Generalized Enemy Wave Generator for Survival Shooter Games

Yiqun Hao, Kangwei Liao, Dengyu Liang
[1]COMP 5900J Game Design & Development Technologies

Abstract

Motivation: Enemy wave generation is the most indispensable part of survival shooter games. Designing and implementing a wave generator is both time-consuming and labor-intensive. **Objective:** We propose a generalized enemy wave generator that could be easily used in survival shooter games similar to Left4Dead. The generator we built supports a standardized game process with three enemy population modes alternating during the game. It can spawn enemies with specified properties such as health and speed at wanted positions. More importantly, our generator could constantly measure how challenging the game is for the current player so that it could dynamically adjust enemies' properties to ensure a tailored player experience. **Result:** We used the sample survival shooter game provided by Unity as a testbed to evaluate our generator. Results have shown that our generator could provide a good user experience with tailored difficulty for different players. **Conclusion:** Our work could be used as a standardized and player-adaptive enemy wave generator for survival shooter games. Developers using our generator could easily customize it to meet their requirements and build their own game with much less effort.

Introduction

In recent years, games adopting procedural content generation (PCG) have gained remarkable success. The major goal of PCG is to create game content automatically as opposed to manually (Shaker, Togelius, and Nelson 2016). Intuitively, PCG is largely used by small-scale game development teams to simplify and accelerate development because it can replace hand-crafting with computer-generating. However, the power of PCG is far beyond that. Advanced PCG could provide a noticeably different playthrough every time the game starts. Infinite Mario Bros (Shaker et al. 2012) and Minecraft (Salge et al. 2019) could be the most typical games adopting PCG. In those two games, levels or maps are randomly generated by algorithms instead of hard-coded. Therefore, players will rarely get exactly the same experience making them have "infinite" playthroughs.

PCG, as a general field, has many sub-fields with different approaches to procedurally generate game content. Player-Driven PCG (Shaker, Yannakakis, and Togelius 2012) is one of the sub-fields that provides a way to tailor game content such as plots and game difficulty according to specific player behavior. By virtue of player-driven PCG, tailored Super Mario Bros' levels that induce the desired experience for the current player could be generated instead of merely playable levels (Pedersen, Togelius, and Yannakakis 2010). To produce desired results, player-driven PCG usually involves the use of Drama Management (Roberts and Isbell 2008), Player Modeling (Carneiro, da Cunha, and Dias 2014)(Hooshyar, Yousefi, and Lim 2018), and Dynamic Difficulty Adjustment (DDA) (Sepulveda, Besoain, and Barriga 2019)(Hunicke 2005).

In this research, we build a generalized enemy wave generator for survival shooter games by using techniques in the field of player-driven PCG. In survival shooter games, generating enemies is always a critical thing that directly leads to the success or failure of the game. However, there are only a few simple enemy generators available in the Unity Asset Store (Store 2014). Furthermore, all of those enemy generators can only spawn enemies with fixed values with no player-adaptive adjustments. Thus, we decide to build a generator that not only supports enemy generation but also dynamically controls the difficulty of enemies generated according to current player skill captured during the game.

Specifically, our generator supports enemy spawning in three difficulty modes: easy mode, normal mode, and hard mode each with a different enemy population and enemy properties. The game will alternate among those three modes to ensure a varied game progression as most survival shooter games have such as what the AI director does in Left4Dead (Booth 2009). Game developers could use this generator to build a survival shooter game with the least effort but the greatest controllability as they could specify all parameters for enemies. More importantly, the generator could adjust the game difficulty in a player-driven way so that players with different skill levels could all have a corresponding difficulty level. Specifically, we consistently monitor the player's performance to identify how challenging the current game is for the player. A numerical value, called intensity, is used to represent that difficulty. Then, we could dynamically adjust the game difficulty according to the intensity value. The goal is to let every player have a dynamic, immersive game experience that is neither too easy to lose motivation nor too difficult to be frustrated and quit the game.

In short, this generator is significant because by using our generator, small-scale development teams could develop a survival shooter game supporting difficulty adjustment with much less effort, time and budget.

Problem Formulation

In this work, we set out to build an enemy wave generator that satisfies the following requirements:

- **Spawn Enemies**: The generator should be able to spawn enemies with specified properties and numbers at the specified positions.
- **Simple Game Flow**: The generator should support three game modes similar to Left4Dead (Booth 2009). During the game, three modes should alternate to give a standard survival shooter game flow.
- **Player Modeling**: The generator should notice how stressful the player is and represent it using an intensity value. By doing so, the generator could adjust the game difficulty based on intensity values. For instance, if the intensity value remains very low meaning the game is not challenging at all, the generator should upgrade enemies to increase the overall difficulty.
- **Difficulty Adjustment**: The generator should adjust enemies' properties to make them easier or harder to fight against corresponding to the result of player modeling. For example, if the game difficulty should be decreased, the enemies could have a slower speed and lower health.
- **Generalizability**: The generator should be generalized and exported to public use so that game developers could adopt and customize it to make their own games.

We believe a generator having those abilities is valuable as, to the best of our knowledge, there is no similar generator available on Unity.

Related Work

In this section, we introduce some critical work that is closely related to our work. First, the AI director of Left4Dead is presented as it is the major work we refer to. Then, general concepts in player modeling and dynamic difficulty adjustment will be illustrated.

Left4Dead AI Director

Our work is inspired by the AI Director of the game Left4Dead (Booth 2009). Left4Dead is a classic survival shooter game in which four survivors cooperate to escape environments swarming with enraged infected. This game was published in 2009 and has been remarkably popular since then. The key for Left4Dead to success is its dynamic game pacing controlled by the AI director. AI director is constantly monitoring players' performance and adjusting the infected population on the fly in order to maximize player excitement. They represent the player's excitement as a numerical value called intensity. Under certain conditions, such as the player getting injured by the infected, the intensity will increase. If there is no nearby infected, the intensity will decay towards zero over time. Based on the intensity,

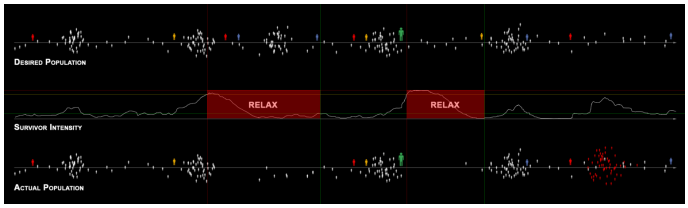

Figure 1: Population and Intensity in Left4Dead

the AI director modulates the infected population by using the following four modes:

- **Build Up:** In this mode, a full threat population is generated until the intensity crosses the peak threshold.
- **Sustain Peak:** After reaching the peak threshold, the full threat population will continue for another 3-5 seconds to keep players excited but not completely overwhelmed.
- **Peak Fade:** After sustaining peak mode, the game will switch to a minimal threat population and monitor the intensity until it decays out of peak range.
- **Relax:** The game will maintain a minimal threat population for another 30-45 seconds. Then the build up mode is resumed meaning another cycle begins.

The procedurally generated population by the AI director is shown in Figure 1. Even though the intensity estimation is quite crude as the author admits, the resulting pacing works and has gained great success. Therefore, we decide to produce a generator with the idea of intensity and modes used in Left4Dead, because we believe this pattern is common in many survival shooter games.

Nevertheless, there are some major differences existing between our work and the AI director in Left4Dead. First, the generator is published online to allow any future use and it could be generalized and used in many similar games. In contrast, the AI director is not published. Second, our generator supports dynamical difficulty adjustment while the author of Left4Dead clearly said the difficulty is not changed during the game but only the population of enemies (Booth 2009). Last but not least, we provide good controllability for users using our generator. They could define their own conditions that rise the intensity, the speed of intensity decaying, the peak threshold, etc. As a result, our work is significant and unique.

Player Modeling & Dynamic Difficulty Adjustment

Player modeling and dynamic difficulty adjustment (DDA) are related techniques that are often used together in video games to create more engaging and realistic gameplay experiences. Player modeling involves using machine learning algorithms to predict and simulate the actions and behaviors of players, while DDA involves adjusting the difficulty level of the game in real time based on the player's performance. By combining these two techniques, game designers can create games that are tailored to the individual player's abilities and preferences, providing a more personalized and enjoyable gaming experience.

Player modeling is a process used in video game development to create a model of how a player interacts with a game (Yannakakis et al. 2013). The model is created by gathering data on how a player plays the game and then analyzing that data to better understand the player. This data can be gathered through surveys, tests, or even through real-time measured metrics. Player models can be used to create a better gaming experience by optimizing the game's difficulty, providing feedback to players, and creating personalized content. In the context of our work, intensity can be thought of as a player modeling technique that measures how difficult or engaging a game is for a particular player. Intensity refers to the level of emotion or engagement felt by players during a game. For example, continuous shooting at an enemy means that the enemy is a threat to the player, and a character being injured means that the player is stressed or the difficulty is beyond the player's skill. And the long silence means that the player is at ease. We balance the intensity of a game to provide a challenging and engaging experience for players without overwhelming them. We achieve this through DDA. By measuring the intensity of a game, we also determine the conditions for entering different difficulties.

The goal of DDA is to provide players with a challenging but enjoyable experience by continuously adapting the difficulty of the game to match their abilities (Zohaib 2018). This can be done by adjusting various factors in the game, such as the number and types of enemies, the amount of health or resources available to the player, or the time limits for completing certain tasks. By using DDA, game designers can ensure that players are always challenged. In our work, we compare the player's performance to a set of intensity targets that we set in advance and then use this comparison to determine the difficulty of the next wave. In different difficulty modes, the intensity targets are different. If the player's performance is below the intensity targets, the game could automatically increase the properties of enemies in the next wave to provide a more challenging gameplay experience. On the other hand, if the player's performance is above the intensity targets, the game could decrease the properties of enemies in the next wave to make the game easier. In short, our generator combines player modeling and dynamic difficulty adjustment to procedurally generate enemies that could provide a better player experience.

Proposed Approach

In this section, we present our approach to build the enemy wave generator.

Overall Architecture

With our enemy wave generator, developers can spawn enemies that adapt to a player's skill level. We implemented an interface for our generator to give developers more control over DDA. The interface contains two types of custom scriptable objects: GEGCharacter and GEGProperty. They allow developers to fit their characters (players and enemies) in our generator. Additionally, we provided a custom inspector that allowed developers to configure two key components of our generator: Intensity Manager and Spawning (or "Spawner"). The inspector is bound to a prefab named GEGManager. Modifying it allows developers to customize intensity manager's and spawner's behaviors, such as when, what, and how enemies spawn. In addition, our generator relies on two types of custom scriptable objects: GEGCharacter and GEGProperty. Here is a flowchart showing how to use the generator (Fig. 2).

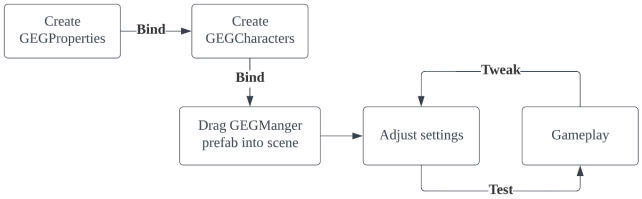

Figure 2: Steps to use the generator

Next, we will introduce two key components of our generator: intensity manager which is responsible for tracking intensity, switching difficulty mode, and adjusting enemies' properties, and spawner which is responsible for spawning new enemies with updated properties.

Intensity Manager

Our intensity manager is inspired by the adaptive dramatic game pacing shown in Left4Dead (Booth 2009) in a more concrete way (Thue 2015). The emotional intensity value of players is maintained and calculated throughout the game in the same manner as Left4Dead. The emotional intensity indicates how challenging the game is at the moment, with a higher intensity indicating a more challenging game; it will be increased proportionally by two events:

- When the player gets injured by an enemy
- When the player causes damage to enemies

In order to make our DDA system work, developers must implement our interface: IGEGController, in their character controllers (e.g., PlayerHealthController, EnemyHealthController, etc.) where they are responsible for storing properties' values (e.g., health value).

To calculate an event's contribution to intensity, we used the equation:

$$IV_{new} = IV_{current} + \frac{V_{current}}{V_{default}} * S_{prop} * S_{mode}$$

where IV_{new} is the new intensity value; $V_{current}$ is the current property's value (e.g., 75hp); $V_{default}$ is the default property's value (e.g., 100hp); S_{prop} is the intensity scalar which can adjust the contribution of this property in prefabs scripts (Fig. 4). As an example, a reduction in health will have a significant impact on intensity; S_{mode} is the intensity scalar in each mode.

In addition, there are three difficulty levels defined based on intensity in gameplay; they are:

- **Easy mode**: This mode consists of a small number of enemies and lasts for a predefined period of time. It's meant

for players to explore the map and get prepared. Therefore, the intensity value shouldn't be elevated much under easy mode. Otherwise, our generator will adjust the properties of enemies, such as lowering enemies' initial health, if the intensity of the player reaches a predefined threshold in easy mode.

- **Normal mode**: This mode consists of a greater number and more powerful enemies (e.g., higher health and damage). It is therefore expected that the intensity will increase until it reaches the entry threshold for hard mode. Similarly, our system adjusts the game difficulty when the intensity increases too rapidly or too slowly. A difficulty upgrade is automatically applied if the intensity value remains low for an extended period of time. In contrast, if the intensity value soars and reaches the hard mode threshold quickly, the game difficulty will be reduced.
- **Hard mode**: When the intensity reaches a predefined threshold in normal mode, this mode will be activated. Compared to normal mode, this mode has a greater number of and more powerful enemies. In this mode, players are challenged for a predefined period of time. After a predetermined period in hard mode, the game will switch to easy mode (e.g., after 3 waves of enemies).

In our design, the game flow should begin with the easy mode so players can prepare for upcoming challenges. As soon as the easy mode has ended, the game will switch to normal mode. In normal mode, enemies will keep coming until the player's intensity triggers hard mode. Then, the hard mode will last for a predefined time. After the peak, the game will back to the easy mode and begin another cycle.

It was considered that the normal mode may be too easy for some players and that the intensity of the game would not rise to the level of hard mode. In order to solve this problem, every wave will increase the difficulty by a predefined percentage. In addition, in case of the absence of enemies (e.g., The enemy is on its way to the player), the intensity will diminish at a predetermined rate over time.

In the inspector of intensity manager (Fig. 3):

- Auto Decrease Amount: Decrease in intensity per second after "Auto Decrease Cooldown"
- Expected Flexibility: Indicates the range of expected intensity in each mode setting (e.g., "Expected Flexibility" = 5, and "Expected Easy Intensity" = 5, so the expected intensity in easy mode is [0, 10]).
- Adjustment: In each difficulty adjustment, the adjustment percentage of each enemy's property. (e.g., "Adjustment" = 5%, enemy's health will increase/decrease 5% in each difficulty adjustment triggered)
- Cumulation Rate: Percentage of difficulty accumulated over time. (e.g., "Cumulation Rate" = 0.1%, after each wave, the difficulty of each enemy's property increases by 0.1%)
- Increment Scalar: Scale up/down the intensity increment under this mode.
- Decrement Scalar: Scale up/down the intensity decrement under this mode.

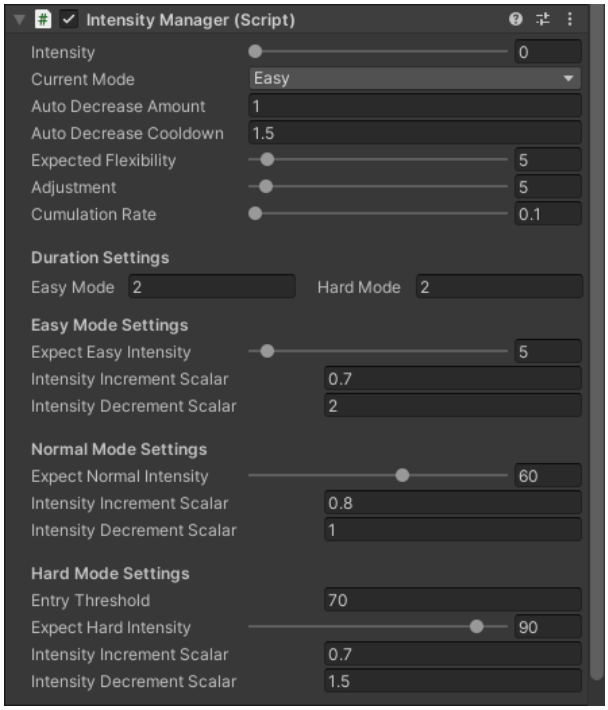

Figure 3: Intensity manager's settings

Figure 4: Example: A health script bind to player's prefab

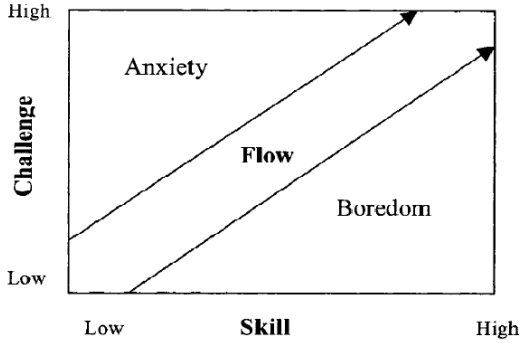

Figure 5: Flow theory (Csikszentmihalyi and Csikzentmihaly 1990)

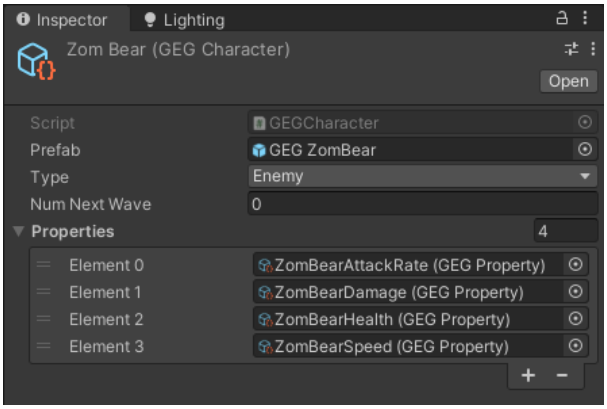

Figure 6: GEGCharacter Example

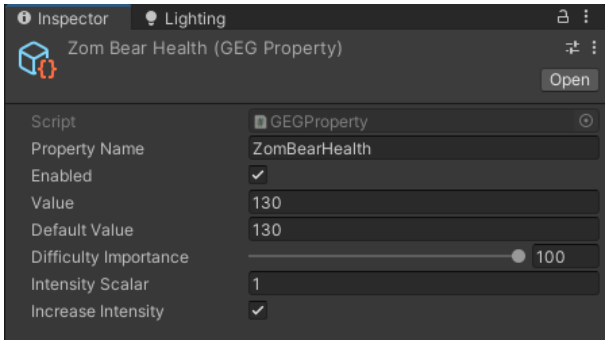

Figure 7: GEGProperty Example

Intensity manager is responsible for monitoring the player's current flow channel (Fig. 5) that defined the player's situation in two dimensions: skill and challenge. We allow developers to customize their DDA by giving them more control over the intensity manager. Developers can implement DDA in their games by tweaking settings to fit the majority of players' flow channels.

Spawner

Having determined the desired difficulty level and evaluated the player's intensity, we use a spawner to instantiate adjusted enemies in the current game scene. Spawner will use the scriptable objects modified by the intensity manager as input to spawn enemies that match the player's current emotional model. Inputs to spawner are as follows:

- Enemies' prefabs, which the following customized scriptable objects are bound:
  - GEGCharacter scriptable object (Fig. 6)
  - GEGProperty scriptable object (Fig. 7)
- Number of enemies to spawn

In our generator, spawner's algorithm consists of the following steps:

1. In the current scene/wave, check if any enemies are alive:
   (a) If true: Start spawning enemies in new waves.
   (b) If false: Wait until the current wave has been completed by the player.
2. If the spawning process started, spawner selects random types of enemies and spawns a predefined number of instances of each type at a random spawn rate and position.
3. After all enemies have been spawned, stop spawning.

The key components in the spawning process are GEGCharacters and GEGProperties.

A GEGCharacter contains a prefab of a certain type of character (e.g., player or enemy), a number of characters to be spawned in the next wave (field "Num Next Wave" in Fig. 6), and list of properties (e.g., health, movement speed, etc.). Intensity manager will modify the field "Num Next Wave" to adjust the number of enemies to be spawned in the next wave.

A GEGProperty contains the template value (field "Value" in Fig. 7) for certain properties that will be referenced when spawning instances of this type of character. This template value is modified by the intensity manager to adjust the difficulty of each type of enemy. After that, the spawner will spawn instances with the modified properties.

To calculate an instance's property, we used the formula:

$$V_{default} * (A + C) * I_{prop}$$

where A is the field "Adjustment" in Fig. 3; C is the field "Cumulation Rate" in Fig. 3; I_{prop} is the field "Difficulty Importance" in Fig. 7

Discussion of Results

To evaluate our generator, we use the survival shooter game tutorial provided by Unity as the testbed [1]. A screenshot of the game with a debug UI is shown in Figure 8. It is noteworthy that we just need to fine-tune the parameters in the inspector for a few rounds. Then the generator could generate enemies as planned. During the gameplay, difficulty modes could alternate smoothly so that it is already a satisfying survival shooter game after fine-tuning the generator.

Moreover, we invited seven players to play our game and each of them has a different skill level. To be specific, we group them into three groups based on their skills. Two players having rich experience in survival shooter games are grouped as advanced players. Three players who even rarely play video games are grouped as novice players. The rest two players who have limited past experience in similar games are grouped as average players. We observed the performance of each group respectively and found the following facts:

- For advanced players having a high skill level, they could eliminate enemies easily without getting hurt at the early stage. However, that means the intensity value will remain relatively low. Therefore, our generator could find out the game is too easy for the current player and decide to upgrade enemies' properties. As a result, the difficulty level

[1] https://learn.unity.com/project/survival-shooter-tutorial

Figure 8: Sample Survival Shooter Game Using Our Generator

could match the player's skill level after a few waves. Besides, the difficulty cumulation rate also made some impact which makes the game harder over time. Overall, advanced players have a good experience playing our game.

- For average players having an average skill level, the default difficulty level almost fit them at the beginning. Thus, they could have an immersive experience faster than advanced players. However, we found that if an average player played better than usual even for one wave, the generator will relentlessly increase the difficulty. Then, the player will struggle with a much higher intensity which then makes the generator lower the difficulty. As a consequence, the difficulty of the game is bouncing depending on the predefined adjustment percentage. If the adjustment percentage is high, the player experience will be affected drastically.
- For novice players having bad skills, we find that the easy mode is very useful for them to get familiar with, meanwhile, the generator could find out they are not good at this game. After that, the difficulties of normal mode and hard mode are both lowered to avoid overwhelming them. However, we notice that if the adjustment percentage is not great enough, the generator will need more time to lower the difficulty which could result in player death.

From observing invited players, we conclude that our generator could generate enemies appropriately with the ability to adjust game difficulty based on simple player modeling. We also found some drawbacks of our generator. First, it is vital to have proper default parameters which might takes a lot of time to find out. A bad default setting is very likely to ruin players' experiences. Second, the two metrics we used to increase the intensity value are not good enough to model players. We will add more metrics in our future work. Third, the difficulty adjustment amplitude is a tricky issue. We propose to make it dynamic. For example, when the intensity value is bouncing up and down, as happened to some average players, the difficulty adjustment amplitude should be smaller to find a balance point where the difficulty level matches the player's skill level.

Conclusion & Future Work

In conclusion, we produce a player-adaptive enemy generator for survival shooter games. By using our generator, developers could develop a similar game faster with little effort spent on generating enemies. Another core feature the generator offers is dynamic difficulty adjustment based on the player's performance. Developers could specify what enemies' properties should be changed with difficulty. Then, we use an intensity value to track how hard the current game is for players. As a result, if the intensity remains high, the generator will spawn weaker enemies with worse properties. Otherwise, if the intensity remains low, the generator will spawn stronger enemies with better properties. The way the intensity changes could also be fully customized depending on the specific game. We have used the survival shooter game tutorial provided by Unity as the testbed to evaluate our generator. Results have shown that our generator could produce a satisfying enemy generation with effective difficulty adjustment based on the player's performance. We believe such an enemy generator is valuable and insightful in the field of Procedural Content Generation (PCG).

In the future, we will further improve our generator. First, with current metrics that have an impact on intensity, players' skill levels cannot be accurately assessed. Hence, we will add more measures (e.g., the time between each kill) that could better model players. Second, we will let developers decide the number of each enemy in each mode. They could also select which specific enemy types should be spawned in each mode by using check boxes on the Unity inspector. Third, we would like to add enemy number adjustment in our DDA approach to make the game more flexible. Finally, we will publish our work on Unity to get user feedback. As a result, we could extract valuable feedback as our future work.

Acknowledgment

We would like to thank Prof. David for being a helpful professor.

References

Booth, M. 2009. The ai systems of left 4 dead. In *Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford, 2009*.

Carneiro, E. M.; da Cunha, A. M.; and Dias, L. A. V. 2014. Adaptive Game AI Architecture with Player Modeling. In *2014 11th International Conference on Information Technology: New Generations*, 40–45. IEEE.

Csikszentmihalyi, M.; and Csikzentmihaly, M. 1990. *Flow: The psychology of optimal experience*, volume 1990. Harper & Row New York.

Hooshyar, D.; Yousefi, M.; and Lim, H. 2018. Data-driven approaches to game player modeling: a systematic literature review. *ACM Computing Surveys (CSUR)* 50(6): 1–19.

Hunicke, R. 2005. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 429–433.

Pedersen, C.; Togelius, J.; and Yannakakis, G. N. 2010. Modeling player experience for content creation. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1): 54–67.

Roberts, D. L.; and Isbell, C. L. 2008. A survey and qualitative analysis of recent advances in drama management. *International Transactions on Systems Science and Applications, Special Issue on Agent Based Systems for Human Learning* 4(2): 61–75.

Salge, C.; Guckelsberger, C.; Green, M. C.; Canaan, R.; and Togelius, J. 2019. Generative design in Minecraft: chronicle challenge. *arXiv preprint arXiv:1905.05888* .

Sepulveda, G. K.; Besoain, F.; and Barriga, N. A. 2019. Exploring dynamic difficulty adjustment in videogames. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, 1–6. IEEE.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'neill, M. 2012. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 304–311. IEEE.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural content generation in games*. Springer.

Shaker, N.; Yannakakis, G. N.; and Togelius, J. 2012. Towards player-driven procedural content generation. In *Proceedings of the 9th conference on Computing Frontiers*, 237–240.

Store, U. A. 2014. Asset store.

Thue, D. J. 2015. Generalized Experience Management .

Yannakakis, G. N.; Spronck, P.; Loiacono, D.; and André, E. 2013. Player modeling .

Zohaib, M. 2018. Dynamic Difficulty Adjustment (DDA) in Computer Games: A Review. *Advances in Human-Computer Interaction* 2018: 1–12. doi:10.1155/2018/5681652.