every one of the three playstyles, thus increasing the number of tests to 24000. And last, but not least, the "AI" library I use in the game is non-deterministic, ergo capable of producing vastly different results every time. An average of ten "runs" per configuration would be useful to combat the effect of randomness, leaving us with a final value of **240000 experiments to conduct!**

At this point, dear reader, you are probably asking how long such an experiment takes. Unfortunately, due to the behaviour of the Unity engine, and due to the system not being optimised for any sort of parallelisation, the *fastest* I could force my NPC to complete levels was *one minute per iteration*, on average. This gives us a remarkable 240000 minutes required, which equates to 4000 hours, which is equal to **166 ⅔ days**.

I may be a simple undergraduate with a penchant for tackling overly ambitious projects, but I don't consider myself *completely* insane; it was clear that this would be a foolish strategy to attempt.

4.3.3 The Unfortunate Infeasibility of Evolutionary Strategies

As stated in the introduction of section 4.3, I enjoy evolutionary algorithms. I would have much liked to use evolutionary strategies – indeed, like a lot of the papers in the field do [12] – to determine which levels are best for which playstyle, but this strategy suffered from the same drawbacks that applied to grid search.

Evolutionary strategies, or genetic algorithms in general, attempt to provide an improving and "evolving" solution to a problem through processes inspired from real evolution in nature. In short, an experiment begins with a randomly-selected *population* of individuals, all of which are evaluated for their fitness. After determining which ones perform best, and which don't, we perform *selection* (using various different strategies that shan't be discussed here), and choose individuals amongst the population that would "breed" to produce the next *generation* of individuals. Genetic algorithms also introduce the aspect of *random mutation* into the equation, sometimes providing a bit of extra randomness into the next generation. The process is then repeated from the start using the next generation, with the hopes that following generations shall continue improving their aptitude towards problem-solving.

It should be relatively obvious how I would apply this to the level evaluation problem: we select an initial population of e.g. 100 levels, and evaluate them for a playstyle. We need 10 "runs" per level (for the same reasons described in the "grid search" section above), which is 1000 experiments. This is multiplied by 3, as we need to do it once per playstyle, which leaves us with **3000 experiments**, a number far lower than the 240000 discussed previously.

Unfortunately, this is all needed for a *single generation*. Every subsequent generation would add **an additional 3000 experiments** (i.e. 3000 minutes of computational time) to this; and whilst I could posit an educated guess that it *may perhaps be possible* to have the effects of evolution become apparent before 80 generations (i.e. the amount of computational time required by the aforementioned grid search algorithm), I sincerely doubt that 80 generations would be enough for anything at all.

This rends us of the possibility to use evolutionary algorithms, which is quite the shame, really.

4.3.4 The *Feasibility* of Clustering, Without Automation

What our failures heretofore described have taught us is that automated level evaluation using NPCs takes too much time and is thus unfeasible. What do we do then? We can't ask the players to play 8000 (or more) levels! They would be long gone before that. The solution (or rather, *a* solution) lies with clustering.

Clustering algorithms are one of the most well-known among the whole "family" of machine learning techniques. Whilst usually regarded as a method for unsupervised learning, in some cases clustering algorithms can prove rather useful during supervised training.

My final choice for a learning mechanism lay with a supervised, nearest-centroid clustering model, tied closely to player feedback, and using no automated testing. Instead of answering the second question (of the two detailed at the start of section 4.3) – as both automated strategies above attempted to – I decided to focus on the first one instead: *How do we learn what the player likes?*

The answer is surprisingly simple: we *ask*. After all, we would have done that anyways. *Asking* means that we don't know everything to begin with – as the other two strategies would have wanted, but that we actually *learn*; it also means that we indeed adapt to the preferences of *every* player, instead of adapting to what we believe is a certain way of playing the game.

In short, the system keeps track of three weighted clusters per player profile – one per each playstyle. After the player completes any level, they are asked to give it a rating (on the scale of 1 to 10), and to state which playstyle of the three they found most appropriate, according to their own tastes, for this level.

The level (the set of its parameters, as a three-dimensional vector) is then added as a node to the respective cluster selected by the player, and its weight is the rating given. This gives us a set of "labelled" clusters, their nodes weighted according to this specific player's preferences.

When suggesting a level for this same playstyle, the algorithm recommends the set of parameters "located at" the centroid of the cluster, plus a small amount of random deviation to ensure a unique level.

Alternatively, if the player selects a random level (one with parameters generated completely randomly), we need to determine which playstyle would fit it best (according to the player). For this, we can find the centroids of all three-clusters, then "classify" the new level according to nearest-centroid distance.

This results in a simple, efficient, "always on-line" system. Unfortunately, it also has flaws, such as the fact that players need to complete some levels before the system begins giving proper recommendations; requiring players to pick a style and rating when completing levels; and so on. The clustering algorithms are discussed in greater detail in section 5.3.2.

Chapter 5

The Game Never Ends in a Draw: Implementation Details

Here, we examine in greater detail what the final product – the playable game – consists of, what functionality it provides to players, how its systems were implemented, and how the game itself tracks player activity and adapts to their preferences.

A point I would like to underline and make as clear as possible, dear reader, is that the entirety of this system – room generation, level generation, models and metamodels, and the learning approach – has been built from scratch, for the express purposes of being used in this project, at the cost of *thousands* of work-hours. I would ask that this be kept into consideration when it comes to the several cases of a supreme lack of elegance that I shall describe in this chapter, and that were unfortunately inevitable during the implementation process.

NB: This chapter is lengthy, as it describes all systems present in the game in great detail. The first sections concerns itself with the generation of rooms and levels; the second one discusses the specifics of the gameplay; and the third one describes the operation of the aforementioned playstyles, as well as the clustering techniques used for learning.

5.1 Different Levels, Different Challenges

In this section, we shall examine in great detail the implementations of the room and level generators, as well as the exact way in which a metamodel creates a full level model.

We shall do this in an ”ascending” manner, starting with the process of room generation, then examining how levels build and use these rooms, and finally cover the equations for generating a full level model from a metamodel. I have chosen to order them in this way, so as to hopefully introduce the reader to the different uses of the parameters, a comprehension of which should ideally help one’s understanding of why the respective values and equations for them were designed.

5.1.1 The Room Generator

For our purposes, an individual room is the smallest self-sufficient unit of interest. Rooms are essentially built independently of each other, and every room has its own layout, theme, set of enemies, and objects (as described in section 4.2.1).

Here, we shall examine the process of room creation, and the behaviour of the aforementioned room generator responsible for it, in greater detail.

Firstly, we shall go over some ”axioms” of the system I created:

1. As previously discussed, rooms consist of ”cells”. Every cell is a ”box” (rectangular parallelepiped) in 3D space with a side length of 6 metres and a height of 5 metres.†
2. Rooms may consist of multiple ”floors”, thus providing a vertical element to gameplay. If a room has more than one floor, it creates staircases that allow all floors and platforms to be reached by the player.
3. Rooms are built according to a set of parameters supplied by the level generator.
4. Furthermore, the room ”creates itself” in world space, thus relying on being given exact position coordinates by the level generator.
5. Rooms are connected to each other using doors‡, the locations of which are specified by the level generator.
6. Every room spawns its own ”squad” of enemies, which patrol it and engage the player if necessary.

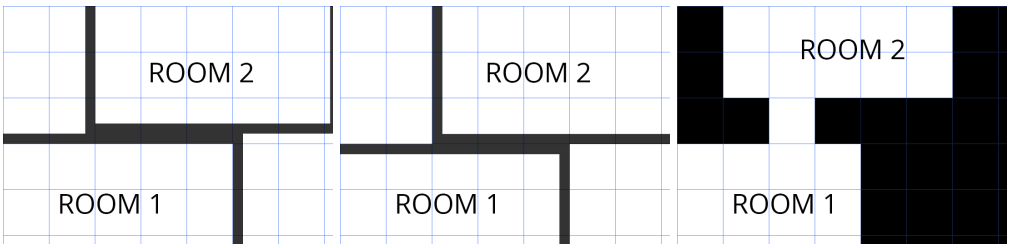
Note †: The game world assumes realistic metric dimensions for objects and characters, e.g. the player character and enemies are about 170-180 cm in height.

Note ‡: The initial design allowed for *corridors* to connect non-adjacent rooms. This shall be reflected in the implementation; yet the feature had to be scrapped late in the development process of the project, for reasons that shall be explained accordingly.

5.1.1.1 On Walls and Their Placement

Firstly, however, I must present this minute musing over the nature of walls. I can think of three distinct ways to deal with the placement of walls in a cell-based environment such as the one *Greyheaven* uses; they are highlighted in figure 5.1. We examine them as follows:

- **”External” walls**: The cell is 6 metres wide, the wall is half a metre thick and is positioned *outside* the cell. As can be seen in the figure, this creates alignment issues and is practically unfit for purpose;
- **”Internal” walls**: The cell is 6 metres wide, the wall is half a metre thick and positioned *inside* the cell. This reduces the effective free space of the cell, but prevents any alignment issues;
- **”Wall cells”**: Instead of taking special care about where to place walls, we can simply force some cells to be ”walls” instead; however this makes ”thin” boundaries between rooms impossible. Whilst arguably easier to implement and



(a) External (b) Internal (c) Wall cells

Figure 5.1: Three different ways to place cell walls. The blue grid denotes cell boundaries in the implicit matrix; Black lines denote walls. Usage of ”external” walls causes alignment problems and is infeasible. *Greyheaven* uses ”internal” walls.

manage, I was reluctant to use this method because of its (non)aesthetic ramifications.

Initially, *Greyheaven* used ”external” walls, but I quickly came to realise the impossibility of keeping cells within their theoretical boundaries, ergo aligning correctly at all times. Thus the game switched to ”internal” walls, which removed these issues.

5.1.1.2 Room Parameters and Their Effects

Despite their relative ”independence”, rooms are built in adherence to a set of parameters supplied to the room generator by the level generator.

Parameters shall be listed with their full names, along with their type and interval of possible values.

- `size` : `Vector3i` – The size of the room in ”cells”;
- `numFloors` : `int` – How many floors there shall be in the room;
- `originPosition` : `Vector3` – The global position of the room’s origin point;
- `chaos` : `float` $[0,1]$ – How much randomness to introduce during generation. 0 = total order.
- `floorLayout` : `FloorLayout` – What the layout of the room’s floors is to be. (Options discussed below.)
- `floorsAreIdentical` : `bool` – Whether all floors shall have an identical cell layout or not.
- `openness` : `float` $[0,1]$ – How ”open” a room with multiple floors will be. In other words, how large will the open vertical spaces in the room be; how much space will be left for ”open air”.
- `linearity` : `float` $[-1,1]$ – Whether the room is to keep its ”direction”, or reverse it when picking a direction for the room’s ”exit” door.

- `suggestedDirection : Direction` – The explicit suggested direction of the room's exit.
- `verticality : float` $[-1,1]$ – Whether the room is to ascend or descend, i.e. position the exit above or below the level of the entrance.
- `lights : LightsConfig` – The configuration of the lights that shall be placed along the ceiling. (Options discussed below.)
- `mainEntrancePos : Vector3i` – The position *index* of the "main" entrance of this room.
- `doorPositions : List<Vector3i>` – The list of position *indices* of all doors the room must create.
- `doorHolePositions : List<Vector3i>` – The list of position *indices* of all *door holes* (i.e. cells without walls, but without creating a door) the room must create.
- `corriDoors : List<Vector3i>` – The list of position *indices* of all *corridor doors* the room must create (they look differently).
- `noCeilingPositions : List<Vector3i>` – The list of position *indices* at which no ceiling must be created (for a vertical corridor connection using a lift).
- `numStaircases : int` – How many staircases there should be in the room; 0 uses as many as is appropriate.
- `stairsLayout : StairConfig` – How the staircases should look. (Unused.)
- `challengeRating : float` $[0,1]$ – The challenge rating to take into account when spawning enemies.
- `optimalEnemySpawnRating : float` $[0,1]$ – The optimal enemy spawn rating (a scoring metric for different *types* of enemies) for this room.
- `enemySpawnDensity : float` $[0,1]$ – The "density" of enemy spawn points, in terms of the ratio of "cell where an enemy may spawn" to the number of all adequate cells in the room.
- `maxEnemies : int` – The maximum number of enemies this room may spawn.
- `theme : RoomTheme` – The theme of the room. (Options discussed below.)
- `propCoefficient : float` $[0,1]$ – The suggested ratio of "cells with props" to the number of all adequate cells in the room.

I make use of a number of non-standard data types, that are the following:

1. `Vector3`: A three-dimensional vector of floating-point values. An important asset when working with three-dimensional linear algebra.
2. `Vector3i`: A three-dimensional vector of integer values. Ideal for indexing a three-dimensional matrix.

3. `FloorLayout`: A set of possible floor layouts that we may want to see. Allows for *Closed* (solid floors with ceilings spanning the entire room), *HalfOpen* (a vertical opening across half the room), *HalfMezzanine* and *Mezzanine* (mezzanine ramps across two or more walls), *FullyOpen* (no floors at all; the entire room is open as if a hangar), or *RandomMixed* (every floor gets a random options of the aforementioned ones, unless `floorsAreIdentical` is true) as its options.

4. `Direction`: An enumeration consisting of a single direction (Forward, Backward, Leftward, Rightward), or any combination of them. By engine convention, ”Forward” corresponds to the positive Z axis; we could also think of it as ”North”, though this distinction is arbitrary.

5. `LightsConfig`: A set of possible light configurations, mostly for the sake of variety. The options are *Most*, *Chessboard*, *More*, *Fewer*, *None*. Affects how many lights the room creates on the ceiling and on the undersides of applicable floors.

6. `StairConfig`: Initially planned to allow for some variety amongst built staircases. Unused, all staircases are built the same way.

7. `RoomTheme`: A set of themes. Determines what props the room shall spawn, as discussed in section 4.2.1. The implemented options are *LivingQuarters*, *GeneralLab*, and *Engineering*.

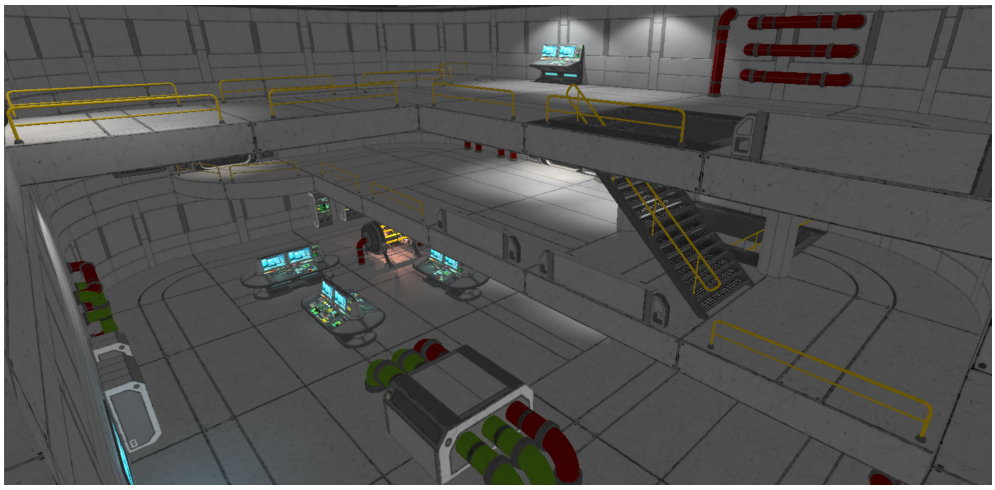
After this onslaught of textual information, I believe some visual examples may help illustrate the effects of some of these parameters, and to highlight what I mean when referring to ”floors”. Please observe the screenshots present in figure 5.2.

The first picture in the figure portrays a simple, small room with three floors (the ground floor plus two ”mezzanine” levels that form walkways). The theme of the room is ”Engineering”, as can be seen by the technical equipment and industrial pipes visible. The `propCoefficient` parameters is high in this case: there are plenty of props present, and some of the fences around the walkways have been replaced with additional barricades, providing extra cover to the player or any NPCs in the room.

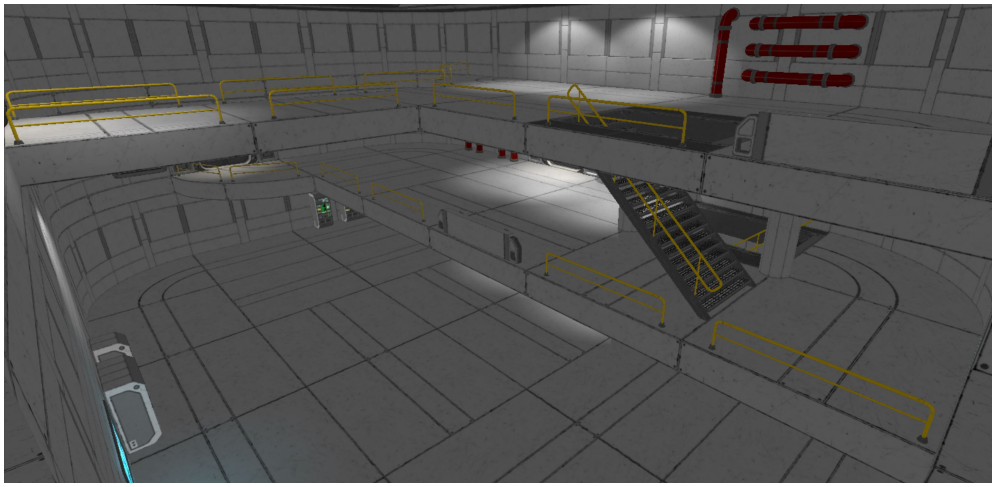
The second picture illustrates the significant difference that lack of cover presents in an otherwise identical room. The lack of props and reduced number of barricades reduces the effectiveness of a close-quarters playstyle (though this specific room remains quite small) and increases the effectiveness of other styles.

The third picture, despite looking dramatically different, is in fact of the same room. The theme has simply been changed to Living Quarters (visible from the higher number of computers and the presence of beds), and the room size has been increased, with an additional fourth floor being added into the mix. The amount of cover, i.e. props, is equivalent to the one in the first picture. Note that the ”second” and ”fourth” floors (”first floor” = ground floor) in this picture have been randomly selected to be ”Open”, providing no actual floor coverage whatsoever, aside from that found in the staircase platforms.

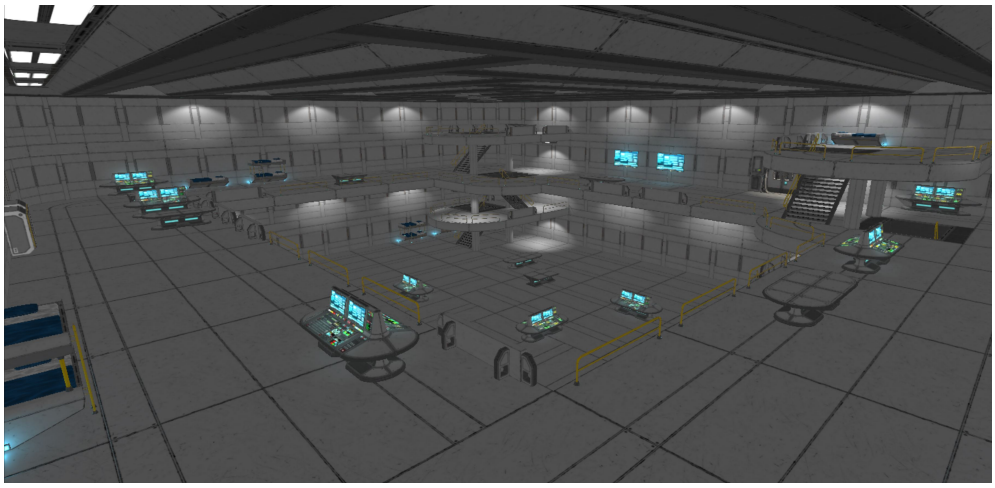
The immense number of parameters allows this generator to produce levels with a significant degree of variety, both visual and functional, though the main ”visual style”



(a) Small `size` with a high `propCoefficient`, and the Engineering room theme.



(b) Identical to the above, but with a drastically reduced `propCoefficient`.



(c) Different theme and increased room size, plus an extra floor, creating a dramatic difference.

Figure 5.2: The ”same” room under three different parameter configurations.

of the rooms remains the same, largely due to the time constraints of the project.

5.1.1.3 The Room-Building Algorithm

With the basics – the set of parameters and some degree of understanding how the system reacts to them – covered, we shall now dive head-first into the room generation algorithm itself. Due to its remarkable length – about 1600 lines of code, a mark of shame or necessity rather than an achievement – I shall provide highly abridged summaries in the form of prose.

The time needed to generate a single room using this algorithm ranges around circa 200-400 milliseconds, mostly depending on room size.

NB: The assets – models and textures – for different pieces of cells, i.e. walls, floors, ceilings, lights, stairs, and so on, were obtained through the purchase of an asset pack from the Unity Asset Store, for the purposes of speeding up development. They may thus be considered placeholders; though I believe they are adequate and consider them to be ”complete”.

Firstly, the algorithm prepares the in-game world objects for the room. It then demarcates the size limits of the room, and picks the floor indices that shall be utilised. For an example, a room with a Y (vertical) size of 7 and `numFloors` = 3 shall always have a floor at ”level” 0 (i.e. the ground floor), but the ”locations” of the other two floors need to be determined. This is done randomly, depending on the `chaos` parameter – ”orderly” rooms tend to have them distributed equally across the vertical space, whereas ”chaotic” rooms tend towards randomness.

I then initialise the ”cell matrix”, which is used to hold all relevant data (which shall be described in stride) for all cells.

We then get to the first loop. The entire algorithm consists of several loops across the entire volume of the room, since the system suffered from design revisions during the implementation time, and again, that time was quite limited; it is certainly less efficient (and a lot uglier) than it could be, however it *is* fully functional.

The first loop concerns itself with the floor layouts of the level, and is responsible for delineating those cells that we’d like to remain empty from those that require the spawning of floors at their position. Furthermore, it marks the *loci* of all cells; the *locus* in this case is a property of the cell that signifies where it is located in the room (i.e. whether the cell is ”outer” – a part of a wall – or ”inner”, and whether or not it is in a corner of the room). Loci are important, as we need to know where the subsequently create walls, and where – not. For simplicity, and for the purposes of retaining my sanity, no ”inner walls” of any sort spawn inside rooms, and all rooms have the form of a rectangular parallelepiped.

Floor layouts are calculated by the first loop using specific mathematical procedures (i.e. finding the complete ”coverage” of a mezzanine platform along any ”wall” in a rectangular grid, given constraints about its ”width”) for each option of the `FloorLayout` enumeration (listed in the previous section), that are too insignificant (yet would be

rather space-consuming) for consideration here. Of course, all code used for level generation may be found in the provided files; alternatively, this can be left as an exercise for the reader.

Additionally, the first loop selects appropriate locations for spawning lights, according to the selected `LightsConfig` option for the room.

After this is done, the algorithm marks cells with doors or ”door holes” at their respective positions, which have been supplied by the level generator using the respective lists amongst the room parameters. Cells with ”doors” require the explicit flagging that there is a door at this location, preventing the cell from spawning a wall in place of the door, and the creation of a door object. However, any single door is ”shared” between two rooms, one of the rooms creates a door object, and the other simply marks the cell as containing a door, i.e. creating a ”door hole”; this is the nature of the difference between the two. Which room creates the door and which leaves a hole to accommodate it is chosen by the level generator.

As briefly mentioned as a note at the beginning of section 5.1.1, the initial design included corridors that would connect non-adjacent rooms, and allowed for the presence of lifts that would establish connections with rooms from above. Corridors between rooms were to be created using Jump-Point-Search, an optimised variant of the A* algorithm; however, they proved too problematic for use, and were cut from the final build of the project. However, the parameters (and logic in the algorithm, though commented) for them still stand, and are presently unused.

After flagging the locations of doors and door holes, we reach the second loop – the one responsible for staircases. It performs a search through all possible staircase locations (by design, a staircase occupies a 2×3-cell horizontal space, as clearly seen in figure 5.2.c), and selects the best ones according to the number of staircases needed in the room, such that all floors (and doors/door holes) may be reached by the player, no matter where they’re located.

With both door and staircase locations now in place, we perform an additional check through floors with doors – ensuring that there is a walkable path from the staircase landing to the door, thus making certain that all doors can be properly navigated through at all times.

Following this, flags for all cells in the matrix have been set; all cells contain all information about their location and the purpose they are to server. The algorithm thus begins the third loop: the one responsible for creating the in-game objects accordingly.

Initially, the loop goes over all cells and determines which of them (those with sides adjacent to open vertical spaces) need fences or barricades along their edges. It also replaces ”square” ”hanging” cells with ”rounded” ones that look nicer. (Again, reference the staircase landings seen in figure 5.2.c for a visual example.) Additionally, all corners of all rooms are rounded, largely for aesthetic purposes.

The algorithm then fills the room with props according to the `theme` and `propCoefficient` parameters, and the capacity of the room to accommodate them. Since props are a significantly important part of room generation, they shall be examined in greater detail

on their own in section 5.1.1.4.

After dealing with the props, the algorithm chooses the spawn points and types for the enemies the room shall contain, according to those several parameters that concern them. Again, since the player's foes constitute the main gameplay element, the method for their creation is given more space for its own, in section 5.1.1.5. Note that enemy agents are *not* spawned at this stage due to technical reasons; enemy spawns are done by the level generator once the entire level has been created.

Finally, after creating the props and marking the respective cells as spawn points, the algorithm creates the physical objects for all cells in the game world, parents them under a new "room object" in the object hierarchy of the world – a feature of the Unity engine – and positions the entire room at its allocated spot, as regulated by the level generator.

With that, the process of generating an individual room is complete.

5.1.1.4 Props

Props – objects found in rooms – are an integral part of immersion. Empty, hangar-like rooms containing nothing but staircases and enemies are remarkably bland and uninteresting, and a sure way to immediately dissipate any interest the players might have in the game. To prevent this, we need to fill our rooms with props.

NB: The assets – the models themselves, as well as their textures – for these props, similarly to the assets for the walls, floors, and ceilings, were obtained through the purchase of an asset pack from the Unity Asset Store, for the purposes of speeding up development. They may thus be considered placeholders; though I believe they are adequate and consider them to be "complete".

Given that it was the best idea I came up with, I decided to enforce two separate categories of props: "small" and "large", with small ones further split into "wall" or "room" props. The distinction is simple: "small" props occupy only one cell, fitting neatly into it; "large" props in turn occupy a space of 2×1 or 2×2 cells. Looking at "small" props now, "wall props" are those that are aligned with a wall (storage lockers, computer cabinets, ventillation grates, wall monitors), and "room props" are ones that are positioned in the middle (tables, computer terminals, etc.). Some large props are merely offset variations of small props; others are completely different. Note that the terms "small" and "large" used in this case are unrelated to the actual physical size of the prop in question.

Examples of the different types of props, seen in figure 5.2, include:

- **"small, wall"** – computer terminals and "wall pipes" in subfigures *a* and *b*; beds along walls and "wall monitors" in the distance in subfigure *c*;
- **"small, room"** – most computer terminals in subfigures *a* and *c*; small reactor in subfig. *a*; small rectangular table on the ground floor of subfig. *c*, along with other small tables in the same figure;

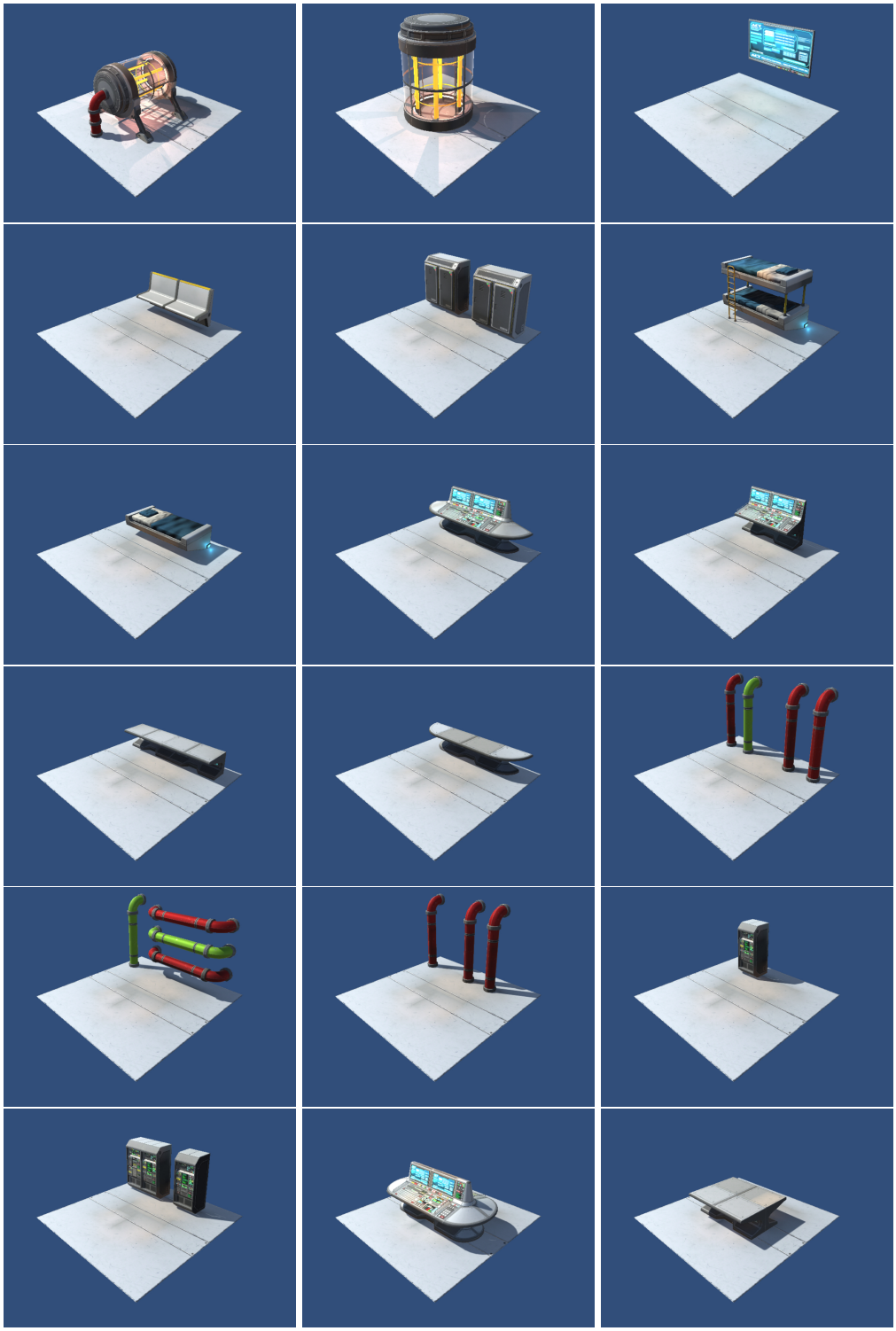


Figure 5.3: A variety of small props available in *Greyheaven*. Cell floor shown for reference. Top to bottom, left to right: two reactors, (wall props:) lockers, two beds, two computer terminals, two tables, three variations of pipes, two computer cabinets, (”room” props;) computer, table.

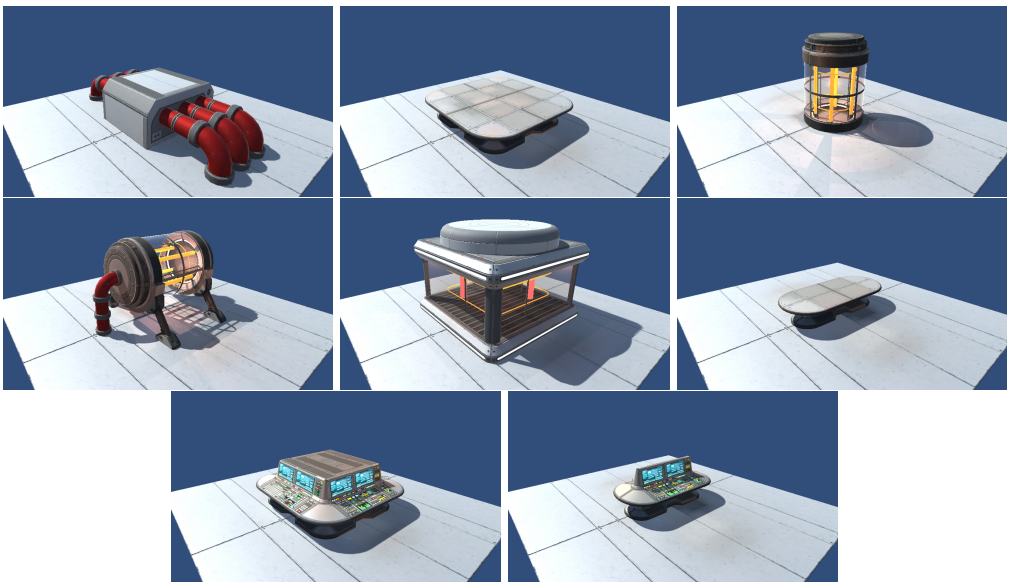


Figure 5.4: A variety of large props available in *Greyheaven*. 2×2 cell floor shown for reference. Top to bottom, left to right: pipes, table, three reactors, table, two computer varieties. They can also include other "small" props. 2×1 configurations include subsets of these that fit in half the space, as well as "small" props.

- **"large, 2x1"** – the large pipes in subfig. *a*; the computer terminal and table in the foreground in subfig. *c*;
- **"large, 2x2"** – no examples.

A subset of all "small" props available to the level generator can be seen in figure 5.3. There are a number of prop types available, each of which has a number of variations, sometimes up to eight different ones per type; all of the individual variations have been positioned by hand. The "wall" prop types are *Lockers, Monitor, Beds, Bench, Computer, Table, Pipes, Cabinets*; the stand-alone "room" prop types are *Computers, Table*, and *Reactor*, with at least one more possibility - *Boxes* that wasn't implemented due to time constraints.

Similarly, some examples of "large" props can be seen in figure 5.4. The large prop types currently in the game are *Reactor, Table, Computers,* and *Pipes*. Again, all possible permutations and variants have been positioned by hand.

Whilst somewhat limited in types, there are a lot of different variations of different props. This helps reduce similarities between different rooms; however the more different props, the better that would work.

The room generator picks a prop type for the current cell in accordance with the parameters of the room, then selects a random variation. Large props are prioritised, if there are spaces to accommodate them.

5.1.1.5 Enemies

Shooting at enemies usually constitutes the majority of the gameplay in an FPS game; *Greyheaven* is no different. In fact, the roster of foes is so important, that we shall examine them in great detail *after* we're done talking about the generation algorithms. Therefore...

NB: This section discusses enemy spawning algorithms *only*, and assumes prior knowledge of some terms; that may be acquired from section 5.2.2.

Enemies are spawned at potential *spawn points* – cells with floors that are not occupied by props or other geometry. Which exact spawn points the room generator uses is chosen at random.

As discussed in 5.2.2, the game features a number of enemy types, and every type consists of several varieties, which we shall refer to as "tiers". Enemies of a higher tier retain the characteristics of their type, but are tougher and more dangerous, thus more difficult for the player to deal with.

Every enemy *type* has the following parameters:

- `name` : `String` – The name of the enemy, i.e. "Enforcer", "Rifleman", "Sniper", or "Shieldbearer";
- `minChallengeRating` : `float` $[0,1]$ – The minimum value that the `challengeRating` (meta)parameter needs to have in order for enemies of this *type* to spawn;
- `optimalSpawnRange` : `float` $[0,1]$ – The optimal value of the "enemy spawn rating" for this enemy type (clarified below);
- `spawnIntervalRadius` : `float` $[0.05, 0.5]$ – The radius of the spawn rating apt for this enemy type (clarified below);
- a list of variants.

Additionally, every *variant* of a given enemy type has the following parameters:

- `optimalChallengeRating` : `float` $[0,1]$ – The optimal challenge rating for the spawning of this particular enemy variant;
- `challengeRatingRadius` : `float` $[0.05, 0.5]$ – The radius of the challenge rating for this variant.

I shall now explain what these parameters do and why they're necessary.

Different enemy types are most effective at different ranges. I use the concept of a *spawn rating* (occasionally referred to as a *spawn range*) to model this. A room is generated with a particular optimal spawn rating, a value between 0 and 1, which denotes the "best theoretical enemy (type)" for this particular room. In general, the lower the value, the more close-quarters foes we'd like the rooms to spawn.

However, setting values in stone is a recipe for a criminal lack of variety. For this reason all enemy types have their *challenge rating radii*, allowing for variance. This can best be visualised as a Gaussian distribution along the $[0,1]$ interval, centred on the

	Min CR	Optimal spawn rating (radius)		Optimal challenge rating (radius)		
				Tier I	Tier II	Tier III
Enforcer	0	0	(0.4)	0 (0.3)	0.5 (0.25)	0.9 (0.15)
Rifleman	0	0.4	(0.45)	0.3 (0.2)	0.6 (0.2)	0.9 (0.2)
Sniper	0.1	0.8	(0.35)	0.3 (0.2)	0.6 (0.2)	0.9 (0.2)
Shieldbearer	0.45	0.3	(0.3)	0.7 (0.2)	1.0 (0.15)	–

Table 5.1: Minimum challenge ratings, optimal spawn ratings and spawn radii per enemy type, as well as optimal challenge ratings and radii per enemy tier. Values listed in brackets are the spawn/challenge rating radii of the respective enemy type/variant. There are no Tier III Shieldbearers in the game.

`optimalSpawnRange`, with a standard deviation equal to one-third of the `spawnIntervalRadius` (indeed, the radius is the range of the ”three sigmas”).

During room generation, random values are selected for every enemy type (filtering out these whose `minChallengeRating` is higher than the player's current challenge rating, preventing ”elite” and difficult enemies from spawning at lower difficulties), according to its respective distribution as outlined above; this value is selected for every spawn point inside a room, and is compared to the room's `optimalSpawnRating` parameter, with an added bit of randomness due to chaos. Using this algorithm, we don't always select the ”best theoretical enemy type” for every spawn point, allowing for a variety of enemies to be encountered, whilst keeping the majority of the ”population” in the room tending towards the optimal enemy type.

Note that for every spawn point, the optimal spawn rating is further modified using ”height bias” as well – increasing it slightly (up to a maximum of 7.5%) depending on what floor of the room the spawn point in question is located. This ensures that riflemen and snipers tend to spawn at higher floors, providing them with a height advantage, and the player – with an additional bit of challenge.

The same principle applies for challenge ratings: the ”optimal” one is the mean of the Gaussian distribution, with the ”spawn interval radius” acting as three times its standard deviation. Challenge ratings are selected at random for every variant of the chosen enemy type, and the closest value to the player's current challenge rating (plus a bit of added randomness due to chaos) is chosen – thus biasing the algorithm towards selecting enemies on par with the player's difficulty preferences.

This process is repeated for all spawn points in the room.

The room keeps track of how many enemies of each type have already been designated for spawning. It is plausible that a ”moderately small” room – fit for close-quarters

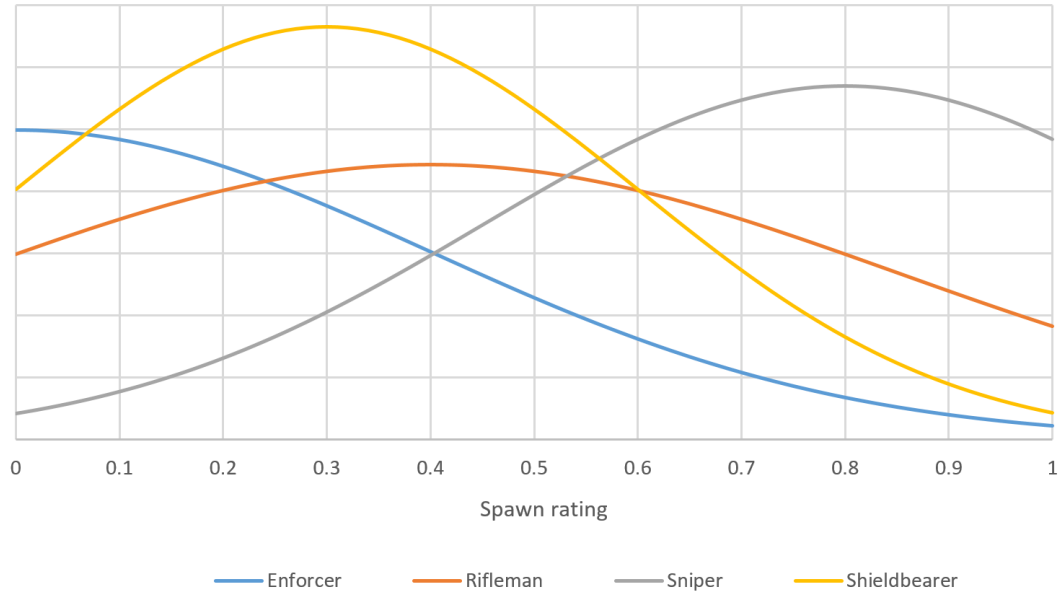


Figure 5.5: The Gaussian distributions for the spawn ratings of all four enemy types. In general, the ”optimal” enemy to spawn is selected as the one with the highest Y value (on this plot) for the ”current” X value (the calculated spawn rating of the specific spawn point). Note however that the choice of enemy is subject to other factors, as detailed in the text.

combat, but not quite tiny – at lower difficulty would generate a roster of Enforcers only. The player would thus have to fight off a mob of identical enemies that do nothing but pursue relentlessly, which is uninteresting and poses a decreased challenge. The same may happen with Snipers in long-range scenarios, or, to a lesser extent, with Shieldbearers on higher difficulties.

To avoid cases of overwhelmingly similar enemies present in any one room, groups of identical enemies are diversified manually, according to the following rules:

1. If the room would spawn a third Enforcer, he is replaced with a Rifleman of the same tier instead.
2. If the room would spawn a third Sniper, he is replaced with a Rifleman of the same tier instead.
3. If the room would spawn a second or third (chosen at random) Shieldbearer of tier x, he is replaced with a Rifleman of tier $x+1$.

The counter for a specific rule resets upon it ”triggering”. The room spawns as many enemies as it is instructed to by the level generator.

The challenge and spawn ratings, and their respective radii, are summarised in Table 5.1. The values of the probability density functions obtained by the optimal challenge ratings and challenge rating radii seen in the table have been plotted in figure 5.5.

Upon a first glance of the plot, it would seem that Shieldbearers have a higher relative probability of spawning than Enforcers, and that they negate the possibility of Riflemen spawning too. Notice however that the plot shows *relative* instead of abso-

lute probability; furthermore, Shieldbearers are bound by stricter constraints in terms of challenge ratings, and are essentially an upgraded version of the Enforcer. Additionally, this plot doesn't take into account any of the aforementioned adjustments mentioned in this section.

When all spawn points have picked the type and variant of enemy they are to spawn, the room generation process continues with the final finishing works (as outlined above).

5.1.2 The Level Generator

Whilst rooms are the scene for the player's adventure, the room generator is incapable of building a complete level on its own. For this purpose we shall examine the operation of the level generation algorithm – the veritable master and commander during the process of level creation.

In the same style as before, we can consider a few "axioms" of the level generation system I created:

1. A level consists of several rooms, connected to each other using doors.†
2. A level must always provide at least one unbroken path from the "entrance room" (the player's spawn point) to the "exit room" (the player's goal).
3. After its creation, the level must feature an adequate navigational mesh (*navmesh*) for NPCs.

Note †: The initial design allowed for *corridors* to connect non-adjacent rooms. This shall be reflected in the implementation; yet the feature had to be scrapped late in the development process of the project, for reasons that shall be explained accordingly.

I shall now provide a description similar to the one of the room generator, detailing the steps taken by the level generator to accommodate these axioms.

5.1.2.1 Level Parameters and their Effects

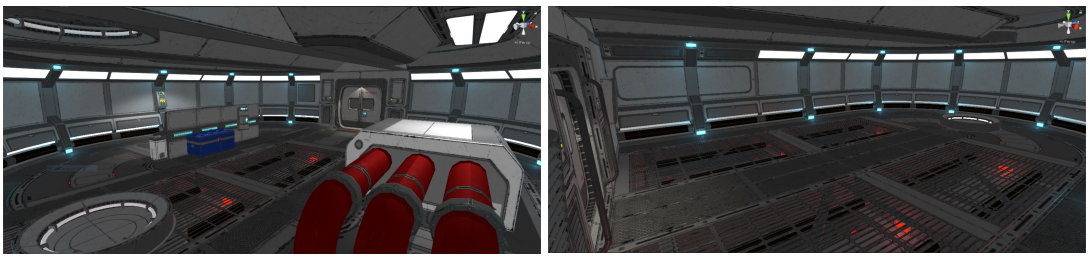
Like the room generator, the level generator relies on a plethora of parameters to generate its output. These were briefly mentioned in section 4.2.2, and listed in figure 4.1.

Unlike the room generator, however, the level generator's parameters don't "come from" anywhere. Instead, they are to be learned and then adapted to the specific player; due to the impracticality of this task (discussed in sections 4.2.2 and 4.2.3), they are generated through the use of a metamodel (using equations outlined in section 5.1.3).

For now, however, we shall investigate the parameters of the level generator and the purposes they serve. Parameters shall once again be listed along with their type and interval of possible values (if applicable).

- `seed` : `String` – The seed for the pseudorandom number generator used by the algorithm;

- `levelLength` : `int` $[-1, 10]$ – How many rooms should the main traversal route be long;
- `maximumExtraRooms` : `int` – How many ”extra rooms” the level can feature;
- `linearity` : `float` $[0, 1]$ – How interconnected the different traversal routes must be;
- `monorouteness` : `float` $[0, 1]$ – How much the level should keep to a single traversal route. If 1, there is only one route to the exit; if 0, there are several instead;
- `extraRoomRatio` : `float` $[0, 1]$ – What proporion of the main rooms (including extra lanes) are to gain extra rooms; affected by linearity. If linearity is high, side rooms may deepen to two-node branches;
- `verticality` : `float` $[-1, 1]$ – The overall vertical direction the level tends to. If -1, the level will rapidly descend in space; if 1, will ascend;
- `openness` : `float` $[0, 1]$ – The overall openness of the rooms in the level. Greatly affected by chaos for variety’s sake;
- `chaos` : `float` $[0, 1]$ – Overall randomness metric;
- `roomSizeCoefficient` : `int` $[3, 15]$ – General room size in tiles. This is effectively the square root of the implied room area size;
- `roomHeightRatio` : `float` – How high an average room should be, proportionally to its X/Z size coefficient. Can exceed 1, resulting in very tall rooms;
- `roomHeightStDev` : `float` – A companion to the previous parameter; room sizes are chosen from a Gaussian distribution with these two parameters;
- `maxFloors` : `int` – A hard cap for the number of floors in this level’s rooms;
- `extraRoomSizeMultiplier` : `float` $[0.33, 1.67]$ – Value applied to further modulate the size of extra (non-essential) rooms.
- `baseRoomDeformationBias` : `float` – How much a room deviates from the simple square. Effecitvely, how much of one side to ”transfer” to the other one;
- `suggestedDirection` : `Direction` – Which direction the entire level will tend to ”move” towards on a horizontal level;
- `directionDeviance` : `float` $[0, 1]$ – Probability of NOT following the suggested direction;
- `basePropCoefficient` : `float` $[0, 1]$ – Base prop density, before being modified by chaos;
- `challengeRating` : `float` $[0, 1]$ – The challenge rating of the level. Higher challenge ratings spawn harder enemies;
- `optimalEnemySpawnRating` : `float` $[0, 1]$ – The optimal enemy spawn rating



(a) ”Entrance” room (b) ”Exit” room

Figure 5.6: Scene-view snapshots of the special rooms used to accommodate the beginning and ending points of all levels. The player spawns at the *”inactive teleporter”* (subfig. *a*, left-hand side) and must find their way to the *”active teleporter”* at the level's exit (subfig. *b*, right-hand side). Particle systems improve the look at run-time.

of the level. Enemy spawn rating determines the composition of spawned enemies.

NB: Some level parameters share names with room parameters. Whilst parameters with the same names can be treated as counterparts, they are *not* completely the same, and do *not* share values.

The variety exposed by these parameters allows us to produce a wide range of possible levels, presenting different challenges suited for different playstyles.

5.1.2.2 The Level-Building Algorithm

Again, due to the significant length of the level generation algorithm, I shall provide abridged summaries of its functionality. The process of this algorithm is simpler, with less different ”components” that require thinking about; however, some aspects of level generation, namely the positioning of rooms, are quite complex and irksome, resulting in yet another ungraceful implementation.

The time needed to generate a full level using this algorithm ranges around circa 800-4000 milliseconds, depending on how large the individual rooms are, and how long the level is. No parallelisation or multithreading optimisations are in use.

Level generation begins at a fixed location, the global $(0, 0, 0)$ coordinate of the game world. At that location is positioned the ”entrance room”; the level generator builds the rest of the rooms in order and ultimately positions the ”exit room” at its allocated spot.

Unlike all other rooms, the starting and ending rooms are *not* procedurally generated. Instead, they are always the same, allowing players to easily recognise them. Furthermore, they have a unique aesthetic that isn't shared with any procedurally-generated room; this is deliberate. The ”entrance” and ”exit” rooms have been visualised in figure 5.6.

The algorithm's first job is to calculate the amount of ”traversal routes” the level will require. For our purposes, a ”traversal route” is an unbroken, unique path from the

starting to the ending room. A level always has at least one traversal route, but may have up to four – based on the `chaos` and `monorouteness` parameters. A traversal route shall also be referred to as a ”lane”.

I then build a graph that holds all rooms and the connections between them. The nodes of the graph are rooms, and edges represent a connection. The main traversal route – the first lane – is added initially by simply connecting a number of sequential nodes (as many as the `levelLength` parameter) to each other, and denoting the last node as the ”exit room index”.

Any extra lanes are added after that. Extra lanes branch off of the main lane at some point; for each extra lane, this is picked using a Gaussian distribution with a mean value selected randomly between 15% and 25% of the length of the main lane. The standard deviation of this distribution is 0.35, i.e. 35% of a room, thus allowing for two sigmas falling within the same room and the third sigma falling outside it.

To illustrate: in a level with a 10-room-long main lane, the extra lanes’ branching points shall be picked randomly from a Gaussian distribution centred around the second or third room, with the standard deviation allowing for a limited possibility of this interval moving up by one room in either direction, i.e. potentially expanding from the first to the fourth room.

The length of an extra lane is calculated as 50% to 70% of the length of the main lane, further modified by $\pm 10\%$ due to chaos. The extra lane’s ending point is calculated thusly, on the basis of its length. Extra lanes – connecting new rooms that have not been used – are added to the graph.

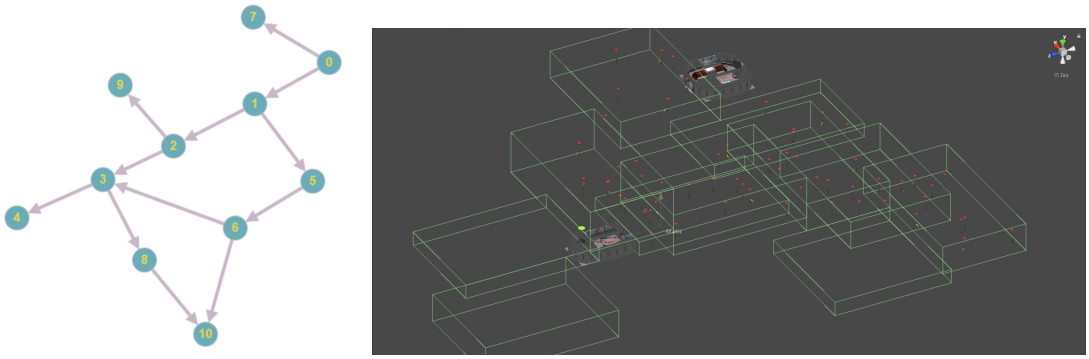
Note that rooms used for extra lanes are, counter-intuitively, *not ”extra” rooms*. Instead, they shall be referred to as ”essential” rooms. Thus a level with a single traversal route and `levelLength` of 10 shall have 10 essential rooms, but a level with the same length and three lanes shall have, e.g. 28 essential rooms. (For that purpose the length of levels used in our experiments is kept as a fixed constant with a low value.)

With extra lanes done, we get to extra rooms. Extra rooms, as mentioned a couple of times by this point, are rooms that are not a part of a traversal route (i.e. they are non-essential), and usually represent a ”dead end”. (In a real game, such rooms would contain supplies for the player to collect, but that was not implemented due to time constraints.)

Extra rooms – up to the maximum amount of those, specified by a level parameter – are attached to random existing rooms, i.e. existing nodes on the graph.

After setting up all lanes and extra rooms, the algorithm makes a pass over the graph, potentially creating ”lateral” connections between lanes, based on the `linearity` parameter. These connections serve to make the level more open, not to move the player closer to the exit. They are thus done amongst nodes that are equidistant from the root node (the entrance room). Additionally, some extra connections between different extra rooms may be formed, removing their ”dead end” properties and providing extra routes for exploration, resulting in more maze-like levels.

This completes the ”logical” layout of the level. All we have left now is to generate it



(a) The graph layout: node 0 is the first room of the level; node 4 is the ”last” room of the level.

(b) The bounds of the rooms of the same level.

Figure 5.7: An example of a level generated using the algorithm – both of its ”logical” layout, i.e. the graph the algorithm forms; and an isometric scene view of how the bounds of the rooms are positioned in world space. In subfigure *b*, the starting and ending rooms are visible, as well as the enemies spawned by the level. The boxes represent the ”bounds” of the rooms, and are presented as such for greater visual clarity.

for real within the game world.

A visual example of the sort of graphs the algorithm generates can be seen in figure 5.7.a:

- Node 0 is the ”first” node – the one that the ”entrance room” is attached to; node 4 is the ”last” node – the one that the ”exit room” is attached to (level length is fixed at 5, but that doesn't include the entrance and exit rooms);
- The main traversal route is thus the set of nodes $\{0, 1, 2, 3, 4\}$;
- The level has a second ”lane”: $\{0, 1, 5, 6, 3, 4\}$;
- Nodes 7, 8, 9, and 10 are non-essential (extra) rooms that have formed some connections to others.

The algorithm does this using a recursive iteration of the graph, starting from the root node and creating the main lane and exit before everything else. On paper, this creates a direct, straight-forward route from the start to the finish; however this is difficult to notice by the players, given the different layouts and spacial direction of the rooms.

The algorithm recursively calculates the bounds of the required rooms in world-space, and computes the locations they must be positioned in, as well as the locations of any doors and other similar connections between rooms. The mathematics responsible for these calculations is unsightly, complex, has been revised dozens of times, and is too painful for me to describe; plus, it would occupy a lot of space better used for more important descriptions; and this document is too long as-is. The reader is recommended to examine the code, should they feel the need to do so.

Individual rooms are created in order using the room generator. The level generator calculates the set of parameters for each room; the process for this is described in the

next section, namely 5.1.2.3.

After the rooms have been built, the algorithm generates a navigational mesh (*navmesh*) for the level. A navmesh is a type of geometry used for pathfinding for non-player characters – the player's foes, in our case; the absence of a navmesh (or other pathfinding methods) renders NPCs unable to move. The navmesh created here is *complete* – spanning all rooms of the level, allowing foes to navigate everywhere.

With the navmesh in place, we spawn enemies and position the player in the entrance room. For optimisation purposes, only enemies in nearby rooms are spawned; additional foes are created as the player navigates through the level.

Furthermore, the level generator, at the end of its operation, initialises the custom-built occlusion culling system, again for reasons of performance. (*Occlusion Culling* (OC) is the process through which objects and surfaces unseen by the player during run-time are not rendered by the GPU, improving the game's performance. The Unity engine doesn't provide an operational OC system for procedurally-generated objects, so I had to write one of my own.)

We can see that a generated level looks a lot different than what its logical layout would have us imagine. (Figure 5.7.) This is easily explained – it's a property of the three-dimensionality of the space we create rooms in.

And thus a level is built, and the player is free to explore it, doing combat with whatever foes await.

5.1.2.3 Generating Room Parameters

The level generates separate sets of parameters for every room it contains.

Given the relative complexity of some of these parameters, and to not bore the reader with what would essentially become code rewritten with mathematical notation, I shall instead provide a short description of how each parameter is obtained.

NB: In the following list, *R* stands as a short-hand notation for *"a new floating random variable with a value between 0 and 1 (inclusive), chosen using the uniform distribution"*.

NB: In the following list, "half of (the) current chaos" means that we take the value of the level's `chaos` parameter, divide it by two, and take the result. Similarly, "a third of *R*" implies the same but for the random value define above. For the sake of example, if chaos is equal to 0.5, then "half of current chaos" would be 0.25; if *R* happens to be exactly 0.6, "a third of *R*" would be 0.2. Again, this is a short-hand notation used for clarity.

- `size` – Determined using *R*, the room's base size coefficient, and the level's base room deformation bias. This helps produce rectangular rooms of varying sizes; furthermore, if the room is non-essential, its size is further modified using the respective multiplier, mostly with the purpose of *reducing* its size;

- `numFloors` – Determined as equal to the room's Y (vertical) size, with potential reductions based on chaos;
- `chaos` – Set the same as the level's chaos parameter, with with up to a randomly-selected $\pm 10\%$ deviation;
- `floorLayout` – Fixed to *RandomMixed* at present to ensure greatest variety in floor layouts;
- `floorsAreIdentical` – Chosen only if chaos exceeds 30%, and even then is set to true only 20% of the time.
- `openness` – Kept as the level's openness, plus half of the current chaos 67% of the time, or minus one-third of the current chaos one-third of the time;
- `linearity` – Set as 1, of which we subtract a quarter of the current chaos, a third of R, and another third if the target room is non-essential;
- `suggestedDirection` – If the room is essential, keep the level's suggested direction; if not, a small chance (based on R and chaos) to rotate the direction by ± 90 degrees;
- `verticality` – Kept as the verticality of the level, $\pm 60\%$ of the current chaos;
- `lights` – Fixed to *More* by default;
- `numStaircases` – Calculated as the room's area (X$\times$Z size) divided by 40;
- `stairsLayout` – Unused, as mentioned a few times;
- `challengeRating` – Retained exactly the same as the level parameter of the same name, but with a 33% to be increased by 10% of current chaos;
- `optimalEnemySpawnRating` – The same as the level parameter, but with a $\pm 1\%$ increment;
- `enemySpawnDensity` – Fixed to 0.111, providing one enemy per 9 cells on average;
- `maxEnemies` – Equal to four times the magnitude of the 2D vector formed by the room's X and Z sizes, rounded to an integer, and bounded between 3 and 25;
- `theme` – *GeneralLab* is picked 50% of the time; *LivingQuarters* is chosen 30% of the time; and the rest (20%) goes to *Engineering*;
- `propCoefficient` – Equal to the level's base prop coefficient, with a chaos-dependent chance to add $\pm 40\%$ of R.

All unlisted parameters, namely `originPosition`, `mainEntrancePos`, `doorPositions`, `doorHolePositions`, `corriDoors`, and `noCeilingPositions` are set individually for each room using the complex calculations of room bounds and positions that I prefer to not address.

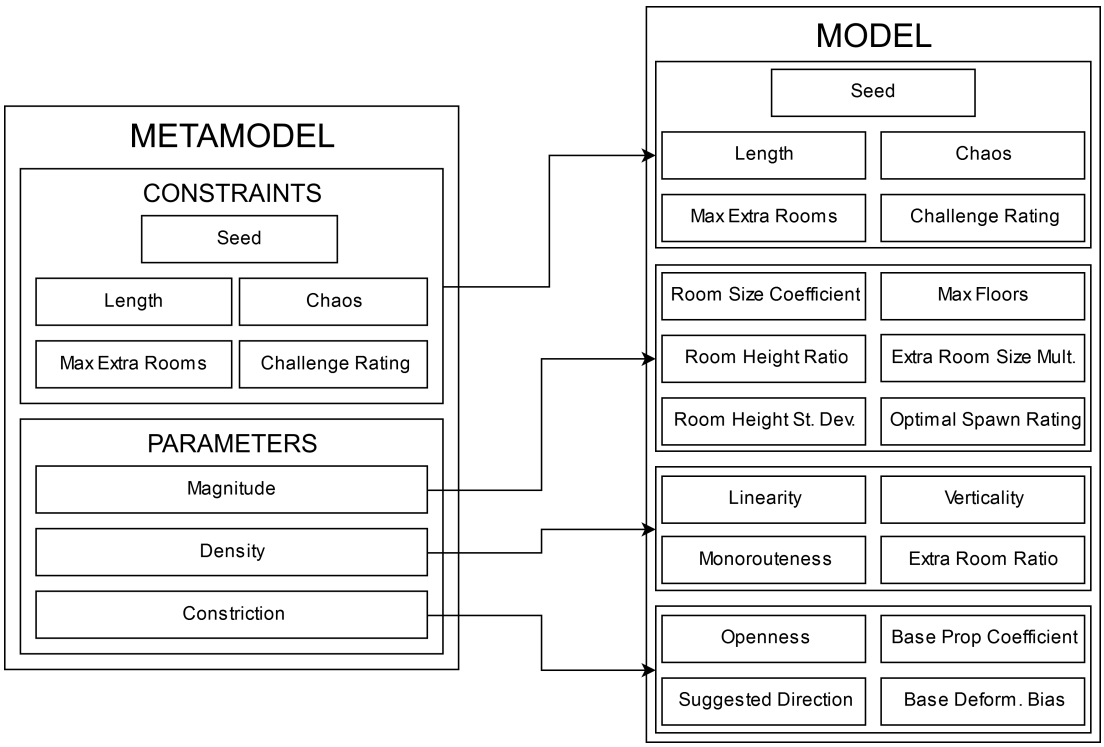


Figure 5.8: Reminder: A visual representation of the relationship between the metamodel and the level model it generates.

5.1.3 The Specifics of Metamodel-to-model Conversion

In section 4.2.3, we went over the theory and reasoning behind the metamodel, and briefly discussed, *in abstracto*, the way it maps to a level model.

There is little more to theorise about this subject, as I've striven to keep the process as straight-forward as possible. Instead, I shall simply present the relevant diagram once again, for clarity for the reader – figure 5.8 – and introduce the specific equations for mapping the metamodel's hyperparameters to the full set of parameters of the level model, as I believe all shall be made quite obvious by a simple look at the formulae.

So, without further ado, here are the equations for level model generation from metamodel parameters (note that, for purposes of clarity, they shall be listed in the respec-

tive order seen in figure 5.8):

$rsc = 4 \xrightarrow{M} 9$	Room size coefficient
$rhr = 0.1 \xrightarrow{M} 0.4$	Room height ratio
$rhs = 0.05 \xrightarrow{M} 0.2$	Room height standard dev.
$mfl = 3$	Max floors; const
$esm = M \cdot 0.15 + 0.6$	"Extra room" size multiplier
$osr = M \cdot 0.75 + (1 - C) \cdot 0.25$	Optimal enemy spawn rating
$lnr = D$	Linearity
$mnr = D$	Monorouteness
$ver = D \cdot (\pm 1)$	Verticality
$err = 0.2 \xrightarrow{1-D} 1$	"Extra room" ratio
$opn = 1 - C$	Openness
$sgd = (C \cdot 10) \bmod 4$	Suggested direction
$bpr = 0.1 \xrightarrow{C} 0.95$	Base prop coefficient
$ddv = 1 - C$	Directional deviance
$brd = 0.2 \xrightarrow{C} 0.5$	Base room deformation bias

Where M, D, and C are the metamodel parameters of magnitude, density, and constriction respectively; and the $x \xrightarrow{p} y$ notation refers to the **linear interpolation between** x **and** y **based on the value of the parameter** $p \in [0,1]$.

In other words:

$$x \xrightarrow{p} y \equiv x(1-p) + py$$

NB: Metamodel constraints map directly to the model variables with the same names, as outlined in figure 5.8.

Note that randomness is only used when selecting verticality and suggested direction – and then only randomness derived from the actual parameters themselves.

The use of metamodels to create level models reduces the possible variety that a level model can produce, but makes the problem of learning a lot simpler. In practice, metamodels can provide a sufficiently varied array of levels to be considered practical.

5.2 The Mechanics of Weapons, Enemies, and Playstyles

We have so far examined the behaviours of the room and level generation algorithms. Those processes, whilst crucial for our purposes, are unseen by the players; in fact, players only interact with the generated levels. It is the manner of this interaction – what is usually referred to as *the core gameplay loop* – that we'll discuss in this section.

In an FPS game, the gameplay generally consists of *shooting* at enemies whilst moving from point A to point B. *Greyheaven* adheres to these classical principles of the genre:

1. Players are spawned at the "entrance room" of the level and are tasked with navigating to the "exit room" (discussed in section 5.1.2.2).
2. Players are given a full arsenal of distinct weapons, allowing them to tackle the enemies in the level in any way they desire (discussed theoretically in section 4.1).

In this section, I shall describe the behaviours of weapons and enemies in detail, and the way they are related to playstyles; furthermore, I shall present an overview of the menus the game presents to players, in an attempt to systematise the available functionality.

5.2.1 Firearms: Respecting the Relevant Roles

As we have discussed previously, there is an infinitesimal possible variety of weapons that may be featured in an FPS game, especially one with a "sci-fi" theme like *Greyheaven*.

In section 4.1, I briefly described the weapons featured in the game. Those four "weapon archetypes", as they arguably are, have been chosen due to being intimately familiar to all players of the FPS genre; furthermore, all of them fit neatly into one of the playstyles I have defined.

NB: All weapon assets – models and textures – have been created specifically for this game. The sniper rifle asset is a modification of the assault rifle one due to time constraints.

Here is a more in-depth examination of the weapons in their final, implemented state, and of the more interesting points in designing them:

- a **sabre** – requiring no ammunition, and being devastating at extremely close ranges. Initially planned to support a variety of attack animations, "power attacks" (stronger but slower types of attack), deflection of enemy missiles, and so on. However, due to time constraints, the sabre only possesses a single, unappealing attack animation, and deals a static, yet rather high amount of damage (90) with a moderate attack speed (a "fire-rate" of 1.25 swings per second), and is an uninteresting weapon.
- a **shotgun** – designed to be deadly at close ranges, yet to be useless at longer ranges; firing 15 pellets in a conical spread to eliminate the need for precision

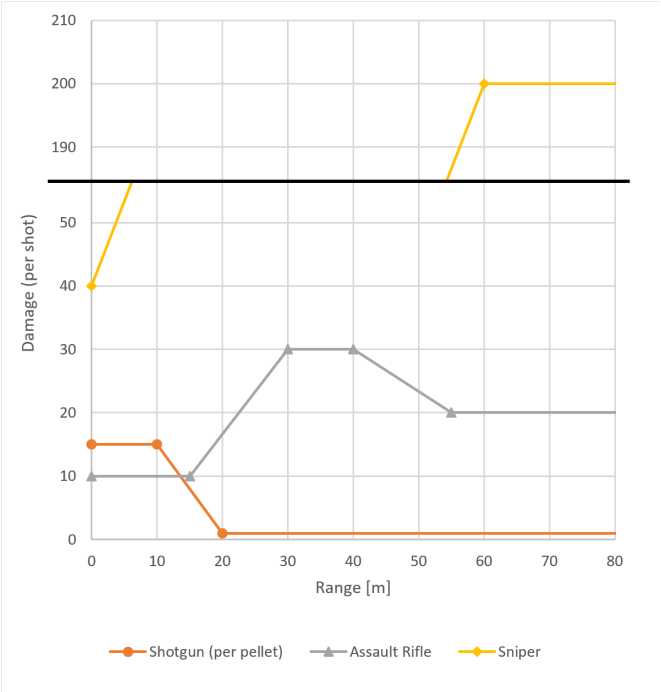


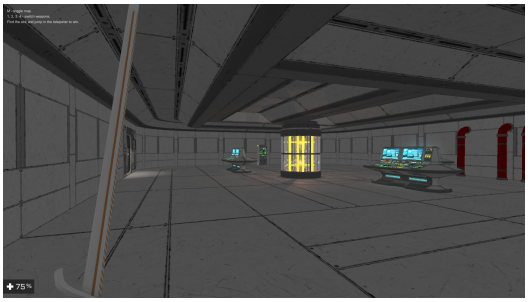
Figure 5.9: Damage per range for ranged weapons. Note that the shotgun fires 15 pellets at once; and that the sniper rifle has a very low fire-rate. The extra vertical space between 50 and 190 damage points has been elided, as it only contains the linearly interpolating damage value for the sniper rifle.
The ”basis” for the health value of a character is 100 points; headshots from all weapons deal double damage.

at close ranges, and to allow for the simultaneous targeting of multiple clustered enemies. Its fire rate is slow (circa 55 rounds per minute), however, and its damage drop-off is utterly despicable – a measure taken to prevent the shotgun from being ”overpowered”.

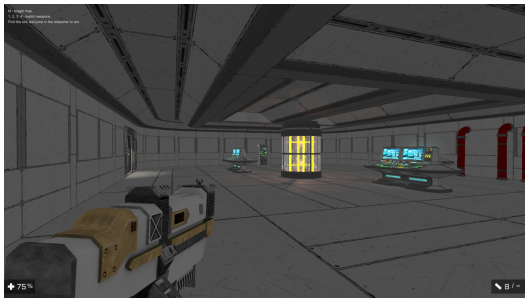
- an **assault rifle** – supposed to shine at medium range, which presented an interesting design problem – making it less capable than the shotgun at close ranges, whilst also less capable than the sniper rifle at long ranges. In the end, the rifle has a moderately-high rate of fire (500 rounds per minute), an unusually high damage at medium range, but its iron sights are lacking for long-range engagements in comparison to the scope of the sniper rifle.

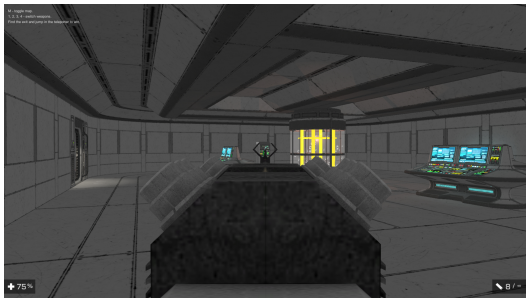
- a **sniper rifle** – creating a sniper rifle isn't a difficult task, as the way it should behave is familiar to more or less every human that has ever seen a war film or played a game. However, in terms of game design, one must be wary of making the sniper rifle – usually a weapon with high-powered projectiles – too effective at close range. To prevent this, I chose to employ an unorthodox method – giving the sniper rifle a low amount of damage at close range that increases with distance, essentially inverting the FPS concept of ”damage drop-off”. Furthermore, I added a **”time dilation”** ability available whilst using the sniper rifle, allowing players to slow down time for two seconds every so often, allowing them to line up (head)shots more accurately. The sniper rifle's rate of fire is 50 rounds per minute.

A better illustration of the projectile weapons' effectiveness is present in figure 5.9. The sniper rifle, though the most powerful in terms of damage per shot, suffers from a very slow rate of fire, which makes it inadequate for use in close quarters; the shotgun quickly becomes useless beyond close range. This leaves the assault rifle in a perfect

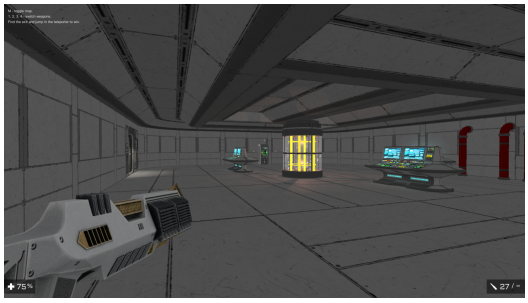


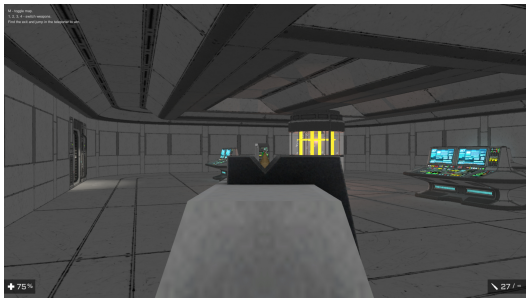
(a) The sabre. Currently not a very inspiring weapon to use. As a mêlée weapon, the sabre has no iron sights and cannot be aimed.



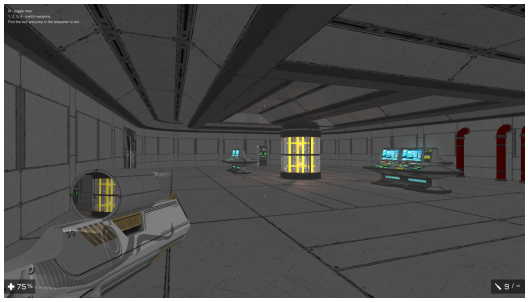


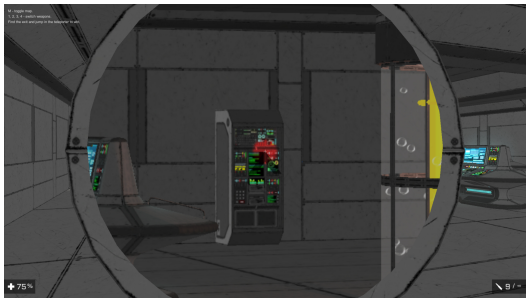
(b) The shotgun. Aiming down sights sacrifices some situational awareness, but tightens the spread pattern considerably.





(c) The assault rifle. Aiming down sights substantially increases accuracy, but the sights are bulky and not quite apt for long-ranged engagements where precision is necessary.





(d) The sniper rifle. The sniper scope's magnification is rendered in real-time (even when not aiming down sights), and, coupled with the sniper rifle's "time dilation" ability, makes long-distance engagements substantially easier.

Figure 5.10: The player aims at a technical cabinet across the room, showcasing the arsenal in the game. *Greyheaven* uses left-handed weapon models, because dedicated left-handed protagonists are underrepresented in FPS games (and this theoretically makes the game a little more original).

spot to fill its niche, especially given the ”damage spike” between 15 and 50 metres. (Remember that a single ”cell” has a length and width of 6 metres.)

The weapons are presented visually in figure 5.10: the images on the left-hand side displaying the unaimed (”hip-fire”) modes, and those on the right-hand side – the ”aiming down sights” (ADS) mode. ADS provides a marginal amount of visual zoom – even when not using the sniper rifle – and increases accuracy, at the cost of reduced movement speed and situational awareness (due to the sights occupying a large part of the screen). The default key for aiming down sights in virtually all FPS games is the right mouse button (as the default key for firing a weapon is the left mouse button); this is the same in *Greyheaven*.

NB: Additional standard shooter mechanics featured in the game include *sprinting* – default key `Shift`, increasing movement speed at the cost of accuracy; can't be used whilst ADS – and *crouching* – default key `CTRL`, improving accuracy at the cost of movement speed (can be combined with ADS). Crouching can be used to take cover behind barricades or other objects.

I thus posit that the game features a simplistic, yet diverse arsenal of familiar weapons, all of which have a clearly defined role, as well as visible strengths and weaknesses. Four weapons in total is certainly a very low number for a commercial game, but I strove to select only these who fit exactly into one playstyle – after all, this is a research project first and game second.

The initial design of the game would have had weapons using limited ammunition pools, and having defeated enemies dropping ammunition for the respective weapon type – i.e. dead Riflemen drop ammo for the assault rifle; dead Snipers – for the sniper rifle. Additionally, ”non-essential rooms” would have had the possibility of spawning additional ammo and health supplies for the player to use. Unfortunately, this feature was not implemented due to time constraints – all weapons in the current implementation have an unlimited amount of ammunition. On the one hand, this allows players to stick to their preferred playstyle at all times; on the other, it removes a gameplay element (searching for ammo and health) that may be used to adjust difficulty on the fly.

The same applies for the player's health: ideally, the player would have to find and use medical kits that spawn across the level. However, due to time constraints, this could not happen; currently, the player's health regenerates up to 75% of the maximum if it has fallen below 75%; at a rate of 1% (1 health point) per second, after 10 seconds of not taking damage.

Given that players are equipped with all four weapons at all times, and may select between them as they wish, it is impossible that they will be placed in a situation where they simply don't have the means to defeat the enemies present in the room. Player death is certainly quite possible, especially if a player charges into a room head-first with no regard for tactics and care, but that is to be expected of an FPS game.

Figure 5.11: A visual showcase of a subset of enemies in *Greyheaven*. Top row, left to right: Rifleman I, II, and III; bottom row: Enforcer I, Sniper I, Shieldbearer I. Due to time constraints, all enemies use the same default soldier model, and the same animation set. These are to be replaced with drastically different models in the future. Cell floor present for reference.

5.2.2 Foes: Situational Strengths and Shortcomings

Perhaps even more important than the weapons are the enemies that the player will face. In section 4.1, I briefly described the enemy types and how they correspond to every playstyle. In section 5.1.1.5, I described the principles of enemy spawning, and what parameters govern the inclusion of different enemy types and variants in rooms.

Here, however, I shall describe the enemy types in detail, including their behaviours during gameplay, and shall provide statistical information about them.

First of all, however, I shall present the enemies and their variants visually – consider figure 5.11 A few notes are in order:

- There are four enemy types: the *Enforcer*, a hand-to-hand combatant; the *Rifleman*, the average ranged combatant; the *Sniper*, a long-ranged opponent; and the *Shieldbearer*, an elite mêlée combatant, who doesn't actually carry a shield.
- All enemies have three *tiers*, sometimes called "variants" (except the Shieldbearer, who has only two). These tiers are denoted using Roman numerals after the name of the enemy.
- A tier improvement is an upgrade to the soldier variant, essentially producing tougher enemies of the same type. The behaviour of higher-tier enemies is the same, except that they have more health, their attacks are more damaging, and they sometimes move faster.

Due to time constraints during the development of the project, all enemies use the same

Type	Tier	Health	Move/Run speed [m/s]	Melee damage	Ranged damage	Comment
Enforcer	I	60	5/6	9	–	–
	II	110	6/7	9	–	–
	III	170	6/7	12	–	–
Rifleman	I	100	5/6	4	3, *1-2 shots*	Rifleman shots travel at 30 metres per second
	II	160	5/6	4	3, *2-3 shots*	
	III	215	5/6	6	3, *3-4 shots*	
Sniper	I	80	5/6	2	8	Sniper shots are almost instant; snipers are highly accurate
	II	125	5/6	2	8	
	III	185	5/6	3	8	
Shieldbearer	I	360	6/7	10	–	–
	II	520	7/8	15	–	–

Table 5.2: A summary of the most important statistics for all enemy types. See text for details.

placeholder model, with colour-coding and minor differences used to distinguish them. The guidelines for differentiating between enemies, as partly visualised by the picture, can be summarised as follows:

- **Enforcers** and **Shieldbearers** carry no weapon, but Shieldbearers's bodies are blue.
- **Riflemen** carry a variant of the rifle available to players;
- **Snipers** carry a brightly-coloured placeholder gun mesh, standing in for a distinct sniper rifle.
- **Tier II enemies of all types** have *green heads*;
- **Tier III enemies of all types** have *red heads*.

These are to be replaced with visually different models separate for each type and variant as time allows it; the same applies to their animations – currently, even Enforcers and Shieldbearers look as if they're holding rifles.

On Shieldbearers: As discussed in section 4.1, the initial plan for this enemy type was that they would carry a large shield, protecting them from all projectile weapons. They would be slow-moving and intimidating, with lots of health, and would only be vulnerable to sabre attacks – allowing the player to knock their shield aside with a "power attack" and dispatch them. However, this was unfeasible due to time constraints – especially given that the sabre also lacks some of its planned "special" functionality. "Shieldbearers" in the present version are fast-moving enemies with a lot of health – essentially an upgraded version of the Enforcer. An *extremely deadly version* of the Enforcer. The name "Shieldbearer" is thus technically a misnomer, but this can be hand-waved away by saying they owe their great durability to an *invisible force shield* of some sort. (Furthermore, players never see the names of the foes, so this is not a practical concern.)

All enemy types have different *statistics* (also known as "stats" in the context of

games), i.e. numeric values that determine their capabilities. Some ”stats” have been compiled in table 5.2. Notes on the table:

- Enemies move around using the first movement speed listed. If they spot the player, they may start running, so as to close the distance faster. Alternatively, enemies will run to get to cover faster.
- Riflemen suffer from some degree of inaccuracy, and their ”laser shots” have a travelling time, allowing a keen player to dodge their projectiles. Snipers, on the other hand, have an almost 100-percent accuracy, and their shots are nigh impossible to dodge.
- Higher tiers of Snipers fire slightly faster.

Additionally, different enemy types exhibit different behaviours:

- Enforcers and Shieldbearers are *reckless* – with no ranged attack options, they charge straight for the player in an attempt to overwhelm them and flush them out of cover. Their high damage makes them dangerous.
- Riflemen are *tactical* – they seek to maintain an average distance from the player, and utilise cover, hiding behind barricades and objects. Furthermore, they often reposition themselves, seeking to flank the player. If approached by the player, they’ll attempt to reposition away.
- Snipers are *cautious* – the most fragile of all enemies, Snipers seek to maintain a maximum distance from the player, ideally hidden behind cover, and are hesitant to approach. Like Riflemen, they may try to escape if approached.

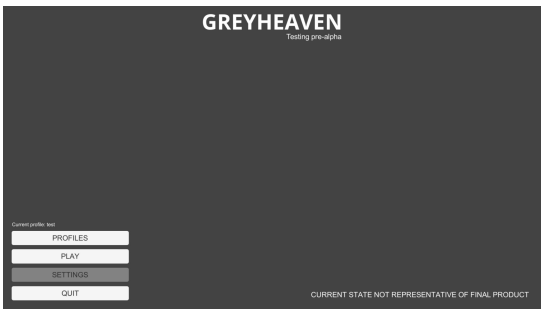
NB: Writing code for NPC behaviours is a daunting and time-consuming task. Due to the time constraints of this project, I’m using an ”AI library” purchased from the Unity Asset Store. The library is badly optimised and less than ideal, and is in need of replacement. It is, however, functional enough to allow for the design of rudimentary behaviours as detailed in the list above.

In a similar way as the selection of weapons, the roster of specific enemy types has been chosen in order to allow for the creation of challenges for every playstyle; furthermore – as outlined in a previous section – enemies are spawned according to the room they are to inhabit. Enforcers (and Shieldbearers at higher difficulties) are frequently seen in smaller rooms; Snipers are present in large ones. Riflemen are seen everywhere (a feature to the spawning rules outlined in section 5.1.1.5), as they are the ”staple” enemy soldier type, and are effective in all environments.

5.2.3 Menus, Navigation, Operation, Presentation

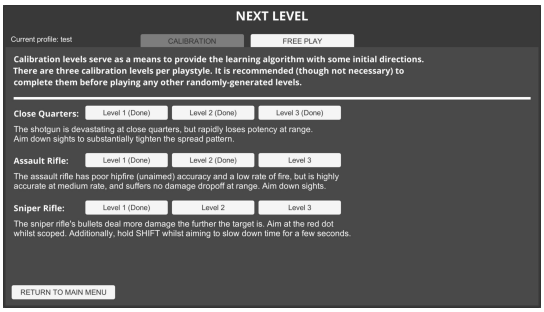
Players interact with the game using menus. This section shall present an overview of the menus currently found in the game, plus a brief description of the *minimap* provided to players for all levels.

Do note that I considered user interface (UI) design to be a task of an insignificant priority in comparison to everything else; as a result, the menus are functional, but far from aesthetically pleasing.

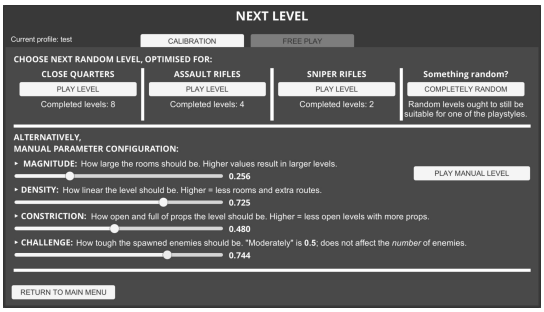


(a) Main menu

(b) Profiles window



(c) Play window: ”Calibration”



(d) Play window: ”Free Play”

(e) Escape (pause) menu

(f) Rating menu

Figure 5.12: Screenshots of all menus in the current build of the game. See text for details.

This section also serves as a companion to the next one – section 5.3 – as player feedback is gathered through directly sourcing players' opinions.

5.2.3.1 Menus

A full presentation of *Greyheaven*'s menus can be found in figure 5.12. A description of their effects follows:

(a) The initial screen that players encounter. Allows access to the *Profiles* and *Play* windows. Note that a settings menu, whilst an integral feature of any game on the PC platform, is a time-consuming thing to create, and has thus been ignored.
(b) The *Profiles* window. Learning and level adaptation is done on a per-profile

basis, allowing multiple players to play the game on the same machine.

(c) The *Calibration* tab of the Play window. Provides gameplay tips for players, and contains buttons for calibration levels. (The specifics of ”calibration levels” shall be discussed in section 5.3.1.1.) Additionally, completed calibration levels are highlighted.

(d) The *Free Play* tab of the Play window. Exposes buttons for levels adapted to each playstyle and provides information about the number of completed levels in this specific profile. Additionally, this window provides direct access to the four parameters (three metamodel parameters plus the challenge constraint), as well as brief explanations of what they affect, allowing experienced players to experiment with the level generation system.

(e) The ”escape” (pause) menu. Brought up when a player presses the `ESC` key during gameplay. Game time pauses whilst the menu is active. This menu allows players to exit the game, holds a placeholder button for the yet-to-come settings menu, and provides information about the specific values of the four parameters.

(f) The ”rating” menu, presented when a player reaches the exit of the level (the ”active teleporter” in the ”exit room”), when a player dies, or when a player aborts the level (through the escape menu). This menu points out some statistics from the current level, and allows players to rate the level itself, as well as its difficulty. Furthermore, it requests players to select which playstyle they saw as the optimal one for this level. (The learning process and the utilisation of these ratings shall be discussed in the next section.) If the player completed the level or died, they are not allowed to proceed until the questions are answered. If the player has aborted the level manually, they are not allowed to answer the questions, and this attempt is ignored by the adaptive algorithm.

It’s worth noting that the ”Play” window is presented again after a level has been rated successfully, if the player has pressed the ”Continue” button in the rating menu.

Pressing one of the buttons that correspond to levels (i.e. all calibration level buttons, or ”Generate...” buttons in the ”Free Play” tab) immediately loads the appropriate level. This will be discussed in section 5.3.

5.2.3.2 Minimap

In a large three-dimensional environment, especially one that tends to be maze-like and features little directional guidance as *Greyheaven*’s, a means of navigation through the level is mandatory.

I initially considered having colour-coded ”trails”, lines painted on walls, that would direct the player towards the exit in a ”natural” manner. However, due to the project’s limited time constraints, this idea was scrapped, as it would’ve been too difficult to implement properly.

To nevertheless provide a means of navigation to players, I created a dynamic minimap. *Minimaps* are a classic feature of video games, and are named so due to usually occupying a small space in the corner of the screen. (Observe the presence of a large

(a) Exit not visible (off-screen) (b) Exit visible

Figure 5.13: An example of the minimap. The minimap is accessible at all times; rooms and doors are presented as grey rectangles, enemies appear as red dots, the player is a white dot, and the exit is a yellow dot. To assist navigation in large levels, the player's marker is ”connected” to the exit marker using a dotted line.

minimap in the lower-left hand side of figure 3.1.a, and the lack of a minimap in figure 3.1.b.)

The minimap in *Greyheaven*, though it can be simply called a *map*, is available to players at all times, and is activated or deactivated using the `M` key. A visual example of the minimap is provided in figure 5.13.

The minimap displays the locations of all rooms and enemies, and of the player and the level exit. Characters and the level exit – a place of importance – are represented as dots, with the player being a large white dot that is always positioned in the centre of the screen. Enemies are smaller, red dots, and the level exit is a large white dot. Rooms and the doors between them are represented by grey rectangles of their respective sizes; additionally, the direction from the player's location to the exit is always given through the use of a dotted white line that connects the player's dot with the one for the exit.

The addition of this minimap proved effective, perhaps too much so: since there are no secondary objectives – as briefly discussed in an earlier chapter, such as ”activating the teleporter” or ”collecting keys for locked doors” – in the levels generated at present, and that reaching the exit is a sufficient condition for completing the level, no matter how many enemies have been defeated, most players felt that the presence of such a functional minimap allows them to simply ”rush through” the level, ignoring enemies, and heading straight for the exit. This, along with other potential changes to gameplay mechanics, shall be discussed more thoroughly in chapter 6.

5.3 Learning Through Player Feedback

In this section, we revisit the notion of playstyles (defined in section 4.1). Furthermore, we examine the way player data is collected in the final implementation, and how the nearest-centroid clustering algorithm mentioned in section 4.3 operates.

5.3.1 Playstyles Revisited and Non-Adaptive Generation

To reiterate briefly, in the context of *Greyheaven*, playstyles are abstracted notions of ways a player can engage with the game. They mostly correlate with the player's preferred weapon, i.e. with the player's preferred range from which to engage enemies.

The three playstyles (with their distinctive features) are the following:

- **Close Quarters**: Optimal for shotgun (and sabre) usage. Small, less open rooms with more objects for cover and a greater presence of Enforcers/Shieldbearers and Riflemen.
- **Assault Rifle:** Optimal for assault rifle usage, as evident by the name. Rooms are larger, levels are more open. Amount of cover is moderate, but not constrained too much. Enemies are mostly Riflemen, with a small presence of all other types.
- **Sniper Rifle:** Optimal for sniper rifle usage. Rooms are the largest and most open possible, often with the least amount of cover. Enemies are predominantly Snipers, though other types also make an appearance.

Playstyles are used exclusively for the learning algorithm.

As briefly discussed in section 4.3, learning strategies such as grid search or evolutionary strategies (mapping all possible levels (or a great number of them), and automatically evaluating them) would have had the benefit of not needing any learning to be done on the players' machines: we would learn "what parameters contribute to what", and then we'd be able to recommend certain levels based on data collected about the player's performance.

These methods, however, would actually learn which levels are better for which playstyle according to *the automatic evaluator* (i.e. the NPC agent I would use instead of a player); *not* according to players themselves. This, coupled with the infeasibility of the outlined strategies, required me to consider a different strategy.

And thus I came up with the idea of using my own notions of a playstyle only as a stepping stone, and allowing the system to ideally do away with them entirely, on a per-player basis. But would that be really necessary?

As the creator of the project and sole developer of all its systems, it would be reasonable to assume that I *know*, at least relatively well, which combinations of metamodel parameters are adequately apt for each playstyle. However, one of the first lessons everyone learns when entering the world of game development is that *players and developers are completely different*. Oftentimes the developers' best intentions can be completely misunderstood by players; elements of the game the developers considered obvious may never be discovered by players; or elements considered extremely useful (i.e. weapons, in the case of an FPS) by the developers may be dismissed as utterly useless by the players.

Knowing this, it was reasonable to assume the same may apply to my definition of playstyles. (Note: some data appears to confirm this assumption; details in chapter 6.) For that reason – and to allow playstyles not to be fixed to my understanding of how

the game is to be played, but to adapt to players themselves – I came up with the idea of using a supervised nearest-centroid clustering. Its operation shall be the theme of the next section.

Firstly, however let us discuss how we generate levels *without* any adaptations in play.

I now present the metamodel parameter intervals for generating levels according to *my personal* understanding about it:

- **Close Quarters**: $M \in [0.1, 0.35]$; $D \in [0.4, 1.0]$; $C \in [0.5, 0.8]$
- **Assault Rifle**: $M \in [0.4, 0.75]$; $D \in [0.2, 0.8]$; $C \in [0.6, 0.67]$
- **Sniper Rifle**: $M \in [0.7, 1.0]$; $D \in [0, 0.4]$; $C \in [0, 0.3]$

Where M, D, and C are the metamodel parameters magnitude, density, and constriction respectively.

If we ask the system to generate a level ”optimal” for one of the playstyles, and there aren't enough data points in the set corresponding to the current profile, a level shall be generated by picking a random value for each parameter, constrained to the intervals outlined above.

The values outlined above were chosen after some amount of personal testing; they attempt to provide the closest possible approximation to a level adequate for the given playstyle, to the best of my personal understanding. Since the system *may* be asked to generate a level even if no calibration levels have been completed,

However, if there is a sufficient amount of data points, the clustering algorithm, which we shall discuss imminently (in section 5.3.2), comes into play.

5.3.1.1 Calibration Levels

”Calibration levels” is the name given to a set of ”pre-defined” levels that – as per the name – are designed with the purpose of providing an initial *calibration* for the adaptive system.

There are nine calibration levels in total, three per playstyle. They are accessible through the ”Calibration” tab of the ”Play” window (pictured in figure 5.12.c). A completed calibration levels is clustered the same way that any other level is (using the system we shall discuss in the next section). Calibration levels can be replayed at will.

The calibration levels were chosen to provide a mix between levels clearly matching one of the playstyles, and some with less clearly pronounced ”distinguishing features”, allowing players to make a meaningful choice in their classification.

The parameters that define all calibration levels are listed below:

- **Close Quarters:**

Level 1: $M = 0.2$; $D = 1.0$; $C = 0.67$;
Level 2: $M = 0.25$; $D = 0.5$; $C = 0.67$;

Level 3: $M = 0.25; D = 0.4; C = 0.67;$

- **Assault Rifle:**

Level 1: $M = 0.5; D = 1.0; C = 0.67;$
Level 2: $M = 0.55; D = 0.5; C = 0.5;$
Level 3: $M = 0.60; D = 0.2; C = 0.2;$

- **Sniper Rifle:**

Level 1: $M = 0.8; D = 0.5; C = 0.0;$
Level 2: $M = 0.85; D = 0.6; C = 0.0;$
Level 3: $M = 0.85; D = 0.6; C = 0.3;$

As usual, the ”Challenge” constraint is dependent on the player’s ratings of previous levels.

5.3.2 Supervised Weighted Nearest-Centroid Clustering for Level Recommendation and Playstyle Suggestion

The nature of level classification, or rather of the task at hand, mandates that data from completed levels of a given playstyle is to be used to find an ”optimal” level of that playstyle. With this in mind, I posit that we can add all of the player’s completed levels (of that playstyle) to a *single cluster*, and that we may recommend the cluster’s centroid as the ”optimal level” for the playstyle – naturally based on the information we have in the moment. This allows us to create a surprisingly elegant level recommendation system that relies on player data only.

In fact, the clustering algorithm’s task is twofold: we need to be able to recommend *levels* based on a playstyle; or we need to *infer a playstyle* from a level’s parameters. I shall now examine how I handled this issue.

5.3.2.1 The Clustering Algorithm: Overview

Remember that a level is generated using the three parameters of the metamodel (ignoring the ”Challenge” constraint), all of which are floating-point values ranging from 0 to 1 inclusive. Indeed, this set of parameters can be viewed as a three-dimensional vector.

Upon the creation of a new profile, we prepare three empty clusters – one for each playstyle. Since the ”optimal” playstyles for all levels are ”known”, each cluster is distinct. For this same reason I would argue that the clustering algorithm is *supervised* – our data points are essentially labelled by playstyle.

Upon a player’s completion of a level, we ask them what the optimal playstyle for this level is, *according to them* (using the ”rating” menu, pictured in figure 5.14.a). We then add this level (i.e. its set of metamodel parameters) into the appropriate cluster, *based on the player’s choice*. The system itself thus makes *no assumptions* about the

most optimal playstyle for the level (except suggesting a playstyle for the level), and all choices are left in the player's hands, allowing them to truly curate their experience.

Furthermore, some levels may be more likely to appeal to players than others, even within the same playstyle category. We thus allow *scoring* – upon completing a level, the player is asked to give a rating from 1 to 10 inclusive of the level. Higher is better. This rating is used for the *weight* of the level in the clustering algorithm.

Clusters thus keep track of the "weight" of every node (set of metamodel parameters). If a previously-completed level is added to a cluster, its previous weight is averaged with the current one. All previous completions of a level thus "weigh as much as one completion", allowing the system to adapt faster to a finely-tuned rating changes. In practice, however, the game encourages players to play different levels every time instead of replaying the same level multiple times.

Upon completing a level (and having it clustered "behind the scenes"), the player is given a choice about their next level (pictured in figure 5.14.a). We shall now examine what these choices are and how the adaptive algorithm handles them.

The player may select to play a level:

1. **Optimised for a specific playstyle**, in which case we need to recommend a level of that playstyle; this is described in section 5.3.2.2;
2. **At random**, in which case we need to suggest a playstyle that would be most adequate for a given level; this is described in section 5.3.2.3;
3. **Manually**, by adjusting the parameters by hand. In this case we also need to suggest a playstyle that would be most adequate for this level (as above).

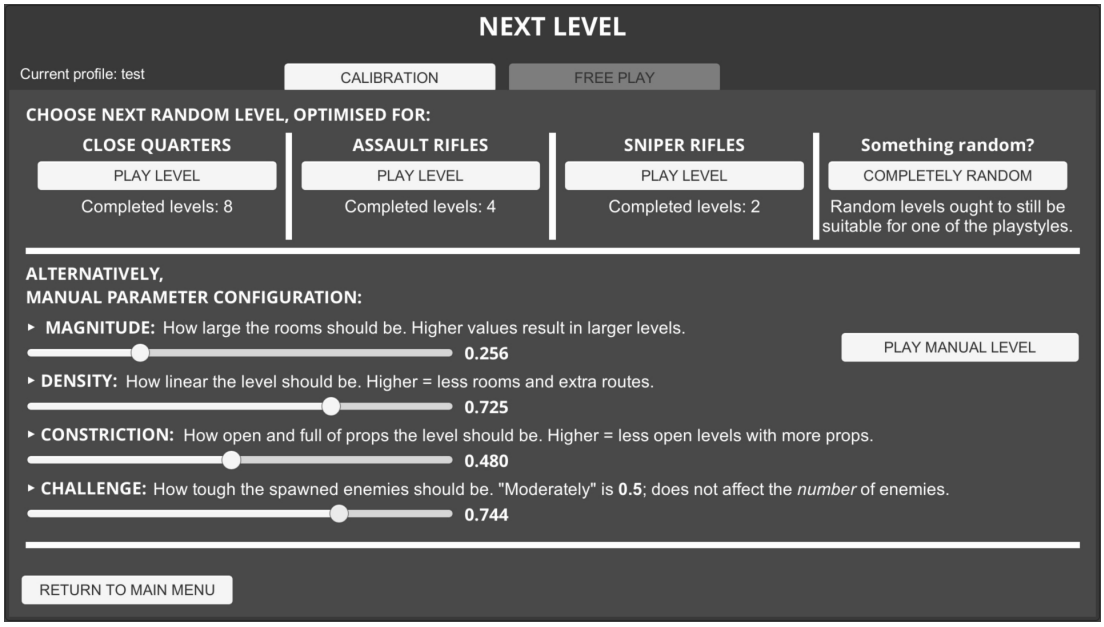
5.3.2.2 Level Recommendation

In most cases (after completing all calibration levels), a player is likely to select a specific playstyle that they would like to play a level of.

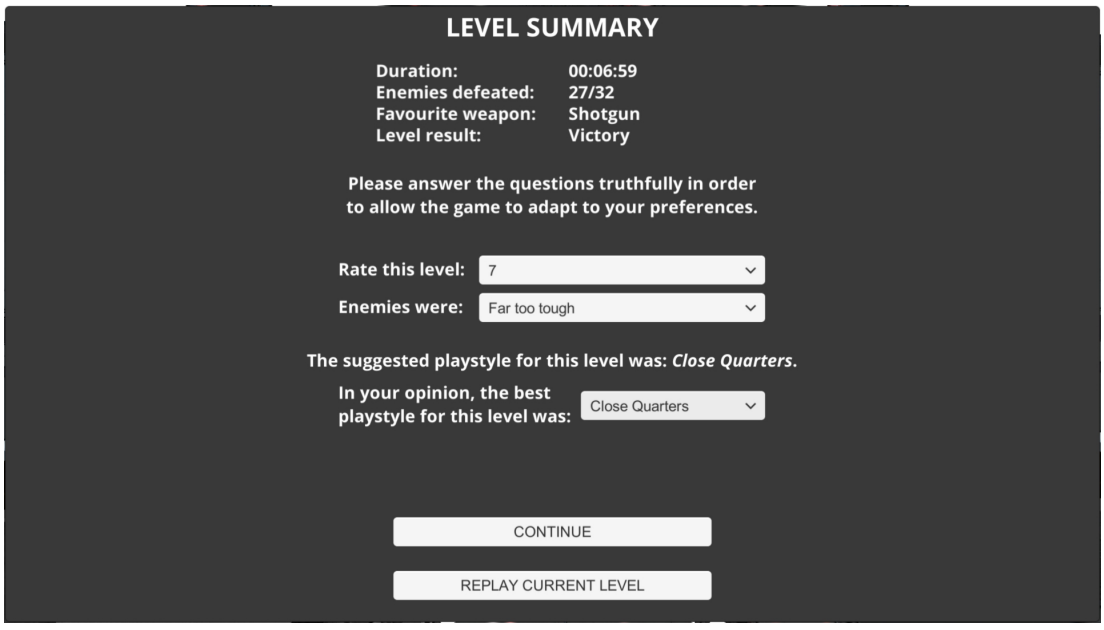
When that happens (and there are three or more data-points (completed levels) in the cluster), the system finds the centroid of the respective cluster, taking individual node weights into account, and applies a small amount of deviance to the result. (to ensure a different level every time, even if it happens to be recommending the same level twice). The result – the effective centroid of the cluster, plus the random deviance – is given to the player as their level recommendation.

The exact process is listed as algorithm 1. Note that in the listing, a set of metamodel parameters, e.g. the resulting value **res** is regarded as a three-dimensional vector of floating-point values, of the format $\{M, D, C\}$, where the three values represent the magnitude, density, and constriction parameters respectively.

For our purposes, we can call this function with *d* set to *true*, we can discard its returned "weight" value, and simply generate a level according to the centroid's parameters, which are stored in the "params" variable of the returned object.



(a) Play window: ”Free Play” menu, allowing players a number of options to choose for their next level.



(b) Rating menu, presented after every level completion.

Figure 5.14: A more detailed look upon two of the important menus present in the game.

Algorithm 1 A function for finding the location and weight of a weighted cluster's centroid. C is the cluster we're interested in, d is a boolean variable responsible for adding deviation or not.

```
1:  procedure FINDCENTROID(C, d)
2:      wts ← 0;                                   // Total sum of weights in the cluster
3:      elem ← 0;                                  // Total number of elements in the cluster
4:      res ← {0,0,0};                             // The resulting vector of parameters
5:      for each node ∈ C do
6:          res += node.params × node.weight;      // Element-wise multiplication
7:          wts += node.weight;
8:          elem += 1;
9:      if d == true then                          // Apply deviation if necessary.
10:         dev = (R × 0.1) − 0.05                 // R is a randomly-chosen value ∈ [0,1]
11:         res += dev                             // The deviance is positive or negative, up to 5%.
12:     res /= weights;                            // Element-wise division
13:     return new Node {params = res, weight = wts/elem}
```

If there are less than three data-points in a specific cluster when a recommendation is needed, we use the "default", assumption-based, non-adaptive level system described in section 5.3. (Ideally, this should not be a frequent occurrence, given the presence of the calibration levels.)

The more levels the player completes, the more the respective cluster grows, and the more accurate the recommendation system gets.

5.3.2.3 Playstyle Suggestion (Level Classification)

Alternatively, the player may pick a completely random level, or tune the parameters manually. In the first case, all three metamodel parameters are chosen at random using the uniform distribution (but the challenge constraint remains chosen as per the default process); in the second, they are chosen by the player (and the challenge constraint is manually set as well).

In these cases we need to essentially do the reverse of what we did before: recommend a playstyle given the level parameters themselves. Unlike the previous use case, which recommended a level, we now need to classify the level into one of the playstyles.

I do this using the "nearest-centroid" method, whereby we find the location (and weight) of the centroid of every cluster, then use distance measurements (whilst taking weights into account) to find the nearest cluster for our level. This is the playstyle we suggest to the player (though again, the player is free to disregard the suggestion).

The process involves calling the same function outlined in algorithm 1, however this time without deviance, with d set to *false*. We do this for all clusters, **if and only if** there are at least three data-points (completed levels) in each cluster. We don't discard the "average cluster weight" values the function also returns, and use them to mark the

weights of the centroids.

We then compare the Euclidean distances between the new level and the cluster centroids, *divided by the weight of the respective centroid*, thus forming a linear relationship between cluster distance and weight (i.e. a twice-as-heavy cluster centroid twice-as-far away will have an "effective weight" exactly the same as the one it's being compared with). Any ties, though rather unlikely, are broken randomly, by selecting one of the minimum-distance clusters.

Nearest-centroid clustering is a better tactic than simply clustering on the basis of all elements, because we know nothing about the *balance* of the data. An extreme (and unlikely, but possible) example would be if a player has completed 200 Close Quarters levels, and only five of Assault Rifle and Sniper Rifle. It is obvious that taking *all* clustered nodes into account would skew the results terribly in favour of the overrepresented cluster.

A question to consider is why one would want to keep track of "centroid weights" at all. The answer is simple: imagine that the average rating (i.e. weight) the player has given to Close Quarters levels is 9, whereas the average rating of Sniper Rifle levels is 4. No matter the amount of completed levels, we may infer that this player has a strong preference for Close Quarters levels in comparison to Sniper Rifle ones; using "centroid weights" takes any such preferences into account, and allows us to bias prospective levels to the Close Quarters playstyle accordingly.

If there are less than three data points in any cluster: The system uses an ad-hoc method to attempt to determine what the optimal playstyle for this level would be. The formula is as follows:

$$val = M + (D - 0.5) \times 0.2 + (C - 0.5) \times 0.25$$

where M, D, C are the three metamodel parameters (magnitude, density, constriction), and *val* is the resulting value. If *val* is lesser than 0.375, the suggested playstyle is Close Quarters; else, if val is less than 0.75, the suggested playstyle is Assault Rifle; alternatively, the level is deemed to be optimal for the Sniper Rifle playstyle.

Magnitude is considered to be the largest predictor of a level's "class". Density is not considered much of a predictor, except that higher values of it tend to reinforce the Close Quarters archetype; Constriction is considered to be a little more important than Density.

I make no claims about the accuracy or validity of this metric, as it was implemented hastily as a "fallback plan".

5.3.3 The "Challenge" Constraint: A Singular Difficulty Metric

The concept of dynamic difficulty adjustment – as we have observed in chapter 2 – has been an area of interest for more than a decade.

Unfortunately, mostly due to time constraints, and the focus of this project being more concerned with the effects of level generation than on level difficulty per se,

Greyheaven has a single, simple metric for difficulty adjustment – the metamodel's `Challenge` constraint.

As stated multiple times now, the challenge constraint is a floating point value ranging from 0 to 1 (inclusive) like all metamodel parameters; it is however not a parameter, and is kept separately, tied to the user's current profile, and is updated upon level completion without making use of clustering or any learning techniques.

This constraint governs the difficulty of spawned enemies (as thoroughly described in section 5.1.1.5), but not their numbers; it changes nothing else. challenge value of 0.5 is regarded as the default and "moderate" difficulty, and is the starting value for a newly-created player profile.

As may be seen in figure 5.14.b, the player's impression of the difficulty of the enemies is gathered as a separate data value, unrelated to the rating of the level or its playstyle. The challenge constraint of the player's profile is adjusted upon level completion, depending on this selection, in a very simple way – the chosen value modifies the current value of the challenge constraint by some amount:

- **Far too tough**: -0.1;
- **A little too tough**: -0.05;
- **Okay**: +0.0;
- **A little too easy**: +0.05;
- **Far too easy**: +0.1;

In other words, if the player feels that enemies are too strong, the challenge constraint (and with it, the difficulty of enemies in subsequent levels) will be decreased immediately by a small amount; or vice-versa if the player believes the enemies are too weak.

This is unfortunately the extent of *Greyheaven*'s dedicated difficulty adjustment metric, at least for now. A suggestion for improvement is briefly discussed in section 7.2.3.

Chapter 6

Evaluation

In this section, we discuss the evaluation methods chosen for the project, and analyse the results of all gathered data. We obtain sufficient evidence to suggest that the proposed hypothesis (”learning can improve a fully-random PCG system”) is promising.

NB: This chapter relies on a familiarity with the implemented system that may be obtained by reading the previous chapters. No special references will be made for concepts addressed in previous chapters.

6.1 The Testers

It is my belief that the selection of testers for an experiment is an important part of the equation. Testers – at least in a system evaluation case like this – should be familiar with the sort of system they'll be evaluating, and should be able to provide at least somewhat competent feedback.

For this purpose, I decided to recruit testers from an online forum dedicated to video games. The site in particular is known as the *TaleWorlds Forum*, the official forum of TaleWorlds Entertainment, developers of the *Mount&Blade* series of first/third-person tactical hack-and-slash genre-bending role-playing games.

I chose this forum solely because of personal reasons, as I have been a member of that community for quite a long time, and was certain that my request for aid would be answered by a sufficient – though by no means large – amount of people. It's worth noting that all of the testers were volunteers, were not paid for their time, and participated due to curiosity and a desire to help.

The total number of testers I managed to enlist was **23** – a far cry from what one would consider a good sample size for a statistical survey, but then again, this is an Honours project, and I have a firm belief that a population of this size is sufficient to draw reasonable conclusions from, at least in the context of this project. (Anecdotally, I have seen other Honours project theses that have done testing on ten people or less.)

All of these testers were people of various ages and backgrounds, predominantly from Western Europe and the United States. The data presented in this chapter is anonymised for reasons of privacy; furthermore, I shan't be considering demographic data to be relevant.

A very relevant characteristic, however, is the fact that all of the testers are fond of video games, have a lot of experience with various types of games, and are at least familiar with the FPS genre. All of them may be considered "qualified" testers, and all were able to give adequate feedback.

Testers were aware that they would be contributing to a research project, and were requested to answer all questions given as truthfully as possible according to them – for questions such as level and difficulty ratings and preferred playstyle..

6.1.1 Testing Process

Every test followed a simple process that shall be documented here.

All testers were initially asked to complete the set of calibration levels, and were told that this was necessary to allow the adaptive system to learn. Testers were allowed to replay calibration levels if they so desired, but were advised that that wouldn't be necessary.

Additionally, different testers had no contact between each other.

Unbeknownst to them, *they were randomly split between two groups* – the *control group* (12 testers in total), members of which were given a build of the game *without* the adaptive systems active; and the *"treatment" group* (11 members in total), members of which were given a fully-operational build of the game. Control group testers instead had the "default", non-adaptive recommendation and suggested systems at play (described in sections 5.3.1 and 5.3.2.3 (under the *"If there are less than three data points..."* paragraph) respectively). It's worth noting that the difficulty adjustment algorithm (modifying the *"Challenge"* constraint, as described in section 5.3.3) was kept the same across both groups.

NB: Since the term "treatment group" seems remarkably out of place in this context, I shall refer to that group as the **"adaptive group"** – an accurate description, since only members of this group interacted with the adaptive procedural esystem.

Despite the crucial difference between the two groups, *all* testers were led to believe that they were playing a game with adaptive procedural generation.

After completing the calibration levels, players were tasked with completing at least ten levels in "free play" mode.

Some data was recorded automatically upon the completion of every level; some data was gathered in the shape of a largely free-form survey given to the tester, in which they were encouraged to provide feedback about several subjects. (All of which shall be discussed in the next section.)

The separation of the tester base into a control and treatment group is a tactic that should be quite familiar to every researcher from any domain, and surely doesn't require a detailed explanation. In this case, it allows me perform a type of "A/B testing" procedure in order to try and determine whether or not the adaptive PCG system does indeed perform better than a "generic" PCG system, based on conclusions drawn from the collected feedback.

6.2 Indirect versus Direct Feedback

There are multiple ways to gather data, as data is usually abundant for any activity. For this investigation, I collect both *direct* and *indirect* feedback. In this section, I shall define what meanings I assign to these terms.

- **Direct** feedback happens, as the name suggests, directly between the tester and me, in the form of a free-form survey after the completion of the game-playing task. Testers were encouraged to describe how they found the game, present any suggestions for its improvement, share their opinions on the level generation system, and state what they think of the concept of adaptive procedural generation. This data, being less precise and "curated" than that from indirect feedback, I would regard as a description of the "general state" of the players' satisfaction with the product. Despite its rather personal nature, I believe this feedback can also prove useful for our analysis.
- **Indirect** feedback, in comparison, is the "hard numbers" – the data we gather when players complete levels. This includes the parameters of the current level plus the player's current challenge rating constraint, as well as the rating that the player has given the level, and the preferred playstyle that he's chosen for the level. I also collect the suggested playstyle for the level, which provides an interesting insight into the differences between players' understanding of what the playstyles are, and mine. These provide real data that we can analyse.

Using these two "types" of data should provide us with as clear a picture as possible – both of the numerical perspective of things, and of how players (believe they) felt about the experience.

6.3 Analysis

In this section, we shall analyse both types of feedback gathered during the testing process.

6.3.1 Indirect Feedback: Level Completion Data

This section shall present the numbers gathered from level completion events.

Firstly, however, let's discuss the data gathered in detail. As mentioned earlier in the chapter, "indirect feedback" was gathered upon a player's completion of a level. **Every one of these "level completion events" included:**

- **The tester's unique ID**, which we consider irrelevant for this discussion;
- **The "chosen playstyle"** that the player classified the level as upon its completion; this is the playstyle that the clustering algorithm would use;
- **The "suggested playstyle"**, i.e. the one that was suggested to the player by the game. This was either the recommendation of the clustering algorithm, or was "known" in advance – determined by the respective button the player chose ("level optimised for..."), or was suggested by the "fallback" system;
- **The parameters of the completed level**, largely used for determining whether this level was a calibration level or not;
- **Rating**, i.e. the out-of-ten score the player gave the level;
- **Current Challenge**, i.e. the value of the player's challenge constraint at the time of level completion;
- **Is Control** (inferred by the user ID) – whether or not the player was in the control group or not;
- **Is Calibration** (inferred by the level parameters) – whether or not the completed level was a calibration level.

Ultimately, the results were interesting, and support my hypothesis – suggesting as much as 35.4% improvement in level ratings when the adaptive system is active, amongst other conclusions.

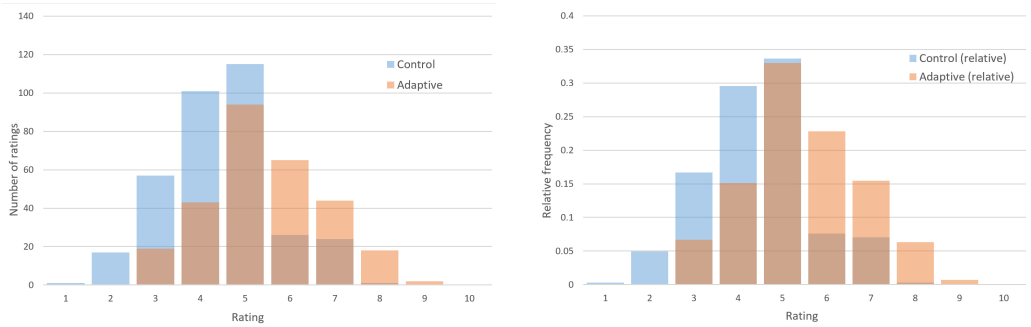
We shall examine the results from three separate perspectives: one for level ratings, as they describe the users' satisfaction with their experience; one for *misclassifications* (cases where the suggested and chosen playstyles differ); and finally, one investigating the potential correlation between the challenge rating of a level and the ratings given – as common sense dictates that a greater challenge results in a greater satisfaction, at least among some competitive players.

6.3.1.1 Level Ratings: A Metric of Satisfaction

Of the "indirect" data gathered, I would argue that level ratings are the most important part. Assuming users were giving them truthfully – as they were instructed to do – those ratings are the best and most accurate way we have of measuring player enjoyment.

The following list outlines the most important aspects of the data gathered about level ratings:

1. The 23 testers were split into the **12-member Control group** and the **11-member Adaptive group**; again, the Control group was given a build of the game *without* the adaptive system running; the Adaptive group received the operational game; however, *both* groups believed that they were playing the adaptive version of the game.



(a) Absolute distribution of ratings across both groups.

(b) Relative distribution of ratings across all groups.

Figure 6.1: Histograms of the absolute and relative distributions of level ratings for both groups. Ratings from the adaptive group are fewer overall, given that less levels were completed; the relative distribution however displays that the normalised amount of **5**s given in both group is more or less identical. The Adaptive group displays a larger (relative and absolute) number of higher votes.

2. There were **627 level completions in total**, **342 by the Control group** and **285 by the Adaptive group**.
3. Of those 627 levels, **279 were on calibration levels** and **348 were on "free" levels**.
4. The **average level rating on both calibration and free levels** in the Control group was **4.44**. The average level rating in the Adaptive group was **5.47**.
5. **On calibration levels alone**, the results were quite similar, with the average level rating in the Control group being **4.41** and the average rating in the Adaptive group being **4.68**.
6. **On free levels, however**, the difference is more pronounced: the Control group gave free levels a rating of **4.46** on average, whereas the Adaptive group gave them a **6.04** rating on average.
7. The most frequent rating for both groups was **5**, but the Adaptive group saw a larger amount of ratings higher than 5.
8. **Ratings of 1 and 2**: Members of the Control group rated one level with a rating of 1, and gave a rating of 2 to 17 levels; members of the Adaptive group found *zero* levels to be that dislikeable.
9. **Ratings of 9 and 10**: In contrast, members of the Control group gave a rating of 9 or 10 to *no levels at all*! Members of the Adaptive group gave a score of 9 to two levels in total, but no level received a score of 10.

The distribution of level ratings in both group types can be seen in table 6.1. The number of levels completed, as divided by the group that completed them and the type of levels they were, may be seen in table 6.2. The average level ratings, divided by group and level type, are presented in table 6.3.

Rating	Control	Adaptive	Total
1	1	0	1
2	17	0	17
3	57	19	76
4	101	43	144
5	115	94	209
6	26	65	91
7	24	44	68
8	1	18	19
9	0	2	2
10	0	0	0

Table 6.1: Number of level ratings per group. **5** is the most common ratings in both groups, but the Adaptive group has given a greater relative amount of higher ratings.

	Control	Adaptive	Total
Calibration	160	119	279
Free	182	166	348
Total	342	285	**627**

Table 6.2: Number of level completions per group and level type.

Table 6.3: Level rating averages at a glance. *"Gr."* is short-hand for *"group"*, and *"avg"* – for *"average"*.

Control gr. Calibr. level avg	4.41
Adaptive gr. Calibr. level avg	4.68
Control gr. Free level avg	4.46
Adaptive gr. Free level avg	6.04
Avg level rating, Control gr.	4.44
Avg level rating, Adaptive gr.	5.47

Additionally, two histograms plotting the level ratings are presented in figure 6.1. We see that both distributions bear a slight resemblance to the normal distribution. Even though the Adaptive group completed less levels in total, the number of levels given a rating of 6 or higher is higher in the Adaptive group; furthermore, the relative frequency distribution of ratings points out that the relative frequency of the rating **5** is almost identical between the two groups (with values of 0.3362 for the Control and 0.3298 for the Adaptive group), as well as that the distributions are almost symmetrical.

These distributions point towards a higher average rating given in the Adaptive group, suggesting that the adaptive procedural system performs better than the non-adaptive one. The average level ratings (listed in table 6.3) confirm this, with the total Control group average rating being **4.44**, and its Adaptive counterpart being **5.47** – a significant difference of more than a single unit!

Observing the averages per ”level type” – that is Calibration or Free – demonstrates that ratings for Calibration levels are essentially identical for both groups (averages of **4.41** and **4.68** for the Control and Adaptive groups respectively). This is the expected outcome, given that both groups were tasked to play through the same identical set of calibration levels.

I’d like to draw the reader’s attention to the averages for Free levels – that is, those levels for which the Control group was forced to use the random, non-adaptive level recommendation system, and the Adaptive group was presented with the real game.

The differences in this category are staggering: an average of **4.46** for the Control group, and an average of **6.04** for the Adaptive group – a difference of over one and a half units, or, alternatively, of **35.4%**!

Given the aforementioned similarity in Calibration level ratings, I posit that the difference in Free levels seen here is direct evidence of the benefit that the adaptive procedural system provides. I believe this is reinforced by the fact that the average Control group rating suffers to see any improvement during Free levels, suggesting that the non-adaptive system is seen by players as no better or worse than the set of pre-selected Calibration levels.

6.3.1.2 Misclassifications: A Metric of Learning

Even though the previous section contains evidence that players seem to enjoy the adaptive experience more – which is the most important factor under consideration, and which essentially proves my hypothesis – I would still like to investigate the results further. Whilst there are many reasons why players may enjoy adaptive levels more – such as levels being generated with more optimal sets of parameters for a certain playstyle – a good question to consider is *whether the adaptive system learns **better***. To investigate this, we shall examine the cases of level ”misclassification”.

Misclassifications, that is instances in which the player chooses a different playstyle than the one the system suggested – cases where the ”chosen” and ”suggested” playstyles differ. Misclassifications are worth looking into, because they underline cases where the player's preferences differ from those defined in the ”fallback systems”. Since the game given to the Control group doesn't learn, one may reasonably expect that the number of misclassifications in the Adaptive group, especially in terms of Free levels, would be lower. If that is the case, an argument can be made that the Adaptive system learns to better predict/suggest playstyles according to the players' preferences.

Of all 627 completed levels, 198 were misclassified by the procedural system, whereas 429 were classified correctly. The misclassified levels were thus 46.15% of all levels, which is a less than ideal a result, and suggests that there may be issues with the classification or recommendation systems. (Naturally, we seek to minimise cases of misclassification so as to ensure that the game recommends the ”correct” levels to players.)

Of these 198 misclassifications, 96 occur between Close Quarters/Assault Rifle levels (i.e. either the suggested or the player-chosen playstyle was one of those), and 102 – between Assault/Sniper Rifle levels. There were no cases of Close Quarters/Sniper Rifle level misclassifications. This suggests that at least shotgun- and sniper-rifle-optimised levels are sufficiently different to prevent any and all cases of uncertainty.

The absolute distribution of misclassified levels per group and level type is presented in table 6.4. Free levels were misclassified less frequently than Calibration levels for both groups, which suggests that both the adaptive and ”non-adaptive” procedural suggestion systems perform somewhat well in recommending levels more fit to the players' requests.

	Control	**Adaptive**	Total
Calibration	63	47	110
Free	52	36	88
Total	115	83	**198**

Table 6.4: Number of level misclassifications per group and level type.

	Control	**Adaptive**
Calibration	39.38%	39.50%
Free	28.57%	21.69%

Table 6.5: Percentage of misclassified levels per group and level type.

Table 6.5 presents the percentage of misclassified levels per group and level type, i.e. the ratio of misclassified levels of a given type by a given group, to the number of *all completed levels* of that sort – allowing us to see the ”normalised” look of the data, ignoring disparities in the number of completed levels.

We can see that in the Control group, **39%** of all Calibration levels were misclassified, whereas only **29%** of the Free levels failed to correspond to the player's idea of the optimal playstyle; this matches the suggestion presented in the previous paragraph.

More interestingly, however, the Adaptive group performs similarly in terms of Calibration levels, with **40%** of them misclassified; the amount of Free levels with a ”wrong” suggested playstyle however **is just 22%**, compared to the Control group's 29%.

This suggests that the clustering-based level recommendation system (even the ”fallback, default, assumption-based” one provided to the Control group (and used when adaptation is impossible for the Adaptive group)) presents a substantial improvement over my assumptions used when creating the static set of Calibration levels.

Furthermore, the adaptive clustering algorithm appears to further reduce cases of misclassification by some 7% – a marginal improvement, but an improvement nonetheless; and coupled with the improved ratings for levels of that type (discussed in the previous section), I would posit that the adaptive system provides a significant benefit to the players' experience.

6.3.1.3 A Flaw in the Plan: On ”Unlearnable” Clusters

The misclassification information outlined above seems positive – adaptive levels are misclassified less often. **However, this doesn't negate the issue** that 46% of *all* completed levels *were* misclassified.

Upon some consideration over the gathered data, I may present a hypothetical scenario that may describe a possible reason for this phenomenon: Perhaps this is a side effect of the way the recommendation system works: imagine player A is very much fond of the assault rifle weapon, and considers all Close Quarters levels to be better completed using the assault rifle. The player will classify every ”Close Quarters-optimised” level as an Assault Rifle level, and the level data shall be added to the ”Assault Rifle” cluster. Any Close Quarters level that the player subsequently requests will be similar to those

before it, as the system won't have been able to learn what levels exactly the player believes are better fit for shotgun usage (i.e. belong in the Close Quarters cluster), and will be unable to recommend levels that aren't going to be misclassified.

On the other hand, the system would then be able to generate "Assault Rifle-optimised" levels with – what I would imagine to be – a very low chance of misclassification; however, the game doesn't enforce a particular playstyle choice on players, potentially voiding this benefit (e.g. if the player continues to request Close Quarters levels instead of requesting Assault Rifle levels).

Cases like these – that I honestly considered highly unlikely whilst designing and creating the system – appear to be more common in practice than I believed they would be. This creates a flaw in the recommendation system; a flaw that must be addressed in any future iterations. Some suggestions on this topic will be presented in chapter 7.

Fortunately, we have already seen that player ratings are better for the Adaptive groups, which suggests that if an "edge case" issue like the one described above were in effect, then they weren't as dramatic as described. On the other hand, players were requested and encouraged to try out levels of different types – which would *not* be the case in a real-world environment, allowing this potential flaw to bloom to its full capacity.

No matter whether or not the present system was affected by something like this, this is evidence that the clustering system must be redesigned to better accommodate possibilities like these.

Additionally, one of the reasons for this occurrence may be an imbalance in the available weapons. As shall be discussed in section 6.3.2.2, it is possible that the Assault Rifle was simply too good of a weapon, and would outclass the other weapon choices in some environments designed to be better-suited for them.

Of course, nothing is stopping these two factors from combining and being at play at the same time.

6.3.1.4 Correlation between Difficulty and Ratings

An interesting experiment to conduct is an examination of whether level ratings would have any correlation with the preferred difficulty at the time the rating was recorded. Note that an investigation about the potential correlation between two variables such as these is a problem in its own right; this brief section will merely present my results obtained after a spontaneous spur of curiosity.

My looking into this was inspired by the discovery of positive correlation between level rating and difficulty in "the Quake paper" [10]. However, the case with *Greyheaven* appears to be different.

In the world of video games, especially on the PC platform, games with high difficulty are usually regarded as "better" than easier games; in fact, there are entire communities of players who believe that a game cannot be too difficult, and any others who have

difficulties with specific titles simply don't possess the skills required to play them, or that such players (often referred to as "casuals", i.e. gamers who are not serious enough) are "unworthy" of playing games, and that it's their fault for the "dumbing down" of franchises considered "elite".

Famous titles regarded as too difficult by a substantial portion of their players include the action RPG series *Dark Souls*, and in more recent times, the platformer *Cuphead*. Whilst I personally appreciate challenges in games and would indeed recommend to players having difficulties that they should practice more, I was curious to see whether some trace of this social phenomenon may have been present in the results I gathered for my game.

I thus calculated, naïvely, the correlation coefficient between the entire set of level ratings and their corresponding challenge ratings. The correlation coefficient between these two variables is **-0.033**, suggesting that there is no correlation between difficulty and player ratings in *Greyheaven* levels.

I personally am of the belief that a heightened degree of challenge in a game is a good reason for evaluating the experience as "better", so this is an interesting result. My educated guess would be that this occurs for several reasons: testers were asked to give objective ratings for all levels, and knew they were a part of a research experiment; there was no air of "elitism" about the title – as "difficult games", such as those mentioned above, usually rely on word of mouth and marketing to attract the crowd of players who very much enjoy hard challenges; and finally, *Greyheaven* may simply not be challenging enough – as numerous testers suggested this in their direct feedback (which we shall shortly discuss).

Again, an investigation of this specific topic is well-suited for a study of its own, and my "experiment" here was simple and insignificant. It would have been interesting if there was a correlation, though.

6.3.2 Direct Feedback: Tester Opinions

In the previous section, we analysed the numeric data gathered from "indirect feedback", i.e. level completions. In this one, we'll examine the "direct feedback" left by testers upon completing their involvement in the experiment. Instead of directly supporting or disproving the hypothesis – whether the adaptive system performs better than the non-adaptive one – it is my belief that this data provides a more "flexible" representation of players' overall enjoyment of the game.

Indirect feedback is best summarised with several lists, which shall be listed imminently. Lists are separated by broad topic.

6.3.2.1 Levels, Overall Ratings, Adaptivity

- When asked how they would rate the entire experience, the overwhelming majority of players said it was "okay", and that they believe it has potential to be

a real game. Three testers (including one in the control group) said they were very excited about the game and requested to be placed on a mailing list for any potential news and updates.

- In general, members of the Adaptive group were more excited about the game and the concept of adaptive procedural level generation.
- All testers agreed that the game generates sufficiently varied levels with sufficiently different themes, which accommodate different playstyles. Several testers suggested that more props, room themes, and colour schemes be added to improve visual variety. Some testers also suggested that having non-rectangular rooms would make the game more interesting.
- The overall majority of players reported that they didn't notice the system adapting to their preferences. Whilst expected for the Control group, this was a little surprising to hear for the Adaptive group. Despite this, more members of the Adaptive group felt that the adaptive (Free) levels were "more interesting" than the Calibration levels.

6.3.2.2 Mechanics and Content

- In general, players felt that the weapons were adequately different from each other and fill adequately different niches. Players had no issue understanding the concept of playstyles. Additionally, of those several testers who expressed enthusiastic affection for a certain weapon, the predominant majority highlighted the assault rifle and described its positive properties at length, citing versatility, power, and low recoil. This – along with the substantial number of level misclassifications – suggests that part of the problem may lie with the rifle covering some of the intended roles of other weapons.
- No testers found the sabre interesting or useful, citing the fact that they have infinite ammo and a very effective shotgun at their disposal instead. Some testers believed that the sabre's unfinished animations (that they were forewarned about) make it "weak" (despite the sabre dealing substantial damage per swing).
- Some testers praised the "time dilation" ability of the sniper rifle as well as the real-time scope zooming effect (something not that common in video games).
- In general, players felt that the roster of enemies was different enough to keep gameplay interesting. Most players praised the different behaviours that different enemy types exhibit, such as riflemen attempting to withdraw from an advancing player. Some players felt that Tier II Shieldbearers (on very high difficulties) were too strong and "not fun" to play against.
- However, the more than half of all players stated that enemy types (often excluding Shieldbearers) were far too weak and posed little challenge to a player familiar with the behaviour of the weapons. Overall, they felt that the game may need to be more challenging in order to be more interesting.

6.3.2.3 Issues and Suggestions for Improvement

- Most users felt that the presence of a mini-map that always guides them directly to the exit, coupled with the fact that levels end when the exit is reached (independently of the number of enemies defeated and so on) makes the game too easy to ”rush” – that is, run straight through all rooms, ignoring levels and heading for the exit. I acknowledged this shortcoming, as it was spurred on by the constraints of the project.
- To improve this, testers suggested that other objectives be added to levels – such as collecting keys to unlock locked doors, having to activate the ”teleporter” (level exit) in order to leave, or having to defeat a certain number of enemies before being allowed to proceed.
- Most users reported the known visual bugs – such as weapon muzzle flashes sometimes appearing at wrong locations, or the shotgun ejecting shells from its front side. Some users reported the identical animation sets used by enemies, or the placeholder enemy sniper rifle model (textured in a single bright colour) as bugs; I clarified the nature of these conditions to them.
- Some testers suggested that a greater variety of weapons or enemies would make the game more interesting. Those comments were expected, as they are usually given by players of any FPS game, especially one with a small roster of weapons.
- As noted above, the majority of testers believed that enemies ought to be more challenging. Some suggestions included spawning more enemies, making them ”smarter”, making them coordinate their attacks against the player, and giving them more powerful weapons.

6.4 Conclusions

In general, player opinions give *Greyheaven* an average rating – which is an achievement, considering the game was made by one person, with no budget, in an extremely short time-frame. More importantly, however, numeric data gathered during the testing process suggests that **the adaptive PCG system does indeed perform better than the non-adaptive one**.

The average player rating in the Adaptive group, given the fully operational adaptive system is over a unit (on the scale of 1 to 10) higher than the Control group that had a non-adaptive system – with an average rating of 5.47 from the Adaptive group against an average of 4.44 from the Control group. In the case of Free levels – that are generated differently in the two groups – the averages change to 4.41 from the Control group, and 6.04 from the Adaptive group – a rather substantial difference of over 1.5 units; this amounts to an improvement of 35.4%.

Furthermore, cases of level misclassification (where the system recommends the ”wrong” playstyle for a level from the point of view of the player) are reduced by 7% in the Adaptive group.

This, coupled with the generally mildly positive ”direct” feedback, suggests that *Greyheaven*'s adaptive procedural level generation system **does** perform better than a completely random system of a similar nature, **thus confirming my initial hypothesis**.

However, some flaws have also been exposed – a potential case of inadvertent weapon imbalance, and a potential case of the system being unable to learn correctly, denoting a potential design oversight (discussed in 6.3.1.3).

I would thus argue that the experiment was successful within its scope, providing promising results; but still in need of further improvement. As a proof of concept, I feel that Project Greyheaven serves its purpose and provides a practical example that adaptive PLG systems can be beneficial; however I wouldn't claim that the implementation and algorithms provided here are of optimal quality. Much can be done for the overall improvement of all systems – from the parallelisation and optimisation of the room and level generators, to adding all respective assets in the stead of the current placeholders.

From a pragmatic standpoint, if a playable game using adaptive PLG techniques can be produced by a single developer over just a few months, at the sole cost of several thousand work-hours and some personal funding, and have it be received moderately well by an effectively random sample of testers, it would be logical to extrapolate that a large development team creating an industry-class product would reap substantially greater benefits (in terms of reduced development time and costs, as well as improved player enjoyment) from a system such as the one described in this thesis.

Additionally, I would like to remind the reader that such adaptive systems need not be restricted to video games: any product or project that requires the utilisation of pseudo-random algorithms and involves interaction with various users may benefit of any operational techniques of this sort.

Some suggestions for future improvements shall be presented in the next chapter.

Chapter 7

Discussion and Future Work

In this section, we turn our attention to what could've been and what was not. I provide wistful discussion on what I wanted to do, and what was too grand in scope to fit in the Honours project; additionally, I provide (reasonable) suggestions for improvements derived from the direct user feedback, chiefly related to the more particular aspects of the game.

7.1 Mechanics: What I Couldn't Do, but Wish I Could

As mentioned a lot earlier in this dissertation, this project was inspired by a paper by Jonathan Robers and Ke Chen, discussing the implementation of an adaptive level generation framework on top of the classic shooter *Quake* [10].

I have described my motivation for creating this as a stand-alone game in section 3.2. Aside from the reasons listed there, however, and aside from the research potential, I wanted to experiment with a few mechanics of the first-person shooter genre – to try something from a purely engineering perspective.

7.1.1 First-Person Mêlée

First-person mêlée combat is something that is notoriously difficult to do right. The presence of the sabre in this game – which was also planned to serve a special role, potentially deflecting enemy rifle shots and being the only weapon effective against Shieldbearers – would have been the perfect opportunity to experiment with designing a proper, interesting, mechanically-deep hand-to-hand combat system.

There are multiple interesting things that can be done for first-person mêlée – directional combat, "power attacks", kicking, and so on.

Alas, however, that wasn't meant to be, as this was a research project first and foremost, and time was very limited.

7.1.2 Shieldbearers With Shields

In sections 4.1 and 5.2.2, I discuss the nature of the ”Shieldbearer” enemy type, and how they came to *not* have shields.

In the ideal world, Shieldbearers would have been a heavily-armoured and slow-moving enemy type, equipped with a large ”tower shield”, always facing the player, thus becoming virtually immune to the ranged weapons. This would give the sabre a niche and a specific use, as sabre attacks – ideally ”power attacks” – would allow the player to stun the Shieldbearer, or perhaps rend them of their shield.

As before, I had no time to implement something like this. But it’s certainly something that should be done in the future.

7.1.3 Other Objectives

As described in section 6.3.2.3, many of the game’s testers suggested that other objectives should be added to levels to make them more interesting and prevent the ”rushing” strategy.

Different objective ideas include:

- collecting keys for doors in some order, with keys found in ”non-essential” rooms or perhaps dropped from defeated ”officer” enemies;
- finding and activating a ”generator” to activate the level exit teleporter;
- giving the players an objective to find an item or defeat an enemy instead of always looking for a designated exit room;
- adding a level of persistence to the game and having every completed level have additional meaning depending on the theme – granting ”experience points” that unlock new weapons or enemies; or providing some form of in-game resource that the player needs for some newly-introduced mechanic.

Indeed, there is a plethora of possibilities. What there *wasn’t*, was sufficient time to better work on the ”gamey” aspect of this project.

Should the game be presented to the general audience as is, players would become bored remarkably quickly by having all level’s goals be the exact same, and by the lack of substantial depth in the ”core game loop”. Extra objectives like those described here would be rather beneficial in remedying this.

7.1.4 More Guns

Anecdotal evidence: *Battlefield 1*, a well-known ”AAA” (high-budget) multiplayer-focused shooter released in 2016, had about five primary weapons per player class at launch; with four player classes, this equates to about 20 primary weapons. All of these weapons were quite different from each other, filling a reasonably different niche

in the game, and upheld class distinctions adequately; and this list doesn't include the variety of hand-to-hand weapons (mostly cosmetic variations of the same weapons), as well as the sizeable list of secondary weapons (i.e. pistols and revolvers).

Despite this quite sufficient, at least in my opinion, variety available, most of the online community kept complaining for months that the weapons were too few.

If that is the case for *Battlefield 1* – an incredibly expensive product, created by a large team of people, and a title in a well-known series – it is reasonable to conclude that players would *not* be satisfied by a first-person shooter with just three (or four) weapons. For a real game, a greater arsenal would be needed; additionally, some testers suggested just this as an improvement they would like to see.

7.1.5 Health and Ammo Pick-ups

In the current state of the game, the player has regenerating health and infinite ammunition ("ammo") for all weapons. This was done due to time constraints, and is far from what I envisioned.

The presence of infinite ammo reduces the viability of the sabre to almost literally nothing, and removes any sort of resource management from the game. The player can't be punished for carelessly wasting ammunition, or rewarded for making their shots count.

Regenerating health, especially *slowly* regenerating health as the current implementation provides, means that players are only required to wait for a decent amount of time, outside of combat, if they're at risk of death. This is not an interesting way to operate with health, and I was very much hoping to have "health packs" around the level, maybe dropped from defeated enemies. This would eliminate the prolonged and dull "waiting for my health to regenerate" moments, would bring an element of necessity to avoiding enemy attacks or eliminating enemies effectively, and would reward players for exploring the level and conserving these limited resources.

7.2 Learning: Making it Better

Whilst I'm fortunate to have had my hypothesis confirmed by the testing process, I have pointed out that there are potential flaws in the way the clustering system works.

I remain content with my choice of "three playstyles" that would define constraints for the ways players interact with the game, yet there are things that could have been done better with them as well.

7.2.1 Subdividing Playstyles

The current state of the system assumes that every level is best optimised for one particular level. What I could've achieved with the infeasible learning strategies outlined earlier in this monography (the subsections of section 4.3), however, is to allow for degrees of "partial fitness" whilst classifying levels. Perhaps a level is "good" for the Close Quarters (CQ) style with a score of 0.91, for the Assault Rifle (AR) style with a score of 0.64, and for the Sniper Rifle (SR) with a score of 0.11 (scoring according to some arbitrary metric).

This would better accommodate the fact that different players may have different preferences in levels, perhaps allow me to define learnable "intervals of recommendation": "This player considers CQ levels to be fit for that playstyle only if their CQ score is above 0.8, but *this other player* likes the shotgun more, so give him CQ levels with ratings above 0.7", and so on.

This would also help account for cases where certain players have a similar preference for two different weapons.

In short, it would make the system "smarter" and more flexible, and this would be beneficial.

7.2.2 Removing Edge Cases

In section 6.3.1.3, I describe a potential scenario that may be a contributing factor in the surprisingly high amount of level misclassifications recorded during the experiment. Sometimes plans don't work out exactly in practice, and the only way to see this is to look at the data.

This proves that the clustering system, for all its elegance and effectiveness for both recommendation and classification purposes, was not a perfect match for this problem; or at least it wasn't a perfect match in the way I used it.

The system thus needs to be redesigned to handle "edge cases". Whilst I wouldn't attribute the issue to the clustering itself, I would argue that having playstyles not be completely discrete (proposed in the previous section) would be beneficial in such scenarios.

Introducing some level of interdependence between playstyles would complicate the learning process, but would perhaps remedy cases in which a player simply considers one playstyle "to be" another. Alternatively, coming up with cleverer ways of creating levels without examples of what the player would consider "adequate" for a given playstyle is surely bound to help in that case.

At any rate, more research needs on the subject needs to be done; I don't currently consider myself capable of suggesting a solution which is guaranteed to work better than the clustering system I have at present.

7.2.3 Investigating Better DDA Methods

As briefly mentioned in chapter 2, an interesting and seemingly rather elegant DDA solution (arguably) successfully applied to a first-person shooter game was Hunicke's *Hamlet* system for dynamic difficulty adjustment (DDA) [5].

The initial plans for this project involved experimentation with DDA methods of a similar caliber, but due to the narrow constraints of the project, chiefly the lack of time, implementation of anything similar was quite impossible.

Introducing a "discrete" DDA system to *Greyheaven* is likely to provide additional benefits towards player enjoyment and satisfaction.

7.2.4 Not Asking for Ratings

In "the Quake paper", Roberts and Chen make some poignant points about data collection for games: *"most of existing methods need to identify the target players style/type and to measure their experience by learning from their feedback or behaviour, which leads to a burden to target players."*

They additionally state that *"one of challenges faced in such work was the existence of many outliers, people who behave in an irregular fashion. If one was to collect direct feedback from the public with questions, e.g., by asking them whether they were having fun, it would be expected that many people would be difficult to process not only because they were outliers, but also because they may deliberately or accidentally provide feedback that does not match their true opinions. When crowd-sourcing is applied to PCG, how to deal with noisy data therefore is an ongoing critical issue."* [10]

Whilst the small amount of testers for *Greyheaven* seems to have successfully excluded such outliers or malevolent individuals who would provide rubbish data, this is a very valid consideration; furthermore, real players would be a lot less inclined to bother with answering questions about ratings.

Research should be done on this subject, and different methods for circumventing these issues should be investigated. Reducing the number and severity of interruptions of gameplay to collect feedback may well see players' overall satisfaction improve.

7.2.5 "Inter-Player Knowledge Transfer"

As-is, the system learns what every player likes after the player completes some amount of levels. However, knowledge about *other* people's preferences is not used in any way.

It sounds reasonable to assume – though I admit that I am unable to devise a concrete plan at the time of writing – that it would be possible to apply other players' preferences to the experiences of new players, allowing the system to learn faster or produce better results.

Such a feature would greatly improve the operation of learning system, especially for new players.

Bibliography

[1] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK, 1996.

[2] Luigi Cardamone, Georgios Yannakakis, Julian Togelius, and Pier Luca Lanzi. Evolving interesting maps for a first person shooter, 04 2011.

[3] Julie Dorsey and Holly Rushmeier. Advanced material appearance modeling. page 3, 08 2008.

[4] K. Hartsook, A. Zook, S. Das, and M. O. Riedl. Toward supporting stories with procedurally generated game worlds. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 297–304, Aug 2011.

[5] Robin Hunicke. The case for dynamic difficulty adjustment in games. 265:429–433, 01 2005.

[6] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: Dynamic difficulty adjustment through level generation. 06 2010.

[7] R. Lopes, E. Eisemann, and R. Bidarra. Authoring adaptive game world generation. *IEEE Transactions on Computational Intelligence and AI in Games*, PP(99):1–1, 2017.

[8] Ricardo Lopes, Ken Hilf, Luke Jayapalan, and Rafael Bidarra. Mobile adaptive procedural content generation. In *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games, co-located with the Eigth International Conference on the Foundations of Digital Games*, Chania, Crete, Greece, may 2013. Society for the Advancement of the Science of Digital Games. ISBN 78-0-9913982-1-8.

[9] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience in super mario bros. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 132–139, Sept 2009.

[10] Jonathan Roberts and Ke Chen. Learning-based procedural content generation, 2013.

[11] Noor Shaker, Georgios N. Yannakakis, Julian Togelius, Miguel Nicolau, and Michael O'neill. Evolving personalized content for super mario bros using gram-

matical evolution. In *Proceedings of the 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2012*, pages 75–80, 2012.

[12] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, Sept 2011.

[13] Georgios N. Yannakakis and Julian Togelius. Experience-driven procedural content generation. *IEEE Trans. Affect. Comput.*, 2(3):147–161, July 2011.