SOEN 6971 Report

Amr Abdalla
Concordia University

***Abstract*—This report presents work on a current project that introduces a game engine specializing in 2D platformer games with an integrated deep reinforcement learning (DRL) tool designed to assist playtesters and level designers in evaluating and balancing game levels. The first part of the report is focused on reinforcement learning, where I explored hyperparameter tuning and reward function modifications to address the problem of agents failing to complete simple test levels. The results indicated that the main cause was issues with the observation space of the agents. In the second part of the work, I focus on the design and development of a statistics observer system using aspect-oriented programming to separate gameplay logic from data collection. I initially proposed a Decorator-based implementation which was later replaced with a YAML configuration approach that allows developers to define tracked gameplay events, recorder functions, and parameters externally, making the collection of gameplay metrics modular and extensible. The resulting system allows developers to create games, train DRL agents, and gather gameplay data that gives insights on gameplay issues, such as unattainable goals and an imbalance in difficulty.**

***Index Terms*—Deep Reinforcement Learning, Aspect-Oriented Programming, Game Testing, Game Balancing, Observer Pattern, Decorator Pattern.**

I. General Introduction

Game balancing and testing are challenging and time-consuming issues. While playtesters can find many issues related to game levels, they cannot possibly test levels exhaustively, and they play much more than the average player, making their experience different from that of players, which could hinder their judgment.

This project was initially developed by Cristiano Politowski, Al Shifan, and Kevin Chua at Ontario Tech University, and I joined it at a later stage. The goal of this project is to create a tool powered by deep reinforcement learning (DRL) to assist playtesters and level designers in finding insights and potential problems related to game level design. Potential applications of the tool include balancing game difficulty modes, such as having different hard and easy modes for the same game; automatically finding issues with level design, such as unreachable goals or mispositioned enemies; allowing DRL agents to propose changes to designed levels or create new levels given certain criteria; and even helping developers learn about reinforcement learning, similar to how Stable Baselines3 Gymnasium works [1], [2].

The scope of this project is limited to 2D platformer games, such as Mario, Sonic, and Super Meat Boy. While this is a limited scope, it can provide feedback on the practicality of this goal, and much of the work can later be expanded to handle additional types of games. In addition, it is not sensible to start such a project on a large scale by trying to support all types of games.

To achieve this goal, we are developing a game engine with integrated DRL agents. This way, game developers can create games, design levels, and configure gameplay mechanics, such as movement speed and acceleration, and then have the agents try the levels and automatically provide measurements of predefined statistics, allowing game development and testing to happen within the same tool. We chose a game engine over simply integrating DRL agents into an existing project because, while platformer games have similar mechanics, available open-source projects implement them very differently and are not generic. Therefore, creating a game engine provides a unifying architecture that DRL agents can depend on while training, making the creation of a new game and the training of an agent on it easier and more efficient.

II. Reinforcement Learning

A. Introduction

When I joined the project, it already included a sample 2D platformer game where the player, whether it is a human or a DRL agent, controls

a character that can walk, run, jump, collect coins, and kill enemies by jumping on them. A YAML-based configuration was created to control player attributes, such as jump height and movement speed. The pipeline for training and testing agents was already present, but the agents were performing poorly and were unable to complete all of the defined levels, which were simple and easily completable by a human player.

I spent the first weeks on studying the fundamentals of reinforcement learning, as I had no prior experience in the field. The learning process involved reading the documentation of Stable Baselines3 [2], which we used to implement reinforcement learning within the project, and gaining familiarity with the project's existing codebase. It should be noted that my understanding of RL remains at a foundational level, limited to the concepts and practical usage required for this specific project.

Reinforcement learning is a machine learning approach where agents learn through their interactions with the environment and through trial and error. Deep reinforcement learning is the combination of RL with deep learning, where neural networks are used to help agents learn behaviors. For example, in a platformer game, a neural network would learn the movement patterns that result in high rewards from observations such as player and enemy positions. Like any other machine learning model, results in RL can vary from one run to another because training starts with random weights. To learn anything useful, a large number of samples is needed. For example, the number of steps used in the project to allow an agent to complete a level adequately, but not perfectly, is one million steps. During training, the agent is evaluated every n steps, where n is defined by the developer, and the average reward per step is used as an indication of its performance. This is usually a good indication, but it can also be misleading if the reward function is not well designed. There are different RL algorithms available in Stable Baselines3 [2]. The default algorithm, and the only one currently used in the project, is Proximal Policy Optimization (PPO). PPO is an algorithm that gradually updates the agent's strategy in small steps and is a standard algorithm in practical RL.

An agent's action space defines what the agent can do and can be either discrete or continuous. Not all algorithms work for all action space types. The project contains a multi-discrete action space, which means that the actions are discrete (jump, move left, move right, etc.), but an agent can choose multiple actions at the same time (jump + move left, jump + move right). Fortunately, PPO works with multi-discrete actions. In the context of the project, a persona is the reward function the agent uses. For example, a speedrunner persona is a function that rewards finishing the level as quickly as possible, while a coin collector persona rewards collecting coins more than finishing early. The observation space of an agent defines what the agent can see. This could range from gameplay elements such as enemies and coins to other helpful parameters, such as Dijkstra's pathfinding results.

B. Problem

The main problem during this phase of the project was that the DRL agents were performing poorly and were unable to complete the test game levels given to them. This is significant because the agents are the central component of the project. Without agents, the project would be incapable of generating metrics that help game designers with their levels. The project's educational purpose that allows learners to experiment with reinforcement learning would also be impossible without functioning agents. The project would simply be a 2D platformer game engine with nothing special.

This is a difficult problem because there are multiple factors that could potentially contribute to the agent's poor performance. The personas used during training, the observation space available to the agents, and the selected hyperparameters could all be affecting the agent's performance negatively. This makes identifying the exact source of the issue difficult. Furthermore, experimenting with reinforcement learning is time-consuming, because evaluating any modification would require retraining agents from the beginning in order to check the changes. This makes the process of identifying issues take days instead of hours.

C. Solution

The main task for me at this stage was to make the agents complete the test levels. I used two

approaches to address this problem: tuning hyper-parameters and modifying personas.

The first approach was to tune the parameters of the model itself. This was because several personas had already been implemented, and there was no obvious issue with them. In addition, the project was using the parameters provided in the Stable Baselines example project without modification. This introduced a new problem: changing a parameter value and then training an agent multiple times to observe the results were extremely time-consuming. To address this, I modified the training script to allow training multiple agents with different values for a given parameter automatically by taking as input the start value, a step value, the number of iterations, and the number of models to train per value. For example, if it is needed to tune the learning rate, the following values can be provided: a start value of 0.001 and a step value of 0.002 for 5 iterations, with 3 models trained per step. In this particular case, the script would train 15 models with learning rate values of 0.001, 0.003, 0.005, 0.007, and 0.009. While this did not reduce the total training time, it allowed multiple models to be trained overnight using a single command.

Another approach I tried was modifying the existing personas and checking the results. I did not create new personas because the current ones looked sound. For instance, the master persona rewarded collecting coins, killing enemies, progressing through the level, and winning the game, while punishing losing and progressing too slowly. Therefore, my focus was on improving the master persona instead. One problem that I discovered with the master persona was that the agent overshot the goal and became stuck at the end of the level past the goal. This happened because the goal position was not included in the observation space; the agent was only rewarded for progressing forward, and the goal was not located at the leftmost position of the level. As an attempt to fix this, a distance-to-goal variable was added to the observation space and the reward function was adjusted based on this value instead of simply rewarding forward progress. Figure 1 shows the implementation of the master persona with the distance to the goal considered.

```python
def master(score_inc: bool, terminated: bool, info: Info, score: int) -> float:
    """
    MASTER: Balanced—progress, coins, and kills.
    """
    if terminated and not info.get("won", False):
        return -1.3

    dx = float(info.get("dx", 0.0))
    delta_dist_to_goal = float(info.get("Dist_to_Goal", 0.0)) - float(info.get("last_dist_to_goal", 0.0))
    coins = int(info.get("coins_delta", 0))
    kills = int(info.get("kills_step", 0))
    powered = 1.0 if info.get("powered_up") else 0.0

    r = 0.0
    r += 1.0 * coins
    r += 1.8 * kills
    r += 0.002 * powered
    if delta_dist_to_goal < 0:
        r -= 0.01
    if dx < 0.05:
        r -= 0.01
    if info.get("won", False):
        r += 10.0
    return r
```

Fig. 1. The implementation of the master persona during testing. Instead of depending solely on dx (which discovers whether an agent moved further left), I depend on the change of goal distance, meaning how much closer or further to the goal an agent has reached each step.

D. Evaluation

To test the parameters, I modified the learning rate with values ranging from 0.0033 to 0.0073, increasing by 0.001 per iteration, with 2 models per iteration and with 1 million training steps. Figure 2 shows that while the overall reward results were positive for most models, there was no clear relationship between the models with different learning rates. In addition, the performance of all agents was similar; they were all failing at the same levels and getting stuck at the same positions.

Regarding the results of modifying the master persona, the issue of getting stuck past the goal was resolved, but the overall performance of the agent was still poor. In a certain level, the agent repeatedly jumped into pits, which should have been marked as dangerous, but nothing indicated that the agent perceived pits that way. This led us to believe that the issue was with the observation space rather than the reward function or the agent's parameters.

E. Conclusion

While trying the discussed solutions, other team members were working on reengineering the code for agent training and the spatial hashing of the game environment, which controls how the agent observes the environment. They also fixed an issue where not all data from the observation space was passed into the reward function and added more ways to inspect more closely which values were affecting the agent's training.

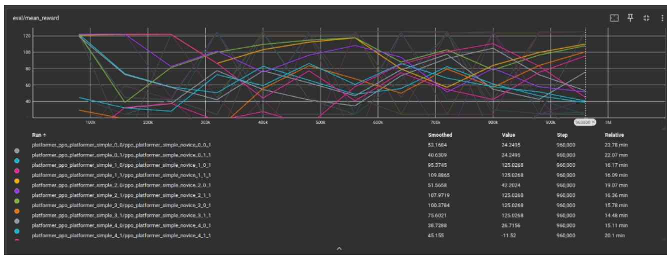


Fig. 2. The training results of testing different values of learning rate. The numbers at the end of each run name determine the index of the trained model, so ppo_platformer_simple_novice_0_0_1 means the first iteration of the first model, so this is the first model of learning_rate=0.0033. Results show there is no clear relationship between learning_rate and mean reward value

Because they were still in progress when the discussed solutions failed to make the agents complete the levels, I was tasked with testing sample RL projects from Gymnasium on my local machine to learn what they were doing differently. However, that approach failed quickly when I tested on Tetris, because it required multiple days of training, and nothing was obviously different in the Tetris project compared to this project.

In the end, because it was difficult to work with the other team on refactoring the project, the refactoring took a long time to complete, and no further progress could be made without it. I started working on the next part of the project: assuming agents learned to complete levels, how could we gather data from them so that they could actually help level designers and playtesters with level design?

III. Statistics Observer

A. Introduction

Having game levels and an agent that can complete those levels solves only part of the problem. It allows designing basic levels and learning more about RL, but that is not what is special about this project. Data still needs to be gathered to draw conclusions about level design. As an example of the final output, the system should be able to answer how difficult a level is by checking the success-to-failure ratio and the average time taken to complete the level for different agents with different personas and skill levels, where a skill level represents the number of training steps an agent went through.

The Observer design pattern was chosen for this task [4]. Statistics would be stored in files in a separate directory from the project, and would simply save the required data at different points determined by the developer. For example, whenever the jump function is called, the jump counter variable in the observer should increment by 1. While this by itself is easy, the main challenges are gathering such data while having minimal dependency between the game code and the observer code and making the observation of new data straightforward, quick, and optional.

To mitigate these challenges, I used aspect-oriented programming (AOP) [3]. AOP is a programming paradigm that aims to increase the modularity of a program by separating cross-cutting concerns. Instead of modifying the code directly, it adds behaviors by specifying which code is modified through a pointcut specification. A pointcut is a query that specifies points in the program called join points. Logging can be an example of AOP. A need to log multiple unrelated functions could arise. Modifying those functions would result in logging being scattered throughout the codebase and those functions becoming inflated and possibly harder to read. This would also result in lower modifiability, as changing the logging behavior could require changing all affected functions. AOP tries to solve this problem by allowing the developer to define cross-cutting concerns in modules called aspects [3]. Aspects contain the code joined to the specified points in the program, which would be the implementation of logging in the previous example, and their corresponding pointcuts. A developer would only be concerned with what to run, when to run it, and how to specify the join points in AOP. AOP also has pitfalls; for instance, making a mistake in the expression of cross-cutting concerns can lead to program failure, and renaming methods or variables can cause aspects to not work as expected.

B. Problem

The engine will have multiple games, so the statistics aspect should be independent of the game it is observing. Observables specific to a game should be specified in a configuration file by the developer of that game. Because using configurations depends on the names of modules and methods, and there are multiple developers in the project, issues are more likely to happen because it is generally

not expected that changing a method name would cause an error. In addition, statistics are not part of the core gameplay or training logic and should be completely optional, so failing should not cause the program to crash; it should only log a warning message. The idea behind the project is clear: to provide information about level design. However, the exact data to use and process, when to collect it, and how to process it are not known. Such a process depends on the game and what the developer aims to know about it, so the requirement here is to make collecting, storing, and processing data intuitive and simple.

C. Solution

We have explored two different solutions, using annotators and using configuration files.

1) Using Annotators: In the first iteration of the statistics observer, I used Python decorators. I created a class called `stats_observer` to hold variables representing the data of interest. For each piece of data, a corresponding record function existed to update it. For example, the number of jumps starts at 0, and a `record_jump` function increments it by one. Figure 3 shows a snippet of different recording functions.

Once the function was defined, its name could be used as the event type parameter in the tracking decorator. I defined a generic `track` decorator in `StatsObserver` that takes the record function name and automatically calls that function at the end of the tracked method. Returning to the example of tracking jumps, a developer would need to add `@track("jump")` before the jump function in the codebase, and the system would look for `record_jump` in `stats_observer` and call it. Therefore, to track a new metric, a developer would only need to add a new variable to `stats_observer`, add a corresponding record function for that variable, and add the `track` decorator depending on when they would like the record function to be called.

When a game level is completed or the game session is terminated, the `reset` method in `stats_observer` is called. This method calculates dependent data based on the tracked independent data. For example, it would calculate the success rate using `1 / (death_count + 1)`.

```python
class PlatformerStats:
    def __init__(self, world, goal_pos):
        self.world = world
        self.goal_pos = goal_pos
        self.cause_of_death = "Success"
        self.jumps = 0
        self.coins_collected = 0
        self.sum_vx = 0
        self.count_vx = 0
        self.max_x_seen = 0

    def record_jump(self):
        self.jumps += 1

    def record_death(self, cause):
        self.cause_of_death = cause

    def record_coins_collected(self):
        self.coins_collected += 1

    def record_horizontal_velocity(self, vx):
        self.sum_vx += abs(vx)
        self.count_vx += 1

    def record_max_x_seen(self, value):
        self.max_x_seen = value
```

Fig. 3. A snippet of PlatformerStats class that shows how variables can be initialized and how recorder functions can be defined.

player	game	world	cause_of_death	jump_count	coins_colle	avg_vx	progress_ratio	enemies_killed	elapsed_time
expert_adept	Mario	Mario1-1	Enemy	1	0	289.03	0.09	1	2.08
expert_adept	Mario	Mario1-1	Success	22	0	285.06	0.99	6	22.16
expert_adept	Mario	Mario1-1	Enemy	7	6	267.64	0.31	1	8.13
expert_adept	Mario	Mario1-2	Success	24	0	323.17	0.99	4	16.65
expert_adept	Mario	Mario1-2	Enemy	7	0	360.08	0.56	1	8.26
expert_adept	Mario	Mario1-1	Enemy	2	0	328.14	0.09	1	1.63
expert_adept	Mario	Mario1-1	Success	24	0	281.57	1	1	22.89

Fig. 4. A sample output for a csv file that the dashboard uses to show information to developers.

After that, it stores all relevant data in a CSV file, including the attempted level and the corresponding data. Figure 4 shows a sample CSV file output.

While this process allowed easy tracking of data with only three simple steps, it had several limitations. Firstly, because a developer had to add the `track` decorator directly inside the gameplay code, there was a dependency between the observer code and the game code. This also resulted in the decorators being scattered across the entire codebase, which defeated the purpose of AOP. To track simple metrics, I had to add decorators to the physics manager, the player script, and the core game script. Another limitation was the way variables were passed between the game and the observer. Not all metrics are as simple as tracking the number

of jumps, where the value is incremented by 1 whenever the jump function is called. For example, `record_death` requires passing the reason for death along with it. Decorators provide some ways to mitigate this issue, but each approach has drawbacks. Parameters from the original method can be passed to the recorder function, but this results in unnecessary parameters being passed and requires including parameters solely for statistical purposes that serve no purpose in the game code. To control which parameters are passed, they can be specified in the decorator, such as `@track("death", "reason")`, but this would not scale well if multiple parameters were needed. Finally, the way statistics worked during this phase was not generic and worked well only if the engine contained a single game, because all recorders and tracked variables were stored in one class: `stats_observer`.

2) Using Configuration Files: The final solution for tracking statistics I developed uses YAML files. Defining variables for metrics and recorder functions remains mostly the same, because attempting to automate them would take control away from developers. However, the difference is that there is a YAML file for each game and a corresponding observer class. It is expected that each game will have its own observer, and there can be a hierarchy of observers following an object-oriented design approach [4]. For example, there could be a base platformer observer class that acts as the parent for Sonic and Mario clone observers.

In addition to defining the target class, the YAML file also defines the target functions, which act as the pointcuts; the recorder, which acts as the aspect; and the parameters the developer would like to pass to the recorder. Using jumps as an example, the stats class would be the observer class containing all variables and recording functions, the target would be the `start_jump` function, the recorder would be the `record_jump` function in the stats observer, and the arguments to pass would be empty. Figure 5 shows a snippet of a sample YAML file. The overall architecture of this solution is shown in Figure 6.

```
stats_class: code.stats.MarioStats.MarioStats

tracks:
  - target: code.games.modules.Objects.Player.Player.set_horizontal_velocity
    recorder: record_horizontal_velocity
    args: [vx]

  - target: code.games.platformer_core.PlatformerCore._handle_death
    recorder: record_death
    args: [cause]

  - target: code.games.modules.Objects.Player.Player.start_jump
    recorder: record_jump
    args: []

  - target: code.games.modules.System.PhysicsManager.PhysicsManager._handle_player_kill_enemy
    recorder: record_enemies_killed
    args: []

  - target: code.games.modules.System.PhysicsManager.PhysicsManager._collect_coin
    recorder: record_coins_collected
    args: []

  - target: code.games.platformer_core.PlatformerCore._set_max_x_seen
    recorder: record_max_x_seen
    args: [value]

resets:
  - target: code.games.platformer_core.PlatformerCore.complete_level
  - target: code.games.platformer_core.PlatformerCore._handle_death
```

Fig. 5. A sample yaml file for simple metrics for a platformer game.

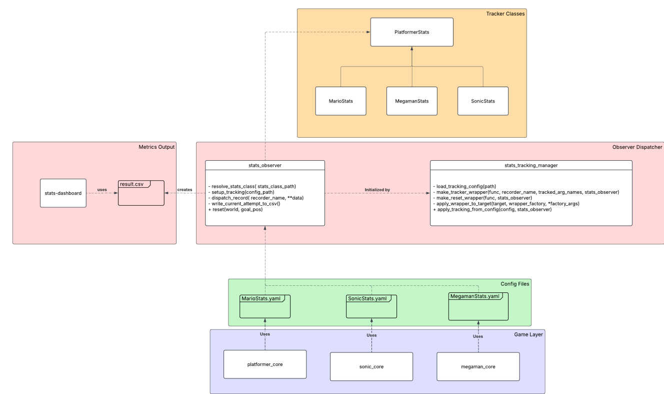
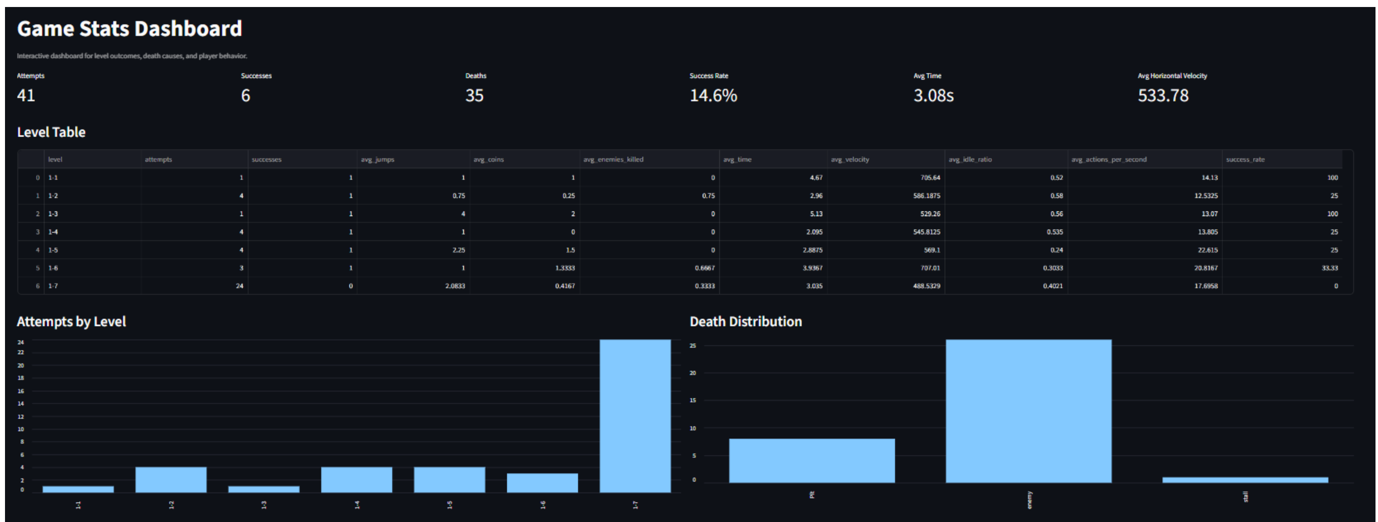


Fig. 6. Architecture of the final statistics observer.

D. Evaluation

I tested the data collection method on a sample platformer game in which the following metrics were tracked: number of jumps, coins collected, enemies killed, elapsed time, average horizontal velocity, number of deaths and causes of death, and the action performed at every step, for example walking left or running right. A sample dashboard was created to show the relationships between metrics and how an agent performed in each level. Figure 7 shows a snippet of how the dashboard looks.

There are some limitations to the final approach. Function parameters remain the only way to pass data from the game classes to the observer classes because the decorator can only read the function definition through reflection, so any variable defined within a function is inaccessible to the decorator. One way to mitigate this issue is to allow passing variables from the game class directly to the ob-



	level	attempts	successes	avg_jumps	avg_coins	avg_enemies_killed	avg_time	avg_velocity	avg_idle_ratio	avg_actions_per_second	success_rate
0	1-1	1	1	1	1	0	4.67	705.64	0.52	14.13	100
1	1-2	4	1	0.75	0.25	0.75	2.96	586.1875	0.58	12.5325	25
2	1-3	1	1	4	2	0	5.13	529.26	0.56	13.07	100
3	1-4	4	1	1	0	0	2.095	545.8125	0.535	13.805	25
4	1-5	4	1	2.25	1.5	0	2.8875	569.1	0.24	22.615	25
5	1-6	3	1	1	1.3333	0.6667	3.9367	707.01	0.3033	20.8167	33.33
6	1-7	24	0	2.0833	0.4167	0.3333	3.035	488.5329	0.4021	17.6958	0

Fig. 7. A snippet of the dashboard with metrics for developers that can be filtered by level.

server class, but that would create a new dependency between them and would be prone to failure because renaming anything in the game class could break the observer of that class.

In general, errors are more likely to occur with the configuration file approach than with the decorator approach because it relies on names as strings rather than annotating code directly. Because observers should never cause the game to crash, and failing silently is poor for debugging, whenever an exception occurs, the observer logs a warning to the console and skips recording that instance.

Another limitation is that the dependency between observers and game classes is not fully eliminated. In the constructor of a game class, the `stats_observer` still has to be initialized using the YAML file path of interest. This dependency could be eliminated by using the already existing game configuration file, but that would either move the dependency somewhere else or make the overall code harder to read. While this issue seems minor, it is problematic because, if missed, nothing would be recorded and no feedback would be given, making the issue difficult to debug.

Tracking only occurs on functions, where the decorator is called after the function ends. Therefore, pointcuts can only be defined at the end of functions, giving developers less control and possibly requiring them to define functions specifically for recorders. However, this is not necessarily an issue because it encourages developers to design functions with the single responsibility principle in mind.

There is also currently no way to compare models within the same dashboard during testing. This is necessary to determine how difficult a level is by comparing the win rates of agents with different skill levels, where skill is measured by the number of training steps an agent has gone through. While it would be easy to modify the architecture to support this, the issue arose because there was no clear understanding of which exact metrics would be used for the project while the solutions were being developed. Measurements without thresholds are meaningless, but there is currently no place in the architecture where such thresholds are defined.

E. Conclusion

At this stage, the engine within the project allows developers to create their own 2D platformer games with the mechanics and level design they would like to add. After that, the steps needed to have agents try the levels and return data on them are: create a new observer class that contains the variables to be tracked and recorder functions that manipulate those variables as needed, create a YAML file that connects game functions to the recorders while controlling which arguments are passed to each recorder, initialize the observer in the constructor of the game, and have the agents play the game. Data is easy to gather from any game within the

engine, and it is independent of whether the player is a human or a DRL agent, meaning it can be used to gather player statistics as well.

Future work includes figuring out what data is needed to obtain useful information about level design and how to validate it. It also includes defining thresholds for measurements and adding the ability to compare the results of different agents for the same level in the same dashboard.

IV. General Conclusion

The main goal of the project is to help playtesters and level designers study and evaluate game levels, because playtesting is one of the most demanding jobs in the gaming industry. To achieve this, a game engine with integrated DRL agents is being developed, where developers can create 2D platformer games, train DRL agents on those games, and define metrics that help identify issues with level design.

My work was divided into two parts. In the reinforcement learning part, I edited and tested hyperparameters and personas, but due to issues with the observation and action spaces, the results were negative. After fixing those issues, adding more debugging tools for agent training, and adding Dijkstra's pathfinding algorithm to help agents make decisions, their performance greatly improved, and they became able to complete the test levels without requiring changes to their parameters.

In the statistics observer part, I used aspect-oriented programming to separate observers from the core game logic. I used Decorators at first but had several issues, most notably the unnecessary dependency between different game files and the decorator, as well as the fact that decorators were scattered throughout the project. Configuration files mitigated those issues and made adding metrics easier.

The project currently allows platformer games to be created easily. Mario, Sonic, and Mega Man clones were created within a few days using the engine. DRL agents can be trained on any of those games and perform well on each of them. Defining metrics and observing them is as simple as creating variables for the metrics and recorder functions for them.

For future work, a way to define thresholds for metrics needs to be developed. A method for comparing agents of different skill levels on the same game and gather metrics based on their relationships also needs to be developed. Finally, there is a need to define and evaluate potentially useful metrics.

References

[1] M. Towers et al., "Gymnasium: A Standard Interface for Reinforcement Learning Environments," arXiv preprint arXiv:2407.17032, 2024.

[2] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP'97 — Object-Oriented Programming*, Lecture Notes in Computer Science, vol. 1241. Springer, Berlin, Heidelberg, 1997, pp. 220–242.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1994.