Developing a Top-Down Shooter with Dynamic Difficulty Adjustment via a Genetic Algorithm

Rory Simpson (2454453)
BSc. Artificial Intelligence and Computer Science
Supervised by Dr. Samuel Montero Hernandez
University of Birmingham

17th April 2025

Abstract

This project investigates the implementation of a dynamic difficulty adjustment (DDA) system in a 2D top-down shooter game with procedurally generated levels. The system uses a genetic algorithm to evolve enemy parameters based on player performance data and incorporates an adaptive loot mechanic to further balance gameplay. The aim was to improve fairness and maintain player engagement by adjusting difficulty in response to individual playstyles. Comparative testing between a static difficulty version and the DDA-enabled version showed that the dynamic system consistently achieved higher and more stable fairness metrics. Player feedback also indicated improved balance and a generally positive reception of the adaptive systems. While limitations remain, particularly in terms of player perceptibility, the results demonstrate the potential of genetic algorithms and context-aware mechanics to enhance procedural game design and player experience.

Contents

Chapter 1

Introduction

1.1 Context and motivation

Dynamic Difficulty Adjustment (DDA) in video games refers to the technique of automatically modifying the difficulty level of the game in real time based on the player's performance. The aim is to maintain an optimal level of challenge that keeps the player engaged without causing frustration or boredom to the extent that they want to stop playing. For example, if the player consistently defeats enemies with ease, the game could adapt by making enemies more aggressive in their tactics, such as increasing their attack frequency or coordinating their movements to flank the player. Conversely, if the player is struggling, the game could reduce their aggressiveness, giving the player a better chance to succeed. The underlying motivation for this project is to explore whether DDA can be an effective tool in enhancing the player experience.

1.2 Problem statement

Traditionally, games have accommodated players of varying skill levels by offering a choice of several static difficulty settings (such as "easy", "normal", or "hard"). While this approach is simple to implement, it comes with a significant limitation: it assumes that players can accurately assess their skill level before they begin playing - a task that is often difficult, especially for newcomers to a genre. A difficulty setting that feels appropriate early on in the game may later become too easy as the player improves, or too difficult as the game's mechanics grow more complex. Although players can usually change the difficulty setting mid-game, doing so can disrupt the flow of gameplay and immersion.. Dynamic difficulty adjustment (DDA) offers a potential solution to this problem and has been adopted in a small number of commercial games such as *Left 4 Dead*, where an AI 'director' dynamically adjusts enemy spawns and item drops. However, implementing an effective DDA system presents several challenges. Accurately determining player skill and adjusting the game appropriately is a complex task, and excessive or intrusive modifications can undermine players' sense of agency and fairness. Enforcing a constant level of challenge can lead to feelings of frustration, as players may feel that they cannot truly master the game and its mechanics, resulting in a diminished sense of accomplishment. In addition, players may want an easier and more relaxed experience (or a particularly challenging one), rather than what the system considers to be the most optimal. These are just some of the issues surrounding DDA in games, which could help explain why it has not yet seen widespread adoption.

This project addresses the problem as an optimisation task: determining how to adapt enemy behaviours and resource allocation in real time to maximise gameplay fairness, based on player performance metrics. This is approached through the use of a genetic algorithm, which evolves enemy parameters over time, combined with an adaptive loot system that complements the difficulty scaling. The goal is to investigate whether such a system can balance challenge and fairness while maintaining player engagement.

1.3 Project aims

The aim of this project is to design and implement my own DDA system, with a focus on addressing the key issues identified in existing implementations. The system will use a genetic algorithm to evolve enemy parameters over time in response to how the player is performing. To streamline the development and testing of this system, a bespoke top-down shooter game will be created. Building the game from the ground up provides complete control over all gameplay mechanics, enemy logic, and

environmental variables, allowing for unrestricted experimentation with difficulty adaptation. This approach ensures that the DDA system can be fully tailored, tested, and evaluated in an environment specifically designed to accommodate it.

The project will be structured as follows:

1. Existing DDA implementations will be reviewed to identify their strengths, weaknesses and common challenges, such as accurately assessing the player's skill in real time.
2. The core game will be designed and developed using a static difficulty setting. This version will operate without any dynamic adjustments. By first focusing on a static model, this will provide a baseline metric for how players interact with the game and help identify which aspects should be controlled by the DDA system.
3. The DDA system will be designed and integrated into the game, then fine-tuned as necessary.
4. Blind play tests will be conducted using two versions of the game, one with a static difficulty and one with the DDA system enabled. Players will not be informed which version is which. This will allow for an unbiased comparison between the two difficulty models.
5. The effectiveness of the DDA system will be assessed by evaluating the fairness of the DDA enabled version compared to the static difficulty version.

By addressing the limitations of traditional difficulty settings and experimenting with a custom DDA system, this project aims explore the viability of such systems in improving the player experience.

Chapter 2

Literature review

This section explores existing DDA systems in games, beginning with their theoretical foundation in a psychological engagement model known as flow theory. It then identifies the common challenges faced when implementing such systems and evaluates their overall effectiveness. These insights will serve to inform the design of the DDA system proposed in this project.

2.1 Historical background

Flow theory, introduced by Csikszentmihalyi, describes a psychological state of complete immersion and engagement in an activity [3]. In this state, time seems to pass quickly, focus is heightened, and the experience is both enjoyable and intrinsically rewarding. Key components for achieving flow include having a clear goal, a sense of agency and a balance between the challenge of the activity and the skill of the individual . He identified that games fit naturally into this framework. Games are uniquely well-suited to supporting these conditions. They typically provide structured goals, immediate feedback, and adjustable challenges, making them ideal environments for inducing flow. However, maintaining this state throughout gameplay is a nuanced task. If the challenge is too low, the player may experience boredom; if too high, they may become frustrated. As illustrated in Figure 1, the optimal experience occurs within the 'flow channel' - a zone where skill and challenge are closely matched and evolve together:

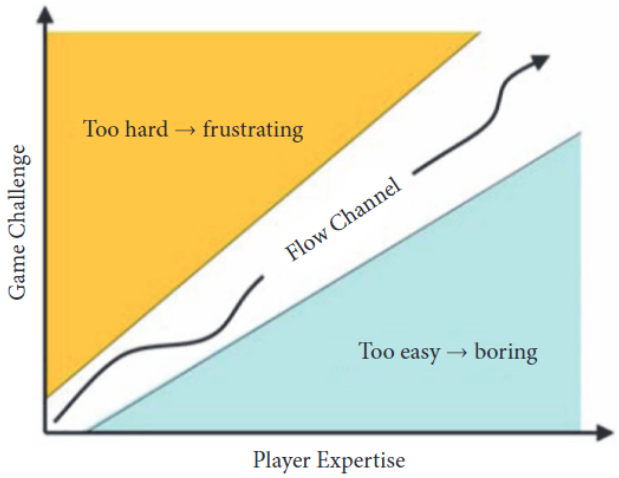


Figure 2.1: Flow channel concept proposed by Csikszentmihalyi

An key implication for game design is that player expertise tends to increase over time as familiarity with the game's mechanics, systems and strategies deepens. To keep the player within the flow channel, the game must adapt its difficulty in step with the player's growth. This creates a compelling case for implementing DDA.

In recent years, DDA has garnered significant attention from both academic researchers and industry practitioners, reflecting a growing recognition of its potential to enhance the player experience [7]. Early discussions of DDA were largely theoretical or limited to anecdotal use in commercial games. However, as game studies and human-computer interaction research have matured, DDA has become a frequent subject of empirical investigation. These studies have examined a wide range of techniques for adaptive gameplay, including rule-based systems, player modelling, machine learning and evolu-

tionary algorithms [21]. Contemporary research focuses not only on how DDA influences in-game performance, but also on its broader impact on player enjoyment, perceived fairness, immersion, and retention. This growing interest reflects the dual challenges of designing DDA systems that are both technically effective and perceptually seamless. To that end, Andrade et al. propose that a robust DDA system should meet three core requirements [1]:

- The system must quickly **assess and adapt** to the player's initial skill level, which can vary significantly between individuals.
- It should continuously **track and respond** to changes in player performance, whether they reflect improvement or decline.
- The system's adjustments must remain **believable and unobtrusive**, avoiding behaviours that might reveal artificial manipulation (e.g., visibly weaker enemies or erratic AI actions).

Together, these principles provide a useful framework for evaluating and comparing the DDA approaches discussed in the remainder of this chapter.

2.2 Existing DDA systems

2.2.1 The Hamlet System

Hamlet is a DDA system built using Valve's Half-Life game engine, operating through a set of libraries that include functions for monitoring game statistics, defining adjustment actions and policies, executing those actions, displaying data, and generating play session traces [8]. It continuously monitors various game statistics to estimate the player's future state. When it predicts an undesirable state, the system intervenes and adjusts the game's settings. An undesirable state is characterised by a shortfall in some part of the player's inventory, such as health or ammunition. The relative abundance or scarcity of items in a player's inventory directly impacts their experience, and Hamlet aims to maintain a balance that keeps players engaged. The main heuristic function for determining the game's difficulty is the damage the player takes over time. A cumulative distribution function $F(d)$ is built, which represents the probability of receiving d or less 'damage' on a given tick:

$$F(d) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{d} e^{(x-\mu)^2/(2\sigma^2)} dx$$

The probability of a shortfall P_s at time t is then calculated as the cumulative probability of damage **exceeding** the initial level z:

$$P_s = P(x(t) > z) = 1 - P(x(t) < z) = 1 - F(z)$$

If a shortfall is predicted, the adjustment process begins. This involves two types of actions: reactive and proactive. Reactive actions modify elements currently in play, such as the accuracy or damage of attacks, while proactive actions adjust elements that are not yet active, like the properties of enemies which have yet to spawn. Proactive changes grant the developer more power over the game's behaviour, but their effectiveness is less certain because they take longer to come into effect. Reactive changes have a more predictable impact, but they run the risk of disrupting the player's sense of disbelief. The authors highlight that DDA systems naturally involve trade-offs in regards to the type, number and frequency of adjustments that are made such that the game's behaviour remains believable yet responsive. Another key takeaway is importance of player inventory in influencing the difficulty of the game, in addition to more typical factors such as the damage or accuracy of enemies.

2.2.2 Dynamic scripting

A DDA system using dynamic scripting was implemented into an online role-playing game called MiniGate [2]. This system was augmented with three machine-learning inspired techniques: dynamic weight clipping, differential learning and adrenaline rush. Dynamic scripting forms the backbone of the system, where a rule database containing behaviours of varying complexity is used to assemble AI agents dynamically. Each rule has a weight that affects its probability of selection. The system continuously adjusts these weights based on the agent's performance against the player, measured using a team fitness function $F(g)$ which is based on the health of each agent in the team g:

$$F(g) = \sum_{c \in g} (\frac{1}{2N_g})(1 + \frac{h_t(c)}{h_t(0)}$$

Where c denotes an agent, N_g is the total number of agents in team g and $h_t(c)$ is the health of agent c at time t. By definition, the team has a fitness of 0 if they lose.

The rules for the agents are grouped into rule-classes based on their complexity (e.g. novice, expert), where each rule-class is associated with a class weight μ_i indicating difficulty. A complexity switching mechanism is used to increase the weights of complex rules and decrease the weights of simpler rules as the player's expertise increases, or vice versa if the player's expertise decreases. To prevent a bias towards certain rules being selected more often, dynamic weight clipping is used to adjust the minimum and maximum weights of each rule-class to enforce variability. The learning rate of the agents is initially high to allow for them to quickly near the player's skill, before being reduced to allow for fine-tuning. This is intended to prevent the player feeling overwhelmed or cheated by the speed of the agents' adjustment. Finally, the adrenaline rush mechanism is used to set a learning threshold for the player. When player learning exceeds this limit, the agent adaptation slows to prevent the player being punished after having improved in some significant way.

Testing found that the agents had greatly improved win rate consistency (lower standard deviation) across players of various skill levels, compared to when using their standard behaviour. Also, the modularity and generality of the proposed system make it applicable across multiple genres, especially where player-versus-AI interactions are central.

2.2.3 Probabilistic graphing

Xue et al. proposed a framework that defines DDA as an optimisation problem aimed at maximising player engagement [20]. Focusing on level-based games, they model the player's progression as a probabilistic graph, where each state has two dimensions, the current level k and trial t. The model allows for three possible transitions: advancing to the next level, retrying the current level, or entering the churn state, which is an absorbing state that signifies the player has left the game and is unlikely to return. The transitions between states are governed by probabilities that depend on the game's difficulty at those states. The framework seeks to prevent players from entering the churn state by optimising the transition probabilities (win rates) in each state to maximise the player's stay time in the progression graph.

The objective function R is defined as the expected total number of rounds a player will play through after reaching a state. This function is computed based on the player progression model, where the reward $R(k, t)$ at a specific state is calculated as the weighted sum of the rewards of adjacent states, incorporating the transition probabilities between those states. The optimisation problem is formulated as follows, where W is the set of win rates across all states:

$$W^* = \arg\max_W R_{1,1}(W)$$

An efficient dynamic programming technique was used to solve this problem. The expected total number of rounds a player will play is defined as the reward function, which is computed iteratively

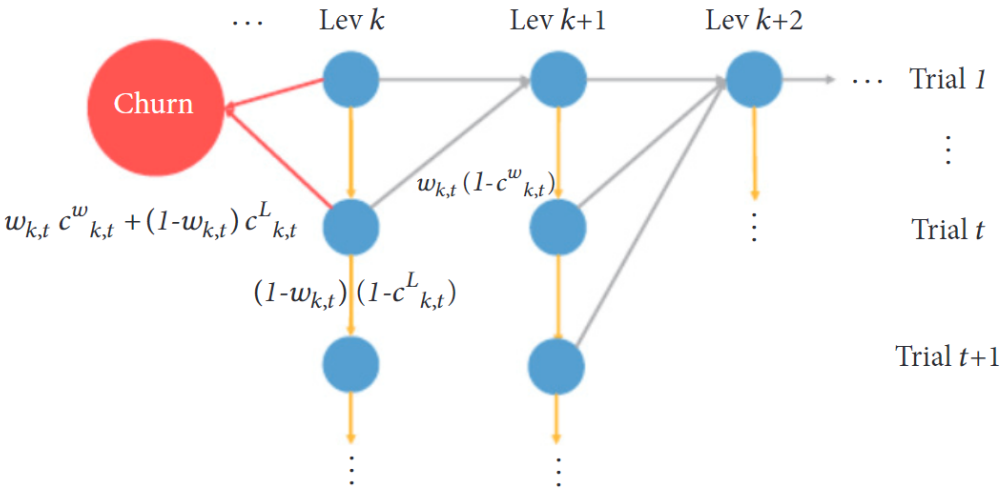


Figure 2.2: Probabilistic graph of the player's progression through a level-based game [20]

based on the transition probabilities between states. The solution involves maximising the reward function while adhering to constraints on win rates. The system was tested in a mobile match-three game developed by Electronic Arts, resulting in significant increases in core engagement metrics, such as the total number of rounds played and gameplay duration, while maintaining a neutral impact on monetisation compared to the control group that did not have DDA enabled. This framework can be extended to more complex, non-linear games (e.g., role-playing games) by incorporating additional dimensions in each state. While probabilistic modelling takes a high-level view of engagement, other systems focus more directly on the player's in-game state. One such approach is the Hamlet system..

2.2.4 Genetic algorithms

A genetic algorithm approach was used to evolve enemy NPCs in a wave-based survival shooter game [6]. A diverse initial population is created using random genes to maximise coverage of the search space. The genes for each enemy are defined below:

Gene	Values	Description
Appearance	0-2	bunny, bear or elephant
Weapon	0-4	punch, flamethrower, missile launch, bomb, thunder
Range	1.5-15	distance between the NPC and its target
Health Points	30-500	character's life
Damage	1-20	character's attack damage
Attack Speed	0.5-2	The lower the value, the faster the attack
Movement Speed	1-5	The reference value is 3 (player speed)
Healer	0;1	1 is healer, 0 is not
Run away	0;1	1 run away with 25% of health
Accuracy	1-100	chance to hit target

Figure 2.3: Genes for a survival shooter enemy implemented in [6]

NPC fitness is evaluated post-death based on three health-related characteristics. 10 points are given for escaping the player or healing another enemy, 0-4 points for every point of damage dealt or

healing given and 0-2 points for every second lived. The normalised fitness function is then given by the following equation:

$$E(i,t) = Min + (Max - Min) * \frac{rank(i,t) - 1}{N - 1}$$

Where Min and Max are the bounds for fitness (0.9 and 1.1 in this case), $rank(i,t)$ is the individual i's rank in the generation t and N is the number of individuals in the generation.

Three selection methods were then tested: cut selection (elitism), fitness-proportional roulette selection and tournament selection. Cut selection was found to consistently yield the best results for fast adaptation. Single point crossover was used to produce offspring from the elites to fill out the next generation. During the game, the player fights waves of enemies, where each wave consists of 20 enemies. The game ends when the player dies, and enemies evolve every 6 waves (120 enemies), with the top 20 enemies being preserved between generations. This cut selection ratio was determined based on testing with various configurations, the results of which were as follows:

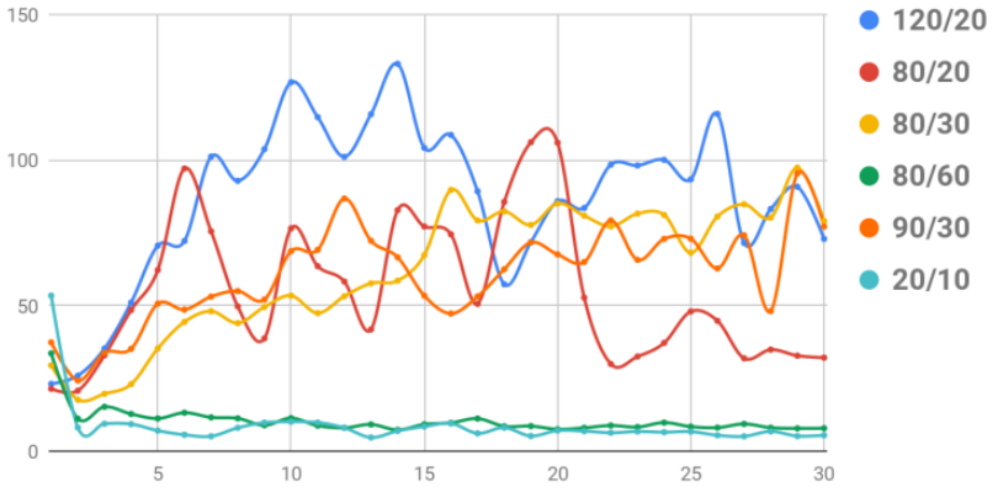


Figure 2.4: Analysis of different cut selection configurations [6]

The 120/20 configuration was chosen over the 80/30 configuration, despite its higher variability over time, due to the importance of fast initial convergence in a game setting. To evaluate the system, 18 participants played through waves until they died. On average, players survived 12 waves (2 generations) and played for 20 minutes. The difficulty was perceived as 'normal' by most players, with no players considering the game very hard, and only one considering it very easy. The majority of players noticed a gradual increase in difficulty. However, the study focused on player's with pre-existing experience in these types of games, so its general adaptability cannot be guaranteed. The authors also acknowledge that alternatives such as neuroevolution (combining genetic algorithms with neural networks) would likely perform better.

2.2.5 Neural networks

Such an approach was proposed in another study, which uses a multilayer perceptron neural network that adjusts its weights via a genetic algorithm [12]. The game is a first-person fighting game between a player and an AI character. The neural network outputs potential actions the agent can take: choosing a weapon, doing a strong or fast attack in a certain direction, or blocking in a certain direction. The agent counters the player's most frequently used weapon and selects attacks based on potential damage. It blocks when anticipating a strong attack or when stamina is low. A fitness function evaluates the agent's performance against the player's skill, considering damage dealt, damage taken, hits blocked, and hits missed. After each fight, weights are recorded, and new sets are generated from the top performing datasets to minimise the skill difference. This approach allows for quick and effective

adaptation to the player's skill level, creating believable and fair AI behaviour. A key advantage is the system's ability to respond to significant skill fluctuations by using a catalogue of the player's long-term performance. However, the neural network requires retraining with any changes to input or output parameters, and online training results can vary across different game instances.

2.2.6 Reinforcement learning

A reinforcement learning system was developed to train a fair AI agent that adapts its attack strategy to balance challenge and enjoyment for the player [10]. The authors implemented a simple turn-based game where the player attempts to win as many battles as possible against a single AI opponent. The player and opponent take turns performing basic actions such as attacking or healing. The game focuses solely on battling enemies, and at the start of each battle, both characters are restored to full health points. The SARSA algorithm was selected for training, as it is typically preferred over Q-learning when the agent's performance during the learning process is more important than the final performance. SARSA is an on-policy algorithm that learns a policy by interacting with the environment and updating its knowledge based on the actions it actually takes. The agent should slowly increase in difficulty according to player performance, rather than finding the most optimal moves possible. The agent's state space is defined by the health points of both itself and the player:

State	Player HP	Agent HP
0	75	==
1	[51-74]	==
2	[31-50]	==
3	[16-30]	==
4	[1-15]	==
5	<Agent HP	[66-74]
6	<Agent HP	[46-65]
7	<Agent HP	[31-45]
8	<Agent HP	[16-30]
9	<Agent HP	[1-15]
10	[66-74]	<Player HP
11	[46-65]	<Player HP
12	[31-45]	<Player HP
13	[16-30]	<Player HP
14	[1-15]	<Player HP
15	0	-
16	-	0

Figure 2.5: States for a SARSA RL agent for DDA

A positive reward is given to the agent for winning a battle, and a negative award for losing. The rewards are swapped if the player wins or loses 5 times in a row. The system was evaluated through two studies. The first evaluated the average moves taken by the agent and the player across 10 sets of 30 battles played against players of various skill levels. It was found that the players were usually quick to discover the best way to use the available moves, while the agent required more time to learn and sometimes overused certain actions such as healing. The second study assessed player experience using five gameplay types: easy (no RL), hard (no RL), and three variations with different exploration-exploitation ratios. Ten participants played three games of each type and rated their experiences via a questionnaire. The best-rated setup used RL with a 30/70 exploration-to-exploitation ratio. Results suggested that RL-based agents can deliver a more engaging and balanced experience, though future work could explore their performance in more complex game environments. Additionally, some penalty mechanism to discourage repetitive agent behaviour could further enhance gameplay quality.

2.2.7 Emotion-based

Frommel et al. propose a novel DDA system that adjusts game difficulty based on the player's emotional state, rather than traditional performance metrics like health or score. This is done through a parametrized difficulty system where difficulty is explicitly defined through adjustable parameters, and changes are informed by self-reported emotions collected via dialogue-based self-reports (DBSR) with a non-player character (NPC) [9]. To test the emotion-based DDA (EDDA) system, a game called SpaceJump was built - a vertical scrolling platformer.

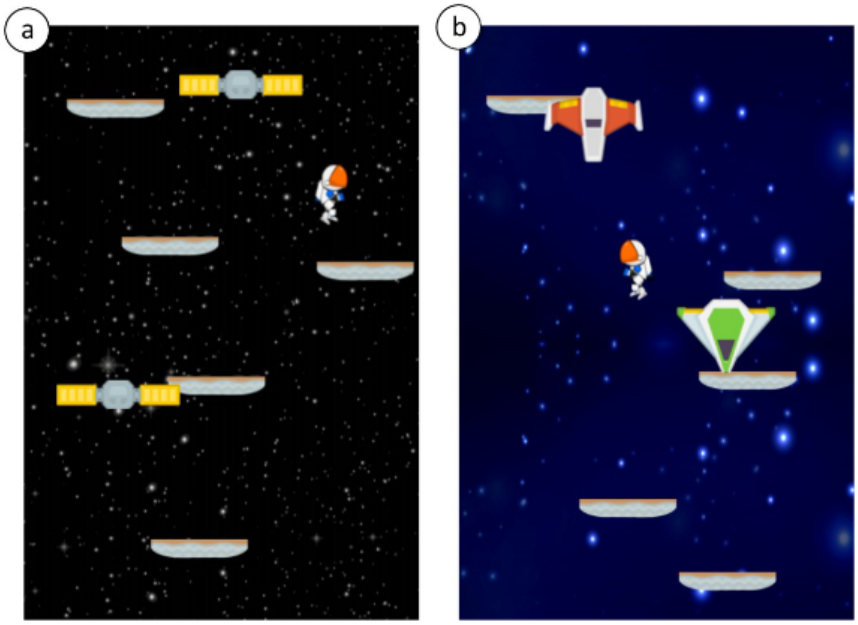


Figure 2.6: Two levels from SpaceJump, a vertical scrolling platforming game

The game consists of five levels, each lasting 1 minute, where the player must avoid obstacles and reach new planets. Players report their frustration and boredom on a 5 point scale (1 = low, 5 = high) after each level via a conversation with an NPC, in order to prevent immersion-breaking interruptions in the experience. Difficulty is defined via three parameters: the number of obstacles, the proportion of those platforms that move horizontally (in addition to the default vertical movement) and the speed at which the platforms move:

Parameter	**Easy**	**Medium**	**High**
Number of obstacles per level	0	5	10
Horizontal movement of platforms	0 %	50 %	100 %
Speed of platforms (in pixels per second)	50	130	210

Figure 2.7: Game parameters used to define difficulty in SpaceJump

These were combined into 17 difficulty levels mapped linearly across the range of parameter combinations. Each time DDA is applied, if the boredom and frustration values differ, the difficulty of the next level is increased or decreased by the sum of both values (clamped to $+-5$). If both values are high (≥ 4), the player's performance is used to decide the direction of change, such that winning leads to a difficulty increase and losing leads to a decrease. If both values are low (< 4), the difficulty remains the same. To test the system, 66 participants played 3 blocks of 5 levels under one of three difficulty conditions: one with EDDA, one with a constant difficulty and one with a linearly increasing difficulty. The EDDA system resulted in fewer deaths over time, higher ratings by players and a greater perceived competence at the game. The DBSRs were found to be as accurate as the traditional questionnaires used for the non-EDDA systems, but didn't significantly improve the experience, possibly due to the repetitive design of the dialogue used. Overall, the paper presents a generalisable, effective and computationally lightweight approach to implementing DDA.

2.3 Key takeaways

While a wide range of DDA systems have been proposed and implemented, several common challenges persist across approaches. Chief among these is the difficulty of balancing responsiveness with believability—systems must adapt quickly enough to maintain engagement without making the player feel artificially aided or unfairly punished. Rule-based systems like Hamlet offer interpretability and control, but they often struggle to generalise across player types and game states. Conversely, machine learning-based systems such as reinforcement learning and neural networks provide flexibility but typically require large amounts of data, extensive tuning, and can suffer from inconsistent or opaque decision-making. Emotion-based systems introduce valuable psychological insight but often rely on self-reporting or intrusive sensing methods that can disrupt immersion.

Additionally, many of the reviewed systems are tightly coupled to specific genres or gameplay structures, raising concerns about generalisability. Several also focus primarily on short-term adaptation within a single session, neglecting long-term player engagement and evolving mastery. There is a noticeable gap in lightweight yet effective systems capable of procedurally adjusting challenge in ways that remain believable, scalable, and sensitive to individual play styles over time.

Given these challenges, this project adopts a genetic algorithm (GA) based approach to DDA. GAs offer a promising middle ground - combining adaptability and scalability with the ability to encode design constraints and evolve behaviours over time. Unlike black-box learning models, GAs provide interpretable parameter sets and can be tuned to optimise for specific fairness or engagement criteria. By drawing on the strengths of GAs while addressing known limitations through careful design and evaluation, this project aims to contribute a practical and generalisable solution to the problem of dynamic difficulty in fast-paced, replayable games.

Chapter 3

Concept, design and methodology

3.1 Research context and objectives

This project adopts a computational and experimental research approach, focusing on the application of a genetic algorithm to address a well-known challenge in game design: maintaining an engaging and balanced difficulty level that adapts to individual players in real time. The core objective is to design and implement a DDA system capable of evolving enemy behaviour and loot distributions in response to player performance, while preserving game balance and perceived fairness. The research aims to contribute to the field by offering a modular, data-driven approach to DDA that can be evaluated both quantitatively (through gameplay metrics) and qualitatively (through perceived fairness and challenge).

3.2 Game concept and core mechanics

The game developed for this project is a 2D top-down shooter called **Mutagenesis**. The player is tasked with fighting their way through endless rooms of enemies in a procedurally generated dungeon. Designed for short, replayable sessions, the game emphasizes fast-paced combat, strategic decision-making, and responsive enemy encounters. This section provides an overview of the core gameplay mechanics. It covers the player's movement and control scheme, the combat system and weapon interactions, as well as the scoring and progression loop that underpins the gameplay structure. These elements are tightly integrated with the DDA system described later on, forming the foundation upon which it is implemented.

3.2.1 Player movement and controls

The game uses a top-down perspective, where the camera is positioned above the player character to provide a clear view of the surrounding environment. The player character is moved using standard directional input, either via keyboard (WASD) or controller (left analogue stick). Rotation is handled independently of movement, allowing the player to move in one direction while aiming in another. The player character rotates to face the reticle used to aim during combat.

Figure 3.1: Top-down view of the player with rotation towards the reticle

3.2.2 Weapons and ammo

Weapons are found as pickups within the game world. When a new weapon is collected, the player's current weapon is automatically dropped at their location. This introduces meaningful trade-offs in resource management, as picking up a new weapon may result in the loss of remaining ammunition in the previous one. Weapons are divided into three categories, each with distinct characteristics:

- **Rifles** — High accuracy and moderate fire rate, ideal for precise, mid-range combat.
- **SMGs** — Fast fire rate with lower accuracy and damage, suited for close-quarters engagements.
- **Shotguns** — High burst damage at close range, with slow fire rate and limited reach.

Firing behaviour depends on weapon type: rifles and SMGs use a hold-to-fire mechanic, enabling continuous fire, while shotguns fire once per button press.

Each weapon comes with a preloaded magazine. Beyond that, players maintain a persistent reserve of ammunition for each weapon type (rifle, SMG, and shotgun) which remains available even after swapping weapons. When equipping a new weapon of the same type, players retain this reserve, enabling strategic decisions about when to switch and reload. This system encourages ammo conservation and forward planning during longer play sessions.

When unarmed, the player defaults to a melee attack. This provides a reliable fallback option, ensuring the player is never completely defenceless. The melee system includes a quick strike for short range damage and a charged lunge attack that propels the player forward and deals increased damage. This offers tactical versatility, allowing players to close distances or finish weakened enemies. Melee combat is especially valuable during intense encounters where ammunition is scarce or a weapon is lost, and it enables a more aggressive, high-risk play style for those who favour mobility and close engagement.

3.2.3 Enemies

Enemy behaviour is governed by a finite state machine (FSM), whereby each enemy transitions between distinct states based on environmental cues and player interactions. This system supports modular, reactive AI design that is efficient and adaptable to dynamic gameplay.

Enemies can exist in one of five primary states:

- **Roaming** – In the absence of player interaction, the enemy patrols random points within a predefined radius of their spawn point.
- **Investigating** – When alerted by nearby gunfire or another enemy, the enemy moves toward the suspected player location. If they fail to locate the player after a short period, they return to the Roaming state.
- **Chasing** – Upon spotting the player, they begin actively pursuing the player in an attempt to reach attack range.
- **Attacking** – Once the player has been spotted and is within attack range, enemies stop moving and focus on aiming and firing at the player.
- **Staggered** – When critically damaged, enemies become staggered instead of dying outright. In this state, they are temporarily incapacitated and can be finished off using a special player ability.

Each enemy is randomly assigned a weapon type (rifle, SMG, or shotgun) upon spawning.

3.2.4 Combat

Combat in the game is designed to be fast, tactical, and rewarding, combining ranged firefights, close-quarters melee, and resource-driven decision-making. Two firing systems are used depending on the weapon type:

- **Raycast Shooting** is used by rifles and SMGs. This method checks for a hit along the firing direction by casting an invisible ray. It ensures highly responsive and immediate feedback during shooting. Raycasts are computationally inexpensive, making them ideal for these higher fire rate weapons.
- **Projectile Shooting** is used by shotguns. Multiple bullet objects are fired in a spread pattern. These projectiles travel through space and their trajectory is affected by the game's physics.

Enemies have a chance to drop weapons or ammunition upon death, introducing a light resource economy into the gameplay loop that challenges players to balance immediate survival with long-term sustainability in firepower and resources.

The player's survivability is managed through a layered damage system that includes both health and armour. The player can carry up to four armour plates, which absorb damage before health is affected. Once armour is depleted, damage is applied directly to health. Health and armour can be restored through pickups found in the environment: health pickups restore a portion of base health, while armour pickups replenish one plate. Additionally, the player can execute a powerful **finisher move** on staggered enemies. This not only instantly eliminates the enemy but also replenishes one armour plate. This encourages players to take calculated risks and rewards aggressive but skilful play. The armour system provides a safety net, allowing players to recover from mistakes and continue high-risk scenarios without instant elimination.

3.2.5 Level design

Each level in the game is a procedurally generated dungeon floor composed of interconnected rooms. The dungeon is organised into two main types of rooms:

Main rooms form the core progression path through the level. Each one contains a group of enemies and cover placements, creating focused combat encounters. Players must clear all enemies in a main room to unlock the exits, allowing travel to other rooms including the next main room.

Side rooms are optional and branch off from main rooms in various directions. These rooms do not contain enemies but instead offer loot chests. Their inclusion encourages players to explore beyond the critical path and allows for a recovery window between combat encounters.

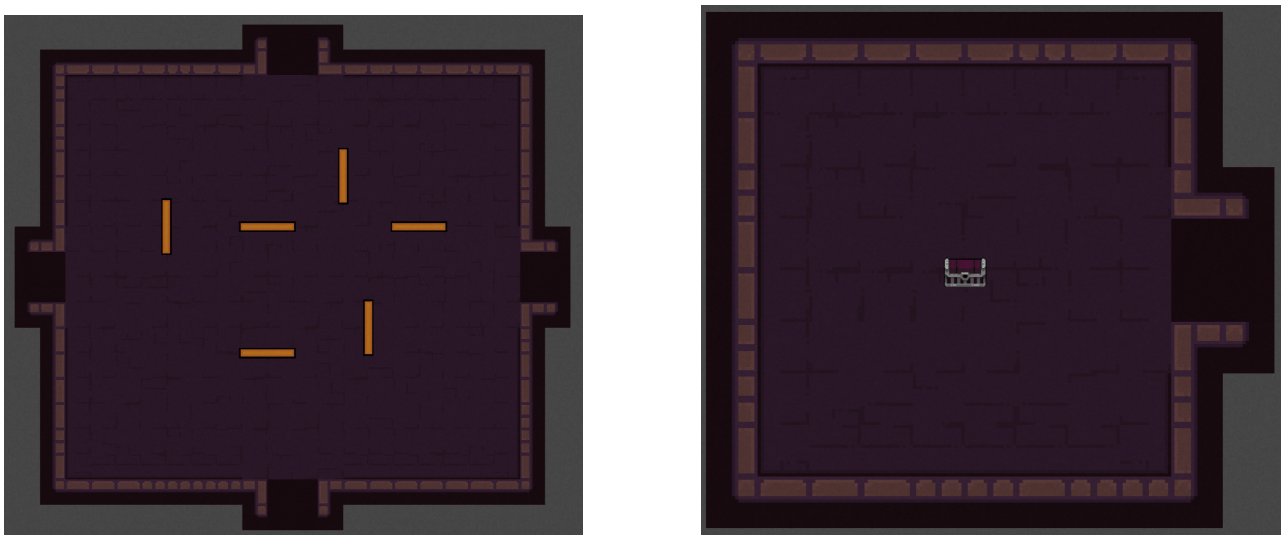
At the beginning of each level, the player spawns in a **start room**, a small safe space to prepare for the challenges ahead.

At the other end of the level lies the **end room**, which contains a portal to the next floor of the dungeon. Reaching this room signifies the completion of the current level.

3.2.6 Score system

The score system serves as the core feedback and progression mechanic in the game, rewarding good performance and encouraging varied combat strategies. It promotes both immediate satisfaction and long-term engagement by awarding points for skilled play, while visually and audibly acknowledging player achievements.

The system tracks several key metrics throughout gameplay:



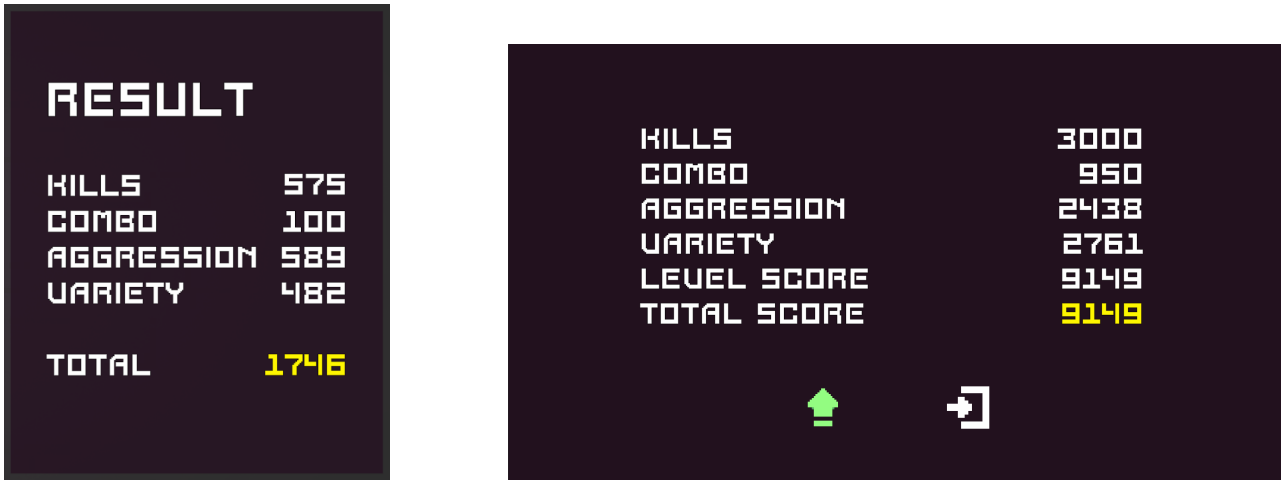
(a) Main room example

(b) Side room example

Figure 3.2: The two main room types in a level

- **Kill score** – Points awarded for each enemy defeated, with bonus points for using a finishing move.
- **Combo score** – Accumulated by chaining multiple kills within a short time window. If the player continues performing without interruption, their combo grows in value before being automatically finalised after a delay.
- **Aggressiveness bonus** – Calculated based on how assertively the player engages enemies. Higher scores are awarded when players consistently push the fight rather than taking a passive or overly cautious approach.
- **Variety score** – Encourages players to switch up their combat approach. This score is based on the diversity of weapon usage, discouraging repetitive strategies and promoting adaptability.
- **Room score** – The total score accumulated within each room, composed of the values above. Room scores are tracked individually and summarised at the end of each level.

When a room is cleared, the system finalises any active combo and calculates aggressiveness and variety bonuses before presenting a breakdown to the player through the HUD. This immediate feedback reinforces good performance and gives players a clear sense of progression between encounters. At the end of each level, cumulative scores from all rooms are tallied and displayed, giving players a sense of accomplishment while also providing insight into how their performance has evolved. A high score is tracked persistently across runs, encouraging replayability and improvement over time.



(a) Results screen for a single room

(b) Results screen for a level

Figure 3.3: Score breakdown upon completing a room/level of the dungeon

Chapter 4

Developing the game

4.1 Tools and technologies

Unity was chosen as the game engine for this project due to its versatility, robust 2D development tools, and strong ecosystem for supporting data-driven design. It offers a well-documented and feature-rich environment suited for both rapid prototyping and complex game systems. Built-in 2D features such as sprite management, tile maps, physics, and animation tools make it an ideal platform for creating a responsive top-down shooter with procedurally generated levels [14]. Another key advantage is Unity Analytics, a cloud platform which allows developers to collect and analyse real-time gameplay data [18]. This is particularly valuable for evaluating the effectiveness of the DDA system. Additionally, my prior experience with Unity allowed for a more efficient development process, reducing the learning curve and enabling me to focus on implementing the more complex systems such as enemy AI and procedural level generation.

4.2 Player system

4.2.1 Input handling

The Player Controller class is responsible for interpreting player input and converting it into in-game actions such as movement, aiming, shooting, and interaction. This is achieved using Unity's built-in Input System, which decouples physical key or button checks from gameplay logic [19]. Instead of hard-coding input, the system allows for the definition of input actions like "Move" and "Attack", each of which can be mapped to multiple bindings. This enables seamless support for different control schemes (such as keyboard and mouse or gamepad) within a single project.

Input actions are organised into action maps, which can be dynamically enabled or disabled depending on the current game state (e.g., during gameplay versus while navigating menus). An Input Actions asset was created to define all available player inputs and their bindings:

When the player character is loaded into the scene, the PlayerController subscribes specific input actions to their corresponding methods. These methods are event-driven and are invoked only during the frame in which the associated input action occurs:

```
1    private void OnEnable()
2    {
3        PlayerInput.actions["Attack"].started += OnAttack;
4        PlayerInput.actions["Attack"].canceled += OnAttack;
5        PlayerInput.actions["Interact"].performed += OnInteract;
6        PlayerInput.actions["Finisher"].started += OnFinisher;
7        PlayerInput.actions["Drop Weapon"].performed += OnForceDropWeapon;
8        PlayerInput.actions["Reload"].performed += OnReload;
9    }
```

Listing 4.1: Subscribing input actions to Player Controller functions

Player movement and rotation are handled in the FixedUpdate() method, which runs at a fixed time step of 0.02 seconds (50 updates per second), ensuring smooth and consistent physics calculations [13]. The movement input is captured as a Vector2, representing the intended direction. This vector is multiplied by the movement speed and applied to the player's current position. At the same time, the player's animation state is updated by toggling the "Moving" boolean parameter in the Animator. For gamepad users, an input dead-zone is enforced to avoid unintentional movement due to joystick drift:

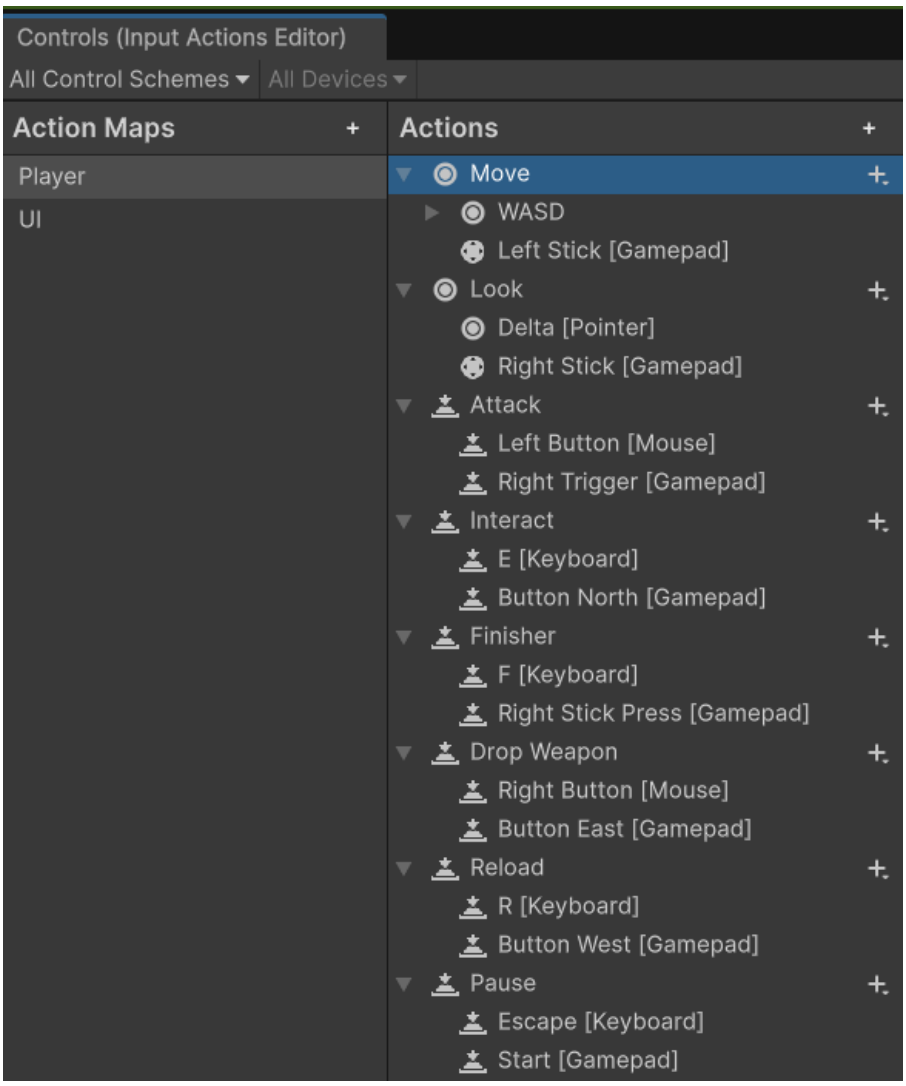


Figure 4.1: Player input actions asset

```
    private void HandleMovement()
    {
        moveInput = PlayerInput.actions["Move"].ReadValue<Vector2>();
        bool isUsingGamepad = PlayerInput.currentControlScheme == "Gamepad";
        if (isUsingGamepad && moveInput.sqrMagnitude > controllerDeadzone *
            controllerDeadzone)
        {
            transform.position += (Vector3)(moveSpeed * Time.deltaTime * moveInput);
            animator.SetBool("Moving", true);
        }
        else
        {
            animator.SetBool("Moving", false);
        }
    }
```

Listing 4.2: Handling movement input

After movement input is processed, the player character is rotated to face the reticle UI element, ensuring that all attacks are directed toward the intended target. Other input actions (e.g., attack, interact, reload) are handled using event callbacks. For instance:

1. OnAttack() triggers either Attack() or HoldAttack() in the Player Attack component depending on the weapon type.

2. OnInteract() searches for nearby interactable objects such as dropped weapons or chests.
3. OnReload() initiates the reload sequence.
4. OnForceDropWeapon() removes the current weapon from the player's inventory and drops it in the world, preserving remaining ammo.

The OnFinisher() function executes a context-sensitive melee execution when conditions are met. It casts a short-range ray (4 units) in the direction the player is facing. If a staggered enemy is detected and no obstacles are present, a finishing move is triggered. During this move, the player and enemy snap to face each other and an animation plays where the player lunges forwards and executes a takedown. This instantly kills the enemy and restores 1 armour plate to the player.

4.2.2 Attacking

The PlayerAttack class encapsulates all other player combat functionality, including weapon firing, reloading, ammo tracking, and armour/health interaction. Weapon properties are stored in a WeaponStats struct, marked as [System.Serializable] to allow it to be modified in editor [16]. Each weapon type has its own stats—such as damage, fire rate, reload speed, clip size, and reserve ammo:

```
1     [System.Serializable]
2     public struct WeaponStats
3     {
4         public int damage;
5         public float fireRate, recoil, reloadSpeed;
6         public int ammoPerClip, reserveAmmo;
7     }
```

Listing 4.3: Handling movement input

When the player activates the attack input, the script selects the appropriate firing method based on the current weapon type:

1. Rifle and SMG fire hit-scan projectiles (raycasts) directly forward, damaging the first enemy they hit. These weapons support both tap-to-fire and hold-to-fire modes. The HoldFireRoutine() coroutine manages repeated fire while the button is held.
2. Shotguns fire a spread of five short-range projectiles in an arc. Each projectile can damage any enemy it collides with but dissipates after a short distance.
3. Melee (no weapon equipped) uses a charged dash attack. Holding the attack input begins charging, while releasing it executes a high-speed dash that damages enemies in the path. Damage scales based on how long the input was held.

The Reload() function animates a progress wheel before refilling the magazine in the current gun (if any reserve ammo of that type is available). The reload speed of the current gun determines how long this takes.

By separating input handling from combat logic and encapsulating weapon-specific behaviour in PlayerAttack, the system remains flexible and scalable, supporting new weapon types or mechanics with minimal changes.

4.3 Procedural level generation

4.3.1 Tilemap system

The dungeon is constructed using Unity's Tilemap system, which enables the efficient placement and rendering of 2D tiles on a grid [17]. The system employs a Tilemap component to draw each room and corridor by assigning tiles from a TileBase asset. This provides a performant and scalable solution for creating visually coherent level geometry.

4.3.2 Room generation

The dungeon comprises a fixed number of main rooms (by default, five), as well as a starting room, an end room, and optional side rooms. Each room is generated procedurally with randomly determined even-numbered dimensions to ensure symmetry and corridor alignment. The starting room is positioned at the origin, and subsequent rooms are placed iteratively in random cardinal directions (north, south, east, west) from the last placed room, with a minimum spacing between room centres to prevent overlap.

4.3.3 Corridor and teleporter connections

Instead of traditional long hallways, the system connects rooms using pairs of teleporters, which provide instantaneous travel between rooms and simplify navigation. These teleporters are dynamically instantiated between adjacent rooms and are responsible for controlling player movement and locking/unlocking rooms during combat encounters. They serve both as functional connectors and gating mechanisms for gameplay pacing, whereby the player is locked inside a combat room until all enemies inside are defeated.

4.3.4 Side room generation

Side rooms are generated probabilistically around each main room. For each cardinal direction adjacent to a main room, a side room may be spawned with probability 0.3, provided it does not overlap with existing rooms. These rooms are smaller and contain a loot chest rather than a group of enemies. Like main rooms, they are connected using teleporters to their associated main room.

Following the completion of dungeon generation, the system rebuilds the NavMeshSurface to allow enemy AI agents to navigate the newly created level. This step ensures that pathfinding is accurate based on the dynamically placed tile geometry and obstacles.

4.4 Enemy AI

4.4.1 Pathfinding

Unity's built-in NavMesh system is commonly used for implementing navigation and pathfinding in 3D environments. It works by baking a navigation mesh (NavMesh) onto the game world's geometry, allowing agents such as enemies to dynamically calculate the most efficient path to a target using the A* algorithm [15]. These agents rely on a NavMeshAgent component, which handles movement, obstacle avoidance, and automatic path recalculations when the target moves or the environment changes.

However, since Unity's native NavMesh system is primarily designed for 3D spaces, additional support is required for 2D games. To overcome this limitation, the NavMeshPlus package was used [5]. This open-source library extends Unity's NavMesh functionality to work in 2D by allowing developers to generate and utilise navigation meshes on 2D tilemaps or sprites. With NavMeshPlus, navigation surfaces can be generated along the X-Y plane instead of Unity's default X-Z plane, enabling effective pathfinding in 2D games like this one.

In this project, NavMeshPlus was used to bake navigation data onto the level tilemaps. Enemy AI agents are equipped with NavMeshAgent components and use this baked data to follow the player around the map. As the player moves through the environment, enemies continuously update their path using the NavMesh to avoid obstacles such as walls or environmental props. This ensures smooth and intelligent pursuit behaviour, which is critical to maintaining the intended challenge level during combat encounters.

4.4.2 State machine

4.5 Score System

The game employs a modular score system to evaluate player performance across individual rooms and entire levels. This system provides feedback to the player, reinforces specific gameplay behaviours (such as weapon variety and aggressive tactics), and contributes to long-term progression through high scores. The system is implemented as a singleton class in Unity to ensure global access and persistence throughout a run [4].

4.5.1 Per-Room Scoring

Each room contributes to the player's score through a combination of five distinct components:

- **Kill Score:** The player receives a base score of 100 points for each enemy killed. If the enemy was defeated using a melee finisher, this value increases to 125 points. The kill count is also tracked for post-run analytics.
- **Combo Score:** The score system tracks consecutive kills within a six-second window. Each time the player kills an enemy, a combo counter increments and resets the timer. When the timer expires, if the combo count is greater than one, the player is awarded a bonus equal to $50 \times$ combo count. This encourages rapid, consecutive engagements.
- **Aggressiveness Score:** The game evaluates player aggression using a heuristic computed from the `PlayerAttack.aggression` metric, scaled to a maximum of 800 points. Aggression increases when the player initiates attacks and defeats enemies at close range, promoting high-risk, high-reward behaviour.
- **Variety Score:** Weapon usage diversity is assessed using a metric derived from the **Herfindahl–Hirschman Index (HHI)**, commonly used in economics to measure concentration [11]. The inverse HHI is calculated as:

  $$\text{Variety Score} = (1 - \sum_i p_i^2) \times 1000$$

  where p_i is the proportion of time spent using weapon i relative to the total attack time. This encourages players to use a broad range of weapons throughout each room.
- **Room Score:** The total score for a room is computed by summing the kill score, combo score, aggressiveness score, and variety score. This value is used for both player feedback and level aggregation.

Each time a room is cleared, these metrics are displayed on the HUD and stored as a `RoomScoreData` object, allowing for later aggregation at the level level.

4.5.2 Per-Level Scoring

At the end of a level, the system aggregates the scores from each cleared room using the stored `RoomScoreData` objects. The following totals are computed:

- **Total Kill Score:** Sum of all kill scores across rooms.
- **Total Combo Score:** Sum of all combo bonuses across rooms.
- **Total Aggressiveness Bonus:** Sum of all aggressiveness scores across rooms.
- **Total Variety Score:** Sum of all variety scores across rooms.

- **Level Score:** Aggregate of the above values, representing the total performance score for the level.

These values are presented to the player through a level results interface and contribute to the `totalScore`, which persists across multiple levels in a run. The score is also compared against a saved high score using Unity's `PlayerPrefs` class and updated if the new score is higher.

4.5.3 Design Considerations

This score system is designed to incentivise dynamic, engaging play styles. Players are rewarded for both mechanical execution (kills and combos) and strategic diversity (weapon usage and aggression). By abstracting scoring into modular categories, the system remains extensible and can be easily adjusted to support balance changes or additional metrics in future iterations.

4.6 Object pooling

In games with frequent instantiation and destruction of objects (such as bullets, enemies, or visual effects) performance can degrade due to the overhead of memory allocation and garbage collection. Object pooling is a design pattern that addresses this issue by maintaining a collection (or "pool") of reusable objects. Instead of destroying an object when it is no longer needed, it is simply deactivated and returned to the pool for future reuse. This approach significantly reduces runtime allocation and improves both performance and responsiveness in gameplay, particularly in fast-paced or resource-intensive scenes.

In this project, a generic and centralised object pooling system was implemented, allowing efficient reuse of Game Objects by tag. The system initialises each pool at startup with a predefined number of inactive instances of a prefab, storing them in queues within a dictionary. When an object is needed (e.g., a bullet or weapon pickup), the pooler attempts to retrieve an inactive object from the appropriate queue. If none are available, it dynamically instantiates a new one and adds it to the pool. Returned objects are simply deactivated and remain available for reuse. This pattern enables consistent performance even in situations where large numbers of objects are spawned and recycled rapidly.

The flowcharts below illustrate the logic used when initialising the pool and when requesting an object from a pool:

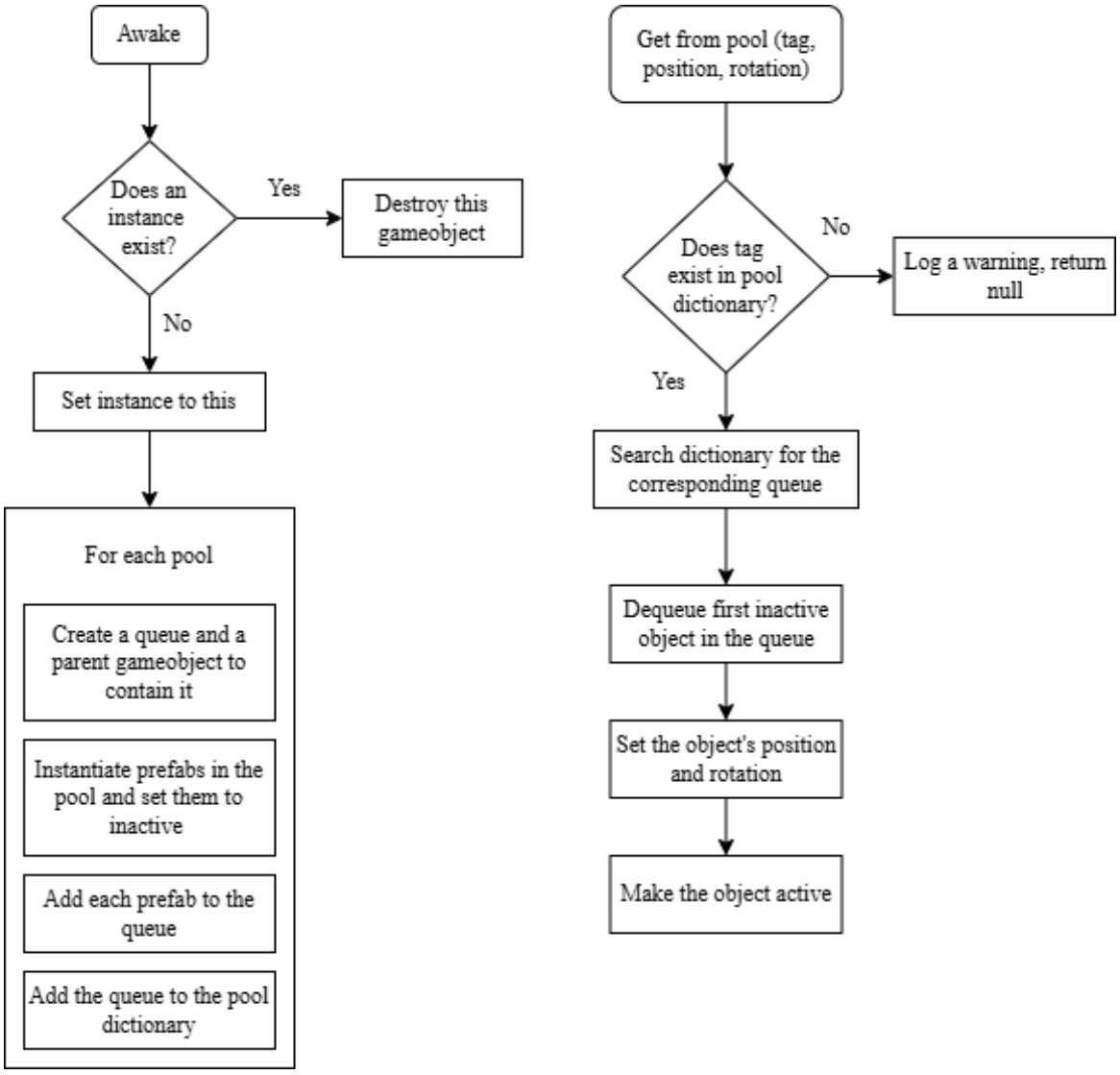


Figure 4.2: Initialising object pools and fetching from a pool

Chapter 5

Implementing the DDA system

This section outlines the implementation of dynamic difficulty in the game, which is achieved through a dual-layered approach: a genetic algorithm that evolves enemy parameters in response to player behaviour, and an adaptive loot system designed to support and reinforce the resulting gameplay dynamics. The genetic algorithm continuously adjusts key attributes of enemies that influence their difficulty, while the adaptive loot system assesses the player's current loadout to provide weapons and items that maintain a reasonable challenge while avoiding frustration. Together, these systems work in tandem to create a gameplay experience that evolves organically to sustain the player's engagement.

5.1 Adaptive loot system

To ensure players remain consistently challenged without being unfairly punished by resource scarcity, the game implements an adaptive loot system. This system dynamically adjusts item drop probabilities by modifying loot table weights in response to the player's current state. The goal is to ensure that loot remains relevant to player needs while retaining enough unpredictability to keep the player on edge.

5.1.1 Base loot table

There are three primary sources of loot in the game:

- An initial pool at the start of the first level in a run
- Enemies encountered in combat rooms
- Chests found in side rooms

All lootable items are defined in a central loot table. Each entry specifies the item's prefab (the in-game object to instantiate), its *baseDropRate* (for probabilistic selection) and a quantity range (*minAmount* to *maxAmount*) representing the amount of health, armour or ammo that item provides.

```
1    public class LootEntry
2    {
3        public GameObject prefab;
4        public float baseDropRate;
5        public int minAmount, maxAmount;
6    }
```

Listing 5.1: An entry in the loot table

The full base loot table is presented in Table 5.1. These baseline values form the starting point for all item drop calculations, prior to any adjustments based on player status.

5.1.2 Adjusting drop rates

The adaptive loot system dynamically adjusts the likelihood of different item drops based on the player's current resource state. This information is gathered through the GetInventory() method, which returns six key values:

- **healthRatio**: Current health as a fraction of maximum health.
- **armourRatio**: Current armour as a fraction of maximum armour.
- **weaponType**: The player's currently equipped weapon.

Item	Drop rate	Min. amount	Max. amount
Rifle	0.5	15	30
SMG	0.5	15	25
Shotgun	0.5	2	5
Rifle Ammo	1.2	15	35
SMG Ammo	1.2	20	40
Shotgun Ammo	1.2	2	4
Armour	0.4	100	100
Health	0.4	100	100

Table 5.1: Base loot table

- **rifleAmmoRatio**, **smgAmmoRatio**, **shotgunAmmoRatio**: Reserve ammo for each weapon type, normalised against an ideal reserve amount specific to that weapon type.

Each ammo ratio is calculated by dividing the player's current reserve by a predefined ideal reserve amount. These values are derived from each weapon's clip size multiplied by some constant, determined through playtesting to represent healthy ammo levels. The ideal reserve ammo for a rifle is 1.5 times the clip size, so 45. If the player has only 15 rifle rounds left in reserve, the resulting $rifleAmmoRatio = 0.33$. For an SMG, the optimal reserve amount is approximately 2.7 times the clip size, reflecting its higher rate of fire and faster consumption of ammunition. For a shotgun, no multiplier is used beyond 1x the clip size, as its slower fire rate and higher damage output necessitate fewer shots.

Before any dynamic adjustments are made, we impose some baseline rules to preserve game balance and player expectations:

- The loot at the start of the first level will always include at least one weapon, to prevent immediate frustration from being forced into melee-only play from the outset.
- Enemies can only drop weapons or ammo of their own type.
- Enemies never drop armour pickups. Armour is already granted by performing finishers on them, so allowing an additional armour plate to be replenished per enemy would detract from the mechanic's importance.
- Health will never drop while the player has full health, and armour will never drop while they have full armour

The drop weights for health, armour and ammo pickups are adjusted using a linear interpolation function that prioritises items the player is low on. The general formula is:

$$W = W_0 \cdot [a + (b - a)(1 - R)] \tag{5.1}$$

Where W is the adjusted drop weight, W_0 is the base drop weight, R is the the player's resource ratio (between 0 and 1) and a and b are constants denoting the minimum and maximum scaling bounds for W. Weights for health and armour pickups scale from 0 to 2, while weights for ammo pickups scale from 0.5 to 2. This is to enforce the rule that health/armour cannot drop if the player already has the maximum amount of it, while there should be no hard limit to the amount of ammo that can drop.

Finally, we adjust the drop rate of weapons based on the player's currently equipped weapon type:

$$W = \begin{cases} 0.5 \cdot W_0 & \text{if } T = T_{\text{player}} \\ 2.5 \cdot W_0 & \text{if } T \neq T_{\text{player}} \end{cases} \tag{5.2}$$

Where W is the weapon's adjusted drop weight, W_0 is its base drop weight, T is the weapon type being considered and T_{player} is the weapon type the player currently has equipped. Effectively, this halves the weight if it's the same weapon type the player is already holding, otherwise it multiplies it by 2.5. This reduces the likelihood of receiving the same weapon type repeatedly, encouraging players to keep switching weapon types throughout the run.

Once all weights are adjusted, a weighted random selection determines the actual drop. This method preserves the excitement and variability of traditional loot systems while ensuring a sense of fairness. Players are unlikely to feel that drops are purely random, yet are also never guaranteed to receive exactly what they want, maintaining tension and encouraging strategic planning. By tailoring drops to current player needs, this system helps prevent resource starvation and pacing issues, aligning with the project's broader goal of creating a dynamic, responsive experience.

5.2 Genetic algorithm

Genetic algorithms (GAs) are a class of optimisation techniques inspired by natural selection and evolution. They evolve a population of candidate solutions, called individuals, over multiple generations using a fitness function to evaluate and select high-performing individuals.These are typically refined through crossover (combining genes from multiple parents) and mutation (random gene changes), gradually improving performance across generations.

GAs are well-suited for implementing DDA in games. They excel in environments with large or poorly defined solution spaces —exactly the kind of space that emerges when trying to find an optimal challenge for a wide range of player skill levels, strategies, and play styles. Also, unlike rule-based systems or supervised machine learning methods, GAs do not have to depend on training data to start converging on better solutions.

This approach is particularly well suited to the modular design of the game. Each level is composed of a sequence of combat rooms, each of which contains a cluster of enemies. As such, it is intuitive and practical to treat each room as an individual, and each level as a generation. A room's chromosome encodes difficulty-related parameters, which adjust the behaviour of all enemies inside it. Once the player completes a level, a new generation of rooms is created based on their performance. The following sections outline how the algorithm's components were designed to adapt gameplay dynamically.

5.2.1 Chromosome design

Each combat room has a chromosome containing genes that modify core aspects of enemy difficulty. Table 5.3 defines these genes.

Gene	Value range	Description
Health	50-∞	How much damage can be taken before dying
Attack range modifier	0.1-∞	A multiplier to the distance from the player within which the enemy enters the attacking state (if alerted)
Accuracy modifier	0.1-∞	A multiplier to the proportion of shots fired at the player that hit them successfully
Damage modifier	0.1-∞	A multiplier to the amount of damage that is dealt per attack

Table 5.2: Difficulty chromosome, applied to enemies in the corresponding room

The purpose of using modifiers for the non-health genes, rather than absolute values, is to preserve the natural variation between different enemy types. For example, an SMG-wielding enemy will always have a faster fire rate and lower damage per bullet than a shotgun. Using absolute values would

homogenise weapon behaviour in unrealistic ways.

Additional genes were considered, but ultimately excluded:

- Fire rate modifier
- Reload speed modifier
- Movement speed
- Investigation time (how long to search for the player before giving up)
- Ammo-per-clip modifier

Investigation time and *movement speed* were omitted to maintain gameplay predictability. These factors are essential to player learning and strategy. If an enemy suddenly chased the player faster or searched longer than expected, the player could feel unfairly punished due to not having sufficient recovery time. Consistency in these areas supports the player's ability to form accurate expectations and adapt their tactics accordingly.

The other weapon-related genes were excluded to prevent unrealistic weapon visuals and behaviours. For example, a shotgun evolving to fire three times as fast as it did at the start of a run would break the player's immersion and undermine the weapon's identity.

Limiting the chromosome to key difficulty modifiers keeps gameplay learnable and ensures that the objective function remains focused and interpretable.

5.2.2 Initialisation

Before any adaptations can occur, the system must generate an initial population of solutions. There are two main strategies for this.

The first approach is to randomly generate values within appropriate ranges for each gene (e.g. 100-200 for health). This offers broad exploration of the search space from the outset, increasing the chances of discovering surprising or unconventional solutions early on. However, it risks producing wildly unbalanced encounters, which could alienate players before the system has a chance to adapt effectively.

The second approach is to manually define a hand-crafted population using values that are already known to be somewhat fair. While this does reduce early diversity, early impressions of difficulty matter and player retention is crucial for gathering meaningful data across multiple levels to ultimately evaluate the system.

Given these considerations, the manual definition approach was chosen. The initial population was designed as follows;

Room no.	Health	Attack range modifier	Accuracy modifier	Damage modifier
1	150	1	1	1
2	125	0.8	1.1	1.2
3	150	1.2	1.3	1.5
4	175	1	0.8	0.7
5	150	1.4	0.9	1

Table 5.3: Initial population for the genetic algorithm

Room 1 acts as a control, using neutral values for all genes. Room 2 simulates 'glass cannon' enemies with higher damage output but lower health. Room 3 introduces a slight difficulty increase

across the board to account for the initial increase in the player's expertise. Room 4 is more of an attritional fight with more tanky but less damaging enemies. Finally, Room 5 focuses on long range combat. Together, the initial population establishes a baseline of varied and fair gameplay for future generations to evolve from.

5.2.3 Objective function

To evaluate the fairness of a given difficulty chromosome, we define an objective function based on how much damage the player takes in that room:

$$\mathrm{E} = \frac{\mathrm{DamageTaken}_{\mathrm{room}}}{\mathrm{MaxHealth} + \mathrm{MaxArmour}} \tag{5.3}$$

An initial consideration was to instead use the player's **current** health and armour upon entering the room. While this reflects the player's actual resilience at that point in time, it introduces variability in the fairness calculation unrelated to the challenge of the room itself. For example, if the player enters a room with low health due to a difficult prior encounter, taking even a small amount of damage could unfairly inflate the enemy effectiveness score. By using the maximum health and armour, we avoid bias from prior encounters. This ensures consistency and allows fairness to reflect the enemy design rather than the player's prior performance or luck.

However, raw effectiveness can be misleading. If the player maintains a consistent playstyle across levels, it can be an intuitive measure of fairness. In practice, player behaviour can be heavily influenced by external game factors unrelated to the parameters being evolved. For example, entering a room with low health often leads to more cautious play, such as staying in cover for longer and engaging enemies at a distance. The degree to which a player acts defensively or aggressively has a effect on how much damage is sustained. A player playing defensively will likely take less damage, potentially leading to the false conclusion that the room is easier. If the objective function is based solely on damage taken, this could erroneously lower the room's difficulty, rewarding cautious play and possibly making the game too easy over time.

To account for this, we factor the player's aggressiveness into the effectiveness of the enemies, defined as:

$$A = \frac{T_a}{T_t} \tag{5.4}$$

Where T_a is the time spent landing aggressive attacks (attacks within 4 units of an enemy) and T_t is the total time spent performing attacks. Only attacks that actually hit enemies are counted, avoiding the misclassification of players who struggle with aiming as being more defensive.

To evaluate the fairness of the enemies rather than their raw effectiveness, we need to measure the difference between their actual effectiveness and some 'expected effectiveness' - a target value that reflects a perfectly balanced encounter. This expected effectiveness serves as a benchmark for how much damage the player should reasonably take, given their current behaviour.

To capture the relationship between playstyle and expected damage, we interpolate between two extremes:

- For fully defensive play ($A = 0$), we expect enemies to deal relatively low damage, with an effectiveness target of 0.3.
- For fully aggressive play ($A = 1$), we expect enemies to deal higher damage, with an effectiveness target of 0.9.

This interpolation ensures that the difficulty target adjusts according to the player's playstyle, preventing the system from mistakenly classifying a room as too easy or too hard based on the player's

behaviour. For a perfectly balanced playstyle ($A = 0.5$), the expected effectiveness falls at 0.6, a midpoint chosen through limited independent playtesting.

This yields the formula:

$$E_{expected} = 0.3 + 0.6A \tag{5.5}$$

Players are generally expected to perform 2–3 finishers per room, restoring approximately 50–75% of their armour. Exploration of side rooms can yield additional health and armour. Setting the expected effectiveness to 0.6 for a balanced playstyle encourages players to engage with these mechanics.

Once the player's aggression score A and expected effectiveness $E_{expected}$ are determined, we can compute a fairness score for the room. This score evaluates how closely the actual enemy effectiveness aligns with the expected effectiveness. It is calculated using an exponential decay function, where a smaller difference between expected and actual effectiveness results in a higher fairness score:

$$F = e^{-k \cdot |E_{\text{actual}} - E_{\text{expected}}|} \tag{5.6}$$

Here, F is the fairness score and k is a tunable scaling factor that controls the rate of decay. In this implementation, we use $k = 1.5$. The following outcomes illustrate the fairness function's behaviour.

- If $E_{actual} = E_{expected}$ then $F = 1$ (perfect fairness)
- If $|E_{actual} - E_{expected} = 0.33$ then $F \approx 0.5$
- If $|E_{actual} - E_{expected} \geq 0.66$ then F drops below 0.1, indicating a poorly balanced encounter.

This function provides a smooth, continuous fairness metric, rewarding well-balanced encounters and penalising outliers without relying on binary thresholds. A visualisation of the fairness function is provided below:

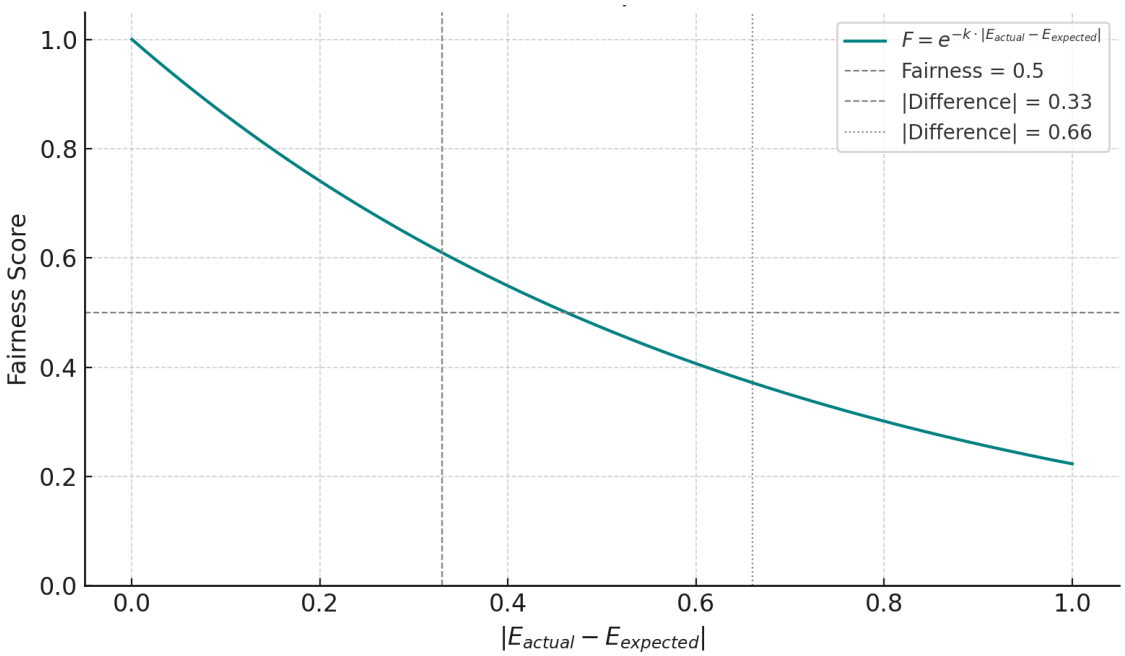


Figure 5.1: Fairness function

5.2.4 Selection

An important part of any genetic algorithm is the selection mechanism, which determines how individuals are chosen to reproduce and pass on traits to the next generation. The goal is to favour high-performing candidates while preserving enough diversity to avoid premature convergence. Three common strategies were considered: roulette selection, tournament selection, and cut selection.

- **Roulette selection** assigns a selection probability to each individual based on their relative fitness. This introduces stochasticity but struggles when fitness values are closely clustered, leading to drift and inconsistent progress. This is usually the case in our system due to the small population size.
- **Tournament selection** selects the best individual from a randomly sampled subset. However, even a minimal tournament size of 2 applies too much selection pressure for the small population size we are using, causing rapid convergence.
- **Cut selection** selects the top n individuals based on fitness. Given the small population and stable fitness metric, a cut size of 1 provided reliable and interpretable results.

Since cut selection of size 1 always selects a single parent, **crossover is not used** in this implementation. This simplifies the evolutionary process and avoids complications arising from recombination, which is unnecessary when a single, high-quality parent is sufficient to guide incremental improvements through mutation.

5.2.5 Mutation

Mutation introduces variation by randomly altering parts of a chromosome, allowing the algorithm to continue exploring new solutions and avoid premature convergence. In our system, each offspring is created by cloning the elite individual from the previous generation and then mutating one randomly selected gene. The selected gene is modified by a random value drawn from a defined range appropriate to that gene. After mutation, the gene's value is clamped to a minimum threshold to ensure it remains meaningful and functional. For instance, enemy health is constrained to a minimum of 50 to avoid trivial encounters where enemies die in a single hit. This maintains gameplay challenge and avoids destabilising the difficulty curve.

This mutation strategy preserves most of the elite's traits while injecting enough randomness to explore nearby configurations in the search space. By mutating exactly one gene per offspring, the algorithm achieves a careful balance between exploitation (refining successful genes) and exploration (searching in new directions for improvements). It also helps maintain a smooth difficulty progression by preventing drastic changes in enemy behaviour between rooms.

The mutation parameters for each gene are detailed below:

Gene	Mutation range	Minimum value
Health	[-50, +50]	50
Attack range modifier	[-0.5, +0.5]	0.1
Accuracy modifier	[-0.5, +0.5]	0.1
Damage modifier	[-0.5, +0.5]	0.1

Table 5.4: Mutation ranges for each gene

Chapter 6

Testing and evaluation

6.1 Testing process

To evaluate the effectiveness of the DDA system, two versions of the game were used for playtesting: one with DDA enabled and the other using a static difficulty model. The static version applies fixed parameters to all enemies, providing a consistent challenge across all rooms:

Enemy parameter	Value
Health	150
Attack range modifier	1
Accuracy modifier	1
Damage modifier	1

Table 6.1: Enemy parameters for static difficulty

Evaluation was based on a combination of performance metrics and subjective player feedback. Each participant completed at least one full run in both game versions (continuing until they either died or chose to exit the dungeon) and subsequently filled out a short feedback form. A total of 13 participants took part in this study.

To prevent bias, the two versions were anonymised as 'Version 1' and 'Version 2', with participants unaware of the presence or absence of DDA. In reality, Version 1 featured dynamic difficulty, while Version 2 maintained static settings. Gameplay data was collected remotely via Unity Analytics to streamline evaluation and minimise the burden on testers.

Unity Analytics is a cloud-based service integrated into the Unity ecosystem that allows developers to track custom events and gather real-time gameplay data from live builds. This enables unobtrusive, large-scale playtesting without requiring manual data submission or in-person observation. Key metrics such as player performance, enemy balance, and level fairness can be monitored remotely through a centralised dashboard.

In this project, Unity Analytics was integrated via a custom `AnalyticsManager` singleton class, which persists across scenes and ensures consistent initialisation. The service is asynchronously initialised during the `Start()` method, after which data collection begins automatically:

```
1 private async void Start()
2 {
3     await UnityServices.InitializeAsync();
4     AnalyticsService.Instance.StartDataCollection();
5     isInitialised = true;
6 }
```

Listing 6.1: Initialising Unity Analytics

The manager exposes a public method, `LevelCompleted()`, which is invoked at the end of each level (i.e., every five rooms cleared). This method gathers key statistics related to difficulty balancing and constructs a `CustomEvent` with relevant data fields.

The `level_completed` event records a range of values representing both gameplay performance and DDA system output. These include:

- **dynamic_difficulty:** Boolean flag indicating whether the DDA system is enabled in this version.

- **level_number:** The current level index, calculated as rooms cleared divided by five.
- **average_fairness:** The average fairness score of all enemy spawns in the level.
- **highest_fairness:** The single highest fairness score achieved in the level.
- **fairest_enemy_health:** Health of enemies in the fairest room in the level.
- **fairest_attack_range_modifier:** Attack range multiplier in the fairest room in the level.
- **fairest_accuracy_modifier:** Accuracy multiplier in the fairest room in the level.
- **fairest_damage_modifier:** Damage modifier in the fairest room in the level.
- **percent_health_armour_remaining:** Percentage of the player's combined health and armour remaining at the end of the level.

These metrics are packaged as a key-value dictionary and sent to Unity's servers using the `RecordEvent()` method:

```
AnalyticsService.Instance.RecordEvent(customEvent);
```

Listing 6.2: Recording a custom analytics event

Unity Analytics is GDPR-compliant by default, transmitting only anonymised gameplay data and avoiding the collection of any personally identifiable information. Additionally, participants were made aware that by playing the game during the testing period, they were consenting to having their performance data recorded. This ensures that all remote data gathering remained ethical and privacy-conscious. To illustrate, the following JSON snippet shows a typical payload for a `level_completed` event. It includes both custom gameplay metrics and default Unity-provided metadata:


```
{
  "average_fairness": 0.576349079608917,
  "clientVersion": "1.0",
  "collectInsertedTimestamp": "2025-04-16 11:38:53.156",
  "dynamic_difficulty": 1,
  "eventDate": "2025-04-16 00:00:00.000",
  "eventID": 3378899146600743835,
  "eventLevel": 0,
  "eventName": "level_completed",
  "eventTimestamp": "2025-04-16 11:38:11.033",
  "eventUUID": "ffa93aa0-ecbf-4bba-97a9-28eaed880404",
  "fairest_accuracy_modifier": 1,
  "fairest_attack_range_modifier": 1,
  "fairest_damage_modifier": 1,
  "fairest_enemy_health": 163,
  "gaUserAcquisitionChannel": "None",
  "gaUserAgeGroup": "UNKNOWN",
  "gaUserCountry": "GB",
  "gaUserGender": "UNKNOWN",
  "gaUserStartDate": "2025-04-16 00:00:00.000",
  "highest_fairness": 0.917725205421448,
  "level_number": 2,
  "mainEventID": 3378899146600743835,
  "msSinceLastEvent": 18054,
  "percent_health_armour_remaining": 0.980000019073486,
  "platform": "PC_CLIENT",
  "sessionID": "4a826221-a2a2-4e9d-bab2-2d1da55cae10",
  "timezoneOffset": "+0100",
  "userCountry": "GB",
  "userID": "1f9f6ca4d64e2404eb351babdcaf9471"
}
```


Listing 6.3: Sample `level_completed` event data

6.2 Evaluation metrics

To evaluate the performance and effectiveness of the dynamic difficulty adjustment (DDA) system, two core metrics were defined and tracked: **highest fairness found per level** and **average fairness found per level**. These metrics were designed to assess how well the genetic algorithm is able to discover and prioritise balanced combat scenarios in each level.

The *highest fairness* value recorded in each level represents the best-performing enemy configuration, based on the fairness function. This metric is important because it shows the system's potential to generate optimal encounters—even if other rooms in the level vary in quality. If the DDA system is working effectively, we expect to see the highest fairness increase after each level, demonstrating that the genetic algorithm is regularly able to evolve enemy configurations that align well with the player's playstyle and ability.

While the highest fairness shows the peak performance of the algorithm, the *average fairness* across all rooms in a level provides insight into its consistency. This metric reflects how balanced the overall gameplay experience is on average, not just in isolated cases. A high average fairness score suggests that the majority of rooms provided an appropriate level of challenge and engagement. This metric is particularly important for player experience, as players do not encounter only the fairest room in a level - they progress through all of them. Therefore, ensuring a consistently fair experience across a full level is a key indicator of a successful DDA system.

Together, these two metrics provide a comprehensive picture of how effective the DDA system is at tailoring gameplay. Ideally, a successful system will show an upward trend in both values over successive levels, indicating not only occasional success but consistent adaptability as gameplay progresses.

6.3 Results and review

Performance data was queried through the SQL Data Explorer within the Unity Analytics dashboard. For example, the following query was used to extract the highest fairness score per level across all runs where DDA was enabled:

```
with fairness_per_level AS (
SELECT
EVENT_JSON:level_number::INT AS level_number,
EVENT_JSON:highest_fairness::FLOAT AS highest_fairness
FROM EVENTS WHERE EVENT_NAME='level_completed' AND EVENT_JSON:dynamic_difficulty::INT=1
)
SELECT level_number as "Level number", avg(highest_fairness) as "Highest fairness found"
FROM fairness_per_level
GROUP BY level_number
ORDER BY level_number
```

6.3.1 Highest fairness per level

In the static difficulty version, fairness peaks in the early stages but declines steadily across subsequent levels, eventually plateauing at a consistently lower level. This trend suggests that as the player progresses and their mastery of the game improves, the fixed difficulty system fails to keep pace, resulting in increasingly unbalanced encounters.

By contrast, the DDA-enabled version exhibits a generally upward trajectory in fairness, albeit with minor fluctuations. This pattern reflects the system's ability to adapt to player behaviour and iteratively refine enemy configurations over time. From the second level onward, the DDA consistently

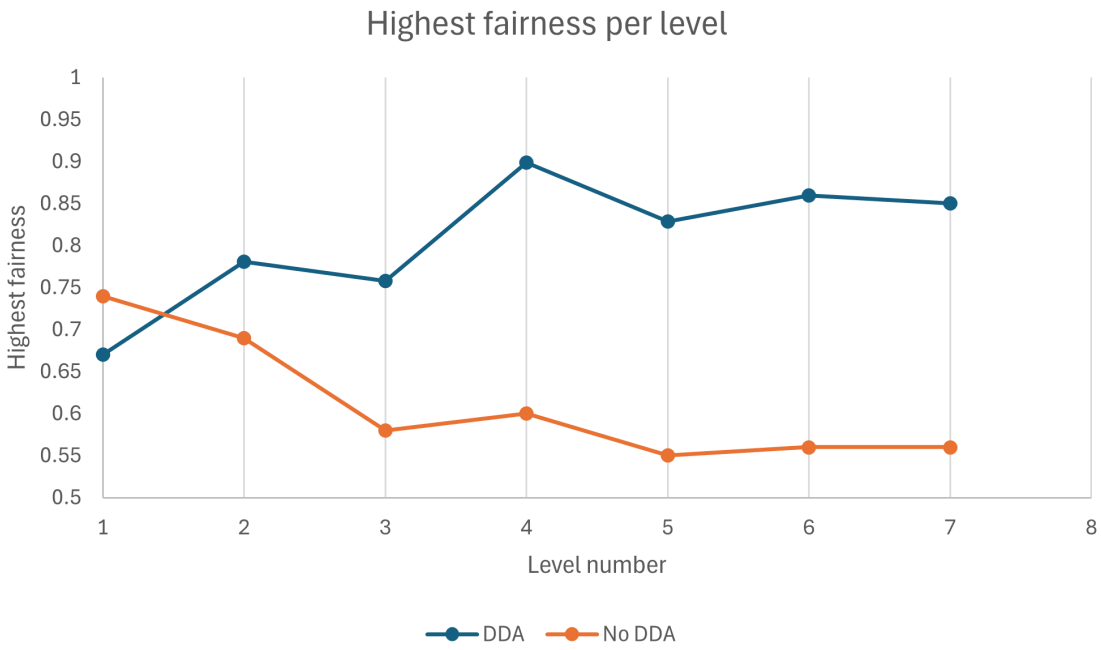


Figure 6.1: Highest fairness found per level

outperforms the static approach, yielding higher fairness across all remaining levels.

The progressive improvement in fairness across levels indicates that the genetic algorithm is successfully guiding the system towards configurations that offer a better match to the player's evolving capabilities.

6.3.2 Average fairness per level

Average fairness, as a more representative measure of overall level quality, further illustrates the differences between the two systems. In the static version, fairness begins at a modest high point but steadily decreases, indicating a growing prevalence of unbalanced enemy configurations. This downward trend suggests that without dynamic tuning, the system struggles to maintain gameplay fairness as the player advances.

The DDA-enabled version, however, maintains a relatively stable level of average fairness throughout. Across most levels beyond the second, it consistently achieves higher fairness than the static counterpart. This stability suggests that the dynamic system is able to consistently generate fair encounters, even if not all configurations reach the peak fairness values observed in the best-performing individuals.

The divergence between the two systems becomes more pronounced in later levels, where the DDA continues to sustain fairness while the static version falters. When considered alongside the highest fairness findings, these results reinforce the conclusion that the DDA system enhances both the peak and the average quality of gameplay. It not only identifies more optimal configurations but also ensures a more consistently fair experience across an entire run, contributing to a more engaging and enjoyable game overall.

6.3.3 Evolution of modifier genes

The evolution of the 'modifier' genes offers some insight into how fairness was achieved. Across levels, enemy damage was gradually reduced while accuracy and especially attack range increased. This suggests the system compensated for lower raw threat with more precise and distant attacks,

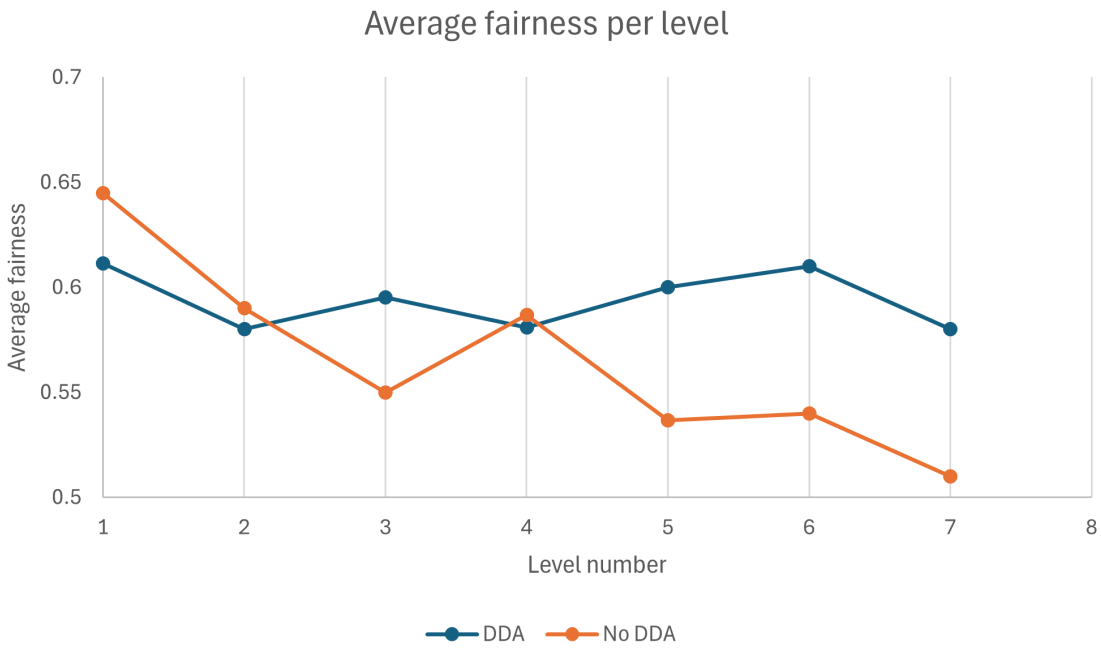


Figure 6.2: Average fairness found per level

promoting fairness not by reducing challenge outright, but by reshaping it into encounters that better matched player skill and response time.

6.3.4 Feedback form

The feedback form results closely reflect the patterns observed in the performance data, providing qualitative support for the effectiveness of the DDA system. Full details are available in the appendix. The first two questions asked players to rate the overall difficulty of each version on a scale from 0 (far too easy) to 10 (far too difficult). Responses for the DDA-enabled version clustered around the midpoint, with most ratings between 5 and 6, indicating a general perception of balanced difficulty. In contrast, the static version received a broader spread of scores, with most ratings falling between 7 and 8, suggesting that players found it noticeably more difficult. This disparity reinforces the notion that the dynamic system was more successful in maintaining an appropriate challenge level, adapting more effectively to individual player performance.

Notably, the majority of players reported that they did not perceive a significant change in difficulty from the beginning to the end of a run. This response was consistent across both versions, but is particularly interesting in the context of the DDA-enabled system. It suggests that the system was able to adjust enemy parameters in a subtle and seamless manner, maintaining a steady level of perceived challenge without introducing jarring difficulty spikes. Rather than feeling artificially manipulated, the difficulty appeared natural and consistent—an ideal outcome for dynamic adjustment systems aiming to preserve player immersion and engagement.

When asked which version they preferred, 38.5% of participants favoured the DDA-enabled version, while only 15.4% preferred the static version. Interestingly, 46.2% of participants reported that they did not notice any difference between the two. This distribution reinforces a key strength of the DDA system: its ability to adapt gameplay in a subtle and unobtrusive way. The fact that nearly half of the players were unaware of any change, yet a significantly higher proportion still preferred the DDA version, suggests that the improvements in fairness and balance introduced by the system were felt intuitively, even if not consciously recognised. Conversely, the relatively low preference for the static version further supports the claim that fixed difficulty settings struggle to remain engaging

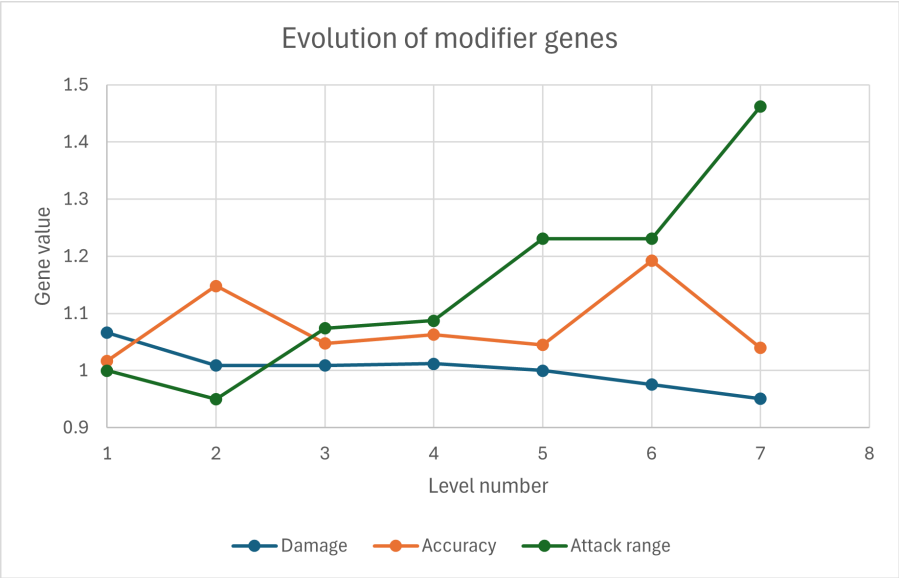


Figure 6.3: Evolution of modifier genes

and appropriate as players progress through procedurally generated levels.

Additionally, the majority of players reported that the adaptive loot system contributed positively to the game's balance and influenced their playstyle and strategies. This suggests that it was an effective complement to the main DDA system.

Chapter 7

Conclusion

This project set out to explore and implement a dynamic difficulty adjustment system for a custom-built 2D top-down shooter, with the goal of improving gameplay balance and player experience compared to using a static difficulty. The system combined a genetic algorithm, used to evolve enemy parameters based on player performance, with an adaptive loot system that responded to player inventory needs to maintain challenge without causing undue frustration. Together, these components formed a layered approach to balancing encounters on a per-level basis, while remaining flexible enough to accommodate aggressive and defensive playstyles.

Playtesting and performance analysis demonstrated that the DDA system was largely successful. Compared to a static-difficulty baseline, the adaptive version produced both higher peak fairness and a more stable average fairness across levels. Player feedback further supported these findings, with a majority expressing a preference for the DDA version or perceiving no difference, and many noting that the adaptive loot influenced their decisions in meaningful ways. However, there are several areas which could be improved, given more time. For example, further refinement of the fairness function could make it less affected by the inherent variability of player behaviour, increasing AI complexity could allow for more sophisticated enemy behaviours to evolve (e.g. flanking manoeuvres), and player progression data could be stored across runs to allow for longer-term adaptation. Overall, this project provides a promising DDA implementation within a top-down shooter, with strong potential to enhance both fairness and player engagement.

Bibliography

[1] Gustavo Andrade, Geber Ramalho, Hugo Santana, and Vincent Corruble. Extending reinforcement learning to provide dynamic game balancing. *HAL (Le Centre Pour La Communication Scientifique Directe)*, Jul 2005.

[2] Joy James Prabhu Arulraj. Adaptive agent generation using machine learning for dynamic difficulty adjustment. *ICCCT'10*, 2010.

[3] Michael Csikszentmihalyi. Flow: The psychology of optimal experience. *The Academy of Management Review*, 16(3):100–107, Jul 1991.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] h8man. Navmeshplus: Unity navmesh 2d pathfinding, 2025.

[6] Alisson S Henrique, Ricardo Martins Brasil Soares, R. Dazzi, and R. Lyra. Genetic algorithm in survival shooter games npcs. *Anais do XI Computer on the Beach - COTB '20*, 2020.

[7] Robin Hunicke. The case for dynamic difficulty adjustment in games. pages 429–433, 2005.

[8] Robin Hunicke and V. Chapman. Ai for dynamic difficulty adjustment in games. 2004.

[9] K. Kutt, Lukasz Sciga, and Grzegorz J. Nalepa. Emotion-based dynamic difficulty adjustment in video games. *2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–5, 2023.

[10] Elinga Pagalyte, M. Mancini, and Laura Climent. Go with the flow: Reinforcement learning in turn-based battle video games. *Proceedings of the 20th ACM International Conference on Intelligent Virtual Agents*, 2020.

[11] S. A. Rhoades. The herfindahl–hirschman index. *Federal Reserve Bulletin*, 79:188, 1993.

[12] M. Shakhova and A. Zagarskikh. Dynamic difficulty adjustment with a simplification ability using neuroevolution. *Procedia Computer Science*, 2019.

[13] Unity Technologies. Unity - scripting api: Monobehaviour.fixedupdate(), 2024.

[14] Unity Technologies. 2d game kit, 2025.

[15] Unity Technologies. Building a navmesh in unity, 2025.

[16] Unity Technologies. Serialization in unity, 2025.

[17] Unity Technologies. Tilemap in unity, 2025.

[18] Unity Technologies. Unity analytics documentation, 2025.

[19] Unity Technologies. Unity input system documentation, 2025.

[20] Su Xue, Meng Wu, John Kolen, Navid Aghdaie, and Kazi A. Zaman. Dynamic difficulty adjustment for maximized engagement in digital games. *Proceedings of the 26th International Conference on World Wide Web Companion - WWW '17 Companion*, 2017.

[21] Mohammad Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018:1–12, Nov 2018.

Chapter A

Full feedback form results

Feedback on final year project

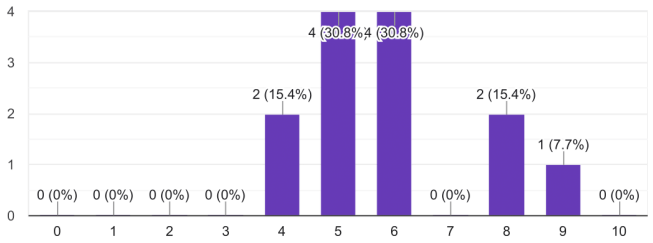
13 responses

Publish analytics

On a scale of 0 to 10, how would you rate the overall difficulty of the game in version 1? (5 = perfectly balanced)

Copy

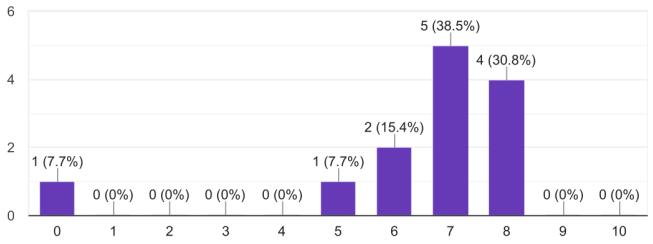
13 responses



On a scale of 0 to 10, how would you rate the overall difficulty of the game in version 2? (5 = perfectly balanced)

Copy

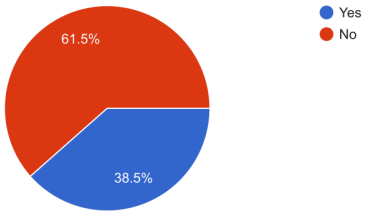
13 responses



Did you feel the difficulty change from the beginning to the end of a run on version 1?

Copy

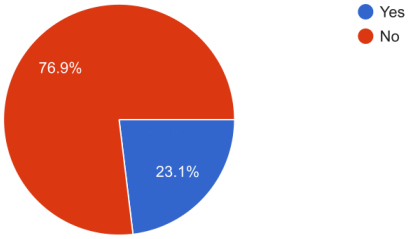
13 responses



Did you feel the difficulty change from the beginning to the end of a run on version 2?

Copy

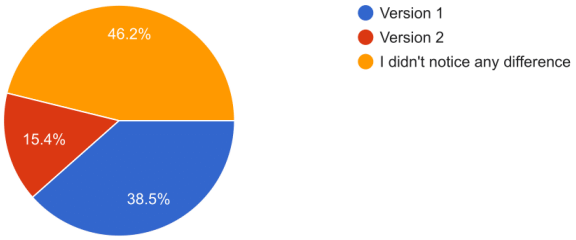
13 responses



Which version of the game did you find more enjoyable as a whole?

Copy

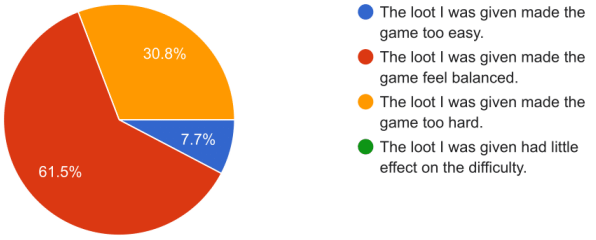
13 responses



How did the loot system affect the game's overall difficulty?

Copy

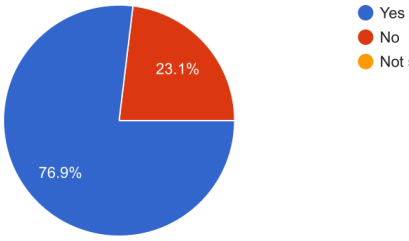
13 responses



Did the loot that you received throughout affect your gameplay strategy? (e.g. played more or less cautiously)

Copy

13 responses



Chapter B

Assets used

- **Dungeon tile set**: https://pixel-poem.itch.io/dungeon-assetpuck
- **Reticle sprites**: https://void1gaming.itch.io/free-mega-crosshairs-pack
- **Chest sprites**: https://ankousse26.itch.io/free-treasure-boxes-pixel-art
- **Portal sprites**: https://frostwindz.itch.io/pixel-art-fantasy-animated-portal
- **UI icons**: https://free-game-assets.itch.io/free-gui-for-cyberpunk-pixel-art
- **Main menu background**: https://craftpix.net/freebies/free-city-backgrounds-pixel-art
- **SFX**: https://freesound.org
- **Music**: https://incompetech.com/wordpress

Chapter C

Other links

GitLab repository: https://git.cs.bham.ac.uk/projects-2024-25/rxs1099
Itch.io page for the game (includes gameplay video): https://rory-simpson.itch.io/mutagenesis