Procedural Content Generation based on Dynamic Difficulty Adjustment outputs to increase player focus.

Kaveh Nejad - 1905933
RGU School of Computing
Honours Project
2023/2024
Supervisor: John Issacs
Code: https://gitlab.com/Kaveh_N_Nejad/uni-honours-cm4105

ROBERT GORDON UNIVERSITY ABERDEEN

School of Computing

Honours Project Report

Procedural Content Generation based on Dynamic Difficulty Adjustment outputs to increase player focus.

Kaveh Nejad

This report is submitted as part of the requirements for the degree of
BSc (Hons) in Computer Science
at Robert Gordon University, Aberdeen, Scotland

I confirm that the work contained in this Honours project report has been composed solely by myself and has not been accepted in any previous application for a degree. All sources of information have been specifically acknowledged and all verbatim extracts are distinguished by quotation marks.

Student Signature: _____Kaveh Nejad_____  **____**

Table of Contents

Literature Review

Project Scope Introduction

Video games play an important part in today's society. The industry is valued at multiple billions of dollars and has been an important driver of new technological breakthroughs (Liang, 2022). Apart from their monetary and technological benefits, video games are often used in education and other areas of our society (Zeng et al., 2020). Just as valuable, if not more, is the ability of video games to help improve the mental health of the players (Kowal et al., 2021). Games have also been proven to have a positive impact on users' well-being, amongst other benefits (Schaffer and Fang, 2019). As an important part of games is the entertainment factor, game designers need to focus on creating an enjoyable experience for players and an important element of this is maintaining a fun yet achievable challenge (Hergenrather, 2020). In other words, the game designers must ensure the game is not too difficult or too easy. This can be achieved using several methods. The main method that most games will use to change the difficulty is to allow the player to select their own difficulty level at the beginning of each playthrough (Smeddinck et al., 2016). A more customisable method is to allow the user to select the difficulty level (or make other changes) of individual components of the game such as in the game Pathfinder: Kingmaker (fandom.com, 2021) (see Appendix A for examples). Another solution to the difficulty adjustment is implementing Dynamic Difficulty Adjustment (DDA) that can update the difficulty for each individual player (Zohaib, 2018). DDA allows the players to jump right into the game without selecting a difficulty and ensures the game is always adjusting itself to maintain an achievable challenge. An additional requirement for games is to maintain player focus. This can be achieved using several methods. One reason players lose focus is because they find the game too easy or too hard, therefore, games can adjust the difficulty (as seen above) as needed (Chen and Sun, 2016) to increase focus. Another method of increasing focus is by switching activities (Westgate, 2019) such as adding puzzle segments to a fighting game. Just as important is providing the player with enough content of high enough variety. This is often done programmatically by an algorithm that can procedurally generate content, such as terrain or platformer levels. This method is called Procedural Content Generation (PCG).

Importance of Flow State

An important part of every game, and something the developers need to prioritise is for the player to enter a flow state. Flow state refers to the player's focus being solely directed at the game, which often leads the player to obtain a rewarding experience (Soutter and Hitchens, 2016). When a person is in flow state, regardless of the activity they are doing, they often find difficult tasks far easier and perform at their best (Harris et al., 2017). Therefore, players in a flow state perform better, and this needs to be considered by any DDA system. But most importantly to the game designer, people in a flow state achieve effortless enjoyment (Harris et al., 2021; Harris et al., 2017; Soutter and Hitchens, 2016).

According to a study by Li-Xian Chen on 266 junior high school students (Chen and Sun, 2016), a player's ability to enter a flow state is linked to their boredom levels. Chen found that when players reported that they felt bored, they would increase the difficulty and were then more likely to enter a flow state. Similarly, when players were struggling, they would lower the difficulty, and were also more likely to enter a flow state.

Shoot 'em ups

While these technologies and techniques are applicable and useful in many game genres, this project will be demonstrated in a "shoot 'em up" style game due to its simplicity and, therefore, ease of prototyping. The game used will also be a newly created in-house game, specifically designed for this proof of concept. Shoot 'em ups are an old style of games that date back to 1962 and involve a main character (often a spaceship) moving upwards while enemies fly towards the player shooting. The player must dodge the enemy fire, and shoot back to clear the level or prevent themselves from being overwhelmed. There can also be boss battles at the end of each level and the levels become increasingly more difficult (Masem, 2023).

Image 1: Example of "Shoot 'em ups" (libertygames, 2013)

Image 2: 2nd example of "Shoot 'em ups" (libertygames, 2013)

Dynamic Difficulty Adjustment

A major part of this project is locating the player's optimal difficulty in order to then be able to detect if the player varies from this difficulty. The simplest approach is to provide the player with several options to choose from (easy, medium, hard, etc.) (Smeddinck et al., 2016), however, this may not lead to the best results for several reasons (Zohaib, 2018). For example, according to a relatively small survey, players may not always choose the difficulty level that best corresponds to their self-assigned capabilities (Hergenrather, 2020). In order to detect the optimal difficulty, Dynamic Difficulty Adjustment (DDA) will be used, thereby, scaling the difficulty of the game based on the player's ability. DDA dates back to Midway's 1975 Gun Fight coin-op game, where if a player takes damage it would provide them with an object to take cover behind (Anaxial, 2023).

Images 3,4: Midway's 1975 Gun Fight (arcadeologia, 2015)

Since its use in Gun Fight, DDA has progressed to become more complicated and, most importantly, more effective. This can be seen in the following DDA methods:

1. Rubber banding: This technique is commonly used in early racing games. However, the concept can be applied in other game types. In rubber banding racing games, if a player is falling behind their opponents the AI opponents will receive a speed decrease. Similarly, if a player is ahead, the opponents will receive a speed boost. With the racers generally close to each other, it ensures the player is always focusing on the game. When in last place, the player is still able to overtake the opponents and win. On the other hand, when in first place the opponents are never too far behind (Mi and Gao, 2022; Missura, 2015).

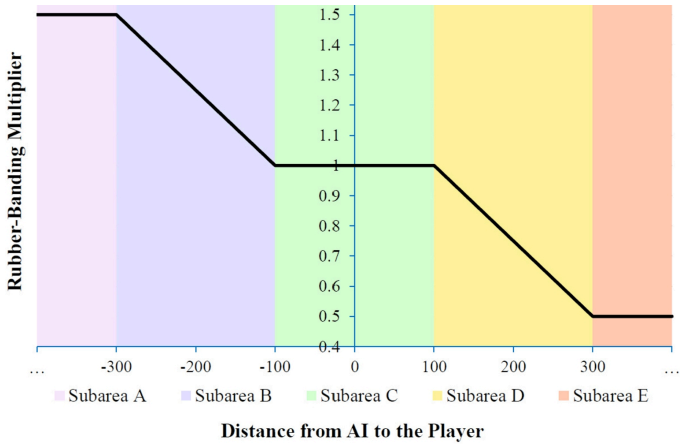


Image 5: Rubber banding AI power/speed multiplier (Mi and Gao).

2. Similar to the rubber banding method of rewarding the players in last place is managing the types of bonuses players receive. This is seen in the multiplayer Mario Kart game where players can pick up mystery power-ups, and unbeknown to them the game will give weaker rewards to those in the lead and more powerful rewards to those falling behind (Vang, 2022; Missura, 2015).
3. A different type of DDA is scaling the enemy level based on the player's level. This can be seen in games like Bethesda's Oblivion, where, as the player becomes more powerful, so do the enemies (Missura, 2015). This allows the player to level up their character while completing side quests without feeling overly powerful when they return to the main storyline.
4. These can also become more complicated using Machine Learning techniques, such as in dynamic scripting. In this method, the enemy AI will be composed of many different rules, which are chosen from a predefined list during gameplay by a DDA system. There are many methods of dynamic scripting as well, particularly due to the fact that it is a continually learning AI. As a continually learning AI, it is possible for it to always outperform the player given enough time and data. Therefore, different types of limitations can be imposed by the designer (Sepulveda et al., 2019).

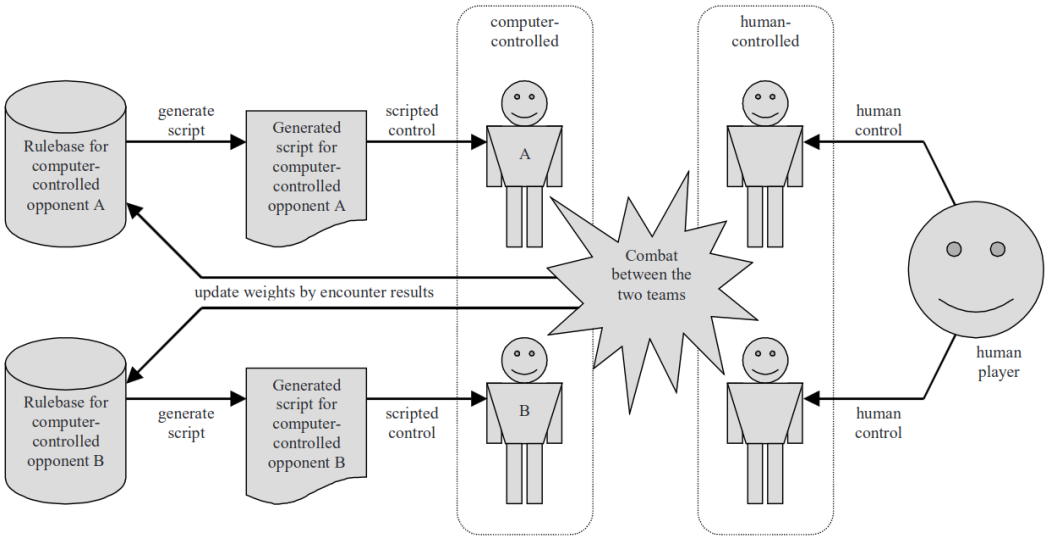


Image 6: Dynamic scripting process during a two player combat game (Sepulveda et al., 2019)

5. Some newer methods that are more invasive can detect player biosignals and modify the game based on these. Detections of biosignals to estimate the player's emotions has been done before, for example in "Automatic Recognition of Boredom in Video Games Using Novel Biosignal Moment-Based Features" (Giakoumis et al., 2011), but a new study that also developed the game Caroline has improved upon these methods. By setting up a method of feeding the player's biosignal outputs into a DDA system, it can increase the difficulty as the player becomes more stressed during the horror game (Moschovitis and Denisova, 2023). The opposite is also possible, and likely preferable in other genres, of reducing the difficulty if the player becomes too stressed. It is likely that this type of biosignal-based games will be seen more often as smartwatches (and other devices that can detect heartbeats and interact with WI-FI and Bluetooth) become more common (Ruby, 2023).

Procedural Content Generation In Games

Procedural Content Generation (PCG) has also been around for several decades in many forms of constantly increasing complexity and effectiveness. One of the first games to include procedurally generated terrain was Rogue in 1980 (Yannakakis and Togelius, 2011).

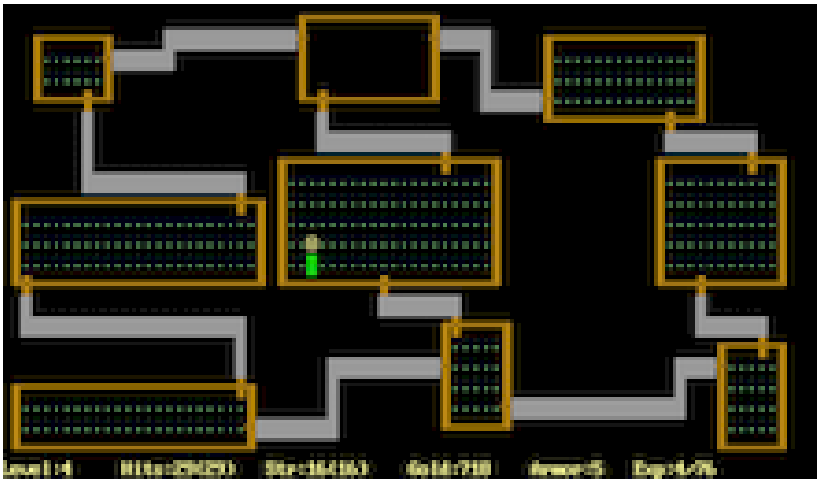


Image 7: Rogue map (Danbloch , 2023)

The definition of PCG can also vary. According to Yannakakis and Togelius in “Experience-Driven Procedural Content Generation”, PCG is “creation of content automatically through algorithmic means” (Yannakakis and Togelius, 2011). However, other studies suggest that PCG should contain “limited or indirect user input” (Shaker et al., 2016) to be considered as such, where “user” refers to both the designers and player. An alternative viewpoint is that the content generation does not have to be generated by a computer, for example Smith argues that several board games have PCG (Smith, 2015). However, in the examples given in Smith’s paper, the player usually assembles the content themselves, often using random dice rolls. Given that the player must interact to the extent of placing tiles or rolling dice, this likely contradicts the aforementioned user interaction requirement, Therefore, it will be ignored in this paper.

The current definition according to Shakers’ paper which is “the algorithmic creation of game content with limited or indirect user input” (Shaker et al., 2016) could also be more precise. The term “indirect” was most likely added to allow for PCG systems based on player’s actions, which is talked about later in both this paper and Shaker’s paper. However, the definition would be better adjusted for the purposes of this project as “the algorithmic creation of game content with limited user input, or when the user does not realise their input is being used for this purpose”. With the new definition, games can use more user input, for instance, in the project proposed in this paper where every user action will be directly fed into the PCG system.
Another example of where user input can be more direct to the PCG system is through conversations the player has with Non-Playable Characters (NPC). These will be dependent on NPC conversations becoming more advanced, as mentioned in a 2018 study by Fraser (Fraser et al., 2018) which will aid in keeping players unaware of their direct input. For instance, it would be possible for an NPC to ask a player if they enjoyed a certain situation and the answer could be given to the procedural content generator.

Another detail that needs to be included in the definition is that “creation” can refer to seemingly brand new content. For instance, in the game No Man’s Sky very basic templates are combined to create “unique” creatures and plants on 18 quintillion “unique” planets and moons (Tait and Nelson, 2021). Unfortunately, No Man’s Sky was unable to fully deliver completely unique life forms as player’s have found numerous life forms on different planets that seem to be nearly identical (Tait and Nelson, 2021; Gravina et al., 2019). Another easier form of PCG is the combination of larger developer created modules as seen in the Minecraft game, where the developers use a bottom-up approach to create a map of the weather (temperature, humidity). They then add the biomes based on the weather, then add the landscapes and finally add pre-created elements (trees, animals, structures, etc.) that fit those landscapes in those biomes (Zucconi, 2023). See appendix B for a detailed example.

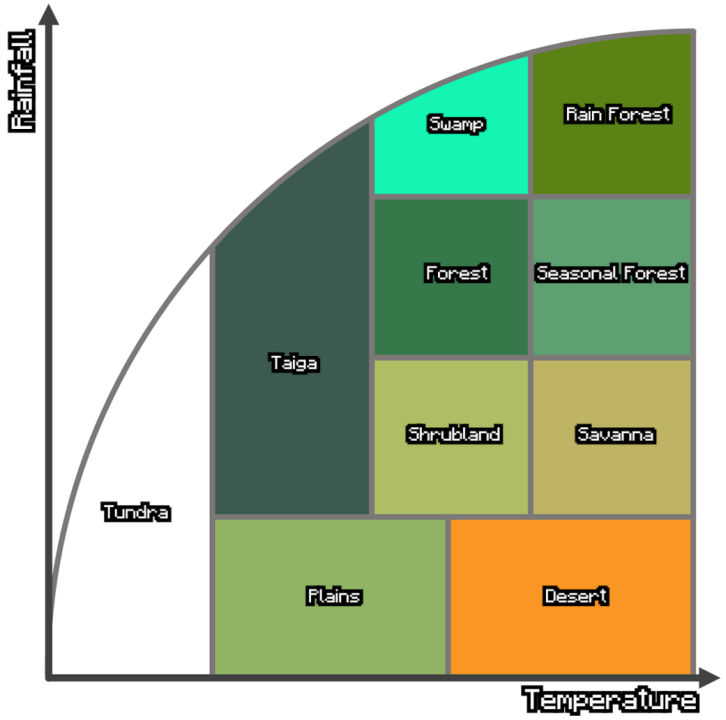


Image 8: Minecraft terrain creation (Zucconi, 2023).

As with DDA, there are many methods of PCG. These include, but are not limited to:

1. Grammatical Evolution (GE) where Backus-Naur grammar is used to describe the base syntax (components) of each possible level. A base level is designed, and then through an evolutionary algorithm it adds new content, based on the grammar. The system then evaluates the level through a fitness operator and then adds another section based on output of the fitness operator and the grammar (Shaker et al., 2016).

```
(1) <exp> ::= <exp> <op> <exp>
            | ( <exp> <op> <exp> )
            |  <var>
(2) <op> :: = + | - | * | /
(3) <var> ::= X
```

Image 9: Backus-Naur form example (Shaker et al., 2016)

2. Supervised learning can also be used. For example, player movement can be analysed in Super Mario games and a learning model can create specific levels for each user (Liu et al., 2020). Another more broad example, also in Mario games, is for human generated levels to be passed through a model to extract design patterns, and then create new levels based on these (Liu et al., 2020).
3. Another example of PCG is for speeding up the process of creating large open-world terrains. For instance, in Skyrim, Horizon Zero Dawn, and Assassin's Creed amongst others, with large worlds where PCG was used to intelligently place the in-game elements. This led to the creation of realistic terrains and landscapes with less developer time needed (Nerevar, 2020).

Image 10: Skyrim terrain (Petitte, 2013)

In this proof of concept project, a simplified Experience Driven PCG system will be used, and it will only be able to select the order of the stages the game will use. Once implemented, the PCG system may also be adapted to combine various game elements (enemies, obstacles, etc.) to create its own stage.

Experience Driven Procedural Content Generation

Experience-Driven PCG (EDPCG) is used when the content generated is based on the experiences the player has within the game, and their interactions. EDPCG relies on the system learning how the player interacts with the game, and which parts they struggle with, or engage with the most, and then generating content mid-gameplay. This is also closely tied to DDA, as in many games if a player is struggling, the best way to reduce the difficulty is to modify the map (Shaker et al., 2016). For example, games can reduce the distance the player needs to jump in a platformer or add additional cover points in a shooter when the player is struggling.

There are also several ways to generate content based on the player's experience, many of which are based on Player Experience Modeling (PEM). PEM data can be obtained through various methods, such as:

- player interactions with in-game elements,
- real-life player reactions (such as the biosignals method seen above, but also through body language),
- and the content the player views.

An example of using PEM is to include the PEM model in the genetic algorithms PCG fitness operator.

Combining EDPCG, DDA, and Shoot 'em ups

There is already a large amount of research put into PCG, EDPCG, and DDA. There is also new research into DDA based on the emotions of the players, for instance, in Caroline mentioned earlier. However, there appears to be a gap in research for EDPCG based on players' emotions, especially boredom. Therefore, a research question of "Can EDPCG based on DDA outputs be used to increase player focus in shoot 'em up style games?" has been chosen.

This project will combine DDA, EDPCG, and "shoot 'em ups" in an attempt to create a game that can change the content of a level when a player is bored to cause them to focus again.

The "shoot 'em up" will have three main components:

1. The classic battles where enemies fly down, and the player dodges their bullets while returning fire.
2. A race through an asteroid maze.
3. And item collection to score points.

When the game begins, a simple method of DDA will collect information about the player's ability based on several factors including:

- time to kill an enemy,
- chances of being hit by an enemy,
- ability to dodge asteroids,
- number of items the player does not manage to pick up.

With the above information, the DDA system will generate a score for the player, and customise the difficulty based on this score. The difficulty can be customised by changing the asteroid density, or enemy and item numbers amongst other game elements.

With the game set to an optimal difficulty, the EDPCG system will initiate and receive data from the DDA system. If the player's performance drops again, the system will assume the player has become bored due to the lack of stimuli. This is based on a combination of Li-Xian Chen's study proving that focus drops when players become bored (Chen and Sun, 2016) and Westgate's paper which states that switching activities can reduce boredom (Westgate, 2019). As the system assumes the player needs new stimuli, it will switch the gameplay mode for a certain amount of time.

In the early stages of the EDPCG system, it will split the game into individual stages dedicated to one type of gameplay (classic battles, maze, collecting). However, it may

become more complex and combine these elements using percentages to indicate how much of each element to include.

Software to Develop

The project will be split into several pieces of software. These will run simultaneously and communicate through websockets or JSON files in a shared folder.
The game will be built in the Godot game engine using GDScript as a language. Initially, Unity was considered, however, as Godot has a simpler language and environment, it makes it a better candidate for fast prototyping of games (Technologies, 2023). Godot also has a dedicated 2D engine which can give it a slight edge (gamedevbeginner, 2022). For example, Godot can handle more 2D objects at certain FPSs (Mean Gene Hacks, 2021).

The DDA and EDPCG systems will likely be coded in GDScript inside the Godot engine, or run externally and written in Ruby. The reasons to not include them in Godot are a) reduce load on Godot resources, b) clearly separate the features.

A third system will also be used for demonstration purposes. This system will be a Ruby on Rails web page hosted locally and constantly updating a display of graphs and statistics to clearly show the intent of the DDA and PCG systems.

Future Benefits

This project will terminate once the proof of concept is complete. However, this system could become more complex and be included in larger games, thus benefiting the industry. An example of a game that could benefit from this is Uncharted as the Uncharted series often switches between puzzles, dialogue, cut scenes, combat, and climbing (Onorem, 2023).

Another sector to which this type of game could benefit is education. Many countries are attempting to create more student personalised education, which can be done through games or computer activities (Zhang et al., 2020). In such a field, a system to detect when students' attention levels are low, and which activities would reignite their attention will be very valuable.

These methods could also help with people who use games to improve their mental health (Kowal et al., 2021), by adapting the game to each player's needs.

Project Scope Conclusion

In conclusion there are many aspects to the gaming industry. This report has focused on Dynamic Difficulty Adjustments and Procedural Content Generation, which are both very large and complex parts of the industry with many variations. DDA and PCG have each existed for decades in various forms, and are both consistently becoming more common in the industry. DDA is commonly based on the user's experiences, and while the methods become more complex, the focus of adjusting the difficulty to allow the player to enjoy the game remains consistent. PCG on the other hand has many reasons to be implemented. The main benefit in the past of PCG was to allow the developers to populate a large world with less effort, however, it can also be used to assist in DDA systems, designing maps for individual players, as well as various other aspects of games.

While DDA output has in the past affected PCG systems, the purpose has been to adjust the difficulty of the game. This paper has suggested a new purpose of passing DDA output into PCG systems which can potentially increase player focus, which can lead to more enjoyment of the game.

Requirements Analysis

Functional Requirements

- Must include a playable “shoot ‘em up” including 3 modes of gameplay (asteroid maze, battling, collecting).
- Must analyse player ability and determine a score to be used in DDA and PCG.
- Must change difficulty based on player ability in the first section of the game.
- Must decide next sections of gameplay based on player ability variation.
- Should display some form of statistics in a second display.
- Should change difficulty based on player ability during the entire gameplay.
- Should keep track of which stages the player stayed focused on the longest.
- Could combine gameplay modes to create its own stages.
- Could include more than 3 game modes.
- Won’t allow the player to input a difficulty setting.
- Won’t need an internet connection.

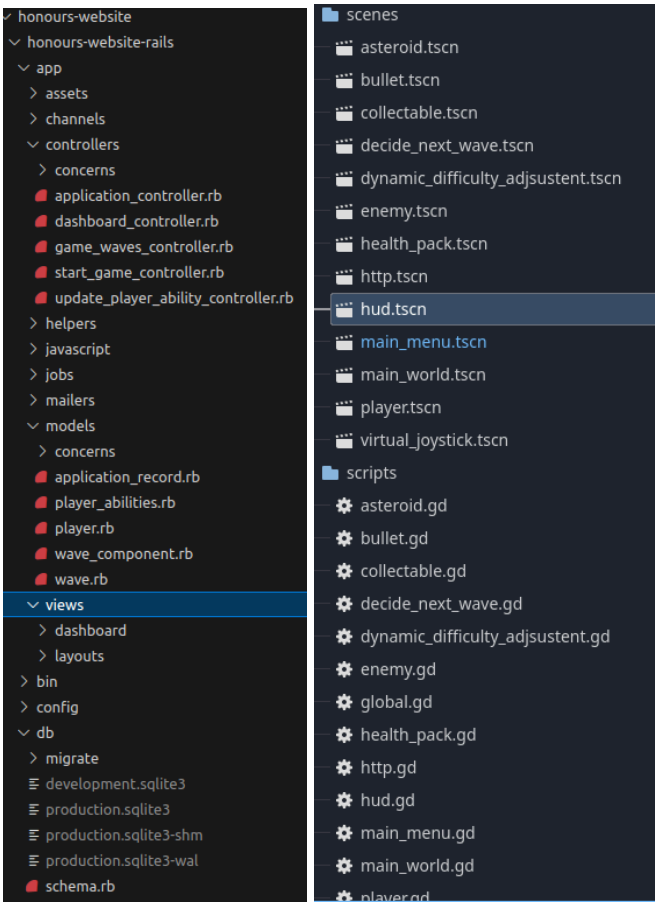
Non-Functional Requirements

- Run on a Dell 5 5590 laptop.
- Run on Godot.
- Run on a ruby on rails application (rails: V7, ruby: V3).
- Game must be played for at least 6 minutes.
- Game must receive better evaluations from players when the DDA and PCG systems are enabled.

Design / Methodology

The game will consist of one wave of objects (enemies, asteroids, collectables) followed by a collection of health boosts on repeat until the game ends either due to the player's health reaching 0 or the player ends/restarts the game. Each wave will spawn objects, with the type of objects depending on the game mode chosen and the quantity and frequency depending on the player's ability score for the current object type. Please view https://youtu.be/z6pE6eM7EoA to watch a full video of the game.

In order to allow more adaptability and improve the development process, all elements were split into their own components. These components fall under three categories: Web, Game, and Random-code. Random-code is for stand-alone scripts to perform tasks such as populating JSON files that are needed at game start, for example a list of possible spawn times that would be too CPU intensive to perform during gameplay. Web and Game have also been split into smaller chunks. In the Game code, each element has its own scene (.tscn, i.e. files that are used to control graphics and connections between base elements) and script (.gd i.e. code) files. Similarly, the Web system has been split into individual models, controllers and one view (MVC (Codecademy)). The web system developed in Ruby on Rails also has database and configuration files following the Rails structure.



Images 11, 12: Screenshots of folder structure (Rails left, Godot right)

The most important aspect of the project is the game itself. This has been coded in Godot for the reasons mentioned in Software To Develop. In addition to the game components and logic, the Experience Driven Procedural Content Generation (EDPCG) and Dynamic Difficulty Adjustment (DDA) systems have also been implemented in Godot in order to allow for more cross-platform compatibility. As all the key components are in Godot, the game as a whole can be exported to any platform that Godot supports, which currently are Android, IOS, Linux, Mac, Windows, and Web. For this project the main export version will be Web.

The scope of the web section allowed for many options of languages and frameworks. Rails was chosen due to personal past experience as well as its fast prototyping abilities (Górniak, 2023). The layout of the Rails application allows for one view file that displays a simple dashboard containing a table with upcoming objects (enemies, asteroids, collectables), and a graph depicting the players ability points over the previous waves among other information that is interesting during development and presentation, but

is not needed during gameplay. As with the backend of the website, the frontend could be developed in many ways. Due to the frontend being intended to be very simple, basic HTML, CSS, and JavaScript were chosen as the implementation methods. The Rails generates HTML through .erb templating files, thus pre-populating them with player information. Erb files suffice for the majority of the data displayed, however Highcharts (Highcharts) was added to simplify the graph's generation.

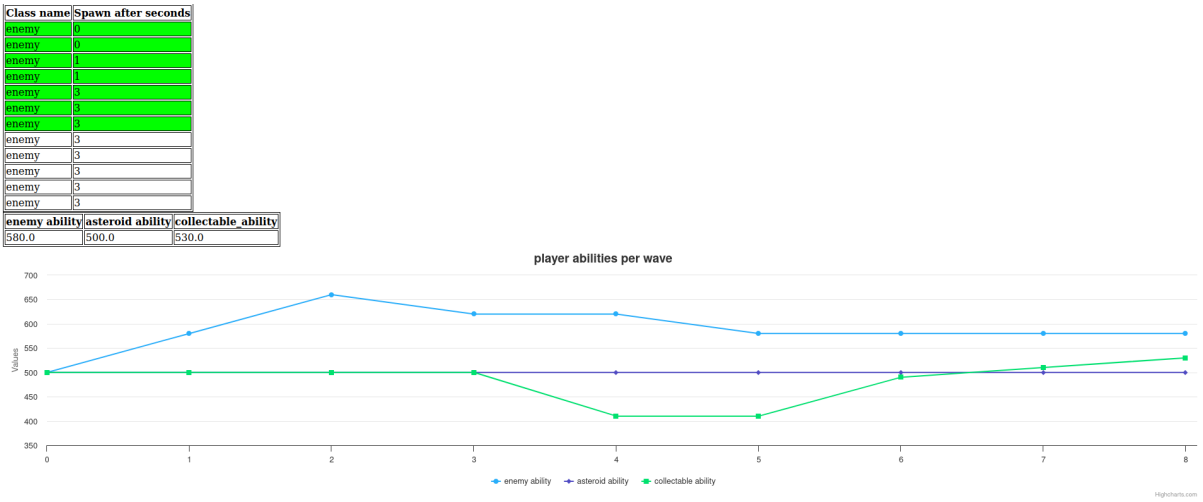


Image 13: Frontend graphs

As previously mentioned, the gameplay consists of three game types: Enemy, Asteroid, Collectables. These three plus health packs can be generated and added to the game through the EDPCG system. In addition, their variables such as speed, can be defined through the DDA system allowing them to adapt to the player's abilities. The player interacts with the above elements by controlling a small ship at the bottom of the screen with limited capabilities. This ship can be controlled using WASD or arrow keys to move, and the spacebar to shoot. The player's movement is restricted by the sides of the screen, the bottom, and a border 450 pixels (arbitrary) above the bottom of the screen to prevent the player from moving too high up. The player also has health that is displayed at the top of the screen in a green health bar.

Image 14: The game with enemies and health bar

The DDA file (dynamic_difficulty_adjustment.gd) stores the player's ability for each game type, and the history of these variables. It also stores the bullet, asteroid, enemy, and collectable vertical speeds and the enemy's horizontal speed and shooting delay. Each time the player takes damage from one of these objects, or completes a round, the above variables will be updated. For example, if the player enemy ability changes, the following code will run setting the bullet and enemy variables:

```
func _player_enemy_ability_change():
    bullet_speed = max(player_abilities_dict['enemy'], 45)
    enemy_speed_sideways = sqrt(player_abilities_dict['enemy'] * 6) * 3
    enemy_speed_down = max(player_abilities_dict['enemy'] / 3, 50)
    enemy_shooting_rate = 500.0 / player_abilities_dict['enemy']
```

Code extract 1: player_enemy_ability_change function

The above code will be executed for two possible reasons: in the first situation the code is executed immediately after the player takes damage, and their abilities are adjusted. The second situation in which the code is executed is after the wave is complete and the player receives 80 additional ability points in the category they just played.

The points that the player loses after being hit also adjusts if the player is hit multiple times during the same wave as depicted by the following code:

```
func _on_player_player_hit_by(object):
    if object.is_in_group('bullet'):
       decrease_player_ability(
         60 / max(times_points_were_lost_during_wave, 1), 'enemy'
       )
       times_points_were_lost_during_wave += 1
       # this is after so the first two are 60
       _player_enemy_ability_change()
    elif object.is_in_group('asteroid'):
       decrease_player_ability(
         60 / max(times_points_were_lost_during_wave, 1), 'asteroid'
       )
       times_points_were_lost_during_wave += 1
       # this is after so the first two are 60
       _player_asteroid_ability_change()
```

Code extract 2: on_player_player_hit_by function

If the player is hit once, they will lose 60, twice will cause 60 + 60, three times will result in 60 + 60 + 30 and so on. This way the ability will not drop too drastically if the DDA sets the difficulty too high, allowing the ability to gradually self-adjust.

The EDPCG (decide_next_wave.gd) file is called at the beginning of each wave, and returns an array populated with the game elements to add. Its first role is to define the game type (the type of objects to spawn). In the next step it will add the elements to the wave, in a similar method for all 3 game types. For example, with asteroids it will function as below:

```
func add_asteroids_to_wave(percentage):
    var asteroid_list = []

    var points = dda.player_abilities_dict['asteroid'] / percentage
    var time_delays = null
    var number_of_asteroids
    var points_to_make_up
    var seconds_to_make_up

    while time_delays == null:
       number_of_asteroids = number_of_objects(
```

```
                            points, cost_of_asteroid, 1.3
                        )

    points_to_make_up = points_to_make_up(
                            number_of_asteroids,
                            cost_of_asteroid,
                            points)
    seconds_to_make_up = points_to_make_up / 15
    time_delays=delay_option(seconds_to_make_up,number_of_asteroids)

var positions_in_x = positions_on_x(time_delays, 20)
for i in range(number_of_asteroids):
    asteroid_list.append(
                {
                    "scene": astroid_scene,
                    "position":Vector2(positions_in_x.pop_front(),0),
                    "time_s": time_delays.pop_front(),
                    "class": 'asteroid'
                }
            )
return asteroid_list
```

Code extract 3: add_asteroids_to_wave function

Executing the above code will return an array populated with dictionaries. Each dictionary will have the scene needed to spawn, the x and y positions, the number of seconds between this object and the previous object, as well as its class. Similar code will run for each game type if multiple are included in the wave (depending on the game mode chosen by the player) and all arrays will be merged and returned to the main_world to handle spawning (a mode with multiple types of objects does not exist yet, but would be easy to create due to the implementation of these components).

Communication between the game and the website is handled via HTTP post requests and is one sided (only the game communicates with the web). When the game starts, or it updates the player's abilities, or when a new wave is started, it will contact the relevant controller on the Rails app. The HTTP elements are handled by the http.tscn and http.gd files, which can be called from various game resources. Within the game, communication is often performed using method calls. However, in some situations such as when multiple functionalities are executed based on one trigger, signals are used.

Implementation

The project started as a blank world with a placeholder image of the Godot logo as the player's spaceship. The first task was to allow the player to move the spaceship within the allowed borders. Once this was complete the ability to shoot was added. When the player's spaceship was complete, simple template enemies were created using the player's script as a default with a few significant changes. Directions were flipped and the movement was automated to allow the enemies to move down and sideways at a fixed speed.

Image 15: Game when enemies were first added.

These speeds will later be updated to account for the player's ability, and will be assigned to the enemy at the spawn time. Without the spawning system the enemies were manually placed slightly above the screen. However, in the final version a more sophisticated method would be needed. So the spawning system was created. decide_next_wave.gd is designed to return an array with various information about the wave. Initially decide_next_wave would return a manually written array with the 'scene', 'position', 'time_s' (delay between the previous spawn), and 'class'. The main world class was updated to be able to handle the decide_next_wave response and create the instances as instructed, including changing the object type depending on the "scene"

passed. With the main world able to create the instances, decide_next_wave was fleshed out to be able to always return a random array of enemies depending on the player's ability (currently static at 100). decide_next_wave was using functions that could later be adapted to work with other classes too. These functions include methods to define the x coordinates and the delay times and functionality to calculate the number of enemies (or objects) as seen below:

```
func positions_on_x(time_delays, size, max_distance=10000):
    var positions_in_x = []
    var positions_used_in_second = []
    for delay in time_delays:
       if delay > 0:
            positions_used_in_second = []

       var x_pos = random_position_on_width()
       var last_x = positions_in_x.back()
       while invalid_x_pos(positions_used_in_second, x_pos, size,
max_distance, last_x, delay):
            x_pos = random_position_on_width()

       positions_in_x.append(x_pos)
       positions_used_in_second.append(x_pos)
    return positions_in_x
```

Code extract 4: positions_on_x function

```
func delay_option(seconds_to_make_up, number_of_objects):
    if !possible_wait_combinations.keys().has(str([seconds_to_make_up,
number_of_objects])):
       return null
    var time_delay_options = possible_wait_combinations[
       str([seconds_to_make_up, number_of_objects])
       ]

    return [0] + time_delay_options[randi() % time_delay_options.size()]
```

Code extract 5: delay_option function

```
func number_of_objects(points, cost_of_object, count_multiplier=1.5):
    var min_number_of_objects = points / cost_of_object
    var max_number_of_objects = min_number_of_objects * count_multiplier
    var max_objects = 100 # from json (all_wait_combinations.json)
    return min(random.randi_range(min_number_of_objects,
max_number_of_objects), max_objects)
```

Code extract 6: number_of_objects function

The delay times in decide_next_wave are calculated based on the number of objects to spawn. “min_number_of_objects” would define the minimum number to spawn as the player's ability in the current game mode divided by the cost of the object. And the maximum number of objects as the minimum multiplied by two (this would later be updated as 1.5 or a custom value passed in). The number of objects will then be a random number within the two. This would then allow for more objects to be spawned than the player's ability suggests resulting in the current points remaining being negative, however, increased delay times compensate for the extra enemies. decide_next_wave would add one second delay to a random object per x ability points needed to reach 0. The allocation of the seconds was done by generating all possible arrays of integers (seconds) with length equal to the number of objects minus one, and with the sum of integers equal to the seconds to make up. The above method, however, would cause lag as the length of the array increased. A pre-built json was then used. In a Ruby script, all possible arrays under length x (number of objects-1) with sum y (seconds to make up) were created and stored in 'all_wait_combinations.json' to be used during gameplay.

```
require 'parallel'
require 'json'

# thanks chatgpt

def generate_combinations_with_length(seconds_to_make_up,
num_of_objects, current_combination, result)
  if seconds_to_make_up.zero? && num_of_objects.positive?
      num_of_objects.times { current_combination.append(3) }

      num_of_objects = 0
  end

  return result.append(current_combination) if seconds_to_make_up.zero?
&& num_of_objects.zero?

  return if seconds_to_make_up <= 0 || num_of_objects <= 0

  numbers = (0..[3, seconds_to_make_up].min).to_a
  numbers = numbers.shuffle
```

```
  while numbers.any? && result.count < 2
      num = numbers.pop
      generate_combinations_with_length(seconds_to_make_up - num,
num_of_objects - 1, current_combination + [num], result)
  end
end

def find_combinations_with_length(seconds_to_make_up, num_of_objects)
  result = []
  generate_combinations_with_length(seconds_to_make_up, num_of_objects,
[], result)
  result
end

all_combinations = JSON.parse(File.read('all_wait_combinations.json'))

Parallel.map((0..40).to_a, in_threads: 22) do |seconds_to_make_up|
  100.times do |num_of_objects|
      num_of_objects += 1 # dont do 0
      next if all_combinations[[seconds_to_make_up,
num_of_objects].to_s]

      puts "x: #{seconds_to_make_up}, y: #{num_of_objects}"
      all_combinations[[seconds_to_make_up, num_of_objects]] =
find_combinations_with_length(seconds_to_make_up, num_of_objects)
  end
end

File.open('all_wait_combinations.json', 'w') do |f|
  f.write(all_combinations.to_json)
end
```

Code extract 7: Code to populate 'all_wait_combinations.json'

However, as x and y increased during testing, the json file became too large, passing 160MB. Initially, the enemy difficulty (speed, and bullet shooting speed) was increased to reduce the chances of x and y becoming too large, but this attempt proved futile. In the final attempt the number of possible arrays was defined to only generate a few possibilities for each x and y combination, and this number decreased (manually adjusting the file) as x and y increased, meaning more likely combinations had more possibilities.

```
{"[0, 1]":[[3]],"[0, 2]":[[3,3]],
"[8, 18]":[[3,2,2,1,3,3,3,3,3,3,3,3,3,3,3,3,3,3]],
"[8,21]":[[2,3,2,0,1,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3],[2,3,2,0,0,0,0,0,0
,1,3,3,3,3,3,3,3,3,3,3,3],[2,3,2,0,0,0,0,0,0,0,0,0,1,3,3,3,3,3,3,3,3]]}
```


Code extract 8: Example outputs from ''all_wait_combinations.json'

In the first example above `"[0, 1]":[[3]]` 0 refers to 0 seconds needed to increase the ability points to 0, and 1 object to spawn. Therefore, an array of length 1 with the value of 3 (no reduced wait time) was created.

With a basic game set up, the website to track waves and abilities during development was created. This website consists of a Rails project run on a docker container and connected to a sqlite database to store the abilities history and the current wave data.

```
create_table "player_abilities", force: :cascade do |t|
    t.integer "wave_id"
    t.float "enemy_ability"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.float "asteroid_ability"
    t.float "collectable_ability"
    t.index ["wave_id"], name: "index_player_abilities_on_wave_id"
 end

 create_table "wave_components", force: :cascade do |t|
    t.integer "wave_id"
    t.string "class_name"
    t.datetime "spawn_time"
    t.integer "spawn_after_seconds"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["wave_id"], name: "index_wave_components_on_wave_id"
 end

 create_table "waves", force: :cascade do |t|
    t.integer "wave_number"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
 end
```

Code extract 9: Website schema

As the game processes, HTTP requests are sent to the Rails API endpoints with json data, that the server will store and display to the developer. This data consists of the objects that will be spawned on the current wave, which is sent at the start of each wave. As well as the player's abilities that will be sent each time they change. Initially, webhooks were considered for the connection between the game and the Rails application, however, basic HTTP proved easier to implement in Godot than originally expected. The website then displays the two datasets to the developer on the dashboard page. The wave information is a simple table that will state the object being spawned and the delay in seconds between the object and the previous one. Each row in the wave table will become green as the object is spawned and if the table grows too large, it will be split into smaller tables along the same horizontal axis.

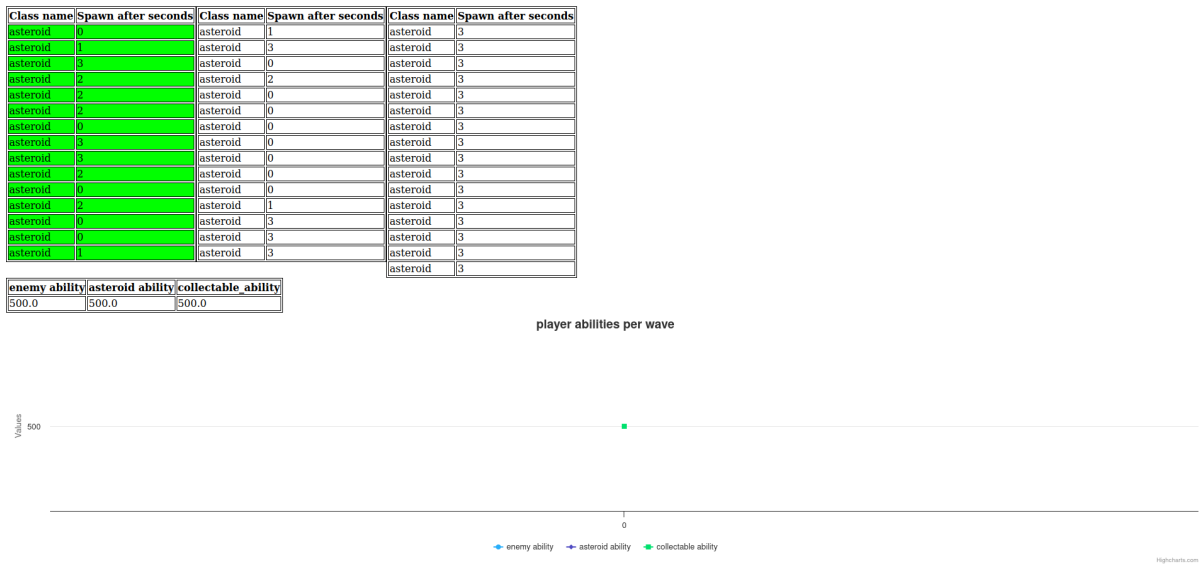


Image 16: Multiple tables of objects to spawn in current wave

The more interesting player ability data is displayed in a graph composed of three points: enemy, asteroid, and collectable abilities per wave. This graph allows the developer to monitor their ability during the game, and therefore, ensure that the game is accurately editing its behaviour to accommodate the player's ability change.

Class name	Spawn after seconds
enemy	0
enemy	0
enemy	1
enemy	2
enemy	0
enemy	1
enemy	3
enemy	2
enemy	3
enemy	2
enemy	0
enemy	0
enemy	0
enemy	0
enemy	2

Class name	Spawn after seconds
enemy	2
enemy	0
enemy	3
enemy	3
enemy	1
enemy	3
enemy	1
enemy	0
enemy	1
enemy	3
enemy	3

enemy ability	asteroid ability	collectable_ability
880.0	1000.0	500.0

Class name	Spawn after seconds
enemy	0
enemy	3

enemy ability	asteroid ability	collectable_ability
100.0	1000.0	500.0

Images 17, 18: enemies to spawn with 1000 ability (left) and 100 ability (right)

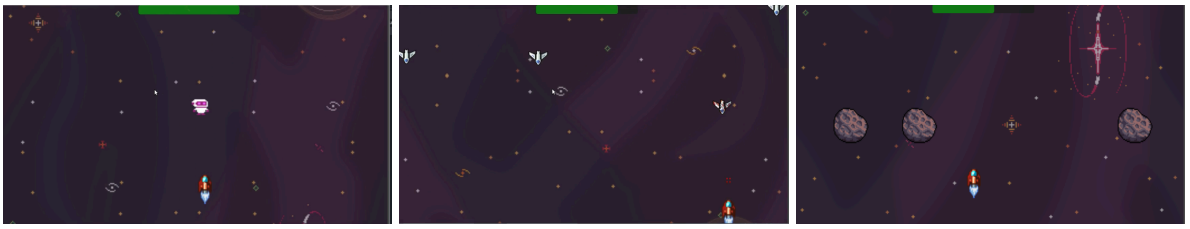
	100 ability	1000 ability
bullet_speed	100	1000
enemy_speed_sideways	73	232
enemy_speed_down	50	333
enemy_shooting_rate	5	0.5

Table 1: enemy variables with 100 and 1000 ability

Player health was implemented as well. The player has five health points, with each hit from an asteroid or bullet costing one, as well as each missed collectable. The player can view their health at the top of the screen during gameplay, and is given an opportunity to add to their health after each wave. After each wave many health packs are dropped in a similar manner as collectables. These are very easy to collect and provide the player with an easy method of filling their health bar while resting after the wave. Enemies also have 3 health which is conveyed to the player by the damage on the enemy ship's sprite.

Image 19: health packs

Asteroid and collectable functionality was implemented in similar ways. Both would “fall” downwards at certain speeds and as all other objects, they are deleted if they passed the screen limits. If the player misses a collectable they will lose health and the asteroids will cause damage to the player if they hit the player’s ship.



Images 20, 21, 22: Final versions of Collectible, Enemy, Asteroid (left to right)

An important element to the DDA system was adjusting each object to the player's ability. The adjusted characteristic which is set at spawn time for all objects (including bullets) is downwards speed, with each object using a different calculation. Enemies have more data which is dependent on the DDA system, these are: sideways speed, and the interval in seconds between shots.

Once all scenes were mostly implemented, with only fine-tuning requirements left, the images were updated. The images were downloaded from open source sites (licences included in the repository). An effort was made to get images from the same artist in order to have a consistent style. However, some images were obtained from other sources.

Images 23, 24, 25, 26, 28: Collectable, Enemy, Asteroid, Health, Ship images (left to right)

Throughout the development the evaluation algorithms were adjusted to allow for more accurate DDA data. The original idea during the Project Scope was to evaluate players on many data points, including how close they are to asteroids, and how many enemy ships they destroy, and how close the collectables were to the bottom of the screen before being collected. However, these approaches were deemed unfit as without implementing a clear incentive to avoid these situations some players might try to increase the difficulty by giving themselves extra challenges. The challenges some players might add could resemble “Can I pass without shooting?”, “Can I pass through that small gap?”, “Can I collect collectables at the last second?” and players attempting those would be providing inaccurate data to the DDA system. Therefore, the DDA system must only use the data that is guaranteed to depict a skilled or unskilled player. The most accurate data is “if a player takes damage” => “they are performing badly”, thus, this was used. In order to increase the player’s abilities similar methods were considered, however, they could all have the same flaws. Therefore, to accurately update the players abilities, a simple method was chosen. After each wave the player's abilities will increase by a standard amount. Therefore, if the player does not get hit, their ability will increase after the wave. If, however, the player gets hit once, their ability score will decrease due to the damage but will also increase due to the end of the wave which has a greater value, so the ability will still trend upwards, but at a lower rate. The third option of the player taking damage multiple times in one wave will result in the ability decreasing more than the end of wave increase, thus the ability will trend down.

In the final version, 6 game modes exist: 3 are intended for development, 2 for testing, and one as the final game. The three for development allow the developer to only include one game type (enemy, asteroid, collectable), and thus enables an easy method of testing individual components. The two game modes for testing are ordered_switch (Mode 1) and random_switch (Mode 3), which will change the game type in a consistent order after each wave, and change the game type randomly (sometimes choosing the

current mode again) after each wave respectively. These two game modes are referred to as “mode 1” and “mode 3” in the questionnaire. The final mode “mode 2” is the main game which will dynamically adjust the game type depending on the player's ability.

Testing & Results

The game is not the main goal of this project: the main goal is the adaptive content designed to increase engagement within the game. Therefore, the testing must focus on how people perceive the entertainment value of the game with different features enabled. In order to allow testers to review the game and the DDA/EDPCG combination the game is deployed on an AWS, EC2, Ubuntu, Node server available at kaveh-nejad.com, as well as a Google form for participants to fill in (please see Appendix C for all questions).

Alpha Testing

During development most features were tested and fine-tuned as they were implemented. However, once all features were complete, the game was tested and many features were adjusted. These features include the default player abilities which increased from 250 to 500 to allow the game to more quickly adjust to the player's levels which is often greater than 250. The maximum quantity of the spawnable objects was decreased to allow for more condensed and quicker waves in order to reduce the time testers would need to play the different modes.

Another issue that was encountered towards the end of the development was the lag when hosted as a web page. This lag and stutter would be introduced because of the browser computing limitations as all the game data is sent to the client through a .wasm file. As the game was designed with game efficiency in mind there were not many parts that stood out for refactoring. The first update attempted was multiplying all speeds by delta (“the amount of time that has passed since the last frame was drawn” (Dragonfly)). Furthermore, the debugging code was removed, such as the HTTP requests being sent out to the Rails application. In addition to removal of debug features, the most impactful change was setting the maximum frames per second (FPS) to 40 which almost completely removed the jitter. Adding a maximum FPS lower than the smallest expected value had the added benefit of making the game's speed more consistent.

Beta Testing

Participants and Methodology

In the first set of beta testing, a copy of the game as well as the questionnaire was given to an initial tester, in which she was able to suggest clarifications on the questionnaire and through monitoring her gameplay a final bug in the PCG system also was caught.

The participants received a questionnaire with sixteen questions. The first two questions are intended to provide information about the tester for future comparisons. This information consists of their age in roughly five year ranges, and how frequently they play video games.

The second to last question is “Any additional comments overall?” as an open-ended question to allow the user to inform the developer of any thoughts they had.

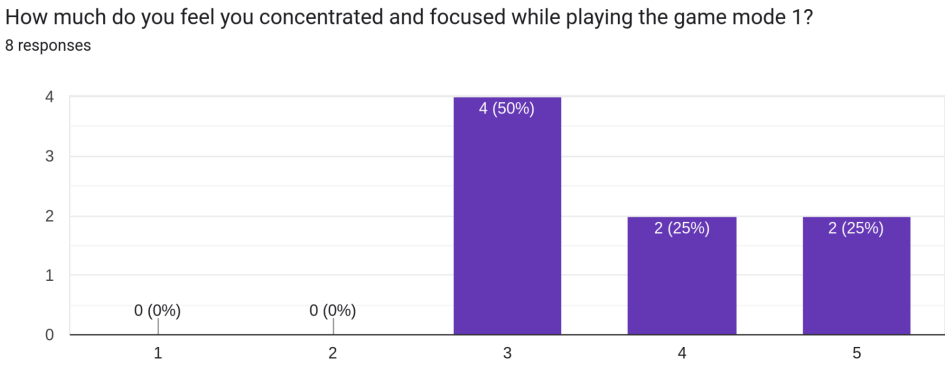
The final question is a text box for the players to input their game data that they can copy past from the main screen. This data is in the form of `[{"abilities":{"asteroid":[...], "collectable":[...],"enemy":[...]},"game_mode":"Mode 1"}]` and will enable the creation of graphs once submitted (see Appendix D for all graphs).

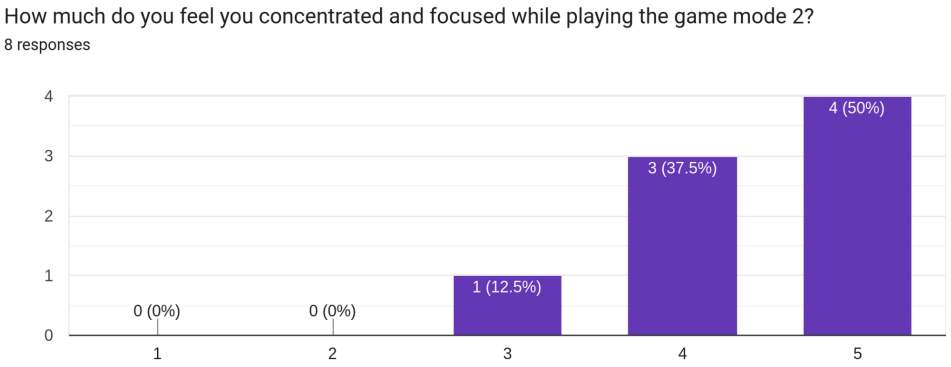
The participants in the questionnaire are from multiple backgrounds, various ages and ranging knowledge of game development and technical experience. However, these questions were not included in an attempt to simplify the questionnaire. An additional point is that some of the participants have already had the concept of this project explained to them in the past.

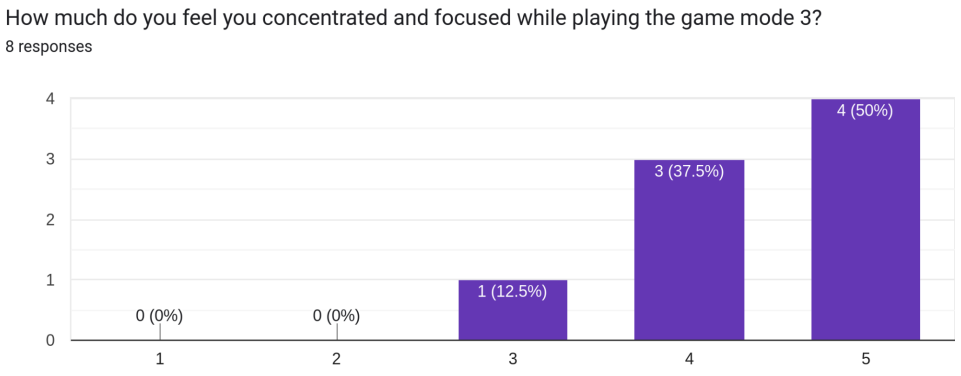
Of the eight participants, four were aged 21-25 while the remainder was split evenly between 26-30 and 51+. The question “How often do you play video games?” had 25 percent responding “multiple times per month” while “less than once per month” and “multiple times per week” received 37.5% (3).

Questionnaire Results

As seen in the results below, the majority of people believe they paid more attention during mode 2 and 3 (random and main) (there are potential issues with this section, please see Evaluation for more details). This suggests that most players lost interest with an ordered switch.

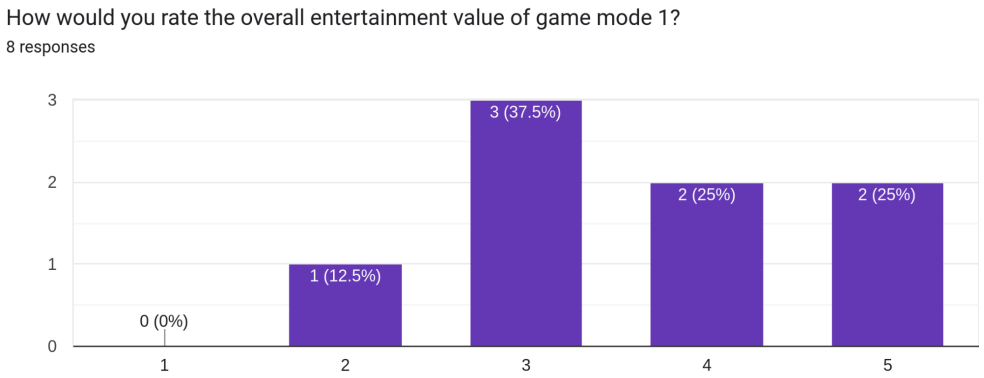


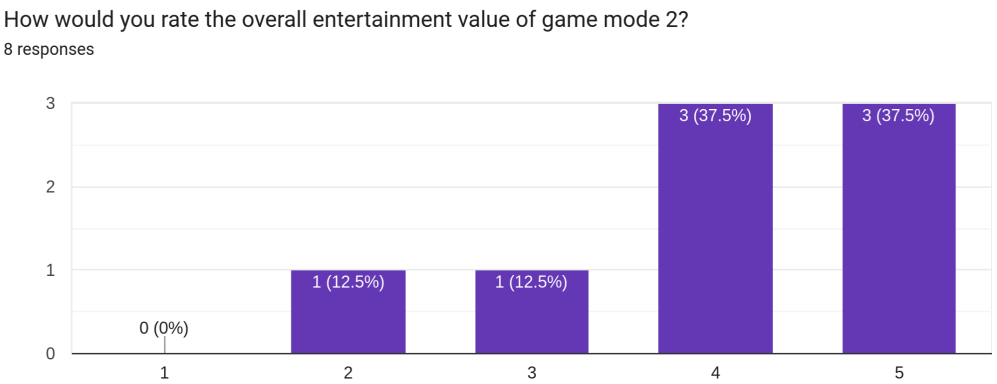


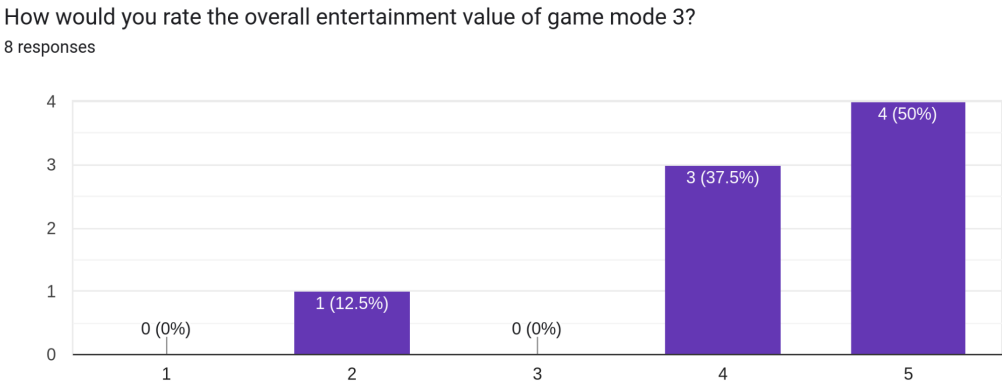


Images 29, 30, 31: Charts depicting concentration and focus values per game mode. From the Google forms survey.

Similar results can be seen with the overall entertainment value of the game:







Images 32, 33, 34: Charts depicting entertainment values per game mode. From the Google forms survey.

Most people were also unable to determine how the game decides which game types to spawn (please see Evaluation for possible issues). In game mode 1 (ordered), three people predicted that it was ordered, one assumed random, and four were not sure. In mode 2 (main), one person answered dynamically, three assumed randomly, two did not know, and one realised that it is adapting to become easier as they made mistakes but did not understand how the waves were decided, and one assumed it was predefined to get harder each wave. As for game mode 3 (random), two predicted random, one dynamically, one assumed the difficulty is based on the performance in the previous round, one assumed it was based on the position of the player (they were possibly mentioning the collectables which are designed to spawn close to each other and therefore, likely around the players ship), and one assumed the difficulty was adjusted based on the previous mode.

Using the DDA ability scores, charts were generated depicting the player's abilities over each wave (see Appendix D). In these charts, most players' abilities increased as they played with few decreases. Regardless of if the abilities increased or decreased to adjust to the player, they appear to stabilise at a score where the player is consistent. The longer a player played the more stable their abilities. This demonstrates that the DDA system can adjust to each player given enough data (see appendix C and D for full results)

Evaluation

Questionnaire Results

The results of the questionnaire did not reflect the intended outcomes of the project. The DDA abilities score graphs do depict that the abilities often adjust until they reach a stable position, and some testers did notice an adjustment of the difficulty. However, the questionnaire was not able to see an increase in the attention or enjoyment due to the EDPCG system as the scores of the random mode were equal to the main mode. This may indicate that longer testing times are necessary or that the EDPCG system needs more fine-tuning.

Dynamic Difficulty Adaptation

The main issue encountered while developing was implementing the DDA system. Unfortunately, the majority of articles/papers found online were either too vague, depicting DDA fundamentals without describing how to implement them, or too narrow focusing on games that are vastly different from shoot em ups. Other methods would have likely produced better results. An interesting method of implementing DDA to explore would have been Bayesian-based player models, in which a model of the player is created, and therefore the game is able to predict rather than retroactively adjust (Gonzalez-Duque et al., 2021).

Experience Driven Procedural Content Generation

The EDPCG system also requires further optimisation. The spawning of objects is not implemented in a method where a skilled player can realistically collect every item in bad situations. This was mentioned multiple times in the questionnaire. More focus could also be required on spawning times to reduce boredom.

Questionnaire

The questionnaire was also flawed. Ideally, it would have included more sections, with more game modes that are able to collect players' opinions on the DDA systems rather than just focusing on the EDPCG system. However, the extra sections would have increased the time needed to fill in the questionnaire and therefore make it unlikely that people would be able to complete it. It also appears that the DDA system was not

accurate enough in the first few waves. Given more time, a better system would have people test the game and fill in a questionnaire solely on the enjoyment of the game types (asteroid, enemy, collectable). This questionnaire would be followed by more development to improve the enjoyment value of the game types. With the recommendations implemented on the game types, a second questionnaire would collect data on the DDA system. More development would follow to improve the DDA system based on the answers to allow for a more accurate evaluation of the EDPCG system in a third and final questionnaire. A final issue with the questionnaire is the consistent ordering of the game modes. As seen in Testing & Results, the majority of players preferred game modes 2 and 3 over game mode 1. This could indicate that the players do prefer those modes, or it could indicate that players prefer the later modes due to already becoming familiar with the game or a number of other factors. The opposite could have also been possible, but was not seen in this questionnaire.

Development Environment

Many tools were used in the creation of this project. The most important tool was Godot. Godot proved very useful due to its easy-to-use 2D interface, methods of connecting objects, and choice of scripting language. However, it also provided challenges due to the performance loss of the high-level language when running on a web platform and an additional difficulty was the integrated IDE which does not have many of the features expected in modern IDEs. Additionally, Ruby on Rails performed exactly as expected making the development of the dashboard web page easy to implement.

Self-Reflection

Personally, I am also happy with the overall outcome, while the DDA and EDPCG systems are not as accurate as I had hoped for, I do see potential in the overall idea and I hope to see it fully implemented in a more advanced game one day. I have also learned more about gamedev than I had expected, such as new ways of optimising the game processes and techniques of structuring the game logic. Most importantly the value of proper game testers to evaluate the individual components was made clear, and I will make more of an effort in the future.

Conclusions & Future Work

This project was an initial attempt at developing a game that can adapt the content to the player's interest based on the data collected by the Dynamic Difficulty Adjustment (DDA) system. Various other projects could be attempted including more sophisticated DDA and Procedural Content Generation (PCG) systems thus allowing for more accurate results to be collected. As mentioned in the Evaluation section, more emphasis should be placed on the DDA systems in future adaptations to ensure the final results reflect on the PCG system rather than the DDA system or other game components. Other game types could also be implemented. In this approach, the three game types appeared to be too similar. Future work would benefit from investigating other possible game types that allow for more distinct gameplay, thus emphasising the change created by the PCG system.

Projects tailor-made to other fields would likely also provide interesting insights into the feasibility of this combination of systems. An interesting example would be in education where the students receive scores based on their abilities in distinct fields. Over the course of hours, days, or weeks the systems adapt the teaching curriculum or subject types in an attempt to increase the learning rate of the students. Or, alternatively, the system could detect the optimal times for students to take breaks, by detecting when the students' attention drops before the students become bored of the task. The system to enable optimal breaks would reduce the chance of students not wanting to return to a subject because their last experiences ended badly (CLEAR, 2018). Of course, these systems would work best for individual students learning by themselves. However, an average could be used for an entire class. Similar implementations for detecting optimal times to stop performing a task could also help individuals in therapy with other tasks if proven successful.

References

ANAXIAL, 2023. Dynamic game difficulty balancing. [online]. Wikipedia. Wikimedia Foundation. Available from: https://en.wikipedia.org/wiki/Dynamic_game_difficulty_balancing [Accessed 8 October 2023].

ARCADEOLOGIA, 2015. Gun fight. [online]. Arcadeología. Available from: https://arcadeologia.es/en/machines/gun-fight-11.html [Accessed 22 October 2023].

BILL, 2022. GameAccess by SpecialEffect: The last of us part II: Motor accessibility options. [online]. GameAccess. Available from: https://gameaccess.info/the-last-of-us-part-ii-motor-accessibility-options/9/ [Accessed 23 October 2023].

CHEN, L.-X. and SUN, C.-T., 2016. Self-regulation influence on game play Flow State. Computers in Human Behavior, 54, pp. 341–350.

CICHACKI, S., 2023. How to turn off swing assistance in Spider-Man 2. [online]. Prima Games. Available from: https://primagames.com/tips/how-to-turn-off-swing-assistance-in-spider-man-2 [Accessed 23 October 2023].

CLEAR, J., 2018, *Atomic habits*. AVERY PUB GROUP: Penguin Random House.

CODECADEMY, 2024. *MVC: Model, view, Controller*. [online]. Codecademy. Codecademy. Available from: https://www.codecademy.com/article/mvc [Accessed 5 March 2024].

DANBLOCH , 2023. Rogue (video game). [online]. Wikipedia. Wikimedia Foundation. Available from: https://en.wikipedia.org/wiki/Rogue_%28video_game%29 [Accessed 22 October 2023].

DRAGONFLY, 2024. *[answered] when should you use Delta in godot?* [online]. Dragonfly. Available from: https://www.dragonflydb.io/faq/godot-when-to-use-delta [Accessed 20 April 2024].

FANDOM.COM, 2021. Difficulty levels. [online]. Pathfinder. Fandom, Inc. Available from: https://pathfinderkingmaker.fandom.com/wiki/Difficulty_levels [Accessed 17 October 2023].

FRASER, J., PAPAIOANNOU, I. and LEMON, O., 2018. Spoken conversational AI in video games. Proceedings of the 18th International Conference on Intelligent Virtual Agents.

GAMEDEVBEGINNER, J., 2022. Godot vs Unity (for making your first game). [online]. Game Dev Beginner. Available from: https://gamedevbeginner.com/godot-vs-unity-for-making-your-first-game/#:~:text=Unity%20is%20generally%20faster%20and,outstanding%20performance%2C%20and%20it%20does. [Accessed 15 October 2023].

GIAKOUMIS, D. et al., 2011. Automatic recognition of boredom in video games using novel Biosignal moment-based features. IEEE Transactions on Affective Computing, 2(3), pp. 119–133.

GONZALEZ-DUQUE, M., PALM, R.B. and RISI, S., 2021. Fast game content adaptation through Bayesian-based player modelling. 2021 IEEE Conference on Games (CoG).

GÓRNIAK, J., 2023. *Why is ruby on rails a good choice for developing your MVP? [2023 update]*. [online]. netguru. Available from: https://www.netguru.com/blog/build-mvp-with-ruby-on-rails [Accessed 5 March 2024].

GRAVINA, D. et al., 2019. Procedural content generation through quality diversity. 2019 IEEE Conference on Games (CoG).

HARRIS, D.J. et al., 2021. A systematic review and meta-analysis of the relationship between flow states and performance. International Review of Sport and Exercise Psychology, pp. 1–29.

HARRIS, D.J., VINE, S.J. and WILSON, M.R., 2017. Neurocognitive mechanisms of the Flow State. Progress in Brain Research, pp. 221–243.

HERGENRATHER, M., 2020. Dynamic difficulty: A player perspective. WRIT: Journal of First-Year Writing, 3(1), pp. 6–6.

HIGHCHARTS, 2024. *Simply visualize*. [online]. Highcharts. Highcharts. Available from: https://www.highcharts.com/ [Accessed 5 March 2024].

KOWAL, M. et al., 2021. Gaming your mental health: A narrative review on mitigating symptoms of depression and anxiety using commercial video games. JMIR Serious Games, 9(2).

LIANG, Y., 2022. Analysis of the video gaming industry. Proceedings of the 2022 2nd International Conference on Enterprise Management and Economic Development (ICEMED 2022).

LIBERTYGAMES, 2013. [online]. A detailed history of shoot ‘em up arcade games. Available from: https://www.libertygames.co.uk/blog/a-detailed-history-of-shoot-em-up-arcade-games/ [Accessed 22 October 2023].

LIU, J. et al., 2020. Deep Learning for Procedural Content Generation. Neural Computing and Applications, 33(1), pp. 19–37.

MASEM, 2023. Shoot ’em up. [online]. Wikipedia. Wikimedia Foundation. Available from: https://en.wikipedia.org/wiki/Shoot_%27em_up [Accessed 15 October 2023].

MASTAZAJEB, 2018. Pathfinder Kingmaker - Beginner’s Guide. [online]. Fextralife. Available from: https://fextralife.com/pathfinder-kingmaker-beginners-guide/ [Accessed 23 October 2023].

MEAN GENE HACKS, 2021. Godot vs. unity in 2D: Who will win? [online]. YouTube. YouTube. Available from: https://www.youtube.com/watch?v=GPgkW0h4r1k [Accessed 15 October 2023].

MI, Q. and GAO, T., 2022. Adaptive rubber-banding system of dynamic difficulty adjustment in racing games. ICGA Journal, 44(1), pp. 18–38.

MISSURA, O., 2015. Dynamic Difficulty Adjustment. [online]. Handle Proxy. Universitäts- Und Landesbibliothek Bonn. Available from: https://hdl.handle.net/20.500.11811/6543 [Accessed 8 October 2023].

MOSCHOVITIS, P. and DENISOVA, A., 2023. Keep calm and aim for the head: Biofeedback-Controlled Dynamic Difficulty Adjustment in a horror game. IEEE Transactions on Games, 15(3), pp. 368–377.

NEREVAR, 2020. The elder scrolls 6 and procedural generation. [online]. UESP Blog. Available from: https://blog.uesp.net/the-elder-scrolls-6-and-procedural-generation/ [Accessed 17 October 2023].

ONOREM, 2023. Uncharted. [online]. Wikipedia. Wikimedia Foundation. Available from: https://en.wikipedia.org/wiki/Uncharted [Accessed 15 October 2023].

PETITTE, O., 2013. Skyrim enhanced terrain mod increases detail of distant textures. [online]. pcgamer. PC Gamer. Available from: https://www.pcgamer.com/skyrim-enhanced-terrain-mod/ [Accessed 22 October 2023].

RUBY, D., 2023. Smartwatch Statistics 2023: How many people use smartwatches? [online]. DemandSage. Available from: https://www.demandsage.com/smartwatch-statistics/ [Accessed 12 October 2023].

SCHAFFER, O. and FANG, X., 2019. Digital Game Enjoyment: A Literature Review. Lecture Notes in Computer Science, pp. 191–214.

SEPULVEDA, G.K., BESOAIN, F. and BARRIGA, N.A., 2019. Exploring dynamic difficulty adjustment in videogames. 2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON).

SHAKER, N., TOGELIUS, J. and NELSON, M.J., 2016. Procedural content generation in games. SPRINGER.

SMEDDINCK, J.D. et al., 2016. How to present game difficulty choices? Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems.

SMITH, G., 2015. An Analog History of Procedural Content Generation. [online]. Available from: http://sokath.com/home/wp-content/uploads/2018/01/smith-fdg15.pdf.

SOUTTER, A.R. and HITCHENS, M., 2016. The relationship between character identification and flow state within video games. Computers in Human Behavior, 55, pp. 1030–1038.

TAIT, E.R. and NELSON, I.L., 2021. Nonscalability and generating digital outer space natures in no man's sky. Environment and Planning E: Nature and Space, 5(2), pp. 694–718.

TECHNOLOGIES, A., 2023. Unity 3D vs Godot: Which Game Engine Is Right for your project? [online]. Unity 3D vs Godot-Which Game Engine is best for Your Project. Available from: https://www.linkedin.com/pulse/unity-3d-vs-godot-which-game-engine-right-your-project [Accessed 15 October 2023].

TRAN, E., 2020. The last of US 2 accessibility / difficulty options: A detailed overview. [online]. GameSpot. Available from: https://www.gamespot.com/articles/the-last-of-us-2-accessibility-difficulty-options-/1100-6478756/ [Accessed 23 October 2023].

VANG, C., 2022. The impact of dynamic difficulty adjustment on player experience in video games. Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal, 9(1).

WESTGATE, E.C., 2019. Why boredom is interesting. Current Directions in Psychological Science, 29(1), pp. 33–40.

YANNAKAKIS, G.N. and TOGELIUS, J., 2011. Experience-driven procedural content generation. IEEE Transactions on Affective Computing, 2(3), pp. 147–161.

ZENG, J., PARKS, S. and SHANG, J., 2020. To learn scientifically, effectively, and enjoyably: A review of educational games. Human Behavior and Emerging Technologies, 2(2), pp. 186–195.

ZHANG, L., BASHAM, J.D. and YANG, S., 2020. Understanding the implementation of personalized learning: A research synthesis. Educational Research Review, 31, p. 100339.

ZOHAIB, M., 2018. Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review. Advances in Human-Computer Interaction, 2018, pp. 1–12.

ZUCCONI, A., 2023. The World Generation of Minecraft. [online]. Alan Zucconi. Available from: https://www.alanzucconi.com/2022/06/05/minecraft-world-generation/ [Accessed 15 October 2023].

Appendix A

Complex difficulty settings and other game options that can assist in difficulty selection.

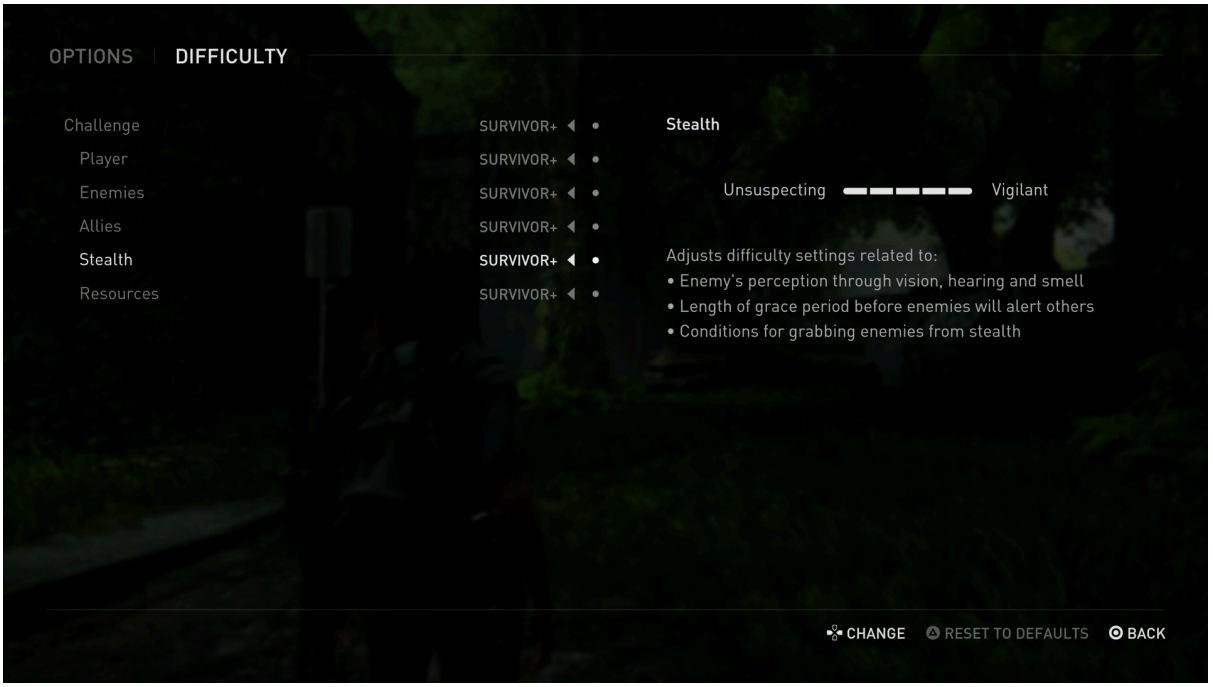


Image A1: The last of us two - various difficulty settings (Tran, 2020)

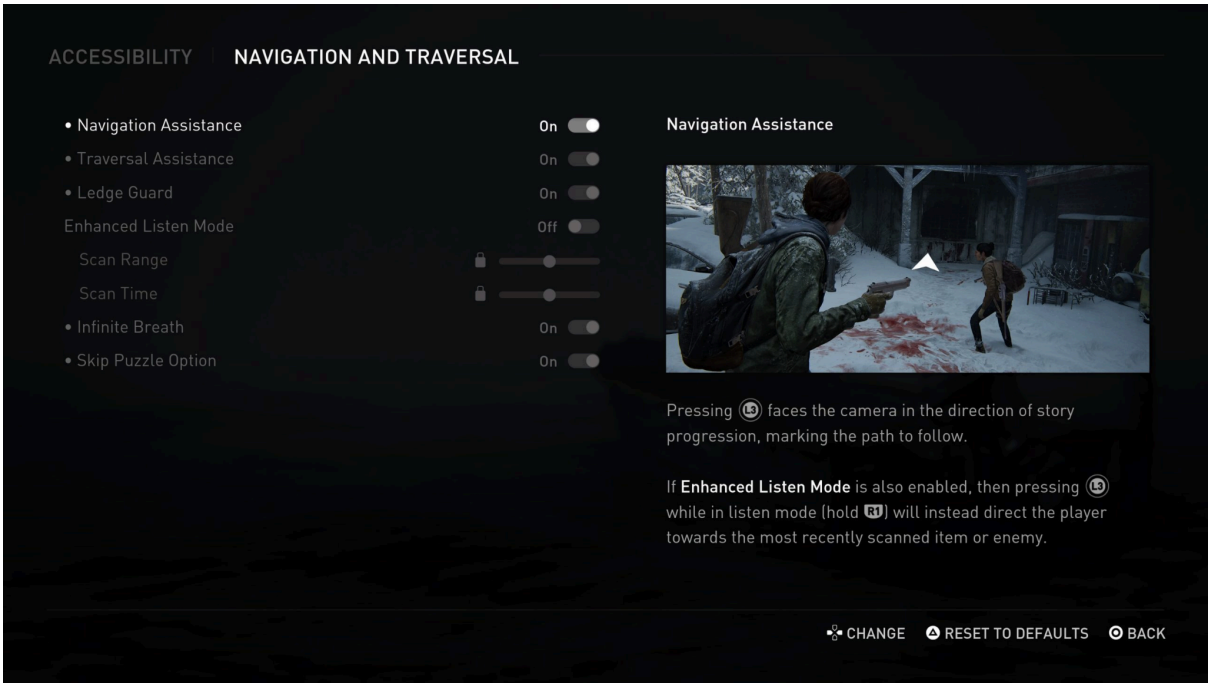


Image A2: More last of us two game options (Bill, 2022)

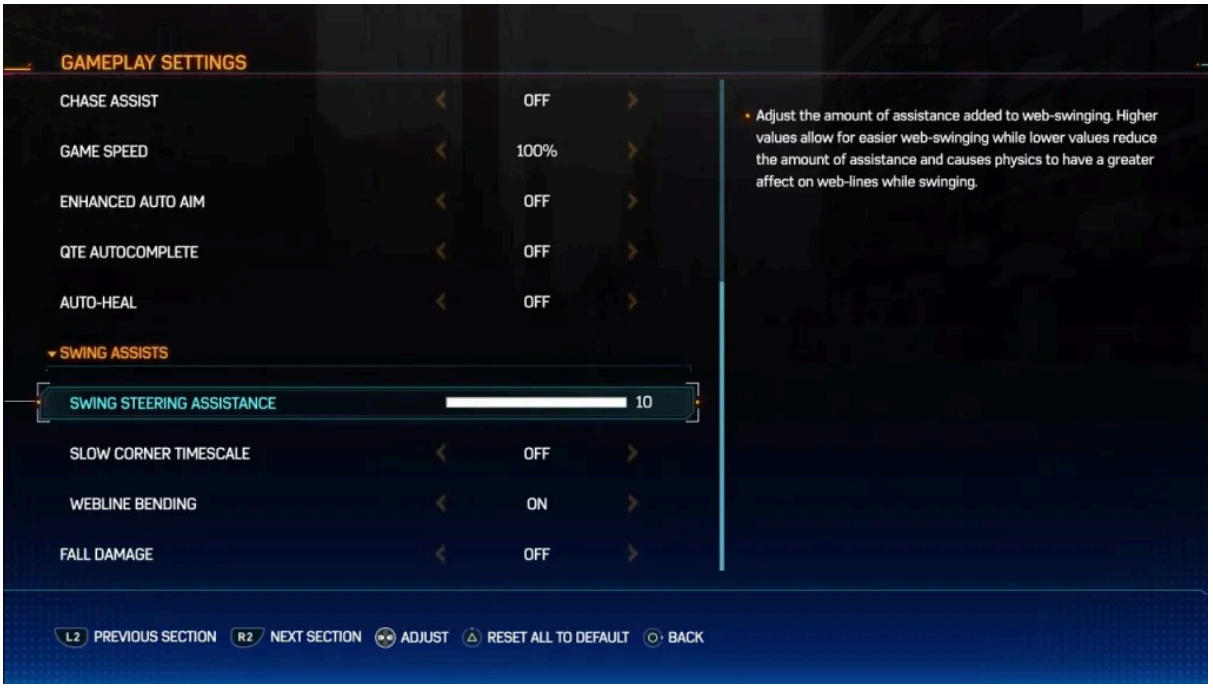


Image A3: Spider-Man 2 Web assist slider and fall damage settings (Cichacki, 2023)

Image A4: Pathfinder: Kingmaker difficulty settings (Mastazajeb, 2018)

Appendix B

Minecraft map generation:

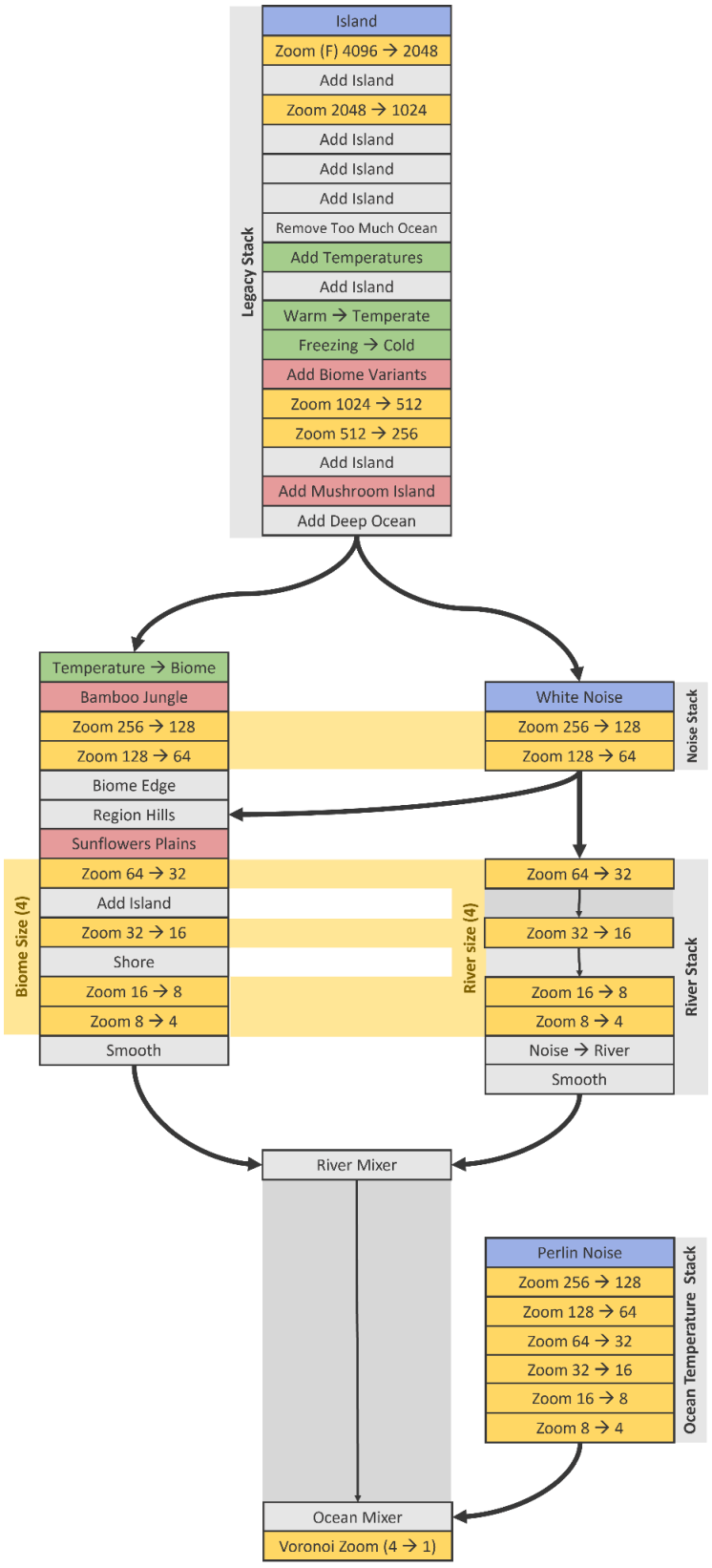


Image B1: Minecraft map generation flow (Zucconi, 2023)

Appendix C

Questionnaire results:

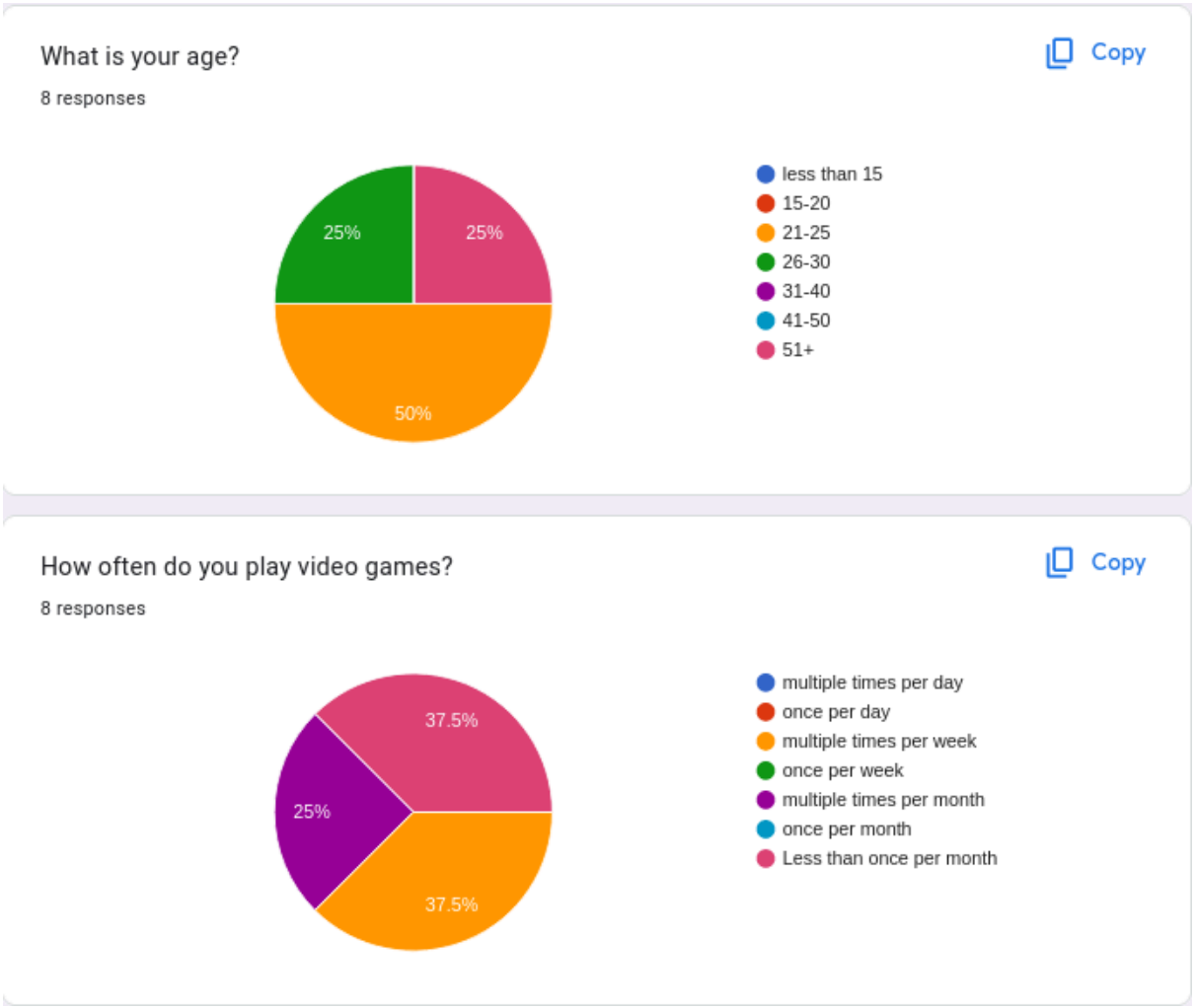


Image C1: Age and video game habit of testers

Image C2: Focus and entertainment scores of game mode 1

How do you think the algorithm of game mode 1 selects the waves (object type [asteroid, enemy, collectable] and number)?

8 responses

- In order
- Not too sure, but I don't think so.
- i guess its preset? maybe i dont understand a question
- Random
- It feels either scripted (just not very engagingely) or random
- No idea
- adjacent items
- All OK

Any additional comments on game mode 1 (experiences, enjoyment, thoughts)?

6 responses

- I liked how fast the astereoids were were going, challenging but not impossible. The spaceships were a bit easy and the humans were challenging too.
- Meh XXD
- Sometimes there were too many astronauts and it didn't seem possible to save them all.
- Pretty boring, also severely missing music
- fun
- Difficult to shoot enemy more than once as they come down fast

Image C3: How the algorithm works and additional comments for game mode 1

Image C4: Focus and entertainment scores of game mode 2

How do you think the algorithm of game mode 2 selects the waves (object type [asteroid, enemy, collectable] and number)?

8 responses

- it adapted to how i played and got easier as i made mistakes
- Dynamically
- randomly, but sometimes it seems to base it on the position of the player
- Predefined to get harder each round
- Either preprogrammed or randomly selecting a number of game states. You have talked to me about the vague idea behind your project so that might influence some of these answers
- No idea
- randomly
- Better I think, need to play some more

Any additional comments on game mode 2 (experiences, enjoyment, thoughts)?

5 responses

- a bit less boring then the first mode
- The game seemed a bit more fun this time, however, the rounds felt like they lasted a little too long compared to the last round
- Still quite boring. Missing sound effects, although the pixel art is cute. It is clearly a very basic game tho. Something you'd find on Ludum Dare, maybe
- had to be more alert
- Good fun, takes me back 30 years to Space Invaders

Image C5: How the algorithm works and additional comments for game mode 2

Image C6: Focus and entertainment scores of game mode 3

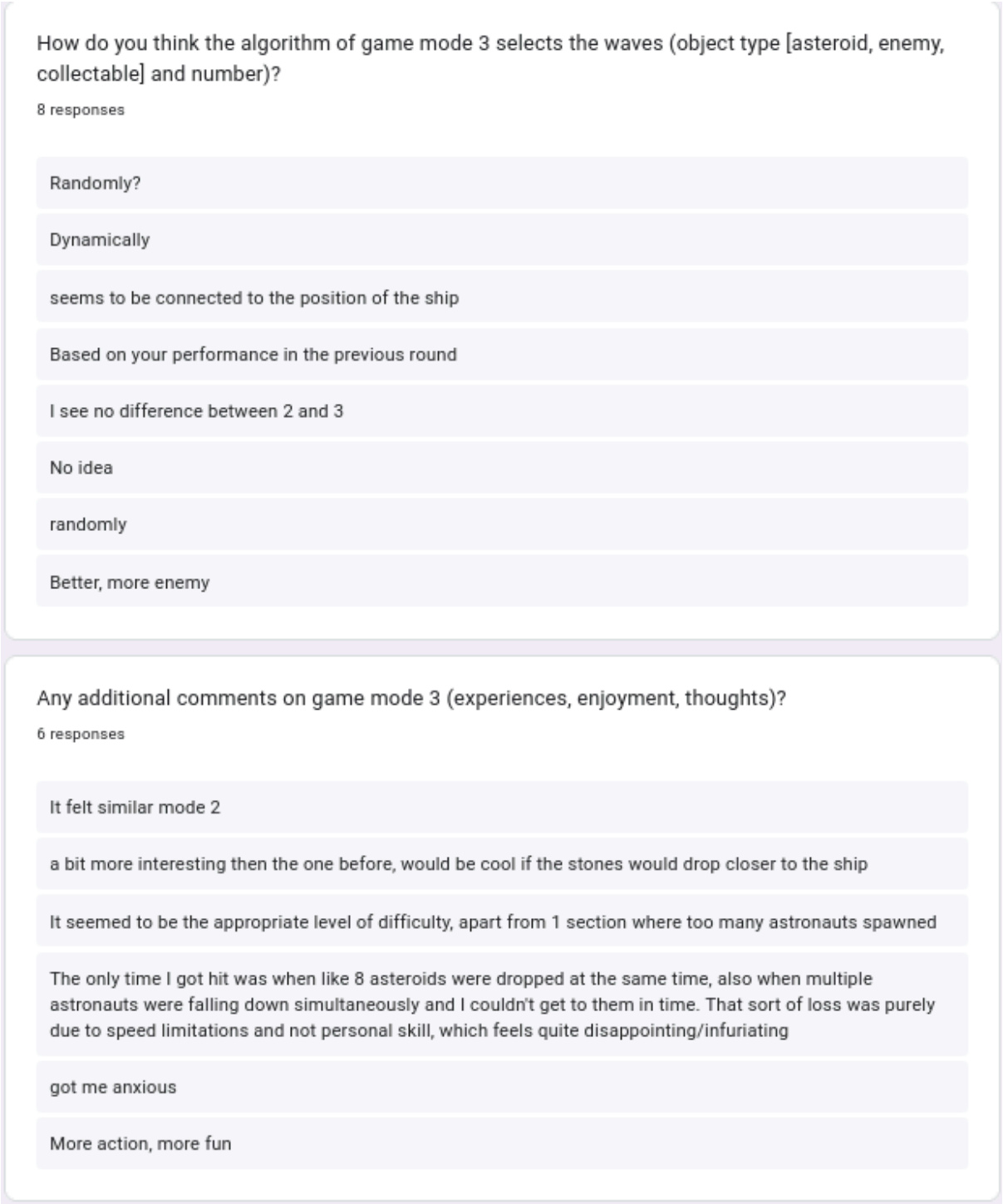


Image C7: How the algorithm works and additional comments for game mode 3

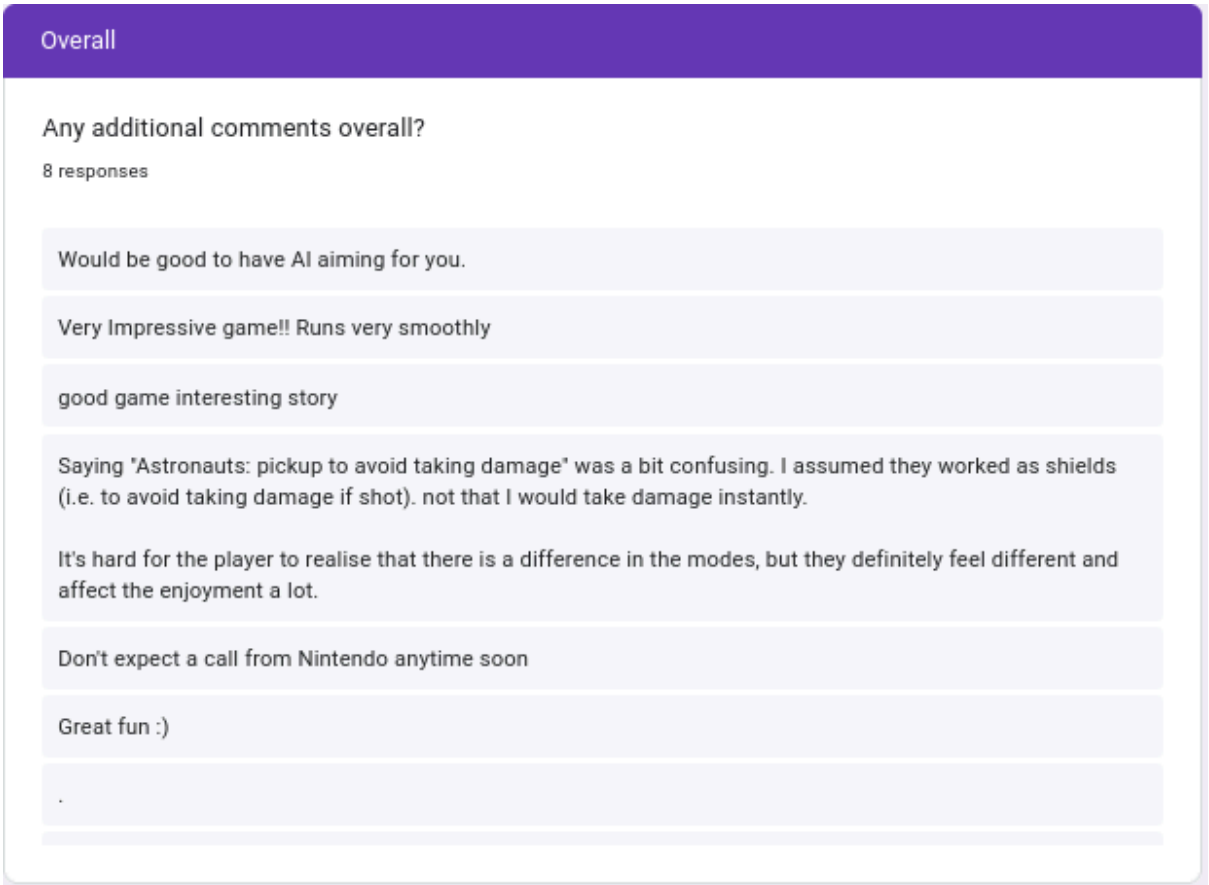


Image C8: Any additional comments overall

Appendix D

Player abilities per wave and game type:

Blue: enemy ability

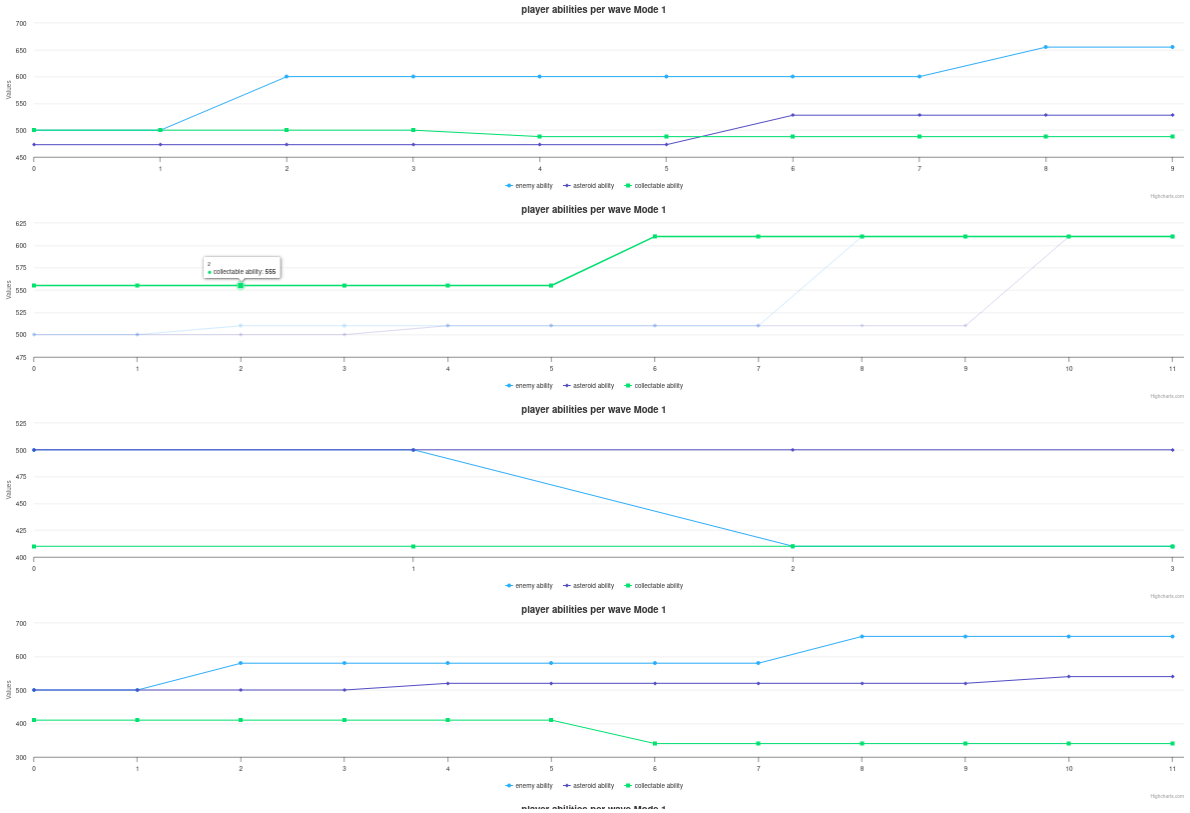
Purple: asteroid ability

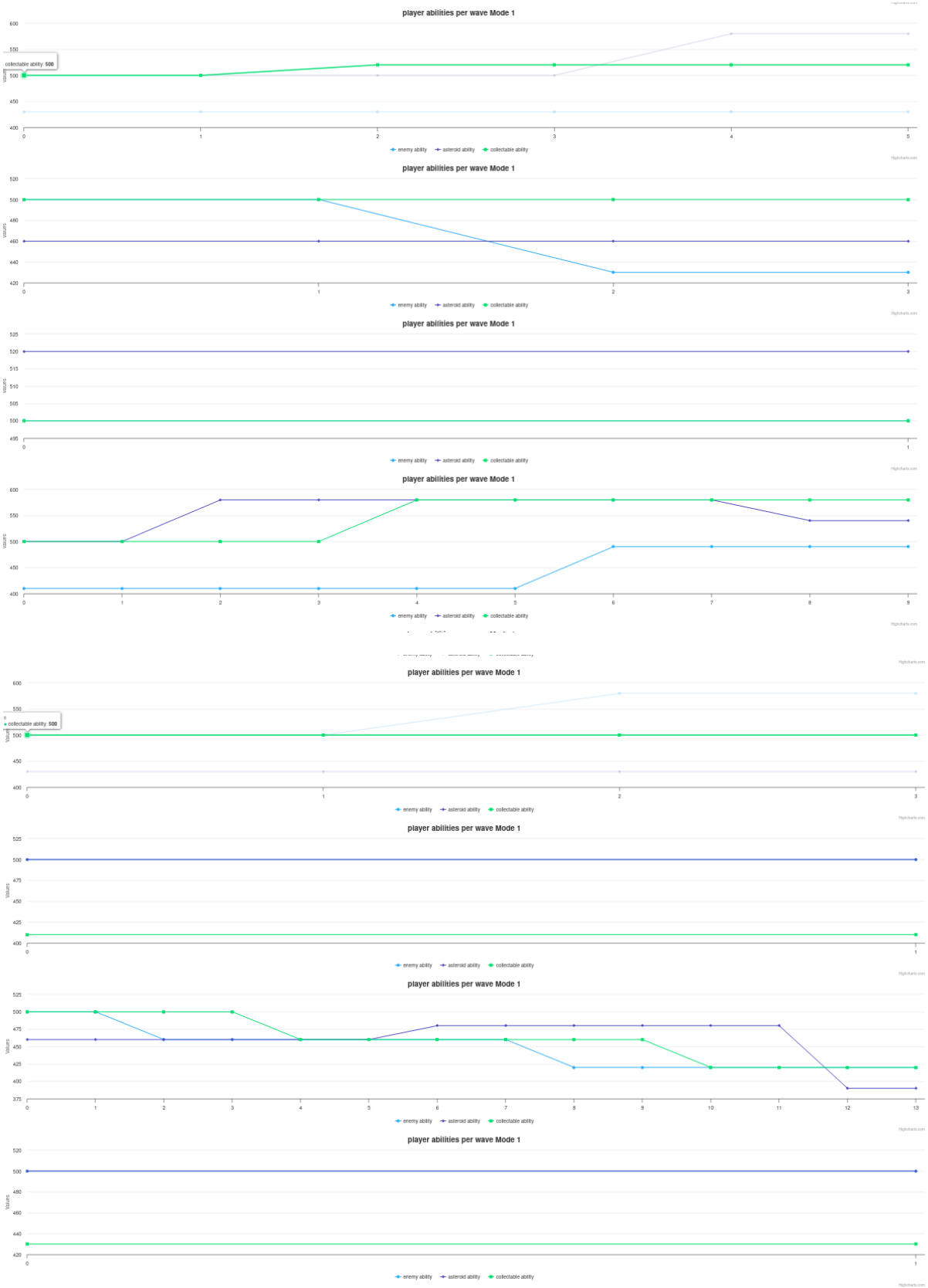
Green: collectable ability

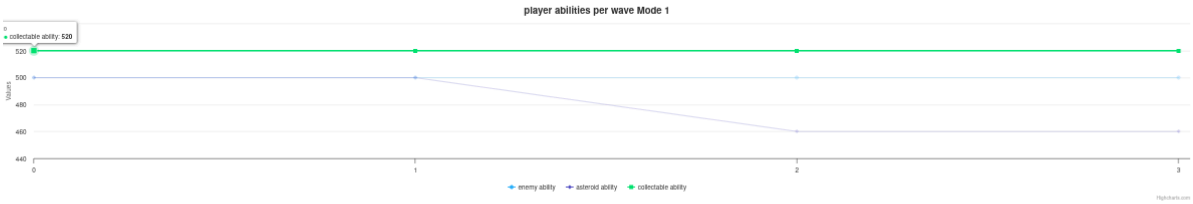
Mode 1: ordered switch

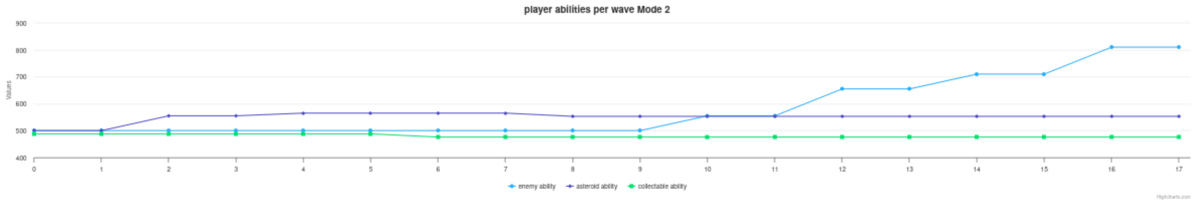
Mode 2: main

Mode 3: random switch

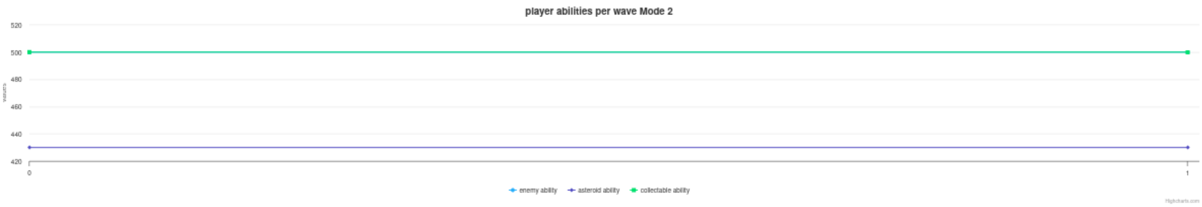


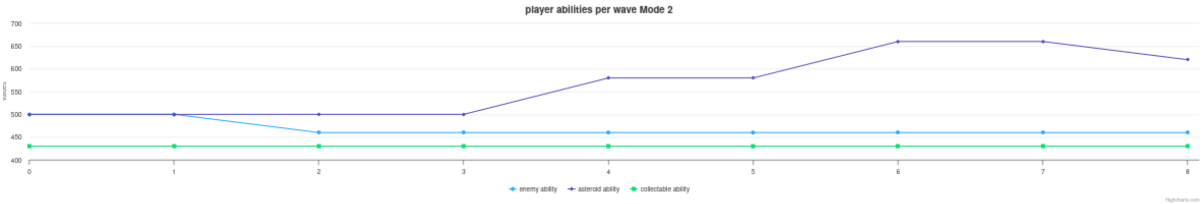

player abilities per wave Mode 1
collectable ability: 500
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
collectable ability: 500
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability
player abilities per wave Mode 1
enemy ability
asteroid ability
collectable ability

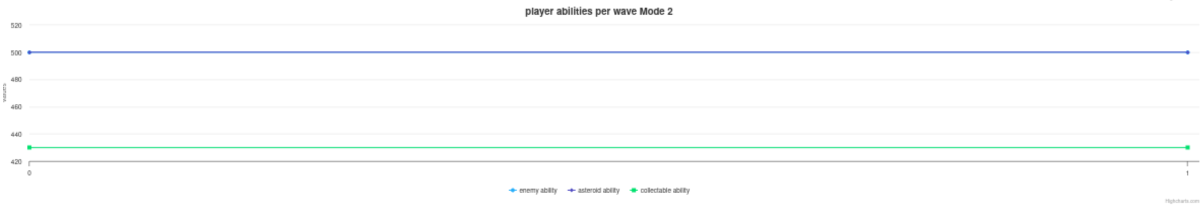




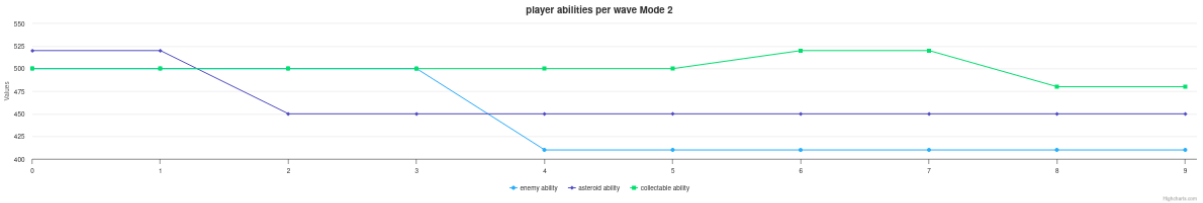
player abilities per wave Mode 2



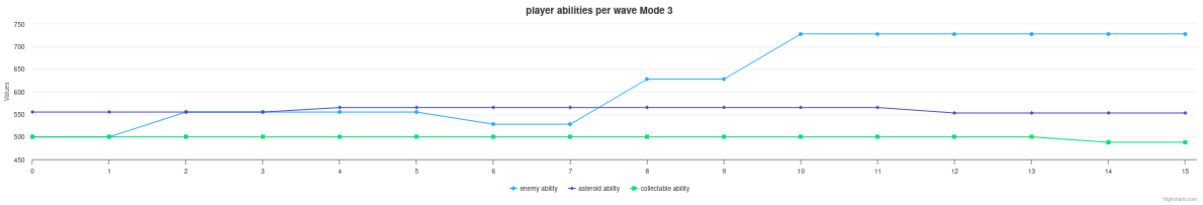


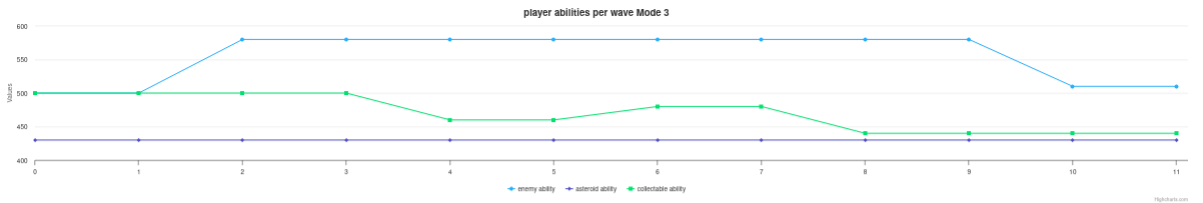


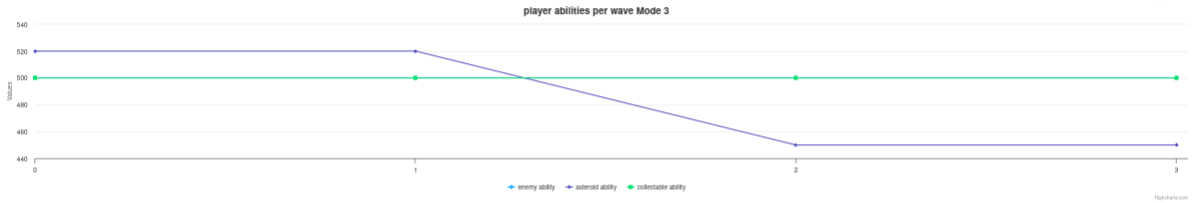

player abilities per wave Mode 2
asteroid ability: 500
Values
enemy ability
asteroid ability
collectable ability


player abilities per wave Mode 2
Values
enemy ability
asteroid ability
collectable ability

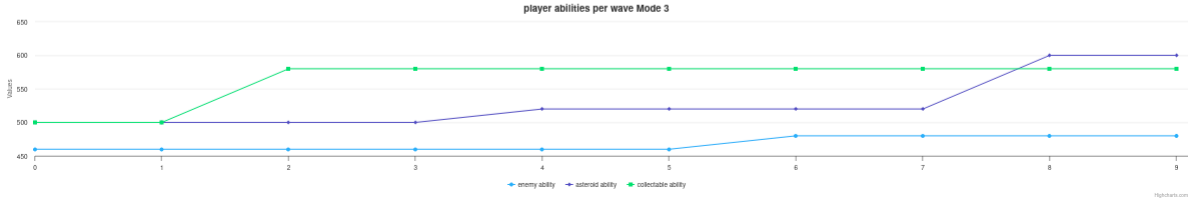

player abilities per wave Mode 2
Values
enemy ability
asteroid ability
collectable ability

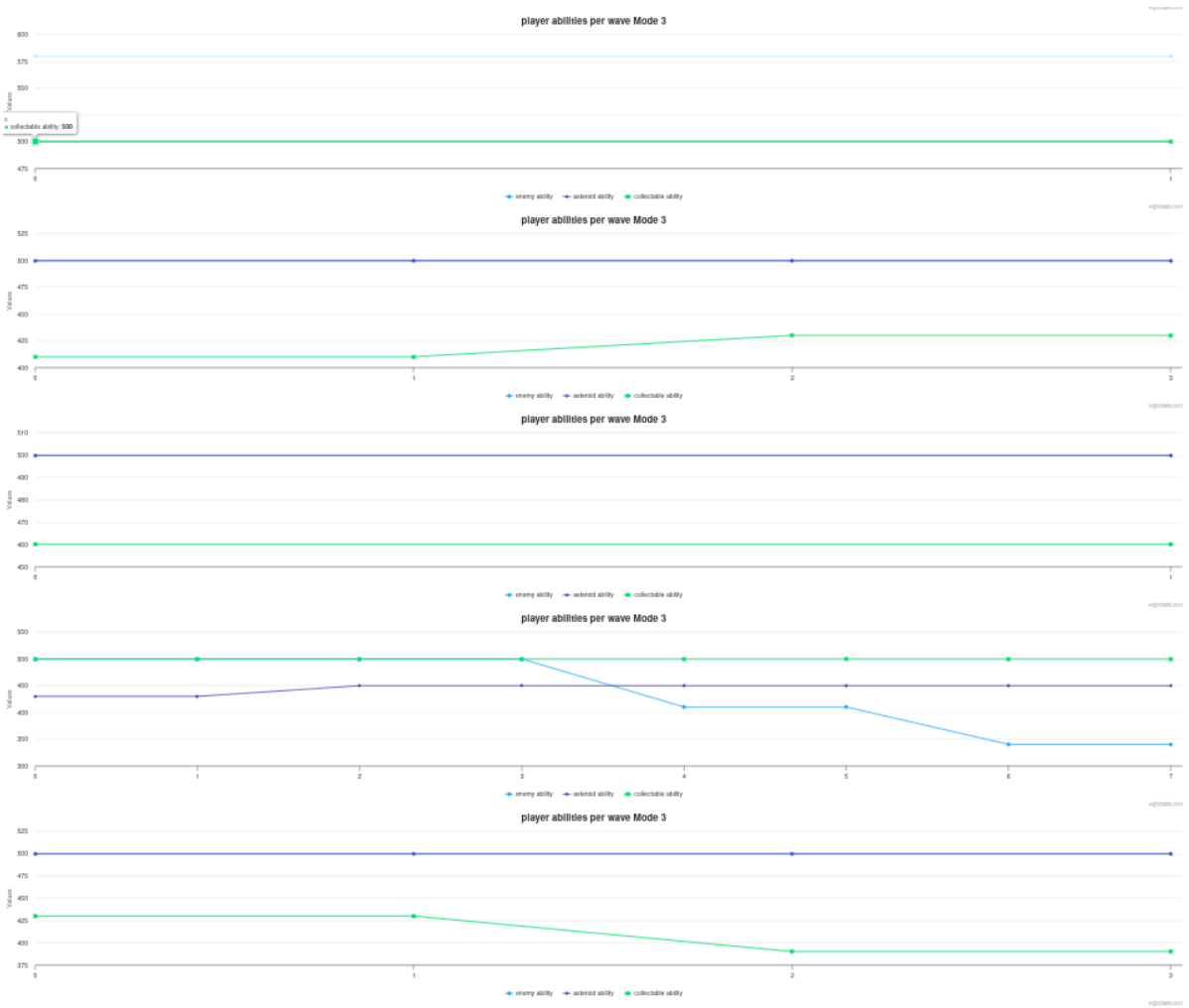

player abilities per wave Mode 3
Values
enemy ability
asteroid ability
collectable ability


player abilities per wave Mode 3
Values
enemy ability
asteroid ability
collectable ability


player abilities per wave Mode 3
Values
enemy ability
asteroid ability
collectable ability


player abilities per wave Mode 3
asteroid ability: 500
Values
enemy ability
asteroid ability
collectable ability


player abilities per wave Mode 3
Values
enemy ability
asteroid ability
collectable ability



Images D1-D6: Testers' ability per wave per game mode

Appendix E
Links

Gitlab (Code):

https://gitlab.com/Kaveh_N_Nejad/uni-honours-cm4105

Game Example (Youtube):

https://youtu.be/z6pE6eM7EoA

Game:

kaveh-nejad.com