Video Game Dynamic Difficulty Adjustment Using a PID Controller

1 CONTENTS

2 INTRODUCTION

There are over 3 billion active video game players (gamers) worldwide, making the industry worth an estimated $197 billion (Howarth, 2023). With the dramatic surge in popularity over the last few decades, the industry continues to grow as more people are interested in playing video games today than ever before.

With the widespread reach video games have, the consumers playing them inevitably have varying skill levels and abilities, as well as differing characteristics such as reaction times, hand-eye coordination, and tolerance for stress and failure (Kowal, et al., 2018). Creating games for large audiences can be challenging due to this, as balancing the difficulty of a game for the spectrum of player skill levels is difficult to accomplish.

One common technique is to use static "easy", "medium", or "hard" difficulty archetypes of which players can choose between to set their own difficulty level manually. This is the simplest approach, but it falls short when players have unrealistic expectations of how difficult the game will be or of their own skill level (Venturelli, 2016).

This report aims to create, test, analyse, and evaluate a dynamic difficulty adjustment (DDA) system for any game, based on a variety of input factors chosen by the game developer. This game-agnostic approach allows for wide implementation and can be built upon to create an idiosyncratic game experience for all consumers.

3 Feasibility Analysis

3.1 Understanding Difficulty

Difficulty in video games can be defined as a measure of how challenging a video game is completed for any given gamer. This is a highly relative and subjective phenomenon however, so a more accurate description of video game difficulty could be how a video game communicates what it wants the player to do and how to do it relative to their own skill level (De, 2021).

Quantifying this is hindered by the inherent ambiguity and unique requirements of any given game. Every game has different components and interdependent systems that interact in a complex way to create a sense of difficulty, and often discerning crucial factors of gameplay that affect difficulty can be intuitively challenging.

A thorough and mathematical approach to identifying factors with statistically significant effects on the difficulty of the game can be done using a 2^k factorial design. Containing k factors, each with 2 values or levels, a full 2^k factorial analysis calculates the effects and significance of each factor and for all possible combinations of the set of factors for each response variable. This information can then provide a vector of effect on a particular response variable (Fraser, et al., 2013).

For demonstration purposes, this paper will consider a simple tower defence game in which the player is a defender of a specific point on a map. The goal is to strategically place towers along the enemy's path to prevent them from reaching their end goal. Towers and enemies can have different abilities and statistics, and as the game continues the enemies will become increasingly difficult to defeat as they come in larger waves with higher statistics such as movement speed or health.

Such a game will allow many different quantifiable metrics for what affects the difficulty for each round of the game. The chosen factors for each round of the game include:

- Number of enemies
- Enemy speed
- Enemy health
- Number of defences
- Defences range
- Defences firing rate
- Player resources/currency
- Level layout
- Player upgrades

Simulations or player testing can be done to identify prominent factors from a matrix of all possible combinations of factors at a high and low level, where a given factor is at an extreme level of influence. For example, high enemy movement speed means enemies move at their maximum speed, whereas low means they move at their minimum speed (Fraser, et al., 2013).

3.2 PID Algorithms

Proportional-integral-derivative algorithms, or PID controllers, are a well-established way of driving a system towards a target position or level using a closed feedback loop mechanism that monitors and adjusts a system's properties such that on the next iteration of the algorithm, the system is closer to its desired set point. The closed nature of the loop ensures that only the desired properties are monitored and affected by the algorithm (Goodwin, et al., 2000).

Usually this is done by monitoring process variables like temperature, pressure, or flow rate, however the versatility of PID controllers make them applicable in a wide range of problems, including video game DDA systems.

A paper by Procedia Technology looks at the use of a simple PID controller in a multiplayer tank game where players try and shoot each other in an arena. The PID uses the difference between how many shots both players have received, and the time elapsed since the last successful shot to determine who is taking advantage. The PID controller uses these and outputs a damage and speed variable, increasing the losing player's statistics to aid them (Álvarez Pato & Delgado-Mata, 2013).

The study found that the PID controller successfully narrowed the skill disparity between the players, creating a more equal matchup. However, this was not enjoyed by the players in a multiplayer setting, as those with lower skill found the advantage made them more interested in the game, but also more easily bored. Conversely, those with higher skill got frustrated and became less interested in the game (Álvarez Pato & Delgado-Mata, 2013).

This shows that in a multiplayer setting, the use of a PID controller for balancing may make the game less enjoyable overall as players end up feeling their true skill is not being reflected within the game. DDA systems should aim to affect the game the minimum possible amount to aid or challenge the player such that they are engaged with the game.

3.3 Survey Analysis

3.3.1 Survey Method

To better understand the experience of difficulty in video games, a survey was taken across a range of participants from potentially many different countries. Shown in Appendix A - Survey Questions, the survey focuses on key aspects of difficulty, including player emotions, engagement, and preferences relating to game difficulty systems.

62 participants responded to the survey, providing valuable data for different demographics on what they expect video game difficulty to be and how they are personally affected by difficulty. Taking the population of video game players with a 90% confidence level, this turnout yields a ±10% confidence interval for the responses (Glen, n.d.).

The participants were a mix of people in the target audience for video games, as well as from outside the target audience to gain a wider perspective on the general population. Using a direct private link to a Google Form, social media was used to share the link with known trustworthy participants, greatly reducing the chances of falsified or computer-generated responses.

The survey aimed to establish the following hypotheses:

- People who have heard about DDA tend to value difficulty levels at a higher importance.
- People who prefer to be challenged in video games tend to value difficulty levels at a higher importance.
- People who like to customise the difficulty in video games don't like DDA systems.

3.3.2 Survey Results

The results of the survey show that 62.9% of players believe the difficulty of a game is important to the experience of playing it. When a game is too easy, although a significant minority do not, most players get bored. When a game is too difficult, 54.8% of players feel frustrated, 48.4% feel discouraged, but 38.7% feel motivated to improve.

Performing an analysis of variance[1] (ANOVA) tests whether certain subgroups of the results have a statistically significant difference in means, showing non-random trends within the sample (Onofrey, 2020).

Table 1 The values in the output of the ANOVA tests and their descriptions.

Name	Description
Degrees of Freedom	Number of groups being compared (Df).
Sum of Squares	Sum of the squared distances between the observed values.
Mean Square	$\frac{Sum\ Sq}{Df}$
F value	Ratio of variance explained by the model to the residuals.
P value	Probability of obtaining the observed F value or a more extreme value if the null hypothesis is true.
Dependant	The variable being predicted or explained by the independent variable.
Residuals	The difference between the observed values of the dependant variable and the predicted values based on the independent model.

Table 2 shows the results of the ANOVA done between the familiarity of DDA and the importance of difficulty on the experience of playing a game. The p-value is larger than 0.05, indicating no statistical evidence that those who are familiar with DDA systems place higher importance on difficulty being a contributing factor to a game's experience. Thus, the null hypothesis can be accepted.

Table 2 Results of an ANOVA on the familiarity of DDA and the importance of difficulty.

	Df	Sum Sq	Mean Sq	F value	P value
Dependent	3	3.57	1.189	1.183	0.324
Residuals	58	58.3	1.005		

Table 3 shows that there is marginal significance to suggest that people who prefer to be challenged in video games tend to value difficulty levels at a higher importance. This could suggest that the question: "How important is the difficulty of a game on the experience of playing it?" may have been too vague, or that the Likert scale used in the question was too subjective and did not meet the required assumption that each category is equidistant in value.

Table 3 Results of an ANOVA on importance of difficulty and importance of being challenged.

	Df	Sum Sq	Mean Sq	F value	P value
Depend.	1	3.99	3.989	3.566	0.064
Residuals	60	67.11	1.118		

Table 4 shows that there is no statistical evidence to support the hypothesis that people who enjoy customising their difficulty prefer doing so manually. This indicates that if a game can find the right difficulty for a player, it doesn't matter whether this is done manually or automatically.

Table 4 Results of an ANOVA on the importance of difficulty customisability and the preference between manually choosing difficulty or automatic difficulty adjustment.

	Df	Sum Sq	Mean Sq	F value	P value
Depend.	1	2.65	2.648	1.878	0.176
Residuals	60	84.59	1.41		

3.4 Survey Conclusion

The survey results and analysis have shown a clear direction for the project to aim towards. Difficulty is a key aspect in video games and generally gamers do not mind how difficulty is approached if the result is a fun and balanced game. The results of the survey will be considered during all aspects of design and implementation.

[1] Performed using R, https://www.R-project.org

4 Technical Analysis

4.1 End User Analysis

The target audience for a tower defence game is primarily casual gamers of all demographics who want to maximise enjoyment from the game (Pauls, 2021).

4.2 User Experience

When a difficulty level is established correctly, psychologist Mihaly Csikszentmihalyi theorises that players will find themselves in a cognitive flow state, where the challenge is not too easy nor too difficult such that the player becomes hyper-focused and engaged on the game (Thirslund, 2020). This is colloquially known as "being in the zone", and can be achieved when gameplay follows three requirements:

1) The activity must be challenging to the player and require skill.
2) The activity must provide clear goals and timely feedback.
3) The outcome must be uncertain but influenced by the player's action or inaction.

Figure 1 shows the area in which a flow state can be achieved, communicating that there is margin for fluctuation of difficulty and that the progression of a game does not have to be linear whilst still residing in the flow state area (Baron, 2012).

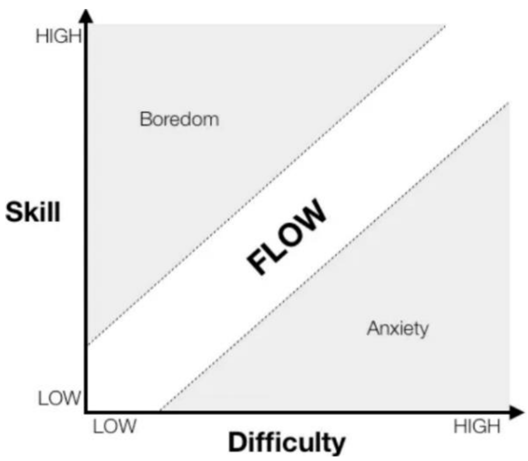


Figure 1 Flow state diagram showing the area of which flow occurs.

The design goal of a DDA system should be to modify the difficulty of the game based on the player's performance such that the difficulty is always inside the flow state region. This will allow for the highest engagement and enjoyment of the game.

4.3 Requirements

The dynamic difficulty adjustment system must input a quantifiable metric that measures the performance of the player, as decided by the game developer. Using a second-order controller from the PID controller family, the system should calculate what level of difficulty the game should be based on previous player performance. The controller should output gameplay adjustment values for the game logic to use.

The system should be tuned to minimise overshooting of the PID controller, as well as remaining discreet from the player so that the changes are not too volatile as to be obviously affecting the gameplay. Most importantly, the system should be adjusting the difficulty to maximise enjoyment of the game, attempting to keep the player in a flow state and subsequently improve player retention.

4.4 System Architecture

The DDA system will be built using Python to demonstrate the algorithm. For deployment in a video game, it is recommended the system is rewritten in a faster programming language such as C# or C++ for maximum performance. However, the system will be designed prioritising accuracy and performance, as well as modularity so the DDA system can be widely used with minimal impact to performance, memory usage, and compatibility with the game's technology stack.

The algorithm should interface with the game's logic using functions with parameters passed each iteration of the game loop.

5 Design Specification

5.1 System Overview

The DDA system consists of the following components:

- GameMetricController: the core component that implements the PID controller algorithm for adjusting gameplay aspects.
- Performance metric(s): functions that measure player performance quantifiably.
- Modifiers: functions that adjust gameplay aspects based on the output of the PID controller.
- Game engine: integrates with GameMetricController to apply gameplay modifications.

5.2 Process Variables

The process variables are what are passed to the DDA system for the PID controller to use when calculating the control variable (the output). This is passed as a function to the PID controller to be called on each evaluation of the controller.

In the case of the tower defence game, this will be the following:

- Enemies per tower describes the relationship between the number of enemies in the round and the number of towers placed.

5.3 Dynamic Programming

By incorporating dynamic programming techniques, the efficiency of the algorithm can be improved by reducing computational overheard. Caching and reusing previous results can significantly increase execution performance when the function is being called frequently.

Memoising the proportional, integral, and derivative terms of the controller prevents redundant recomputing for previously calculated error values.

This will increase the space complexity of the algorithm, so when error values are unlikely to repeat often, the exchange of performance for memory requirements may not be advantageous. This can be assessed on individual implementations of the controller.

5.4 PID Controller

This is represented as a class named “GameMetricController”, encapsulating the functionality of the PID controller algorithm. This can be reused multiple times in a game to provide DDA support for different aspects of gameplay. The controller can adapt based on the performance metrics and modifiers provided and can be tuned by the game developer within the class constructor with the gain values.

Table 5 Property descriptions for GameMetricController.

Property	Description
metric	Callable function that provides the process variable.
modifiers	List of functions that will be called on controller evaluation to adjust game mechanics. Includes weights to affect modifiers individually based on control variable.
Kp (proportional_gain)	Tracks the proportional gain.
Ki (integral_gain)	Tracks the integral gain.
Kd (derivative_gain)	Tracks the derivative gain.
setpoint	The desired difficulty level.
last_error	Stores the previous error for the derivative calculation.
integral_sum	Accumulates the sum of errors for integral calculations.

last_output	Tracks the latest control variable value.

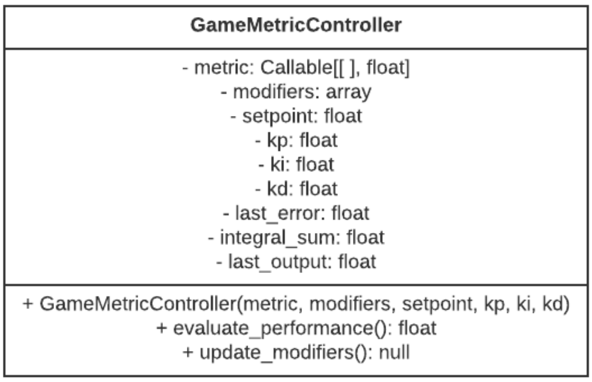


Figure 2 UML class diagram for GameMetricController.

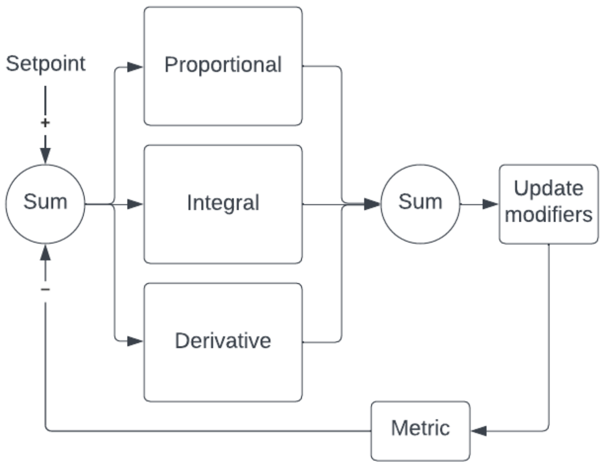


Figure 3 Block diagram of the DDA PID controller.

5.4.1 Data Structures

The metric property makes use of a callable function to allow flexibility in defining the performance assessment from within the game logic. This function takes no arguments and returns a float value.

The modifiers property is a 2-dimensional array representing each gameplay modifier as a pair $[update\ function, weight]$. It provides a simple way to handle any number of modifiers as well as their importance in relation to the control variable.

The remaining properties are used for numerical calculations and so are represented as floating point numbers for precision involving decimal values.

5.4.2 Methods

GameMetricController is the class constructor, assigning the class properties from the arguments. Internally in Python this is represented as an __init__() function.

The function evaluate_performance handles the PID controller calculations, which can be described as:

$$e = SP - M$$

$$P = K_p \times e$$

$$I = \sum_{t=0}^{n} K_{i_t} \times e_t$$

$$D = K_d \times (e - LE)$$

$$C = P + I + D$$

Where:

- e: error value.
- SP: setpoint value.
- M: performance metric value.
- P: proportional component.
- K_p: proportional gain value.
- I: integral component.
- K_i: integral gain value.
- t: index variable for summation.
- n: current iteration.
- D: derivative component.
- K_d: derivative gain value.
- LE: last (previous) error value.
- C: control variable.

The function returns the control variable as the output. The output is heavily dependent on the proportional, integral, and derivative gain values, which need to be manually set for each implementation of the PID controller. Each gain value will affect the behaviour of the controller differently:

Table 6 Gain values for a PID controller and their behavioural effects.

Gain	Effect
Proportional	How aggressively the controller responds to change in the error. Increasing will

	cause the controller to respond faster to change but may cause overshoot or oscillation.
Integral	How quickly the controller eliminates steady-state error, or how well the controller can track a desired setpoint. Increasing will again eliminate steady-state error, but may cause overshoot or oscillation.
Derivative	How quickly the controller responds to change in the rate of change of the error. Increasing this will cause the controller to respond more quickly, and thus reduce overshoot and improve stability.

(Cheung, 2020)

The function update_modifiers updates the gameplay modifiers' values based on the control variable multiplied by the modifier weight. This allows the game logic to make tangible use of the PID controller output.

5.5 Control Variable

The control variable will be a positive or negative decimal number indicating whether the process variables (i.e., the modifiers) need to be increased or decreased to get closer towards the setpoint. This usually manifests as when the player is performing well, the control variable will be a larger positive number, and when the player is performing poorly the control will be a smaller negative number.

Translating this to the game logic can then be done simply with the modifiers; for example, with the enemies per tower metric, the control variable can increase the number of enemies in the next round when the player is doing well, and decrease the number of enemies when the player is performing poorly:

```
def generate_enemies():
   enemies = [Enemy(
      id,
      get_round_enemy_health(),
      get_round_enemy_speed()
                           )
      for id in range(
         get_round_enemies()
      )]

def get_round_enemies():
   return min_enemies + round(.7 *
(game_round - 1)) +
enemies_modifier[-1]
```

6 Development and Testing

6.1 Unit Testing

Table 7 Unit tests for the performance evaluation function.

Input Value	Expected Outcome	Actual Outcome
Setpoint: 10 **Metric: 10**	No error	0
Setpoint: 15 **Metric: 10**	Positive error	6
Setpoint: 5 **Metric: 10**	Negative error	-6
Setpoint: 15 **Metric: 10** **Kp: 1**	Larger than default proportional gain value.	10.5
Setpoint: 15 **Metric: 10** **Ki: 1**	Larger than default integral gain value.	10.5
Setpoint: 15 **Metric: 10** **Kp, Ki: 1**	Larger than previous two gain tests.	15
Setpoint: 15 **Metric: 10** *After caching*	Identical output to non-cached output	6
Setpoint: sys.maxsize **Metric: -sys.maxsize**	Value based on the extreme inputs.	2.21×10^{19}
Setpoint: 15 **Metric: 10** **Kp, Ki, Kd: sys.maxsize**	Value based on the extreme gain values.	inf

The unit testing produced expected results, showing that the algorithm is working as intended. However, the edge case with extreme gain values does cause Python to consider the result as infinity, which is not a useful output for the system. A minimum and maximum validation conditional statement could help stop erroneously large or small control variables from affecting the gameplay here.

6.2 Algorithm Scaling and Efficiency

Performance and efficiency of the DDA system are crucial for its use within video games. If the system has a noticeable effect on the game's performance, it would be unsuitable for wide adoption.

Function performance can be measured using Python's timeit library [2]. This can allow comparison between theoretical and practical time complexity of individual functions or sections of code.

6.2.1 Theoretical Analysis

The functions being tested are:

- Metric calculations
- Performance evaluation
- Updating modifiers

Metric calculation returns a floating-point value, from the calculation of various aspects of gameplay. For example, when calculating the enemies per tower metric, the code is as follows:

```
def get_enemies_per_tower():
    return get_round_enemies() /
len(towers)
```

The Python "len" operation is $O(1)$ due to list objects maintaining a counter of how many objects are within the list (Mayer, n.d.). This may be different in other programming languages, so for other implementations this should be kept in mind for when the player has an extreme number of towers.

The code for get_round_enemies is:

```
def get_round_enemies():
    return min_enemies + round(.7 *
(game_round - 1)) +
enemies_modifier[0]
```

This is also a constant time of $O(1)$ due to the basic arithmetic options.

This will be true for most metric calculations, with few dynamic memory allocations making constant space and time complexity. This should have a negligible effect on the game's performance.

Performance evaluation is where the core PID controller calculations are completed.

```
def evaluate_performance():
    error = setpoint - metric()

    # Proportional term
    proportional = kp * error

    # Integral term
    integral_sum += ki * error

    # Derivative term
    derivative = kd * (error -
last_error)
    last_error = error

    last_output = proportional +
integral_sum + derivative
        return last_output
```

The function performs basic arithmetic calculations with a few local variables, and so would have a constant space and time complexity if the metric calculation function also has a constant complexity.

Lastly, updating the modifiers calls the relevant modifier update functions which assigns the variables:

```
def update_modifiers():
    for modifier in modifiers:
    # modifier = [update_function,
weight]
```

[2] https://docs.python.org/3/library/timeit.html

```
    modifier[0](last_output *
modifier[1])
```

Assuming the modifier update function has constant time (due to the function simply assigning a value), updating all the modifiers has $O(n)$ time complexity and $O(1)$ space complexity due to the list iteration.

As the algorithm scales, it is unlikely to have any effect on the performance of the game.

6.2.2 Performance Validation

Ap. D – Performance Testing Results Figure 7 shows the time taken to perform 10,000 runs of the performance evaluation function, repeated 250 times. On average, this operation took 12.5 milliseconds to complete. Variation can be seen within the results, which can be assumed to be the result of differing system load or garbage collection behaviour.

Ap. D – Performance Testing Results Figure 8 shows the results of the same test, except with a cached version of the function using a Python dictionary to store the output of previously calculated error values:

```
def evaluate_performance():
   error = setpoint - metric()

   if error in evaluation_cache:
      return evaluation_cache[error]

   # Proportional term
   proportional = kp * error

   # Integral term
   integral_sum += ki * error

   # Derivative term
   derivative = kd * (error -
last_error)
   last_error = error

   last_output = proportional +
integral_sum + derivative
   evaluation_cache[error] =
last_output
   return last_output
```

This operation took an average of 6.2 milliseconds to complete, showing an improvement of 50.4%. The cache prevents redundant calculations from being performed, and due to the nature of the output either being positive or negative aiming towards zero, repeats of error values occur frequently meaning the program benefits from the cache significantly.

Overall, both versions of the function would have negligible effect on the performance of the game, as it is unlikely the function would be run more than 60 or 144 times per second (the most typical frame rates of video games).

Ap. D – Performance Testing Results Figure 9 shows the same test on the modifiers update function for a single modifier. This function took an average of 3.9 milliseconds, again having negligible effect on the game performance.

Ap. D – Performance Testing Results Figure 10 shows that with 1000 modifiers, the function takes an average of 2.25 seconds, giving 0.2 milliseconds per single function call. Whilst confirming the $O(n)$ time complexity, this result is still suitable for a real-time game environment due to the function only needing to be called once per game loop, even in extreme cases such as having 1000 modifiers.

7 Analysis and Evaluation

The PID controller implementation for the DDA system is effective and performant. By comparing different implementations for with and without caching, it becomes evident that caching significantly improves performance. Caching allows the reuse of computed results, reducing redundant calculations and enhancing efficiency. However, even without caching, the algorithm remains usable within any video game without a noticeable performance penalty. The simplicity of the design contributes to its efficiency, as it avoids the use of cumbersome data structures that could potentially consume unnecessary memory.

Furthermore, the algorithm demonstrates excellent scalability, accommodating any practical use case even with extreme inputs. GameMetricController can be implemented into a game as many times as needed without sacrificing game performance.

Correct tuning of the gain values in the PID controller is crucial for achieving optimal control variables. When properly tuned, the controller provides effective feedback to the game logic, allowing a tailored experience that is not directly noticeable by the player. However, tuning a PID controller can be difficult and unintuitive, and with each implementation requiring manual unique tuning, this puts additional overhead on the development of the DDA system implementation.

8 CONCLUSION

Overall, the PID controller based dynamic difficulty adjustment system offers a reliable and performant solution to automatically changing difficulty based on player performance. It showcases excellent scalability and resource utilisation, as well as providing a tailored game experience to the user.

The PID controller algorithm, implemented within the GameMetricController class, effectively assesses the difference between current and desired player performance, giving the video game developer full control over the system and implementation details in tandem with the game logic.

The simplicity of the algorithm allows it to be highly efficient, with minimal to no impact on the performance of the gameplay even with multiple implementations or extreme inputs. By applying dynamic programming techniques, this efficiency can be further improved with comparisons and analysis of the improvements being discussed.

However, its worth noting that the manual tuning required for each implementation of the PID controller can be a challenging and unintuitive process, introducing additional developmental overhead in making sure the PID controller is behaving as intended.

For future work further research and exploration can be conducted to automate the tuning process of the PID controller, leveraging machine learning or optimisation algorithms to fine-tune the gain values and improve the system's adaptability.

In conclusion, the developed DDA system using a PID controller stands as a valuable tool for game developers seeking to create immersive and balanced gaming experiences, benefiting both players and the gaming industry.

WORD COUNT: 3650

9 BIBLIOGRAPHY

Álvarez Pato, V. & Delgado-Mata, C., 2013. Dynamic Difficulty Adjusting Strategy for a Two-player Video Game. *Procedia Technology,* Volume 7, pp. 315-321.

Baron, S., 2012. *Cognitive Flow: The Psychology of Great Game Design.* [Online] Available at: https://www.gamedeveloper.com/design/cognitive-flow-the-psychology-of-great-game-design
[Accessed 25 May 2023].

Brown, M., 2015. *What Capcom Didn't Tell You About Resident Evil 4.* [Online] Available at: https://youtu.be/zFv6KAdQ5SE
[Accessed 21 May 2023].

Cheung, P., 2020. *PID Controller.* [Online] Available at: http://www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/Lecture%2018%20-%20PID%20controller%20%28x2%29.pdf
[Accessed 11 June 2023].

De, S., 2021. *What Is Video Game Difficulty and How Does It Work?.* [Online]
Available at:
https://www.makeuseof.com/what-is-video-game-difficulty/
[Accessed 24 May 2023].

Fraser, J., Katchabaw, M. & Mercer, R., 2013. An Experimental Approach to Identifying Prominent Factors in Video Game Difficulty. *Springer International Publishing,* pp. 270-283.

Glen, S., n.d. *Confidence Level: What is it?.* [Online]
Available at:
https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/confidence-level/
[Accessed 1 June 2023].

Goodwin, G., Graebe, S. & Salgado, M., 2000. Classical PID Control. In: *Control System Design.* Valparaíso: s.n., pp. 157-158.

Howarth, J., 2023. *How Many Gamers Are There?.* [Online]
Available at:
https://explodingtopics.com/blog/number-of-gamers
[Accessed 21 May 2023].

Kowal, M., Toth, A., Exton, C. & Campbell, M., 2018. Different cognitive abilities displayed by action video gamers and non-gamers. *Computers in Human Behavior,* Volume 88, pp. 255-262.

Mayer, C., n.d. *Python List Length – What's the Runtime Complexity of len()?.* [Online]
Available at: https://blog.finxter.com/python-list-length-whats-the-runtime-complexity-of-len/
[Accessed 10 June 2023].

Onofrey, E., 2020. *ANOVA Explained by Example.* [Online]
Available at:
https://towardsdatascience.com/anova-explained-with-an-example-4a8c9699b175
[Accessed 1 June 2023].

Pauls, J., 2021. *Analyzing the Target Audience of Ironhide Game Studio's Kingdom Rush.* [Online]
Available at: https://jake-pauls.medium.com/towers-towers-everywhere-9a7931a9ed43
[Accessed 4 June 2023].

Thirslund, A., 2020. *Difficulty in Video Games - Game Design.* [Online]
Available at: https://youtu.be/bxp4G-oJATM
[Accessed 21 May 2023].

Venturelli, M., 2016. *Difficulty Levels And Why You Should Never Use Them.* [Online]
Available at:
https://www.gamedeveloper.com/design/difficulty-levels-and-why-you-should-never-use-them
[Accessed 21 May 2023].

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

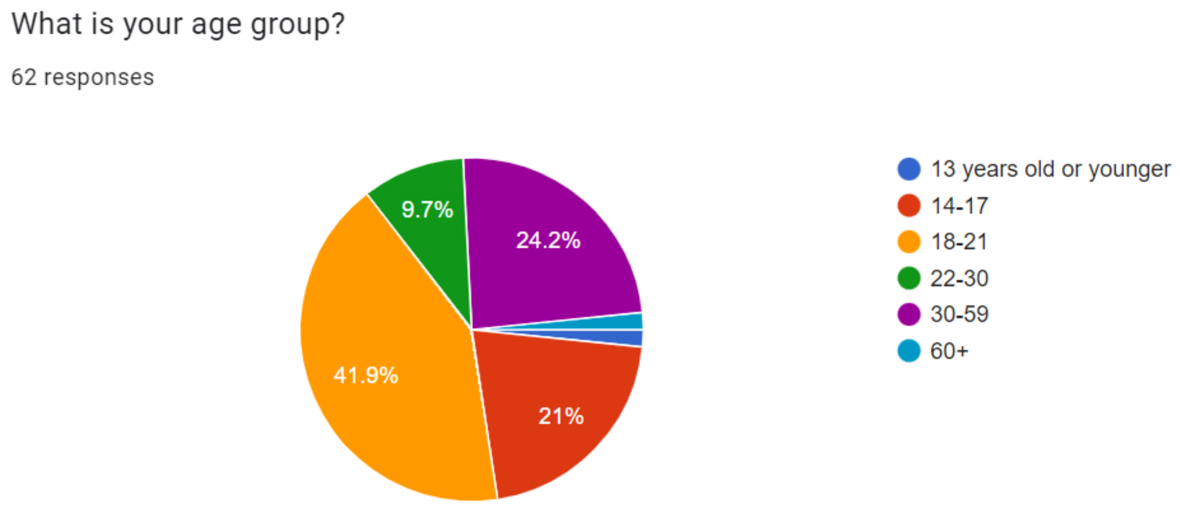
10 Appendices

10.1 Appendix A - Survey Questions

QUESTION	TYPE
What is your age group?	Multiple choice
What is your gender identity?	Multiple choice
How would identify your gaming experience?	Multiple choice

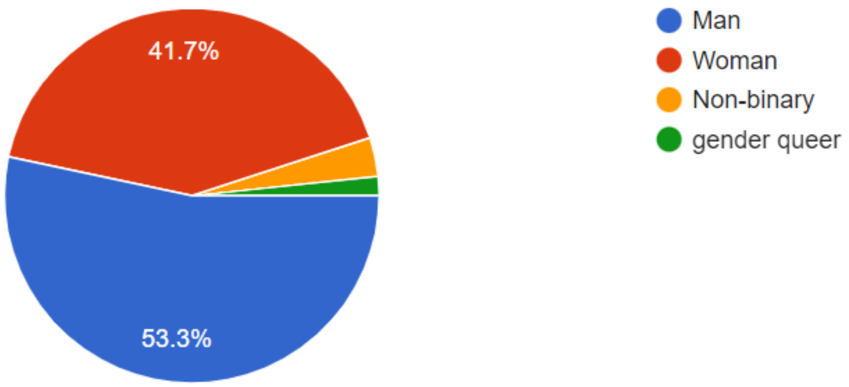
How many days a week on average do you play video games?	Range
How familiar are you with the concept of automatic (or dynamic) difficulty adjustment systems?	Multiple choice
Have you played any games with automatic difficulty adjustment systems?	Yes/No
If you answered yes to the previous question, what game(s) were they?	Text
How important is the difficulty of a game on the experience of playing it?	Range
How do you feel when a game is too difficult?	Checkbox
Do you find it boring when a (non-sandbox) video game is too easy?	Yes/No
Pick the top 3 factors you think contribute to the difficulty of video games?	Checkbox (pick 3)
How important is it for you to be able to customise the difficulty settings in video games?	Range
Would you prefer to manually choose a difficulty, or for the game to adjust its difficulty level automatically based on your performance?	Multiple choice
Do you think having the option for both manual and automatic difficulty adjustment would improve player experience with a game?	Yes/No
Have you ever stopped playing a game due to it being too easy or too difficult?	Yes/No
How important is it for you to be challenged when playing video games?	Range
Across all video games you have played, would you say video games typically have an enjoyable difficulty level?	Yes/No
Are you more likely to replay a game that has varying difficulty options or gets more difficult as you become better at the game?	Yes/No

10.2 Ap. B – Survey Results



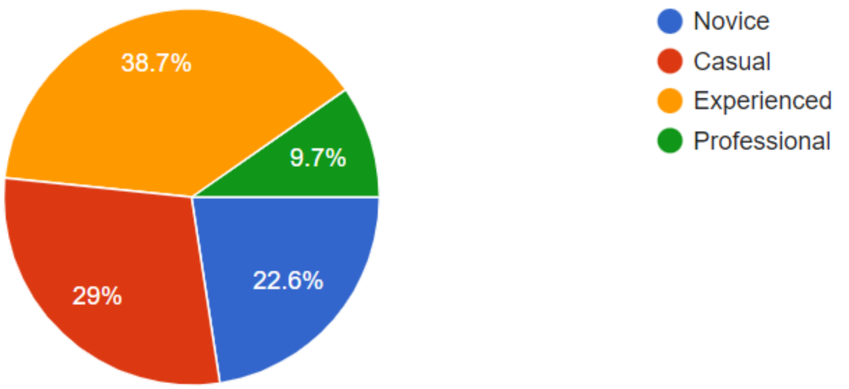
What is your gender identity?

60 responses



How would identify your gaming experience?

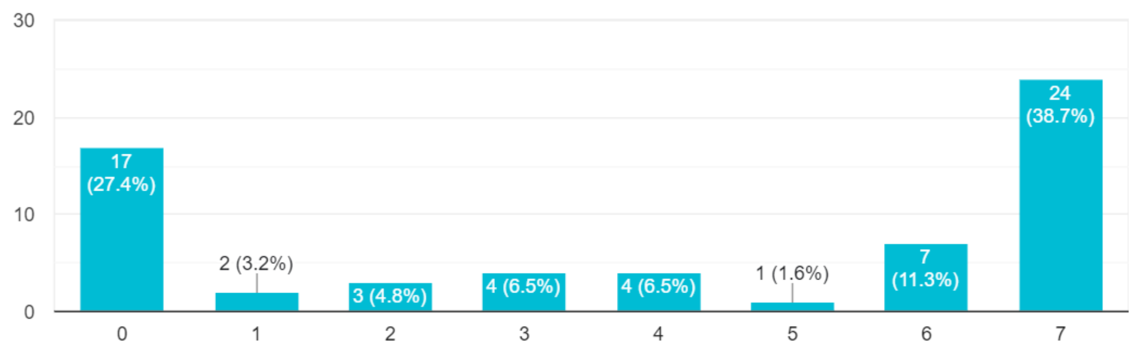
62 responses



How many days a week on average do you play video games?

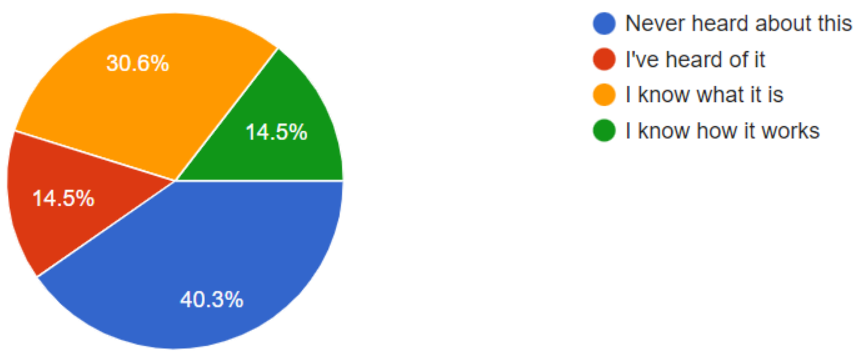
Copy

62 responses



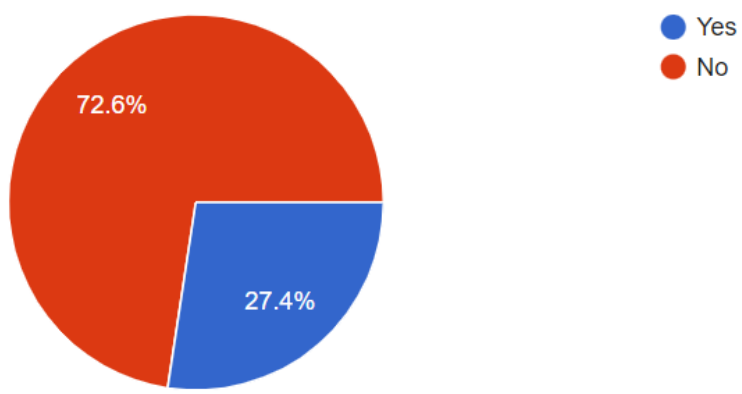
How familiar are you with the concept of automatic (or dynamic) difficulty adjustment systems?

62 responses



Have you played any games with automatic difficulty adjustment systems?

62 responses



If you answered yes to the previous question, what game(s) were they?

17 responses

Alien: Isolation (a bit different but it adapts to how you play to make the game harder), Mortal Kombat, RE4 and onwards, BOTW, Half Life 2, Oblivion, Lego Starwars and other lego games, Left 4 Dead 2, Fallout 3 and some of the other fallouts have it, XCOM reboot (if you do really bad it makes it insanely difficult for you), Civ games arguably, even Dark Souls games could be said to have it (more you die harder it gets the lower health mechanic), and many more games I can't think of right now

Path of Exile, World of Warcraft, Mario Kart

Resident Evil 4, Left 4 Dead, Mario Kart, Mortal Kombat, Pokémon

Fortnite, Apex Legends, Call of Duty

Breath of the Wild, Mario Kart, Nuclear Throne

Tends to be sports games in the past. Madden used to do this at some point. I think the new F1 games do it too

If you answered yes to the previous question, what game(s) were they?

17 responses

Mario Kart

im going to guess PvP shooters MMR counts, so Overwatch, Valorant, CSGO etc.

Minecraft, Jedi Fallen Order, Terraria, etc

Left 4 Dead, Uncharted 1-4, The Last of Us Remastered

Half-Life, Resident Evil, Fallout:3/NV, Left 4 Dead, FIFA

depending what u mean by automatic many games I played make beginners go in thier own queue so they aren't straight up starting against experienced players such as fortnite, overwatch, and fall guys

Metal Gear Solid 5, Half Life 2, Resident Evil 4

Resident Evil 4, Half-Life 2

Muck, forza horizon 4/5 (difficulty just increases if you want it to)

League of Legends, tera online or any mmorpg, etc.

I guess League of Legends counts as this, as you rank up you get enemies that are more difficult to deal with

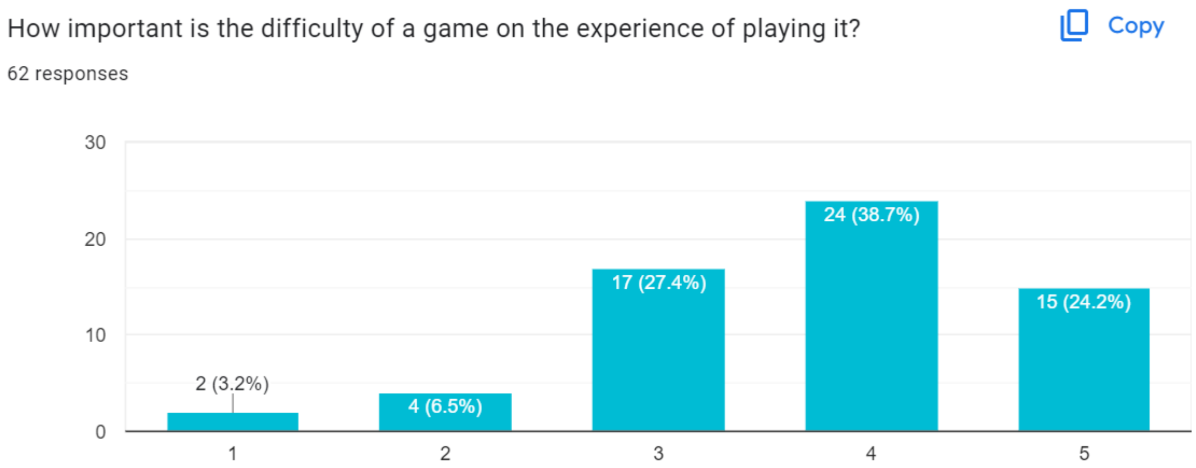
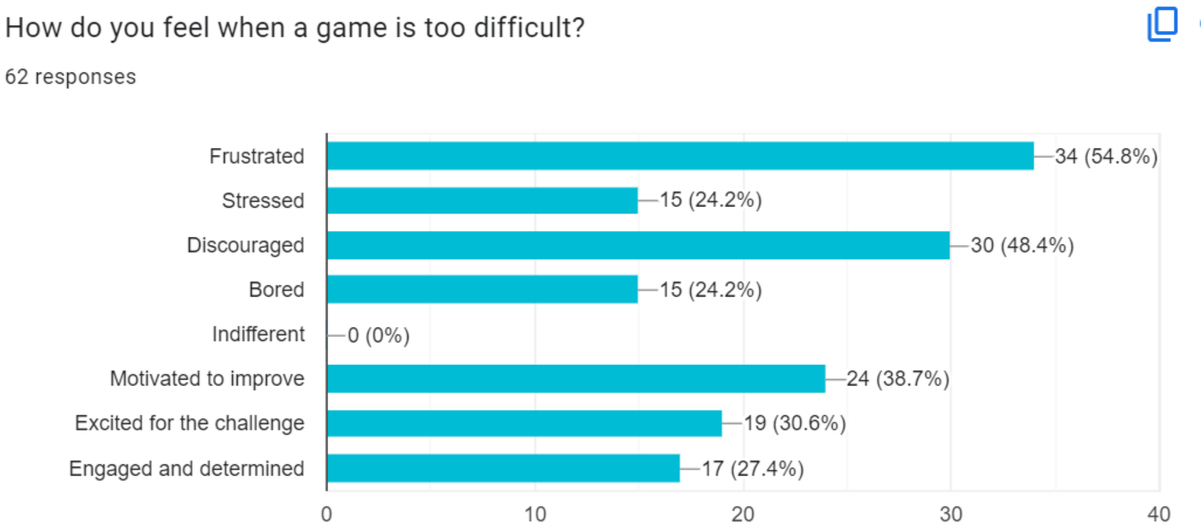
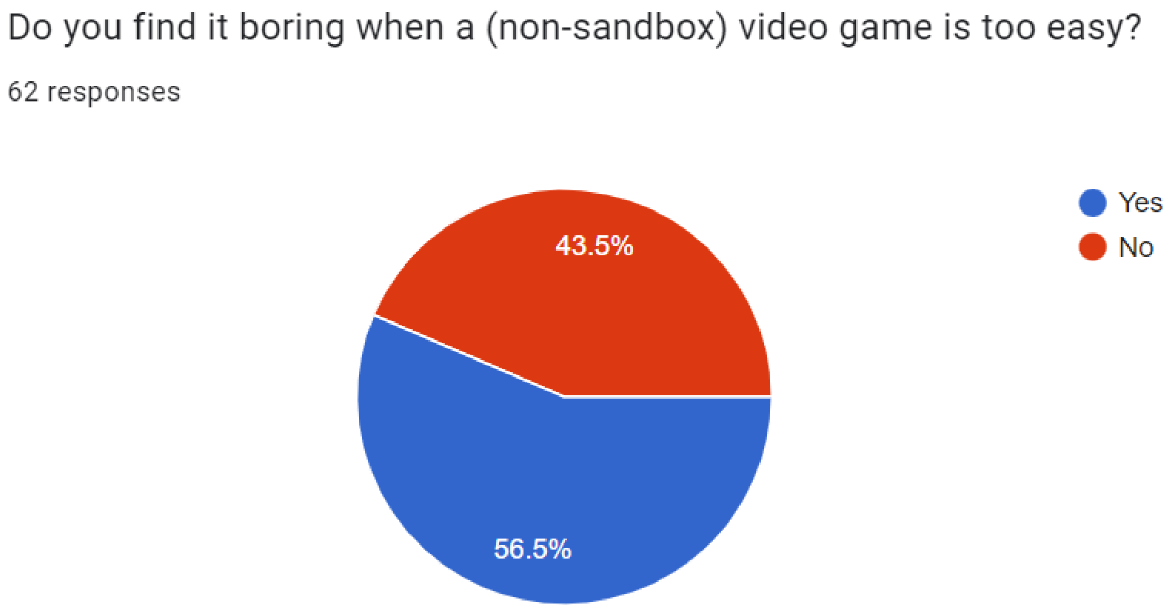


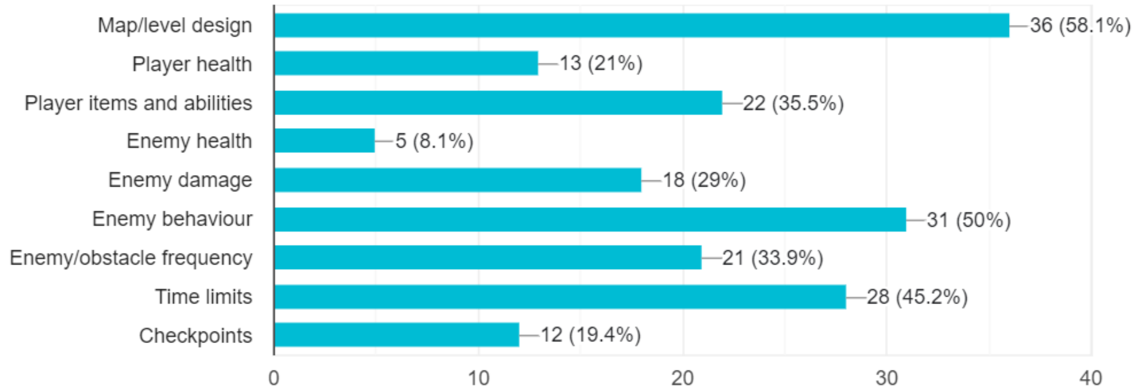
Figure 4 Survey question asking the importance of difficulty in video games, with 1 being not important and 5 being very important.





Pick the top 3 factors you think contribute to the difficulty of video games?

62 responses



How important is it for you to be able to customise the difficulty settings in video games?

62 responses

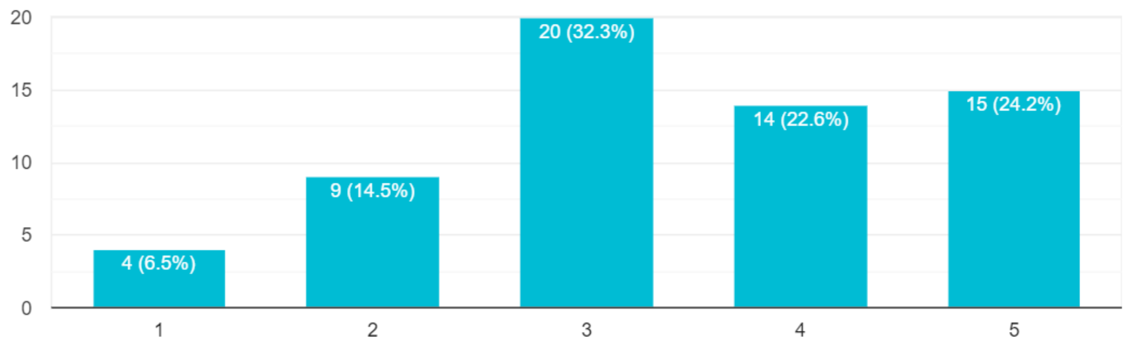
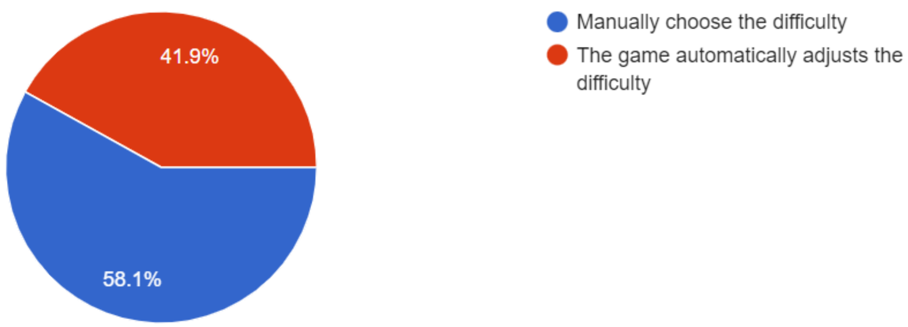


Figure 5 Survey question asking how important customisable difficulty settings are, with 1 being not important and 5 being very important.

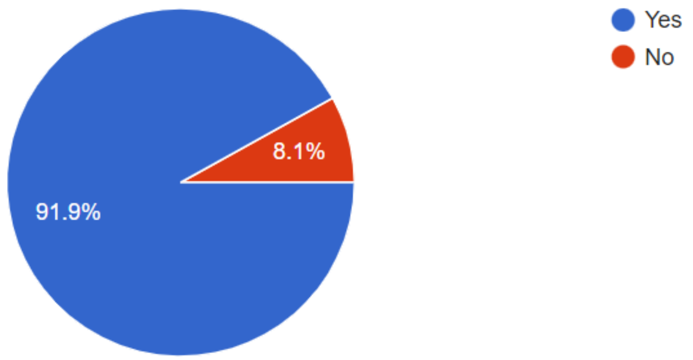
Would you prefer to manually choose a difficulty, or for the game to adjust its difficulty level automatically based on your performance?

62 responses



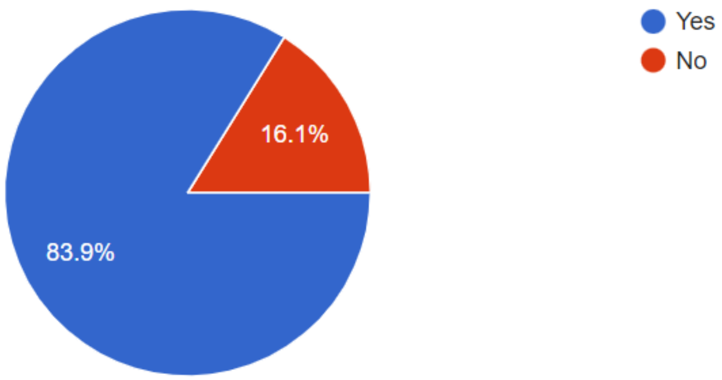
Do you think having the option for both manual and automatic difficulty adjustment would improve player experience with a game?

62 responses



Have you ever stopped playing a game due to it being too easy or too difficult?

62 responses



How important is it for you to be challenged when playing video games?

62 responses

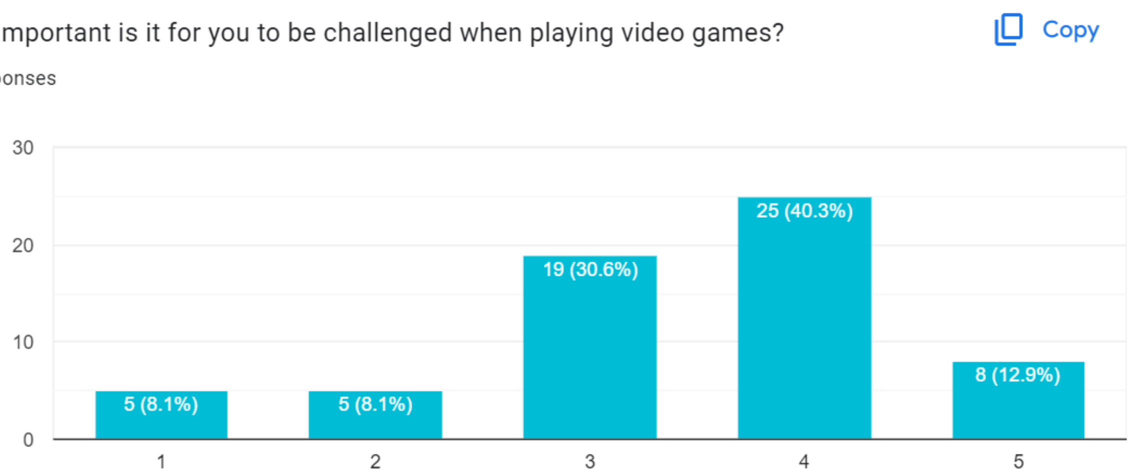
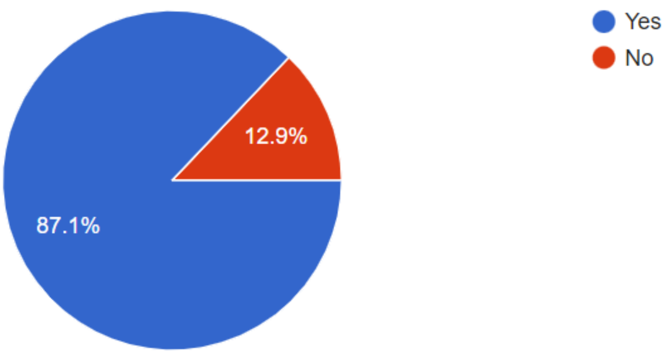


Figure 6 Survey question asking how important it is to be challenged, with 1 being not important and 5 being very important.

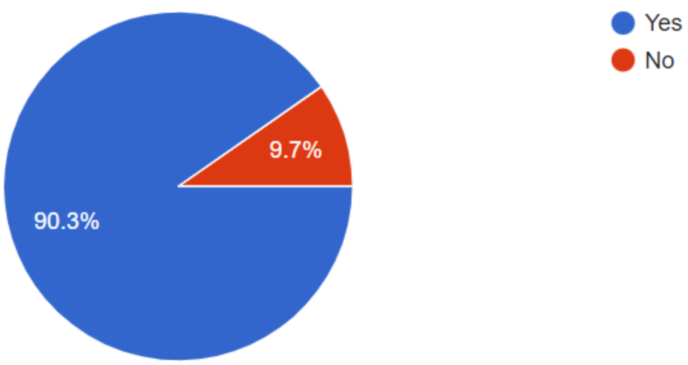
Across all video games you have played, would you say video games typically have an enjoyable difficulty level?

62 responses



Are you more likely to replay a game that has varying difficulty options or gets more difficult as you become better at the game?

62 responses



10.3 Ap. C – R Code For Analysis

```
# Read CSV data
data <- read.csv("form_responses.csv")
# Apply column names as variables
colnames(data) <- c("Timestamp", "Age_Group", "Gender_Identity",
"Gaming_Experience", "Days_Played", "Familiarity_ADAS", "Played_ADAS",
"Games_Played", "Difficulty_Importance", "Feel_Too_Difficult",
"Boring_When_Easy", "Factors_Contribute", "Customise_Importance",
"Difficulty_Preference", "Both_Options_Improve", "Stopped_Playing",
"Challenge_Importance", "Enjoyable_Difficulty", "Replay_Difficulty_Options")
# Categorise Familiarity_ADAS data and store as unique values for the analysis
of variance
data$Familiarity_ADAS <- factor(data$Familiarity_ADAS)
# Create the analysis of variance model with Difficulty_Importance as the
dependant variable and Challenge_Importance as the independent variable
model <- aov(Difficulty_Importance ~ Challenge_Importance, data=data)
```

```
# Output the model results to the console
print(summary(model))
# Create the analysis of variance model with Difficulty_Importance as the
dependant variable
model2 <- aov(Difficulty_Importance ~ Familiarity_ADAS, data=data)
print(summary(model2))
# Modify the difficulty preference data to be binary for the ANOVA calculation
data$Difficulty_Preference <- ifelse(data$Difficulty_Preference == "Manually
choose the difficulty", 1, 0)
# Create the analysis of variance model with Customise_Importance as the
dependant variable
model3 <- aov(Customise_Importance ~ Difficulty_Preference, data=data)
print(summary(model3))
```

10.4 AP. D – PERFORMANCE TESTING RESULTS

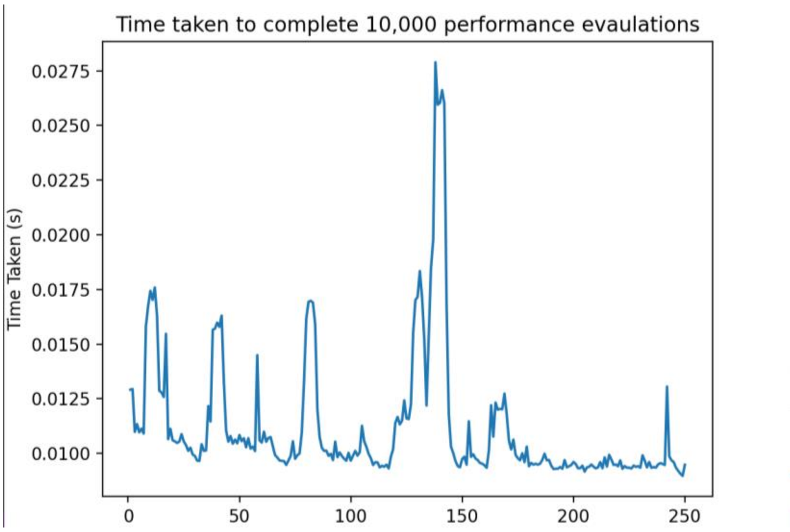


Figure 7 Time taken to complete 10,000 performance evaluations (250 repeats of the test).

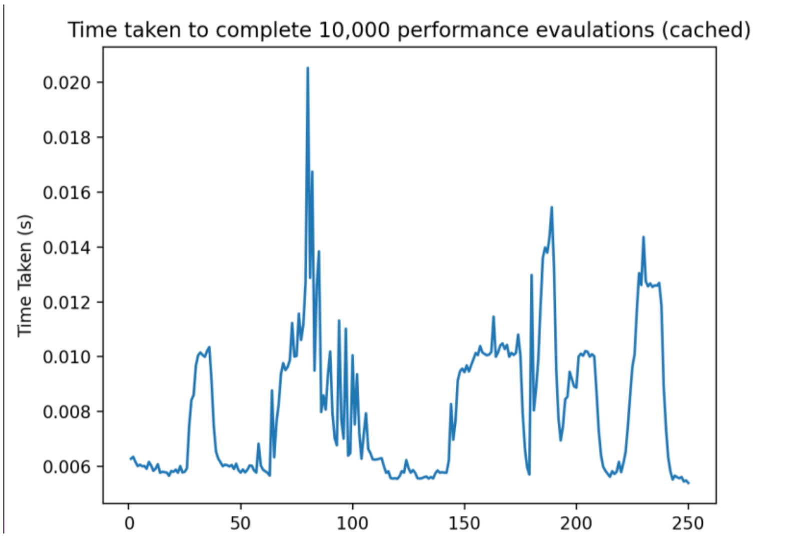


Figure 8 Time taken to complete 10,000 cached performance evaluations (250 repeats of the test).

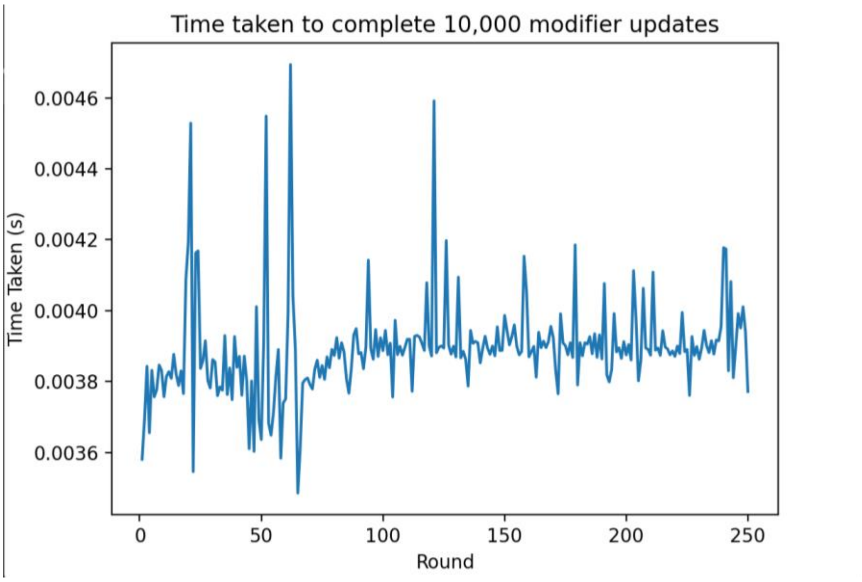


Figure 9 Time taken to complete 10,000 modifier updates (250 repeats of the test).

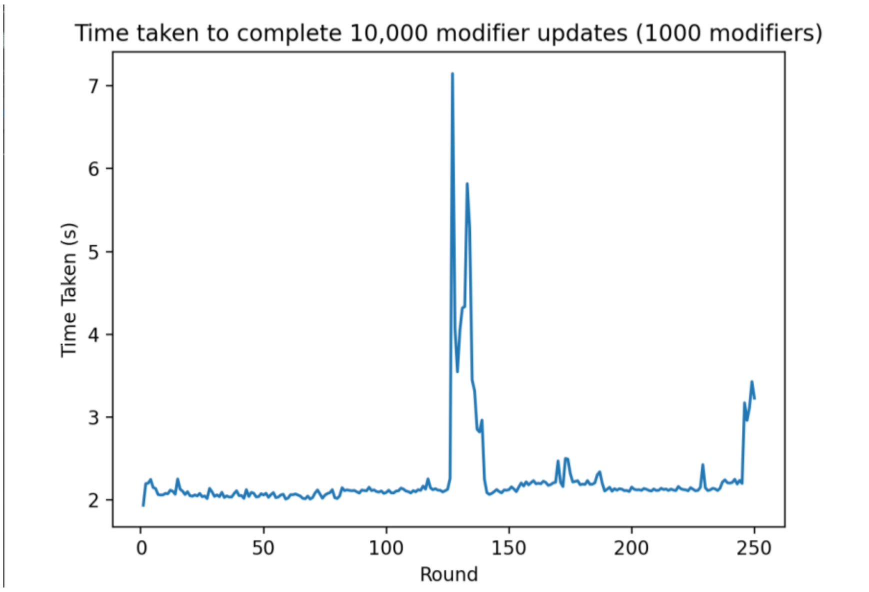


Figure 10 Time taken to complete 10,000 modifier updates with 1000 modifiers (250 repeats of the test).

10.5 Ap. E – GameMetricController Code

```
from typing import Callable

class GameMetricController:
    """
    The DDA system uses a quantifiable metric to assess the difference
    between current performance and the desired performance (setpoint).
```

```
    A PID controller then uses the provided modifiers to adjust gameplay
    aspects aiming to push the player closer to the desired performance.
    """

    def __init__(self,
                 metric: Callable[[], float],
                 modifiers: list,
                 setpoint: float,
                 kp: float = .1,
                 ki: float = .1,
                 kd: float = 1):
        self.metric = metric
        self.modifiers = modifiers
        self.setpoint = setpoint
        self.kp = kp
        self.ki = ki
        self.kd = kd
        self.last_error = 0
        self.integral_sum = 0
        self.last_output = 0

        self.evaluation_cache = {}

    def evaluate_performance(self) -> float:
        error = self.setpoint - self.metric()

        # Check if error has previously been calculated
        if error in self.evaluation_cache:
            return self.evaluation_cache[error]

        # Proportional term
        proportional = self.kp * error

        # Integral term
        self.integral_sum += self.ki * error

        # Derivative term
        derivative = self.kd * (error - self.last_error)
        self.last_error = error

        # Summate terms
        self.last_output = proportional + self.integral_sum + derivative

        # Add to cache and return value
        self.evaluation_cache[error] = self.last_output
        return self.last_output

    def update_modifiers(self):
        if not self.last_output:
```

```
            raise ValueError("last_output not set")
        for modifier in self.modifiers:
            # modifier = [update_function, weight]
            modifier[0](self.last_output * modifier[1])
```

10.6 Ap F – Game Demonstrative Code

```
# In-built modules
from time import sleep
from random import randint
from pprint import pformat
# Third party modules
from numpy import linspace
# Project modules
import DDA

class Game:
    """
    Tower Defence Game

    Demonstrates the use of PID controllers to create a dynamic difficulty
    adjustment system.
    """
    def __init__(self):
        # Game properties
        self.map_length = 1000
        self.round = 1
        self.lives = 3
        self.player_coins = 60
        self.towers = []
        # Enemy properities
        self.enemies = []
        self.min_enemies = 10
        self.min_enemy_health = 60
        self.min_enemy_speed = 20
        self.enemies_modifier = [0, 0] # [previous_value, current_value]
        self.enemy_health_modifier = [0, 0]
        self.enemy_speed_modifier = [0, 0]

    def add_tower(self, tower_type: int):
        if tower_type == 1: # Normal
            self.towers.append(Tower(len(self.towers), 25, 2, 50))
        elif tower_type == 2: # Heavy
            self.towers.append(Tower(len(self.towers), 70, 1, 100))
        elif tower_type == 3: # Speed
            self.towers.append(Tower(len(self.towers), 10, 10, 75))

    def get_active_enemies(self) -> list:
        return [enemy for enemy in game.enemies if enemy.active]
```

```
    def distribute_towers(self):
        # Distribute towers equally along the map
        locations = linspace(0, self.map_length, num=len(self.towers))
        for i, tower in enumerate(self.towers):
            tower.location = round(locations[i])

    def generate_enemies(self):
        self.enemies = [Enemy(id,
                              self.get_round_enemy_health(),
                              self.get_round_enemy_speed())
                              for id in range(self.get_round_enemies())]

    def get_round_enemies(self, game_round: int = 0) -> int:
        if game_round <= 0:
            game_round = self.round
            return self.min_enemies + round(.7 * (game_round - 1)) +
self.enemies_modifier[-1]
        else:
            return self.min_enemies + round(.7 * (game_round - 1)) +
self.enemies_modifier[0]

    def get_round_enemy_health(self, game_round: int = 0) -> int:
        if game_round <= 0:
            game_round = self.round
        return self.min_enemy_health + ((game_round - 1) * 10)

    def get_round_enemy_speed(self, game_round: int = 0) -> int:
        if game_round <= 0:
            game_round = self.round
        return self.min_enemy_speed  + ((game_round - 1) * 3)

    def get_enemies_per_tower(self) -> float:
        return self.get_round_enemies() / len(self.towers)

    def set_enemies_modifier(self, value: float):
        self.enemies_modifier.append(round(value))
        del self.enemies_modifier[0]

    def set_enemy_health_modifier(self, value: float):
        self.enemy_health_modifier.append(round(value))
        del self.enemy_health_modifier[0]

    def set_enemy_speed_modifier(self, value: float):
        self.enemy_speed_modifier.append(round(value))
        del self.enemy_speed_modifier[0]

    def exit(self):
```

```
        print("============================================")
        print("GAME OVER!")
        print(f"Rounds survived: {game.round}")
        print(f"Towers built: {len(game.towers)}")
        exit()

    def print_towers(self):
        pretty_towers = '\n- '.join([pformat(tower.__dict__) for tower in
game.towers])
        print(f"\nTowers:\n- {pretty_towers}")

    def print_round_info(self):
        last_enemy_position = min(self.get_active_enemies(), key=lambda enemy:
enemy.position).position if self.get_active_enemies() else "N/A"
        output = f"""Round {self.round} Info:
    Enemies Alive: {len(self.get_active_enemies())}
    Lives Left: {self.lives * '❤️ '}
    Coins: {self.player_coins}
    Last Enemy Position: {last_enemy_position}
        """
        print(output)

class Tower:
    def __init__(self, id:int, damage:int, fire_rate:int, range:int):
        self.id = id
        self.damage = damage
        self.fire_rate = fire_rate
        self.range = range
        self.location = 0

    def attack(self, nearby_enemies):
        targets = sorted(nearby_enemies, key=lambda enemy:
enemy.health)[:self.fire_rate]
        for target in targets:
            target.health -= self.damage
        return targets

class Enemy:
    def __init__(self, id:int, health:int, movement_speed:int):
        self.id = id
        self.health = health
        self.movement_speed = movement_speed
        self.position = 0
        self.active = False

    def move(self):
        self.position += randint(self.movement_speed//4, self.movement_speed)
```

```
# Driver code
if __name__ == "__main__":
    print("--------- Tower Defence Game ---------")
    game = Game()
    enemies_per_tower_dda =
DDA.GameMetricController(metric=game.get_enemies_per_tower,
                         modifiers=[[game.set_enemies_modifier, 1]],
                         setpoint=10,
                         kp=0.2,
                         ki=0.1,
                         kd=0.5)
    input("Press Enter to start!")
    dda_outputs = []
    while True:
        print("===========================================")
        enemies_difference = game.get_round_enemies() -
game.get_round_enemies(game.round - 1)
        enemy_health_difference = game.get_round_enemy_health() -
game.get_round_enemy_health(game.round - 1)
        enemy_speed_difference = game.get_round_enemy_speed() -
game.get_round_enemy_speed(game.round - 1)
        print(f"""Round {game.round}:
    Enemies: {game.get_round_enemies()} ({'' if enemies_difference < 0 else
'+'}{enemies_difference})
    Enemy Health: {game.get_round_enemy_health()} ({'' if
enemy_health_difference < 0 else '+'}{enemy_health_difference})
    Enemy Speed: {game.get_round_enemy_speed()} ({'' if enemy_speed_difference
< 0 else '+'}{enemy_speed_difference})
    Coins: {game.player_coins}
    Lives: {game.lives * '❤ '}""")

        game.print_towers()
        # Add new towers
        while (game.player_coins > 0) and (input("\nAdd a new tower? (y/n)\n>
").lower() in ["y", "yes"]):
            print("--------------------------------------")
            print(f"Coins: {game.player_coins}")
            print("""\nSelect a tower to purchase:
    1) Normal (30 coins)
    2) Heavy (100 coins)
    3) Speed (50 coins)""")
            tower_costs = {1:30, 2:100, 3:50}
            try:
                user_selection = int(input("> "))
            except ValueError:
                print("Invalid input!")
                continue
```

```
            if user_selection < 1 or user_selection > 3:
                print("Invalid input!")
                continue

            if game.player_coins >= tower_costs[user_selection]:
                game.player_coins -= tower_costs[user_selection]
            else:
                print("You don't have enough coins for that tower!")
                continue
            game.add_tower(user_selection)
            game.print_towers()

        game.distribute_towers()

        input("\nPress Enter to start the round.")

        game.generate_enemies()
        loop = 0
        game.enemies[loop].active = True
        while (len(game.get_active_enemies()) > 0):
            # Adjust enemy positions
            for enemy in game.enemies:
                if enemy.active:
                    enemy.move()
                    # If an enemy goes beyond the map length, a life should be
lost
                    if enemy.position > game.map_length:
                        enemy.active = False
                        game.lives -= 1
                        life_plural = "life" if game.lives == 1 else "lives"
                        print(f"[GAME] An enemy made it through your defences!
{game.lives} {life_plural} left!")
                        # The game ends when the player has no lives left
                        if game.lives == 0:
                            game.exit(dda_outputs)

            # Calculate tower damage
            for tower in game.towers:
                nearby_enemies_condition = tower.location - tower.range <=
enemy.position <= tower.location + tower.range
                nearby_enemies = [enemy for enemy in game.enemies if
nearby_enemies_condition and enemy.active]
                targets = tower.attack(nearby_enemies)
                for target in targets:
                    if target.health <= 0:
                        target.active = False
                        game.player_coins += 1
```

```
            # Activate next enemy
            loop += 1
            if loop < len(game.enemies):
                game.enemies[loop].active = True

            # Print round info
            game.print_round_info()

            # Sleep for dramatic effect
            sleep(0.1)
        print("==============================================")
        print("Round complete!")
        game.player_coins += game.round * 10
        dda_outputs.append(enemies_per_tower_dda.evaluate_performance())
        enemies_per_tower_dda.update_modifiers()
        game.round += 1
```

10.7 Ap. G LLM Reflective Analysis

Throughout the project, I used OpenAI's GPT-3.5 AI ChatGPT to aid in exploring topics and critically reflecting on ideas and outputs. By engaging with the AI in interactive conversation, I was able to brainstorm concepts and ideas, discuss approaches with different algorithm implementations, and refine the scope of the project to align with the assignment assessment criteria. This increased the efficiency and quality of research, design, and analysis for the project and was a useful starting point for expanding my ideas and research.

Large language models will very confidently assert false statements and hallucinate facts. Thus, relying on such an AI for accurate facts and knowledge is a naïve approach and so to ensure the content within the report maintained academic integrity, I only used the AI as an initial starting point for thinking about ideas and identifying areas for research. Then, I used proper research techniques to make sure what the report states is cited and cross-referenced by reputable sources to reduce the chance that false information was presented.

Here is an example of a prompt, and ChatGPT's response:

I'm creating a game-agnostic dynamic difficulty adjustment system using a PID controller, using a tower defence game as a showcase for the algorithm. What might the PID controller look like?

ChatGPT

Using a PID controller for dynamic difficulty adjustment in a tower defence game can be an interesting approach. The PID controller is a control loop feedback mechanism that adjusts a system's output based on the error between the desired setpoint and the actual output. In the context of a game, the setpoint can represent the desired difficulty level, while the actual output can be derived from the game's performance metrics, such as player score, completion time, or survival rate.

Here's a general outline of how the PID controller might look for your game-agnostic dynamic difficulty adjustment system:

1. Error Calculation: Determine the error as the difference between the desired difficulty level (setpoint) and the current performance metric value. For example, if the setpoint is a target survival rate of 80%, and the current survival rate is 70%, the error would be 80% - 70% = 10%.

2. Proportional Gain (P): Multiply the error by a proportional gain factor (Kp) to obtain the proportional component of the controller's output. The proportional component adjusts the output in proportion to the error. Higher values of Kp will result in more aggressive adjustments. The proportional term can be calculated as: P = Kp * error.
3. Integral Gain (I): Calculate the integral component to account for cumulative errors over time. Multiply the error by an integral gain factor (Ki) and sum up these values over time to get the integral term. This term is useful for eliminating steady-state errors. The integral term can be calculated as: I = Ki * (sum of errors).
4. Derivative Gain (D): Calculate the derivative component to consider the rate of change of the error. Multiply the rate of change of the error by a derivative gain factor (Kd) to get the derivative term. This term helps prevent overshooting and oscillations. The derivative term can be calculated as: D = Kd * (change in error).
5. Controller Output: Combine the proportional, integral, and derivative terms to compute the overall controller output. The output can be obtained by summing these three terms: Output = P + I + D.
6. Apply Controller Output: Use the controller output to adjust the game difficulty. Depending on the game, you might increase or decrease the number of enemies, their health, their speed, or other relevant parameters. The adjustment should be based on the magnitude and sign of the controller output.
7. Repeat: Continuously repeat the above steps to dynamically adjust the game difficulty based on the performance metrics and the desired setpoint. It's important to tune the proportional, integral, and derivative gain factors (Kp, Ki, and Kd) to achieve the desired level of control and responsiveness.

Remember that PID controllers are a general control mechanism, and their effectiveness depends on the specific application and tuning. You might need to experiment and fine-tune the PID parameters for your tower defence game to achieve optimal difficulty adjustments.

This response provides key insights into the anatomy of a PID controller, as well as crucial considerations such as tuning. However, without fact-checking the response it would be hard to know whether the maths provided would yield a valid response. Furthermore, it is a very basic overview and leaves plenty of room for further exploration into the topic.

Overall, leveraging the capabilities of ChatGPT alongside external research and my own analysis has allowed the creation of a comprehensive and well-informed report by increasing the efficiency of the process of understanding key concepts and topics before exploring more advanced papers, studies, and websites. It has provided valuable insights and demonstrated the potential power the AI has as a tool for academic and professional work when used correctly and with its limitations in mind.