Assessing Video Game Balance using Autonomous Agents

1st Cristiano Politowski
École de Technologie Supérieure
Montreal, Quebec, Canada
cristiano.politowski@etsmtl.ca

2nd Fabio Petrillo
École de Technologie Supérieure
Montreal, Quebec, Canada
fabio.petrillo@etsmtl.ca

3rd Ghizlane ElBoussaidi
École de Technologie Supérieure
Montreal, Quebec, Canada
ghizlane.elboussaidi@etsmtl.ca

4th Gabriel C. Ullmann
Concordia University
Montreal, Quebec, Canada
g_cavalh@live.concordia.ca

5th Yann-Gaël Guéhéneuc
Concordia University
Montreal, Quebec, Canada
yann-gael.gueheneuc@concordia.ca

***Abstract*—As the complexity and scope of games increase, game testing, also called playtesting, becomes an essential activity to ensure the quality of video games. Yet, the manual, ad-hoc nature of game testing leaves space for automation. In this paper, we research, design, and implement an approach to supplement game testing to balance video games with autonomous agents. We evaluate our approach with two platform games. We bring a systematic way to assess if a game is balanced by (1) comparing the difficulty levels between game versions and issues with the game design, and (2) the game demands for skill or luck.**

***Index Terms*—game, testing, automation, deep-reinforcement-learning**

I. Introduction

Game development is an iterative process [1]. Game developers start with a core mechanic, limited in scope, and then iterate, adding new features until they deem the game complete. For every new change in the game, game testers (playtesters) interact with the game and provide feedback to the game developers who then change the game. They use experimentation and trial-and-error to tweak the game mechanics and make it engaging for the players. Keeping the game challenging while avoiding boredom requires *balancing* it [2], which is hard to translate to the actual game specification and relies on empirical knowledge. This constant experimentation implies there are no clear requirements but a "vision" for games.

In this paper, we propose an approach to assess video game balance (semi) automatically. Instead of manually testing games, we propose an automated approach with autonomous agents to aid game developers assess the game's balance. Schell [2] lists 12 common types of balance in video games. In this paper, we focus on two balance types: *Challenge vs. Success* about keeping the player engaged considering the game difficulty and the player's skills and *Skill vs. Chance* about needing luck instead of skills to succeed. Games of skills are more like athletic contests (which player is the best?) while games of chance are more random. For example, dealing out a hand of cards is a pure chance but choosing how to play them is pure skill.

Similar to unit tests in continuous integration pipelines in traditional software development, with which a system warns developers when a test fails, we want to provide developers with an automated process that would warn them when a new version of their game is too far from a given balance. Our approach brings a systematic and autonomous way to identify if the game is balanced by (1) checking the difficulty spikes and (2) if the game demands more skill or luck.

To do so, we train autonomous agents using Deep Reinforcement Learning (DRL). Training agents to play games with DRL is an ever-increasing research area [3] and machine learning models allow autonomous agents to master games. Video games offer an environment with reduced scope (compared to real life) that suits the training of autonomous agents. These approaches show great success in mastering simple and complex games [4]. In our approach, we incorporate the agents into the game development process and provide a solution for testing the game balance. Most game studios, especially the small ones, do not have the time or the budget to adopt complex and costly solutions. Yet, they can benefit from a more feasible approach.

In the following, we describe our approach, how we implemented it, and how we validated our testing approach with two platform games. Section II discusses the related works and how our approach differs from them. Section III presents the approach. Section IV shows how we technically implemented the approach. Section V and Section VI are the case studies that use our approach. Section VII presents the discussion and Section VIII the Threats to Validity. Finally, Section IX shows the conclusion and future works.

II. Related Work

We read 199 papers about video game testing[1] but, for the sake of space, here we will show the ones more close to the our testing objective: video game balancing. In the following we summarize, to the best of our knowledge, all the papers that relate to video game balancing.

Isaksen, Gopstein, Togelius, *et al.* [5] used A-Star, MCTS and visualization to determine the game difficulty by verifing different attributes of the same game without changing its rules. The authors use metrics to predict the score of a platformer called Flappy bird. Gudmundsson, Eisen, Poromaa, *et al.* [6] tested the difficulty of a game level using autonomous agents, powered by a CNN, to simulate gameplay and the "success rate" against human players. They validated using a Match-3 game. They wanted to predict the difficulty of a new game level automatically. They claimed that the difficulty of new levels could be tested automatically. Roohi, Relas, Takatalo, *et al.* [7] predicted the pass rate (win the game) and churn (abandon the game) in new levels of a Match-3 game. They used gameplay data from autonomous agents and playtesters. The agents were trained using PPO within the Unity-ML engine. They claimed that the pass rate and churn of new levels could be tested automatically.

DeLaurentis, Panchal, Raz, *et al.* [8] defined a framework that predicted the game balance using data from autonomous agents, trained with CNN and MCTS, to play against each other in a RTS multiplayer game. They assessed the play-session with a visualization tool together with the actions performed by the agents. Pfau, Liapis, Volkmar, *et al.* [9] and Pfau, Liapis, Yannakakis, *et al.* [10] used Deep Player Behavior Modeling (DPBM) and data from real players to model autonomous agents and play a MMO game. The idea was to replicate the human behaviour and then assess the playsession manually. de Mesentier Silva, Lee, Togelius, *et al.* [11] and Mesentier Silva, Lee, Togelius, *et al.* [12] created agents using A-Star and MCTS to explore a board game called Ticket to Ride. The focus was more exploring than dealing with the game balance.

Liu, Chaoran, Yue, *et al.* [13] used procedural content generation to create new tower defense game levels to then playtest it autonomously using flat MCTS. Morosan and Poli [14], [15] used evolutionary algorithm to balance different games, like Pacman and StarCraft, using the win-rate as oracle. Preuss, Pfeiffer, Volz, *et al.* [16] used the feature of the open-source game OpenRA, an RTS, to balance the strategies the player can adopt.

III. Approach

We describe, now, our approach. Figure 1 shows the process of the feedback loop of manual game testing and how our approach automates part of it. There are three roles involved in this process: The *Game Developer (GD)*, which refers to Game Designers and Programmers. It is the game developer who describes and implements how things should behave in the game. The *Game Tester (GT)* (Gameplay Tester, Playtesters or QA) find bugs and any other abnormality in the game. Game testers should test game quality by verifying gameplay, logical consistency, observability, progressive thinking, reasoning ability, and exhaustively testing features, game strategy, and functionality [17]. Therefore, game testers should understand the principles and the characteristics behind games and especially understand the game development context [18]. The *Testing Agent (TA)* is the autonomous agent that interacts with the game and reports the findings according to the test objectives.

A. [GD] Modifies and Generates the Game

The modifications in the game vary according to the game tester's feedback. For example, the game developer can simply tweak a parameter, like character speed, or introduce new gameplay mechanics, like the ability to jump. The bigger the change, the bigger the impact on the player's experience.

B. [GD] Sets the Test Objectives

The test objective depends on the modification made to the game. It is defined by the game developer and varies in scope. For example: *find bugs*, *explore the levels*, *check the character collision*, *verify if the level can be completed*, among many others.

As video games have a large scope , developers isolate parts of the game to test. This strategy is similar to what Rare does with the game Sea of Thieves [19] where they use single scripted actions to verify the object's behaviour, like opening a door, for example.

C. [GD] Trains the Agents

Playing the game is a sequential decision-making process, where the players continuously make decisions and take actions based on received observations. Therefore, this problem can be modelled as a Markov Decision Process (MDP) [20]. The *Agent (Player)* interacts with the *Game System*. The *Game State* is the observation (representation) of the Game System. What the Agent can "see" of the game. The Game Action is a set of possible decisions (move, attack, jump, etc). Finally, *Agent's Reward* is the feedback used to measure the success or failure of the agent's actions in achieving some goal (winning, surviving, etc). The autonomous agent (or model) is the output of the training process. It is the autonomous agent who interacts with the game.

D. [GT & TA] Interacts with the Game

Testing a video game means playing it [19]. For every new change in the game, game testers interact with the game and provide feedback to the game developers who then change the game. This trial-and-error process relies on the empirical knowledge of the team and could be faster, more scalable, and more efficient. Indeed, as developers perform multiple changes or permutations thereof, keeping track of what works best for the game quickly becomes overwhelming.

[1]https://doi.org/10.5281/zenodo.7768876

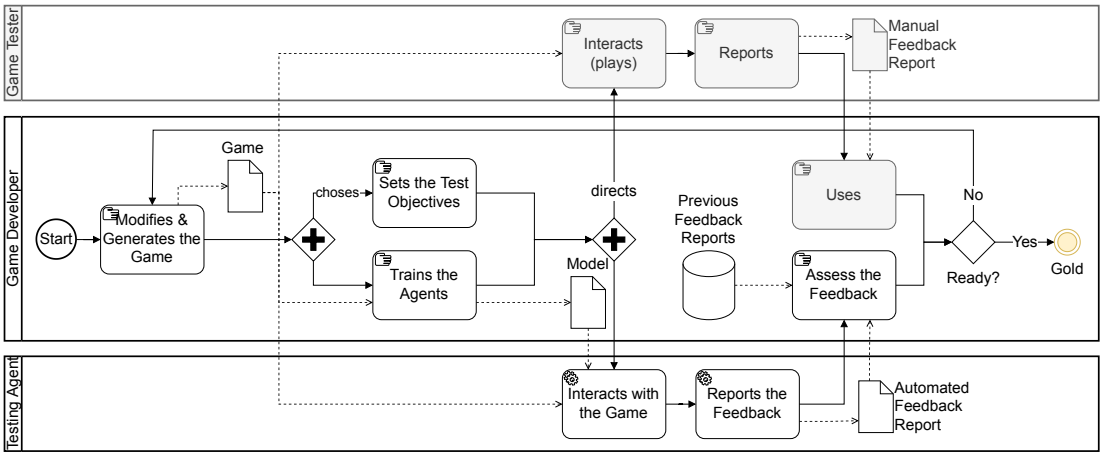

Figure 1. The testing process (in BPMN notation). The activities and artifacts show a typical manual process of game testing and how our approach complements it.

E. [GT & TA] Reports the Feedback

While the autonomous agent interacts with the game, we collect the metrics related to the testing objective. Game testers and testing agents have different testing objectives. For example, humans can assess subjective details of the game, like engagement heuristics, while autonomous agents can handle trivial details that are a burden for the human tester, like repetitive checks in the game versions.

F. [GD] Assess the Feedback

The game developer uses the feedback report to decide if the game is good enough to be released. Otherwise, they restart the process by making new modifications to the game. After receiving the feedback containing the data about the agent's performance, they can compare it with the previous versions of the game.

IV. Implementation

We now describe how we implemented our approach. One of the issues was how to separate the concerns between the game code, the libraries and frameworks (game engine), the DRL training code, and the testing detail (number of runs, logs, etc). Figure 2 shows the UML-like diagram of our architecture. We separated each class and made them responsible for only one job. We started with the first element, the *game logic* which is the game source code and its related assets. The game uses "Pygame" as *framework* (game engine). Then, we used two other *libraries* so we can use DLR methods to create the autonomous agents: Gym and StableBaselines. The *training logic* is the element that handles the training parameters like the reward function. To link the game and training logic, we used some *instrumentation* code based on the design patterns Observer and Command. Finally, we use another element to manage the *testing code*.

A. [GD] Modifies and Generates the Game

We simulate the development process by considering different versions of a game. The changes in version#1 are carried to the further versions. To have different versions of a game, we modify the game according to its game mechanics. We modify the game difficulty using the *Doubling and Halving* balancing methodology [2]. It says the developer should change the game attributes by high amounts: doubling or halving them. The objective of this method is to change something so that you can actually feel the difference right away.

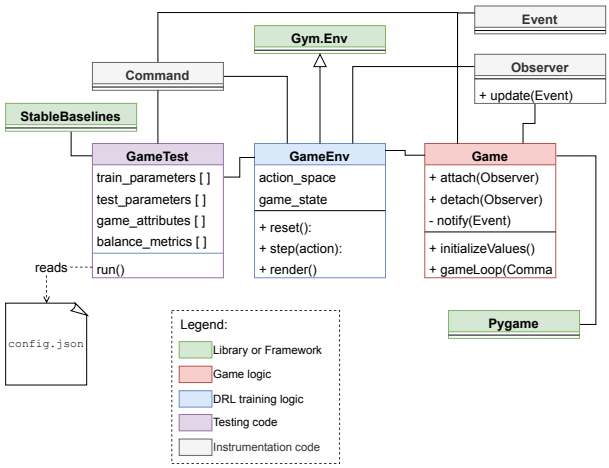

Figure 2. Architecture of our approach.

B. [GD] Sets the Test Objectives

We focused our testing feedback on two game balance types: *Challenge vs. Success* is about keeping the player engaged considering the game difficulty and the player's skills; *Skill vs. Chance* is about games that depend, more or less, on luck instead of the players' skills to succeed. To better assess the game we use game testers (humans and agents) with different skill levels: *novice* and *professional*. Also, we added one random agent that performs actions without any reasoning.

C. [GD] Trains the Agents

Our approach uses DRL to train the models and create the Autonomous Agents. We use the game system to define (a) how to represent the game state and (b) what will be the Agent's Reward. These two details are specific to the game being tested and can be done in many ways.

To train the agents, we use a Python library called *Gym*. It provides a number of already defined environments, which are games with defined action space and reward functions. This library is usually used to assess new machine-learning models.

Using Gym, we defined a custom environment that includes an *action space* with the commands that are possible within the game, and a *reward* function. We also used *Stable Baselines* library, providing a selection of machine learning models, like *PPO* and *A2C*, to train the agents to play the game.

We divided the skill level of the agents by training time, that is, the *novice* is trained with *100K steps* while the *professional* with *1M steps*. The machine used to run the training has a CPU Core i7 2.6 GHz, A GPU NVIDIA GeForce RTX3070, 32 GB DDR4 or RAM, and an SSD hard drive. As for the software, we use the Python language with the Stable Baseline library running on Windows with NVIDIA CUDA.

To train the autonomous agents we used *Training Parameters* (Table I). For our experiments we chose to use the model-free DRL `models` because (1) in our game environment, we cannot predict state transitions and rewards, and (2) the training cost (computation time) is lower. Thus, we chose two different models to train the agents: *PPO* and *A2C*. The `action space`, `reward function`, and `observation space` vary from game to game.

Table I
THE TRAINING PARAMETERS USED TO TRAIN THE AUTONOMOUS AGENTS.

Params	Type/Value	Description
`train`	Boolean	Re-train or not the agents
`model`	String	The machine learning model used to train the agents
`action space`	Array	String indicating the action of the game (LEFT, ATTACK, etc)
`reward function`	Float	Values, positives and negatives, defined by an heuristic
`observation space`	Array	The state of the game, what the agent knows and "see"

Depending on the modification of the game, it is necessary to re-train the models (agents) for each new version. For example, when a new game mechanics heavily modifies the gameplay.

We used human players to validate the performance of the agents. The agent must have a similar performance compared to the human players. Also, the performance across different versions of the game should follow the same pattern/trend.

D. [TA] Interacts with the Game

We defined a play session of 180 seconds where the autonomous agent plays the game two times. We use the median of the two runs to calculate each performance.

Table II shows the *Testing Parameters* used in the game test scenarios. The player plays the game for *N* seconds, *M* times. Either a human or an agent plays the game. Each one is assigned a skill level, with the exception of the random agent. The human professional is someone with gaming experience while the novice is not a regular game player.

Table II
TESTING PARAMETERS.

Parameter	Type/Value	Description
`time`	Integer	The time to be played in seconds
`run`	Integer	Number of runs
`session`	{human, ai-play, random}	The player of the session
`skill`	{novice, professional}	The skill level of the player
`version`	String	The game versions

E. [TA] Reports the Feedback

As we are tackling the issue of balancing the game, we created metrics related to the score of the agent playing the game. These metrics are variables that work as a proxy for the agent's performance and, therefore, the game balance. They vary for each game but, in general, are related to the score of the game.

F. [GD] Assess the Feedback

We compare the balance metrics across the different versions of the game:

- To balance the *Challenge vs. Success* we check spikes on the balance metrics when the agent plays in each game version. Also, we compare the performance of novice and professional skill levels.
- To balance the *Skill vs. Chance* we compare the performance of the random agent with the other trained agents. If the random agent performs better, the game has a balance problem.

V. CASE STUDY A. BATKILL

For Case Study A, we modified a 2D action platformer open-source game called Batkill[2]. We did not touch the rules of the game. It consists of a single screen, where the character tries to stay alive while bats fly toward him. The goal is to kill as many bats as possible without being hit. For each bat killed, the player gets one point (`+1 score`). The player loses one life for each hit (`-1 life`). The bats spawn faster as the players kill them. The bats spawn in random locations. The actions are `LEFT, RIGHT, ATTACK, JUMP`.

A. [GD] Modifies and Generates the Game

We created five different versions of the game Batkill (Table III) by changing a set of variables in the game: `bats` is the number of enemies on screen; `bat_speed` is the enemies' movement speed; `attack_cooldown` is the time between the character's attacks; `jump` if the character can jump or not. We expect an increase in the difficulty on versions #2, #3, and

[2]https://github.com/python-aficionado/batkill

#4. On version #5, with the addition of the jump mechanic, the difficulty should decrease as the player has one more resource to avoid enemies.

Table III
THE GAME VERSIONS FOR THE GAME BATKILL.

Version	bats	bat_speed	attack_cooldown	jump
#1	2	3	10	FALSE
#2	3	6	10	FALSE
#3	3	6	10	FALSE
#4	3	6	15	FALSE
#5	3	6	15	TRUE

B. [GD] Sets the Test Objectives

To assess the balance of the game Batkill, we measure two variables. `bats_killed`, which is the number of enemies killed by the character; and `hits_taken`, which is the number of times an enemy hits the character. To have a balance between kills and hits, the balance metric (`score`) for the game Batkill is given by: $score = bats\_killed - hits\_taken$.

C. [GD] Trains the Agents

To train the agents to play the game Batkill, we started with giving rewards when an enemy is killed (`BAT_KILLED +5`) or a hit was taken (`HIT_TAKEN -5`). After that, we used a small negative reward (`ATTACK -0.1`) to avoid making the agent use the attack for no reason. After seeing humans playing the game, we verified that they did not jump often. That is why we penalized the agent if he jumped (`JUMP -0.2`). The idea is to keep the character on the ground for more time. We also gave a small reward when the player moved toward the enemy (`MOVING_TOWARDS +0.1`). Finally, we gave a small reward if the agent faced the nearest enemy (`FACING_NEAREST_BAT +0.2`). That is a movement humans do naturally, and we try to hint here.

The PPO model had better performance (bigger reward) in all versions when the steps were more than 100K. For the PPO model, training a "novice" agent took around six minutes, while the "professional" took around one hour. The A2C model took about 10% more time to do the same training process.

D. [TA] Interacts with the Game

The trained agent interacts (plays) with the game autonomously. The behaviour of the agent resembles a human playing the game. However, the agent performs movements that are uncanny for humans. That is, a human player would never play like that.

For example, even after putting a penalty for it in the reward functions, the agent used *jump* and *attack* constantly whether the enemy was near or not. Nevertheless, there were moments the agent stayed still, like a human player.

E. [TA] Reports the Feedback

The Table IV shows the results of the feedback report for the PPO and A2C agents, as well as the Random agent and the Human player. The score of each agent is also separated by skill level. The negative values mean that the agent/player got hit more times than kills.

Table IV
MEDIAN OF SCORE (`BATS_KILLED - HITS_TAKEN`) FOR THE GAME BATKILL.

Version	Human		Agent PPO		Agent A2C		Random
	Pro	Novice	Pro	Novice	Pro	Novice	
#1	78	59	18	23	29	13	-13
#2	21	6	-7	7	-44	-47	-27
#3	-67	-86	-53	-63	-112	-122	-73
#4	-74	-92	-96	-86	-121	-123	-98
#5	-36	-1	-40	-47	-56	-51	-56

F. [GD] Assess the Feedback

Figure 3 shows the results of the humans and agents playing the game. The `score` is the average between novice and professional players/agents. Version#1 is the easiest because the agents and human players had the best score among all other versions. Version#4 is the hardest, followed by versions #3, #5, and #2.

In all game versions, the human outperforms any other autonomous agent, except on version#3. The PPO agent is the one that plays similarly to the human player, as the score across the game versions follows a similar pattern. In general, the A2C agent results are the lowest across all versions, except in the first version.

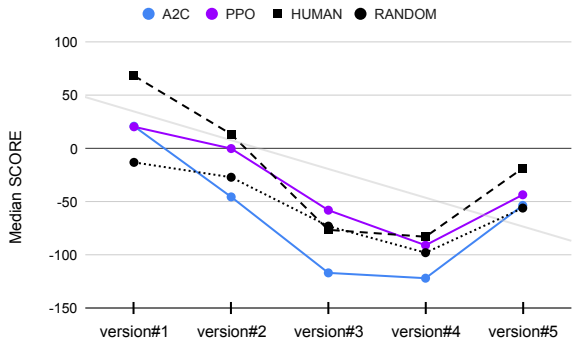

Figure 3. Results of all agents on Case Study A. Batkill.

1) Challenge vs. Success: We can identify three big spikes in difficulty. The first two spikes made the game harder while the last one made the game easier:

- **Difficulty Spike**: on game version#2, when we increase the number of enemies, the game got *harder*.
- **Difficulty Spike**: on version#3, when we increase the speed of the enemies, the game got *harder*.
- **Difficulty Spike**: on version#5, when we added the "jump" mechanic, the game got *easier*.

2) Skill vs. Chance: We identified two game versions where luck was more important than the player's skill. In those cases the random agent had similar (or better) performance to the other agents and human players:

- **Chance**: on version#3, when we increase the number of enemies, the game depends more on *luck* than skill.
- **Chance**: on version#4, when we decreased the time between the character attacks, the game depends more on *luck* than skill.

Summary - Case Study A. Batkill

- The PPO agent is the one that follows a similar pattern as the human players.
- Across the game versions, we found two spikes of difficulty that made the game harder and one spike that made it easier.
- The performance of the random agent on versions #3 and #4, compared to the other agents, shows that these two versions do not favour the player's skill. However, the introduction of jumping on version#5 helped making the game more skill-based.

VI. Case Study B. Jungle Climb

We modified an open-source, 2D "infinite-runner" platform game[3] called Jungle Climb[4]. It consists of a single screen, where several platforms are drawn. Each platform has gaps randomly generated. The player character is initially placed below the platforms and has to climb them by jumping between gaps. After passing through the second platform, the screen will start scrolling upward. The objective of the player is to keep the character below the bottom line of the screen as long as possible, not letting the player be "scrolled down" for too long with the screen.

For each step, the player is able to survive while above the second platform, it gets one point (`+1 point`). The scrolling speed of the platform increases as time passes. The actions are `LEFT`, `RIGHT` and `JUMP`.

A. [GD] Modifies and Generates the Game

We created three different versions of the game Jungle Climb (Table V) by changing two variables in the game: `shift_speed` is the rate the screen scrolls up; and `max_gaps`, which is the maximum number of gaps in each platform. We expect version#2 to increase the difficulty of the game and version#3 to make the game easier.

B. [GD] Sets the Test Objectives

To assess the balance of the game Jungle Climb, we measure two variables: `max_points`, which is the maximum number of points that shows how long the character was alive; and `max_correct_jumps`, which is the maximum of correct jumps, that is, when the jump connects to the next platform.

[3]https://github.com/gamedev-studies/jungle-climb

[4]https://github.com/elibroftw/jungle-climb

Table V
The game versions - Jungle Climb

Version	`shift_speed`	`max_gaps`
#1	1	1
#2	2	1
#3	2	2

The `score` is given by the rate between points and correct jumps: $score = max\_points + (max\_correct\_jumps * 100)$.

C. [GD] Trains the Agents

To train the game Jungle Climb we use the following set of rewards. At the beginning of every step, we compute the `Below Threshold Reward (BTR)`. If the character has not passed the threshold of the screen where it starts to shift, the score will always be zero and the BTR will be greater than zero. We use this value so we can give the agent different motivations so it can exit this initial state more quickly: `BTR = time_elapsed * 5 if score == 0 else 0`.

D. [TA] Interacts with the Game

The trained model (agent) interacts (plays) with the game autonomously. However, aside from our efforts to train the model, the behaviour of the agent does not resemble a human player. The agent jumps continuously and, most of the time, without purpose.

E. [TA] Reports the Feedback

The Table VI shows the results of the feedback report for the PPO and A2C agents, as well as the Random agent and the Human player. The score is separated by skill level.

Table VI
Median of `score` for the game Jungle Climb.

Version	Human		Agent PPO		Agent A2C		Random
	Pro	Novice	Pro	Novice	Pro	Novice	
#1	3262	2890	3597	1576	788	407	1371
#2	1908	1712	2251	850	519	305	683
#3	1885	1591	2154	1102	100	338	615

F. [GD] Assess the Feedback

Figure 4 shows the results (score) of all the agents playing the game versions. Version #1 is the easiest to play. The difficulty across the versions increases on version#2 and maintains on version#3. Even by adding one more gap to jump into each platform.

The curve for the PPO agent and Human players are very similar, which shows that this agent is a good proxy to mimic human players. The PPO agent outperforms, in all three versions, all the other agents. On the other hand, the A2C agent performs poorly, even when compared to the random agent.

1) Challenge vs. Success: We could identify one big spike in difficulty:

- **Difficulty Spike**: on game version#2, when we increase the speed the screen shifts, the game got *harder*.

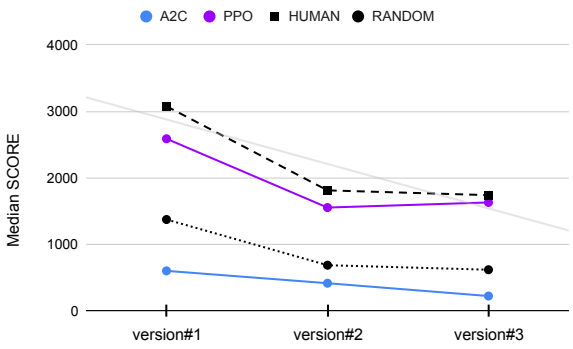

Figure 4. Results of all agents on Case Study B. Jungle Climb.

2) Skill vs. Chance: The random agent has a similar pattern to the human player but with a very low score across all game versions. This indicates that the game does reward the player's skill in all versions. Therefore:

- **Skill**: on versions #1, #2, and #3, the game depends more on *skill* than luck.

Summary - Case Study B. Jungle Climb

- The performance pattern of the PPO agent is similar to the human players.
- There is a spike in difficulty on version#2 that is not balanced on version#3 by adding more gaps to jump.
- The random agent does not outperform the human or the PPO, showing that the game is skill-based.

VII. Discussion

A. Balancing Challenge vs. Success

To deliver a good experience to the players, keeping them away from boredom or anxiety, game development requires trial and error and much experimentation. Our results show that it is possible to systematically automate part of the balance testing process. In our case studies, we showed that the agents could mimic the player's achievements and struggles, even without re-training the models. Thus we showed that it is possible to rely on autonomous agents to proxy the human's skill levels. These agents identified the spikes in difficulties among the versions. This setup provided quick feedback that game developers can use to promote changes in the game design.

B. Balancing Skill vs. Chance

Games are complex systems that mix mechanics that demand both skill and luck (chance). Finding the "sweet spot" requires a subjective aspect that only experienced developers have. Using the random agents proved to be useful in spotting when the game is rewarding, or not, the player's skills. When compared to the trained agents and humans, we noticed that, in certain versions, the random agents got the best results. With a closer look at the metrics, these exact same versions were the most difficult to play. The game developers can use the information to tweak the game accordingly, that is, make the game more skill-based or chance-based.

C. Expectation vs. Reality

With Case Study A we confirmed our design choices (Section V-A) that adding the jump ability would reduce the game difficulty, as it would add a new way to avoid getting hit. However, in Case Study B, we assumed that adding a new gap in the platforms would facilitate the climbing, which was not the case. This shows that the game developer's assumptions must be properly tested in practice. Our approach helps this process.

D. Training the Agents and Balance Issues

Training the agents demands an initial effort that pays off later in development. As each game has different mechanics, it takes time to discover which rewards will result in an agent playing the game well, or close to the human performance. While trying different rewards, the game shows some of its balance issues. For example, even a simple game like in Case Study B. Jungle Climb requires an effort to make the agents play reasonably well.

E. Game Testers and Effort

In our case studies, we asked two people to play the game for three minutes for each version, twice. Even with these two simple game scenarios and a few minutes of gameplay, both of the players reported tiredness after playing each game. This adds to the fact that manual testing on video games is tiresome and demands full concentration. In the game industry, game testers work for hours every day on the same game checking the same issues multiple times. The automation in the game testing aid these testers as well, so they can focus on subjective details of the game.

F. Deterministic vs. Stochastic

The two games used in our experiments are stochastic, they contain random gameplay elements that are not possible to predict. e.g., the bats in Case Study A and the gaps in Case Study B. Games like these pose difficulties when training the agents to play the game. As randomness is a feature in the games, automating the testing is even more valuable for these cases.

G. Cost of Creating the Reward Function

To make the DRL agents play the game, we must give rewards (or penalties). However, finding a combination of rewards that makes the agents play the game properly requires trial and error. Ultimately, the cost to produce "heuristics" to reward agents should be lower than creating a simple script that performs actions to the game. Thus, the process of creating cost-effective reward functions is an open challenge yet.

VIII. Threats to Validity

We faced issues during the implementation of our approach that can become obstacles to its adoption. For example, creating the testing architecture (Figure 2) demands an initial effort. Although this investment pays in the long run, we understand that spending engineering time with tooling instead of the game itself might not be welcomed by developers. However, many libraries allow a rapid configuration of the environment. In our cases, we used Python libraries (Gym and Stable Baselines) to speed up the process of training.

Another issue is to define reward functions so that the agents can play the game properly. Games have different mechanics and even games within the same genre, e.g. platformers, do not have common rewards. Yet, scenarios within the same game can share the same rewards. For example, in larger games, we could split the chunks of the scenarios the developer wants to test and reuse the rewards with fewer modifications.

IX. Conclusion

In this paper, we proposed and implemented our approach to automate game testing to balance video games using autonomous agents. We described the process of training the agents, playing the game, and assessing the game balance. We focused on two game balance types: *Challenge vs. Success* and *Skill vs. Chance*. We validated our testing process with two platform games. We found difficulties spikes in the game versions and identified the versions which demanded more (or less) player skill.

Game development often relies more on the developer's "feeling" rather than on any game specification. The result is an empirical manual cycle of development and testing. Although replacing manual testing is not possible, game developers can adopt a development pipeline with automated testing that provides quick feedback about the game balance.

Our approach brings a systematic and autonomous way to identify if the game is balanced by (1) checking the difficulty spikes and (2) if the game demands more skill or not. We believe our approach is a step toward providing steady game development and games with better quality.

For future work we want to expand the scope of the experiments and find easier ways to train the agents. We also want to add more skill levels for the agents aside from different ways to play the game, that is, a player profile/persona.

References

[1] C. Politowski, L. Fontoura, F. Petrillo, and Y.-G. Guéhéneuc, "Are the old days gone?: A survey on actual software engineering processes in video game industry", in *Proceedings of the 5th International Workshop on Games and Software Engineering - GAS '16*, Austin, Texas: ACM Press, 2016. DOI: 10.1145/2896958.2896960.

[2] J. Schell, *The art of game design: a book of lenses*, en. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2008, ISBN: 978-0-12-369496-6.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, *Playing atari with deep reinforcement learning*, 2013.

[4] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, "Deep Learning for Video Game Playing", *IEEE Transactions on Games*, 2020. DOI: 10.1109/TG.2019.2896986.

[5] A. Isaksen, D. Gopstein, J. Togelius, and A. Nealen, "Exploring Game Space of Minimal Action Games via Parameter Tuning and Survival Analysis", *IEEE Transactions on Games*, 2018. DOI: 10.1109/TCIAIG.2017.2750181.

[6] S. F. Gudmundsson, P. Eisen, E. Poromaa, *et al.*, "Human-Like Playtesting with Deep Learning", in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2018. DOI: 10.1109/CIG.2018.8490442.

[7] S. Roohi, A. Relas, J. Takatalo, H. Heiskanen, and P. Hämäläinen, "Predicting game difficulty and churn without players", in *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, 2020.

[8] D. A. DeLaurentis, J. H. Panchal, A. K. Raz, *et al.*, "Toward automated game balance: A systematic engineering design approach", in *2021 IEEE Conference on Games (CoG)*, 2021. DOI: 10.1109/CoG52621.2021.9619032.

[9] J. Pfau, A. Liapis, G. Volkmar, G. N. Yannakakis, and R. Malaka, "Dungeons Replicants: Automated Game Balancing via Deep Player Behavior Modeling", in *2020 IEEE Conference on Games (CoG)*, 2020. DOI: 10.1109/CoG47356.2020.9231958.

[10] J. Pfau, A. Liapis, G. N. Yannakakis, and R. Malaka, "Dungeons amp; replicants ii: Automated game balancing across multiple difficulty dimensions via deep player behavior modeling", *IEEE Transactions on Games*, 2022. DOI: 10.1109/TG.2022.3167728.

[11] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "AI-based playtesting of contemporary board games", in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, Hyannis Massachusetts: ACM, 2017. DOI: 10.1145/3102071.3102105.

[12] F. de Mesentier Silva, S. Lee, J. Togelius, and A. Nealen, "Ai as evaluator: Search driven playtesting of modern board games", in *AAAI Workshops*, 2017.

[13] S. Liu, L. Chaoran, L. Yue, *et al.*, "Automatic generation of tower defense levels using PCG", in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, ACM, 2019. DOI: 10.1145/3337722.3337723.

[14] M. Morosan and R. Poli, "Automated Game Balancing in Ms PacMan and StarCraft Using Evolutionary Algorithms", in *Applications of Evolutionary Computation*, G. Squillero and K. Sim, Eds., vol. 10199, Springer International Publishing, 2017. DOI: 10.1007/978-3-319-55849-3_25.

[15] M. Morosan and R. Poli, "Lessons from Testing an Evolutionary Automated Game Balancer in Industry", in *2018 IEEE Games, Entertainment, Media Conference (GEM)*, 2018. DOI: 10.1109/GEM.2018.8516447.

[16] M. Preuss, T. Pfeiffer, V. Volz, and N. Pflanzl, "Integrated Balancing of an RTS Game: Case Study and Toolbox Refinement", in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 2018. DOI: 10.1109/CIG.2018.8490426.

[17] S. Aleem, L. F. Capretz, and F. Ahmed, "Critical Success Factors to Improve the Game Development Process from a Developer's Perspective", *Journal of Computer Science and Technology*, 2016. DOI: 10.1007/s11390-016-1673-z.

[18] R. E. S. Santos, C. V. C. Magalhaes, L. F. Capretz, J. S. Correia-Neto, F. Q. B. da Silva, and A. Saher, "Computer games are serious business and so is their quality", in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM 18*, 2018. DOI: 10.1145/3239235.3268923.

[19] C. Politowski, F. Petrillo, and Y.-G. Gueheneuc, "A Survey of Video Game Testing", in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021. DOI: 10.1109/AST52587.2021.00018.

[20] Y. Zheng, X. Xie, T. Su, *et al.*, "Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning", in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019. DOI: 10.1109/ASE.2019.00077.