Dynamic Difficulty Adjustment

Marshall P. Hoffmann
Division of Science and Mathematics
Morris, Minnesota, USA 56267
Hoff0899@morris.umn.edu

ABSTRACT

Difficulty plays a pivotal role in how players experience video games. Because of this, implementing a system that produces a positive experience is very important. This paper will talk about typical video game systems and some of the problem they present. Dynamic Difficulty Adjustment (DDA) is a concept of handling difficulty in real-time in which the game can be altered during play to ensure engagement and immersion. Two methods of DDA: orthogonally evolved AI and dynamic scripting look to improve on prior methods by introducing adaptability and variability.

1. INTRODUCTION

Video games tend to take place in fantastical worlds. This allows for the use of imaginary forces, and mystical powers. Characters in games can harness these forces for spells like fireballs and lightning bolts, as well as fantasy creatures wielding grossly large weapons like swords and guns. In these games there tends to be conflict between multiple characters, which can fuel alot of fights, and goals to eliminate your opponents.

It is estimated that over 60% of American's play video games daily [2], as a source of entertainment, stress relief, educational benefits, etc. Player experience is vital to video game success. Video games are sought after for providing enjoyment, a sense of accomplishment when a challenge has been overcome [3]. When designing a video game, it can be hard to create difficulties that are challenging, but achievable to players of different skill levels. Producing systems that can adapt to various player strategies and change with player improvements, will be effective for players of different skill levels, as well as players with different improvement rates. As the video game entertainment industry continues to grow, new and/or improved systems will need to be developed to help optimize player engagement and enjoyment, as this determines the success of a product.

2. DEFINITIONS

2.1 Flow

Flow is an extreme state experienced by the player where the task is so rewarding, they are willing to perform the task for the sake of undergoing that experience as if the task was an excuse to do so [1]. For the purposes of this paper, this can be thought of as optimal enjoyment in which the player feels extremely engaged and immersed into a game to the point where they can lose track of time. This is the state developers strive to keep the player in, as this level of engagement is associated with the optimal enjoyment, and in educational purposes this provides the player with optimal learning benefits. Flow is achieved when the player has been sufficiently, but not overly challenged (figure 1).

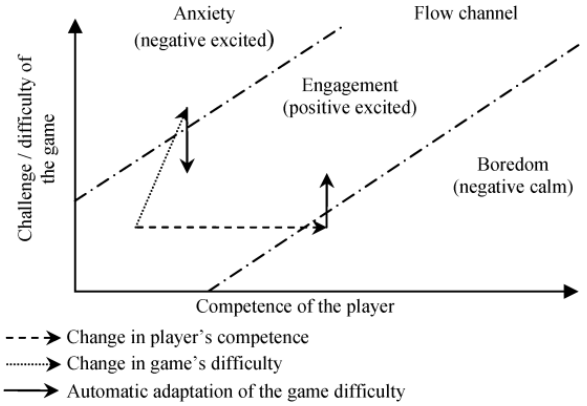

Figure 1: A graph illustrating optimal level of game difficulty vs player competence [4]

2.2 Artificial Intelligence (AI)

Artificial intelligence (AI), is intelligence displayed by non-player characters (NPCs) [10], or agents as they will be referred to in this paper. AI is used more broadly in video games, to describe the algorithms used to control how the agents act. Examples include: How the agent learns from past experiences, and how the agent determines its next actions. AI have incentives or goals, often in games that represent conflict between the player and agents, the AI goal might be to kill the player, or to slow their progress in some way. These goals are assessed through a fitness function which numerically quantifies the AI performance into a fitness value. This is important because using the fitness value can allow the AI to learn the value of different actions.

UMM CSci Senior Seminar Conference, April 2019 Morris, MN.

3. BACKGROUND

3.1 Difficulty Adjustment and Player Experience

Difficulty is important in relation to a player's overall enjoyment in video games. Difficulty is the degree of challenge the player experiences; a game with a higher difficulty is harder to complete. Difficulty is based on factors such as inventory e.g. the weapons or spells the player has access to, agent/opponent skill, environment, etc. Players feel engaged in the game when they feel they have been challenged, and overcame an obstacle [3]. A player interacting with content that has difficulty far away from their skill level will produce negative results. Figure 1 illustrates this correlation; a player who is playing at a difficulty that is too hard for their skill level will become negatively excited causing stress, anxiety and frustration. A player who is playing at a difficulty lower than their skill level will experience boredom. In both of these cases, difficulty needs to be adjusted, either higher or lower, to fall into the region of positive excitement.

Due to varying levels of player skill, traditionally, difficulty has been coded statically. A player is presented with various options such as easy, medium, or hard. This is the starting point for the difficulty which then increases gradually (linearly or stepwise [11]). This practice can be helpful to the developer because it can give them control over how the player experiences the game, and the story they would like to tell. However, this system causes some problems:

1. Forces players to self-rate skill level:
   Before a player begins the game, the player does not know their skill level, or what certain difficulty levels mean.

2. Predictable AI interaction:
   A player might notice tendencies in how the agent performs. For instance if the agent uses certain abilities in the same order, the player might notice this sequence, and develop strategies that trivialize the content.

3. Player improvement variance:
   Players improve at unpredictable, and differential rates. Two players might start at the same skill level, but one player might progress significantly faster than the other, in which the difficulty they have chosen might not remain suitable for their skill level.

4. Development cost
   It can be a long iterative process to create multiple levels of difficulty, as each new stage needs to have modifications made to variables such as the health of the player or agents, or tactics used by agents. These difficulties need to be tested to make sure values chosen provide the interactions and results desired, and that certain combinations do not cause unexpected results.

3.2 Dynamic Difficulty Adjustment (DDA)

Dynamic Difficulty Adjustment (DDA) is a technique for modifying a game's features, behaviors, or scenarios automatically in real-time to adapt to various player skill levels, and strategies [11]. It should be noted that different games will have different variables that can be adjusted. For instance in fighting games, things like health or agent interaction might be the manipulable variables, whereas puzzle games might not have these variables. So in this scenario how much time a player has to solve a puzzle might be the manipulable variable. Adjusting dynamically to the behaviors of players helps maintain engagement, by keeping them in the region of positive excitement (Figure 1), even after the game has been released [1]. For a DDA to be successful, there are three requirements [11]:

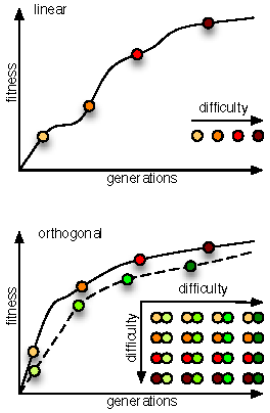

Figure 2: Comparison of linear vs. orthogonal evolution of AI [5]

1. The game needs to track the player ability and rapidly adapt to it.

2. The game must maintain a balance between the player's skill and difficulty.

3. The adaptation process must be seamless, not clearly perceived by players, and coherent with previous game states.

4. ORTHOGONALLY EVOLVED AI

4.1 Evolving AI

AI can be evolved to learn new tactics beneficial for performance. When evolving AI, there is a starting algorithm that performs a task with a fitness value that quantifies how well it performed. The algorithm goes through a series of iterative changes, each creating a new algorithm referred to as offspring. The offspring's performance is then measured, and the best performing offspring is chosen. This chosen offspring is the new algorithm generation. This done repeatedly evolves the AI to have better performance.

4.2 Theory

One method of automating difficulty is through orthogonally evolved AI. Typically, when difficulty is adjusted using evolved AI, only the opponent AI is evolved. In this method, two orthogonal axes are co-evolved to control difficulty: collaborative AI, which controls agents assisting players, and opponent AI, which controls agents competing against players. In this context, orthogonal refers to the fact that the two AI being evolved have independent incentives.

The advantage of this method is that it allows for an increased variety of experiences. AI from multiple generations, and axes, with different fitness values, creates a multiplicatively larger set of unique experiences/difficulties the player

can encounter. In standard opponent evolved AI implementations, difficulty adjustment is restricted to a single axis. While orthogonally evolved AI is able to manipulate these axes to create unique experiences. For example, when the difficulty of a game needs to be reduced this can be done in multiple ways (or a combination). 1) The player is paired with more evolved collaborative agent(s). 2) The player is matched against weakly-adapted opponent agent(s).

4.3 Application

To test this method [5] uses a simple predator-prey simulation. In this simulation, there are two classes of agents: predator and prey. The predator's goal is to capture as many of the prey agents, and conversely the prey's goals are to avoid capture for the entire 120 seconds game duration.

The hypothesis explored is that the use of orthogonally evolved AIs help expand game difficulty options [5]. To test this hypothesis, players interact with the game involving various mixtures of adapted and unadapted AI for both the opponents and collaborators. These levels of adapted and unadapted AI can be obtained through the multiple generations of evolution. Evolution does not occur while the game is in progress, predator and prey agents are controlled by previously evolved AI. In this scenario difficulty is not adjusted in real-time, but the goal is to show that the different axes can be manipulated to create different difficulties, which could in the future be manipulated in real-time.

4.4 Experiment

In the experiment, evolutionary history of the prey as well as the predator were saved every 25 generations. Eventually two instances were chosen, generation 900 and 1900, as they represented two significantly distinguishable levels of adeptness. Agents using AI from generation 900 are considered unevolved, while agents using AI from generation 1900 are considered evolved. Between these generations there is a noticeable difference in predation capability, and a swarming technique developed by the prey.

The game was run through a web browser, using 200 Amazon Mechanical Turk participants. Amazon Mechanical Turk is a crowdsourcing marketplace where companies or individuals can outsource jobs/tasks to allow easier collection of data. At the start of each game, one of the four difficulty conditions was randomly chosen: evolved prey vs evolved predator, evolved prey vs unevolved predator, unevolved prey vs evolved predator, or unevolved prey vs unevolved predator [5]. Each player played for the entire 120 seconds or until caught by the predator. At the end of the game, two parameters were recorded: length of survival, and number of prey agents alive.

4.5 Results

Results of this experiment show that average survival time in each of the four combinations mentioned in section 4.4, differ significantly (figure 3)[5]. This data shows that the difficulty of the game is more heavily reliant on the prey's ability to swarm, rather than the predator's ability to catch. Difficulty determined by a singular axis, for example prey evolution, allows for only two game states: unevolved prey vs predator or evolved prey vs predator. The use of multiple axes allows use of AI from different evolutionary time periods, creating more game variability, ultimately resulting in a smoother difficulty transition.

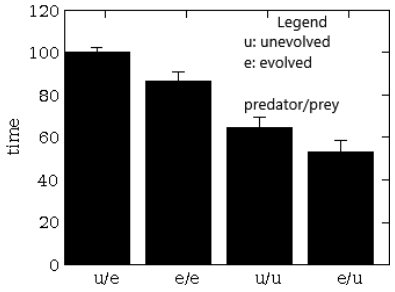

Figure 3: Comparison of player performance versus various evolutionary combinations [5]

5. DYNAMIC SCRIPTING

5.1 Theory

Dynamic scripting is another method that can be used to dynamically adjust difficulty in video games. Dynamic scripting primarily works to create a behavioral script for opponents agents [6]. Each class of agent has a set of rules, called a ruleset that represent each behavior that agent can undergo. Each time a new opponent agent is generated it takes a combination of these rules, called a script. Each rule has its own separate weight, called the rule weight, which can be used to determine which rules are selected from the ruleset to create the agent's script, as well as how the agent uses the rules it has access to. The probability of a rule being chosen from the ruleset into the generated agent's script is based on its rule weight, the heavier the rule, the more likely it is to be chosen. After a conflict, the ruleset changes its weights based on the outcome of the conflict, determined by a fitness function.

There are four main components in a dynamic scripting algorithm [6]:

1. Set of Rules
2. Script Selection
3. Rule Policy
4. Rule Value Updating

The first component of a dynamic scripting algorithm is a set of rules. These are rules are created by the developer, though some research has been done to automate rule creation [7]. Each of these rules can have conditionals that restrict its use in certain scenarios. For example, a rule involving a blizzard spell that can not be chosen for a script unless the agent is in an arctic environment.

The next component is the selection of rules that make up the script. The script has a total capacity n (amount of rules that the opponent can have) determined by the developer. At this point rules are chosen probabilistically, based on their weight, meaning the higher the weight, the more likely the rule is to be chosen.

The next component is rule policy. This is how the agent uses the script it was given. Rules can have additional conditional requirements at this stage that apply for the use of the rule, rather than a conditional applied to the selection of the rule. For example, a rule that allows for healing spells may be restricted until a friendly agent's health is below a $X\%$ threshold. Agents use of the script can be determined

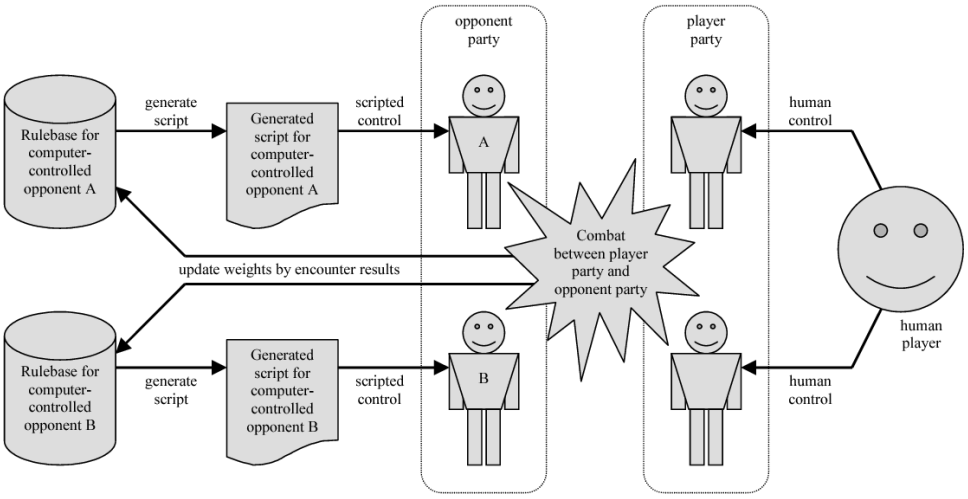

Figure 4: Process of Dynamic Scripting [9]

in two ways. 1) Each rule can have a set priority. This causes a flow chart structure, where the AI will check the highest priority rule to see if its condition has been met: if the condition is met, the agent uses that rule, and if the condition is not met it the next highest priority rule is checked. 2) The weights of each rule can be used to probabilistically determine which rule is used. In this case all rules that have their conditional requirement met will have their weights totalled, and the likeliness of a rule to be used is determined by how much weight the rule contributes to the sum.

Rule weight updating is the last and defining component of dynamic scripting. This is done by two functions created by the developer: A fitness function that determines how well the agent performed in the conflict (on a scale of 0 to 1), and a reward function which changes the values of the rule weights, rewarding positive behavior, and punishing negative behavior. The fitness function uses variables to measure its performance. For example, in a battle between two enemies, health remaining and damage dealt could be two variables used to determine the performance of the agents. The reward function then provides feedback to the weights, based on the fitness value [6]. Only rules that are in the current script are subjected to weight changes. Once these weight changes are made, the remaining rules not selected for the current script have their weight evenly adjusted to maintain an equal weight total in the ruleset.

The dynamic scripting process promotes positive behavior by increasing script weights when the agent performs well, and penalizing negative behavior by decreasing script weights when the agent performs poorly. Over time this will result in an emerging optimal script comprised of highly weighted rules. In a video game setting however, balanced gameplay is desired, where the goal is to create agents that can keep up, but do not surpass the players skill level. Fortunately, dynamic scripting can be modified in three ways to achieve this balance [11]:

1. High-fitness penalizing
2. Weight clipping
3. Top culling

High-fitness penalizing is a form of penalizing optimal behavior when the dynamic AI is out performing the player [11]. In this case, when the dynamic agent defeats the player, instead of the weights being rewarded for positive behavior, they are diminished.

Weight clipping is a technique that reduces how much weight a rule can hold [11]. A rule that is at the set maximum value can not be rewarded further. This results in balanced gameplay with more variability because when weights can not reach as high, less optimal rules have an increased chance of being selected for an agent's script.

Top culling works similarly to weight clipping, but rather than putting a cap on how high the weight can reach, it allows the weight to grow above the maximum value [11]. Once the rule has surpassed the maximum, it can no longer be selected for script generation, forcing other rules to be selected. This avoids rules that are causing frequent wins from being reused repeatedly, opening up room for some of the weaker rules to take their place. Rules can drop below the maximum again if the dynamic agent's other rules are rewarded, decreasing its weight percentage.

Figure 5 represents the results from using these modifications in a study [8] through the game Neverwinter Nights. The authors found that all three modifications were effective at forcing a more even game, but that high-fitness penalizing had high variance in results, meaning that it was unreliable, with some encounters having extremely high fitness values and some extremely low. Weight clipping and top culling provided more consistent results, but still with some variance, which can be an important trait to produce some sort of unpredictability, keeping the player engaged. Top culling stands above the rest, because it has slightly more consistent results, was the only method capable of forcing even gameplay against inferior tactics, and it continues to learn strong behavior even while performing scaled behavior [8].

5.2 Application

Daniel Policarpo, Paulo Urbano, and Tiago Loureiro apply dynamic scripting in the context of a first person shooter [6]. First person shooter is a genre of video game, involving agents in conflict, attempting to kill one another. This application was designed to show the ability for the dynamic

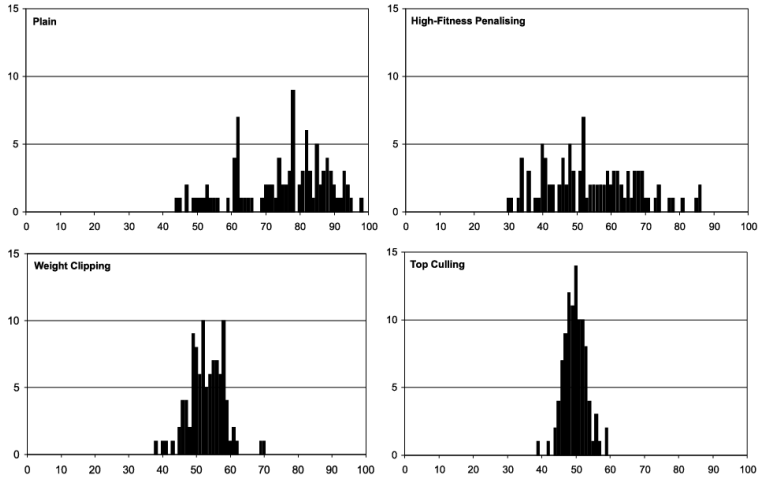

Figure 5: Results of dynamic scripting modifications [8]

AI to adapt to actions taken against its agent. This implementation does not include a human player, however the ability for the dynamic AI to quantify how valuable a rule is compared to the actions of an opponent agent shows that dynamic scripting can be an effective way of modifying difficulty, when paired with methods discussed in 5.1.

In this game there are two agents: a static agent, and a dynamic agent. These two agents are placed in an arena and fight until either one agent is dead, or they reach the maximum game time. The arena has a few items: A health item that bring their health to max (200 health), a barrel that explodes on contact damaging in an area, and an ammo box which restores the agents ammo to max. Each agent has two weapons: A fast firing machine gun which deals 5 damage for each hit, and a slow firing rocket launcher which deals 100 damage at the center, and less the further away from the blast. Each agent has equal access to items and weapons previously mentioned, to ensure that behavior is the only variable. The dynamic scripting agent has access to a variety of rules (discussed later), while the static agent uses the following 4 rules: (1) Patrol - If the opponent is not in range and not visible, the agent moves to predetermined locations in search. (2) Approach Opponent - If the opponent is in sight but not visible, approach the opponent. (3) Shoot rocket launcher - If the opponent is at half or more range, use rocket launcher while approaching opponent. (4) Shoot Machine Gun - If the opponent is less than half range, use machine gun while staying put.

There are 14 rules the dynamic agents chooses from (here are three examples)[6]:

Name	AdvanceGunAttack
Condition	Agent has machine gun ammo and can see an opponent
Effect	Advances towards the opponent and shoots with the machine gun if opponent is in range

Name	TakeAmmo
Condition	Agent does not have ammo in at least one of his weapons and he can see an ammo item
Effect	Advance towards the ammo item

Name	Idle
Condition	Agent cannot see the opponent
Effect	Remain stationary

At the beginning of each conflict, the dynamic agent selects four rules, with an additional default rule used in all scripts: Idle. This action is just the agent standing still. Having a default rule is important because as in most game engines, it is required to have at least one action selected [6]. Rules are used in the priority based system as described in 5.1, where the agent will go through the rules by order of its priority and use the first rule that meets its condition.

After each conflict the performance of the dynamic scripting agent is evaluated with a fitness function [6]. This is a function that generates a score for the most recent episode on a scale of 0 (poor performance) to 1 (perfect performance).

$$F(a, \mathrm{g}) = \frac{4^{*}H(a) + 4^{*}D(\mathrm{g}) + 2^{*}T(\mathrm{g})}{10} \tag{1}$$

In this function, these components are used to reflect the performance. The a parameter refers to the agent, and the g parameter refers to the current match [6]. The following components, $H(a)$ represents remaining health of the dynamic agent a, $D(g)$ represents the total damage done to the opponent in the match g, and $T(g)$ represents the time the match g took. They determined after analyzing different weight values over various scenarios that these values produced the best results [6], which displays health as the primary factor in the fitness value, rather than the time to defeat the opponent. The following equations for the components are shown below [6]:

$$H(a) = \frac{h_t(a)}{h_0(a)} \tag{2}$$

$$D(\mathrm{g}) = \frac{(h_0(o) - h_t(o))}{h_0(o)} \tag{3}$$

$$T(\mathrm{g}) = \frac{t_t}{t_m}, a\ loss; \frac{(t_m - t_t)}{t_m}, a\ win \tag{4}$$

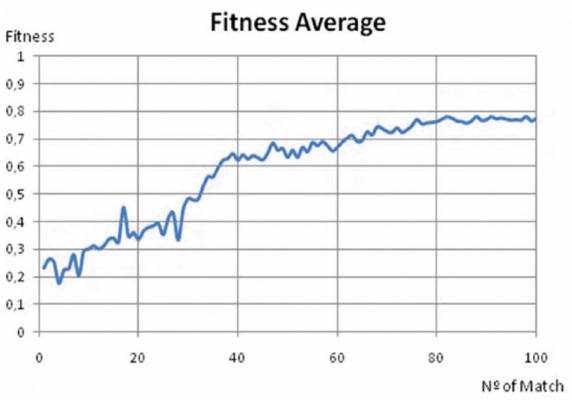

Figure 6: Fitness graph for dynamic agent performance [6]

In these equations, a refers to the agent, o refers to the agent's opponent, and $h_t(x)$ refers to the health of the agent x in time t the end of the match. $h_0(x)$ refers to the health of the agent x at the beginning of the match, t_t refers to the duration of the match, and t_m refers to the maximum game time allowed.

5.3 Results

The researchers in [6] recorded their results in 5 batches of 100 matches. After each batch of 100 matches, the weights are reset to give the opportunity for new learning to occur. These 5 batches were then averaged and represented by figure 6.

These results show that the dynamic agent struggled to perform during the first 30 matches with average fitness values below 0.5 [6], indicating it was losing more than winning. During these matches the dynamic AI is learning which rules are performing well, and working against the static agent's tactics. At around the 40 match point the dynamic agent was consistently able to find victories, with fitness values rising above 0.7. What this shows is that the dynamic AI was able to reliably determine which tactics were powerful against for the rules of the game and/or the static opponents strategies. In this scenario it took the AI 30 to 40 iterations to determine which rules were valuable. This could present a problem for a player being introduced to a game, as during this period the difficulty might not match their skill level; because of this it might be beneficial to have multiple sets of weights established prior to release to help reduce how long it takes the AI to find an optimal balance. In this case the most used rule was SidestepRocketAttack [6], which is a move in which the agent shoots his rocket whilst moving sideways. This is justified because rocket launchers provide more damage than the machine gun counterpart, and moving sideways is an effective measure against incoming fire.

6. CONCLUSIONS

Dynamic Difficulty Adjustment techniques can create a smoother and more engaging video game experience for players. These techniques help increase adaptability, by quantifying how the agents are performing in relation to other agents. When the agent is under or over performing, the ability to quantify its performance, and the actions that lead to that performance, allows the AI to be manipulated to better align with a desired difficulty. DDA techniques can also increase variety in gameplay, as agents the player is interacting with can be changed from one encounter to another; having access to multiple traits or tactics, allows for scenarios where different trait combinations can provide different experiences, while maintaining a similar difficulty.

7. ACKNOWLEDGMENTS

Thank you to my advisor Peter Dolan, and my senior seminar professor Elena Machkasova for their expertise, feedback, and guidance. I'd also like to thank University of Minnesota Alum, Andy Korth for his unique expertise and feedback.

8. REFERENCES

[1] D. Ang. Difficulty in video games: Understanding the effects of dynamic difficulty adjustment in video games on player experience. pages 544–550, 06 2017.

[2] T. E. S. Association. Industry facts, 2018. [Online; accessed 10-April-2019].

[3] B. Baére Pederassi Lomba de Araujo and B. Feijó. Evaluating dynamic difficulty adaptivity in shoot'em up games. 10 2013.

[4] G. Chanel, C. Rebetez, M. Bétrancourt, and T. Pun. Emotion assessment from physiological signals for adaptation of game difficulty. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 41(6):1052–1063, Nov 2011.

[5] A. Hintze, R. S. Olson, and J. Lehman. Orthogonally evolved ai to improve difficulty adjustment in video games. In *EvoApplications*, 2016.

[6] D. Policarpo, P. Urbano, and T. Loureiro. Dynamic scripting applied to a first-person shooter. pages 1 – 6, 07 2010.

[7] M. Ponsen, H. Munoz-Avila, P. Spronck, and D. W. Aha. Automatically generating game tactics via evolutionary learning. *Ai Magazine - AIM*, 08 2006.

[8] P. Spronck, M. J. V. Ponsen, I. G. Sprinkhuizen-Kuyper, and E. O. Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63:217–248, 2006.

[9] P. Spronck, I. Sprinkhuizen-Kuyper, and E. Postma. On-line adaptation of game opponent ai with dynamic scripting. *Int. J. Intell. Games & Simulation*, 3:45–53, 01 2004.

[10] Wikipedia. Artificial intelligence — Wikipedia, the free encyclopedia, 2019. [Online; accessed 12-April-2019].

[11] M. Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018.