University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

# RealmAI: Reinforcement Learning Generated Game Analytics

ECE 498B: Final Report


**Group Number: 2022.38**

Ethan Lee (20731081)
Ken Ming Lee (20722040)
Cliff Li (20721961)
Michael Meng (20666979)
Matthew Tang (20723384)

Consultant: Dr. Mark Crowley

Waterloo, Ontario, Canada
May 8, 2022

## Abstract

As part of the game development cycle, game developers undergo a time-consuming and resource-intensive process of iteratively playtesting, re-designing, and fine-tuning their game before it is released. Using reinforcement learning, RealmAI eases this process by training bots to play games, during which, gameplay data are generated, recorded, and analyzed by the tool. Game developers can then view the analyzed results to make data-driven decisions, without the need of a large playtesting team or other post-release analytic tools to provide these insights. Furthermore, RealmAI abstracts away technical details of reinforcement learning which lowers the barrier to entry for game developers. RealmAI is also publicly available for free.

## Acknowledgements

First and foremost, we would like to thank Professor Mark Crowley, our consultant, for his guidance throughout the entire duration of the project. Professor Crowley played a pivotal role in guiding not only the overall direction of the project, but also specific features that improved the usability of the RealmAI tool. We would also like to thank Professor Sagar Naik for supporting us through our project development and providing a smooth and successful Symposium Day.

We would also like to thank our colleagues from past co-ops and friends who provided crucial feedback for the user survey we did in 4A, and the user interviews we conducted in 4B.

# Contents

# List of Figures



# List of Tables

# 1 High-Level Description of Project

## 1.1 Motivation

Success for indie game companies often depend on making games that deliver a fun and engaging experience for a wide audience. Before the game's release, game developers undergo an iterative process of playtesting, redesigning, and fine-tuning [1]. The iterative design process helps teams find the correct design by building fast and failing fast [1]. The problem with this process is that it is time-consuming, resource-intensive, and prone to mistakes. Oftentimes, large companies utilize playtesting teams to identify problems (e.g., exploits, bugs, glitches, etc.) before the game is released [2]. However, this is not always financially feasible for smaller, independent game studios because they require specialized equipment and expertise [2].

Using reinforcement learning (RL), RealmAI trains agents (i.e., bots) to play the game, during which, gameplay data and video recordings of the agent(s) are collected. Analysis on the data are then performed, generating various heatmaps with user-customizable parameters (Figure 1). The insights are displayed on a custom web application, aiding game developers in making design decisions, without the need for a large playtesting team or other post-release analytic tools. To minimize the barrier of entry for game developers, RealmAI is publicly available at no cost[1] and abstracts away RL technical details.

## 1.2 Project Objective

The main objective of RealmAI is to use RL to provide data-driven insights through the use of heatmaps and gameplay video recordings. This helps developers understand and balance their games before release. RealmAI is built with the following guiding design principles:

**User-Friendliness** RealmAI abstracts the complexity of the underlying RL technical concepts away from users. Additionally, RealmAI provides documentation[2] to minimize the learning curve for its users and help users setup the tool.

**Generalizability** Although RealmAI focuses specifically on the 2D Action-Platformer genre (e.g., Super Mario series), this can be extended to other genres (e.g., First Person Shooters) in the future.

**Flexibility** RealmAI is highly configurable and extensible. For instance,

- It is composed of multiple standalone subsystems that can be used individually.
- Its online components are optional so that users operate the tool completely offline if they prefer to.

**Accessibility** RealmAI is publicly available at no cost on GitHub.

[1] Project website: `https://bit.ly/RealmAI`; source code: `https://github.com/realm-ai-project`
[2] `https://realm-ai-project.github.io/documentation/`

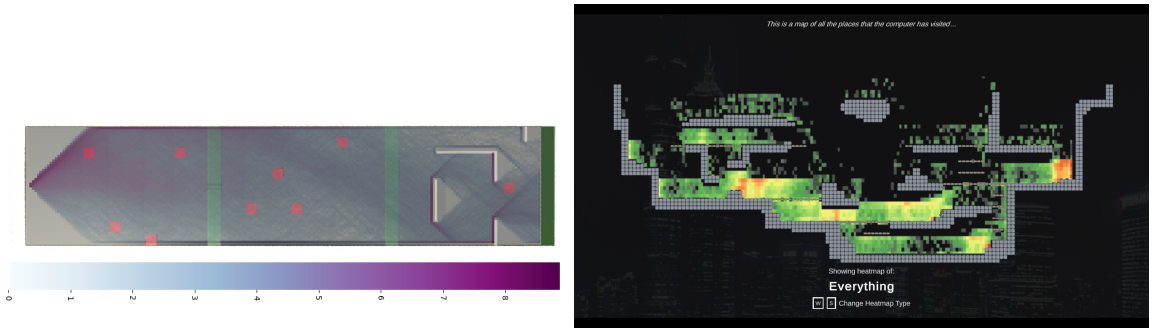



Figure 1: Example of heatmaps generated for 2 different games

### 1.3 Block Diagram

The first stage in the block diagram is the game development phase. A game developer would download the custom plugin into their Unity editor. This process is similar to downloading an extension in Visual Studio Code. They would then set up the agent to play the game by following the public documentation guide.

The second stage is the training phase. In this phase, the agent learns to play the game and while doing so, the agent generates gameplay data. There are two options for the training phase – training with or without hyperparameter tuning. Hyperparameter tuning finds the best performing RL hyperparameters for training. However, this option takes significantly longer, so developers can choose to skip it.

The last stage is the results phase. Once the training phase is complete, developers can view heatmaps and watch video recordings of the agent playing the game. This is all done through a local webapp. The block diagram in Figure 2 indicates the system flow from the input to the output of the system.

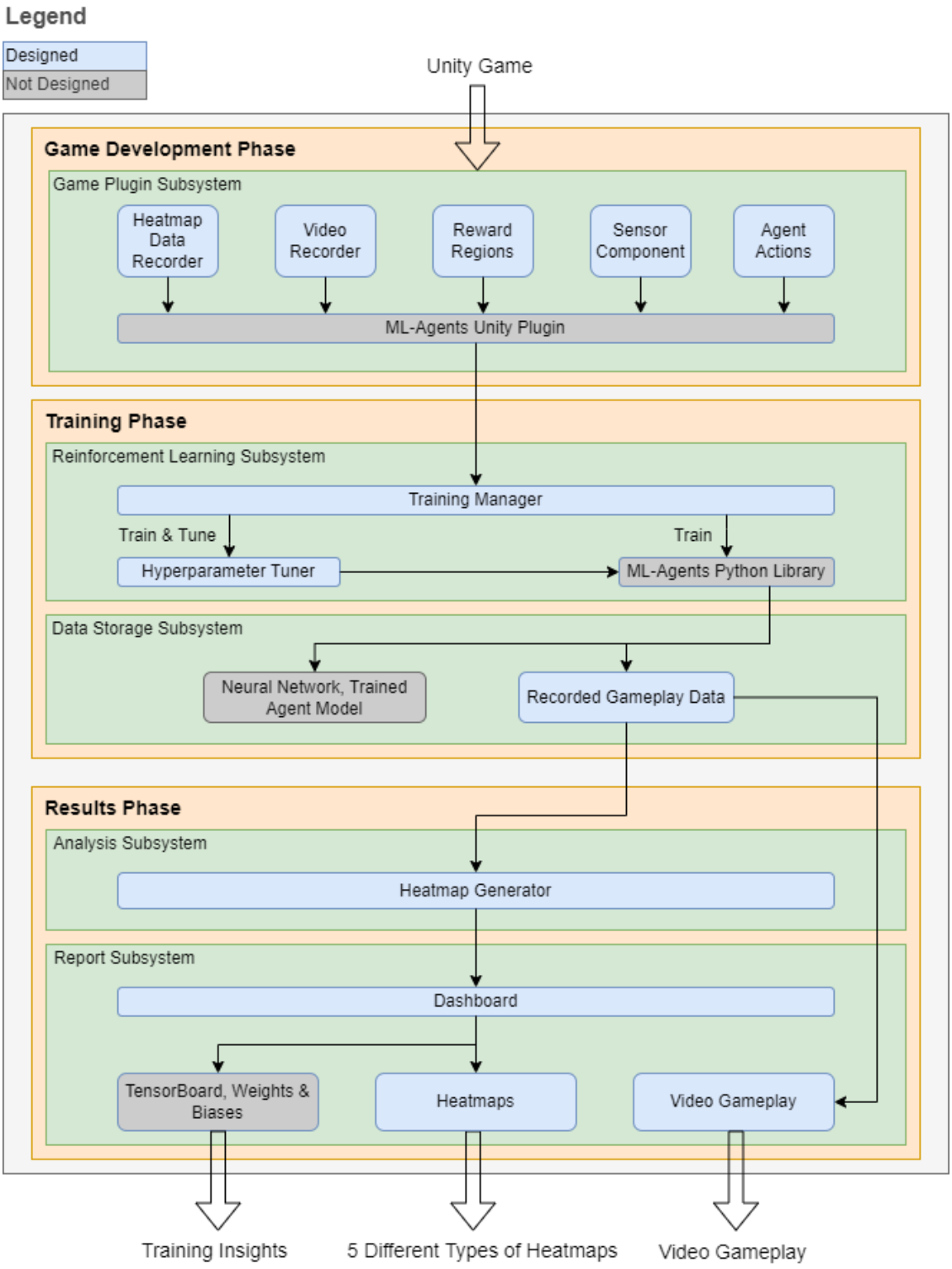



Figure 2: High-level block diagram of the tool's inputs, outputs, and subsystems

# 2 Project Specifications

## 2.1 Functional Specifications

**Essential Functional Specifications**

Table 1: Essential Functional Specifications

| ID | Subsystem: Specification | Description |
|---|---|---|
| FS1 | Custom Game Plugin: Initial Setup for Reinforcement Learning Subsystem | The game plugin should add the functionality for the game to be played automatically, without human input. |
| FS2 | Custom Game Plugin: Data Collection | The plugin must allow game developers to collect key gameplay data and statistics from the game. At minimum, this includes periodic samples of the position of the player character in the game world (as a 2D vector [x, y]) and the gamescore obtained. |
| FS3 | Custom Game Plugin: Video Recording | The plugin must support video recording of the RL training to a video output file. The video must contain gameplay for an entire episode. |
| FS4 | Custom Game Plugin: Video Recording | The plugin video recording system must capture a recording of the agent at periodic intervals as defined by the game developer (e.g., 1 video per 50 episodes). |
| FS5 | Reinforcement Learning Subsystem: Hyperparameter Tuning | The system must be able to perform hyperparameter tuning for RL algorithms provided by the Unity ML-Agents library, and allow users to configure the hyperparameters to tune. |
| FS6 | Reinforcement Learning Subsystem: Configurability | At minimum, users must be able to set the number of hyperparameter tuning trials and the number of training steps per trial. |
| FS7 | Data Storage: Data Persistency | Generated training metrics and gameplay data from the Reinforcement Learning Subsystem must be stored in a persistent file or database. |
| FS8 | Analysis Subsystem: Heatmaps | The system must be able to generate at least 3 different types of heatmaps. |

| | | |
|---|---|---|
| FS9 | Reinforcement Learning Subsystem: Training Progress Indicator | The system must be able to generate training curves[3] of Gamescore vs. Timestep to allow users to monitor the training process. |
| FS10 | Report Subsystem: Displaying Results | The system must have the ability to display heatmaps and training curves from FS8 and FS9. |

**Non-essential Functional Specifications**

Table 2: Non-essential Functional Specifications

| ID | Subsystem: Specification | Description |
|---|---|---|
| FS11 | Data Storage Subsystem: Data Retrieval | All raw collected data can be retrieved by the user. |
| FS12 | Report Subsystem: Displaying Video Replays | Users can view recorded gameplay videos (if any) through the Report Subsystem. |

## 2.2 Non-Functional Specifications

**Essential Non-Functional Specifications**

Table 3: Essential Non-Functional Specifications

| ID | Subsystem: Specification | Description |
|---|---|---|
| NFS1 | Overall System: Monetary cost | The codebase for the system must be publicly available at no cost. |
| NFS2 | Report Subsystem: Heatmap Generation Duration | The generation of the heatmap must take less than 2 hours. |
| NFS3 | Reinforcement Learning Subsystem: Pausing and Resuming | The hyperparameter tuning process can be paused and resumed by the game developer. |

[3]Usually represents a family of different plots showing different metrics against time. These plots are denoted as "training curves" because they give the user information about how the training is progressing.

**Non-Essential Non-Functional Specifications**

Table 4: Non-Essential Non-Functional Specifications

| ID | Subsystem: Specification | Description |
|---|---|---|
| NFS4 | Overall System: Documentation | Game developers should be able to find documentation within 2 clicks on the Report Subsystem. |
| NFS5 | Custom Game Plugin: Plugin Size | The total size of game plugin must be less than 100 MB. |
| NFS6 | Overall System: Privacy | Users should have the choice to operate completely offline, such that all generated data are stored on their local system. |
| NFS7 | Overall system: Cross-Platform Compatibility | The system should work on Windows, macOS, and Ubuntu. |

# 3 Detailed Design

In this section, detailed descriptions, alongside notable key features and design decisions made for every subsystem are provided. In the following subsections, it is assumed that the user is a game developer.

## 3.1 Game Plugin Subsystem

### 3.1.1 Overall Design

To use the Game Plugin Subsystem, it is assumed that the user has already created a game that is playable by a human player (it is not incomplete or dysfunctional). Assuming that such a game is provided by the user, the purpose of the Game Plugin Subsystem is to then:

1. be able to automatically play the game and play it without human input (FS1).
2. be able to collect data while playing that can be used to generate the heatmaps in the Analysis Subsystem (FS2).
3. be able to record video replays of the game being played automatically (FS3, FS4).

To simplify the scope of the Game Plugin Subsystem, it is assumed that the user's game is being developed inside Unity, a popular game production software. The Game Plugin Subsystem is provided to the user in the form of a Unity package containing several C# scripts, which will provide the required functionality once they are setup. A more detailed description of Unity and its packaging system is available in Appendix A.

In order to be able to automatically play the game and play it without human input, the scripts included will leverage the ML-Agents library, which is the official RL library for Unity [3]. ML-Agents can use RL algorithms to create and train an RL agent that can learn to play any game without human input, with the following requirements:

1. State observations must be provided as inputs to ML-Agents. These observations represent what the player sees at each time step and are encoded as a list of floating point numbers (floats).
2. The possible actions that the player can take must be specified to ML-Agents. When running the RL algorithms, ML-Agents will output the actions it wishes to take as a list of floats and integers. When this list is outputted from ML-Agents, the corresponding actions should be taken in the game.
3. A reward function must be provided as another input to ML-Agents. This will be treated as the performance metric in the RL algorithms and the algorithms will try to optimize its actions to achieve the maximum reward.

To implement the required functionality, and to satisfy the requirements for ML-Agents, the Unity package contains the following classes:

1. `RealmAgent`: a class which coordinates all of the other classes with each other and with the ML-Agents library.
2. `RealmSensor`: a class responsible for detecting objects around the player and sending it to ML-Agents as the inputs (state observations) to the RL algorithm.
3. `RealmActuator`: a class responsible for taking the output of the RL algorithm from ML-Agents and performing the actions corresponding to the output.
4. `RealmScore`: a class responsible for generating a performance metric and sending it to ML-Agents to use as the reward signal for the RL algorithm.
5. `DataRecorder`: a class responsible for periodically recording the position of the player, along with any other data required by the Analysis Subsystem, and saving it in required format in the Data Storage Subsystem. The specific data required and the storage format is specified in the Data Storage Subsystem section.
6. `VideoRecorder`: a class responsible for periodically recording video replays of the game. This class must allow the user to configure the frequency these video recordings.

### 3.1.2 Quantitative Analysis

**Choice of Sensors** The system is designed to focused on 2D Action-Platform games, which are characterized by a having player-control character that has to navigate around some terrain, avoid obstacles, interact with objects in the environment and defeat enemies [4]. The key information needed by players and by the RL agent to play the game are the positions of the player in-game and any enemies, items, or terrain surrounding the player. The `RealmSensor` class is responsible

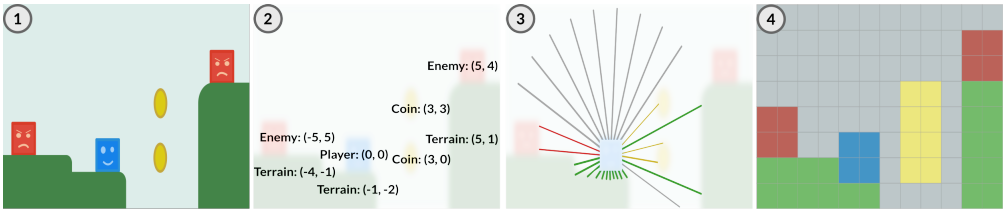



Figure 3: A visualization of the differences between the 4 observation methods. Image 1 is an example game environment as seen by a human player and by the RL agent using method 1. Images 2-4 visualizes methods 2-4 respectively.

Figure 4: A custom 2D game where the goal of the player is to traverse from the left side of the map to the right side, while avoiding red enemies that move around vertically

for gathering this information and sending it to ML-Agents. There are four approaches for doing this (visualized in Figure 3):

1. Use Unity's rendering system to generate an image of the environment surrounding the player. Then, send the image to ML-Agents using the `CameraSensor` class provided by ML-Agents. This generated image is similar to what human players would see when playing the game.
2. Create a class that encodes the positions of all enemies, objects, or terrain near the player into a list of floats and integers, and send it to ML-Agents using its `ISensor` interface.
3. Use the `RayPerceptionSensor` class provided by ML-Agents, which projects a number of rays originating from the player outwards and encodes information about the first object hit by each ray as an list of floats and integers.
4. Use the `GridSensor` class provided by ML-Agents, which projects a grid around the player, and encodes information about an objects colliding with each cell in the grid as an list of integers. If multiple objects lie in a square, the object that is taken is chosen based on an order specified by the game developer.

To determine the best method, an experiment was conducted where each of the four observation methods was used to train an agent to play a simple 2D game (Figure 4). The performance of the agent (how far it can travel towards the goal location) over time for each of the four methods is summarized in Figure 5.

It can be observed that approach 4, detecting objects in a grid, performed the best among the four methods, allowing the agent to learn the highest scoring behaviour in the shortest amount of time. Based on this result, it was decided that the `RealmSensor` is to be implemented based on approach 4.

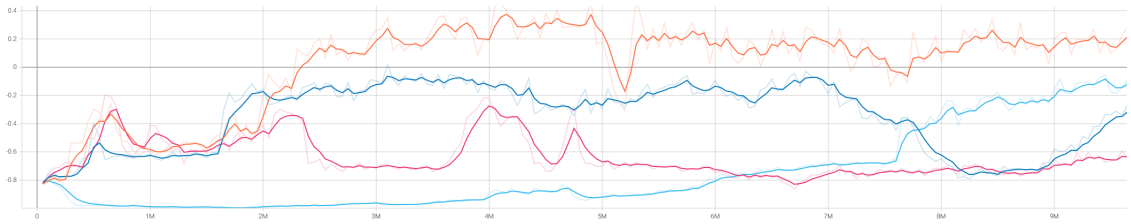


Figure 5: A plot of the total gamescore achieved by the agent over the total number of time steps using methods 1 (dark blue), 2 (red), 3 (light blue), and 4 (orange)

### 3.1.3 Design Alternatives and Iterations

**Actions Interface** When running the RL algorithm, ML-Agents will output the desired actions as a list of floats and integers. In the original design, the `RealmActuator` class was designed to retrieve this list. However, since different games can have very different possible sets of actions, the user was required to manually create and provide to `RealmActuator` some functions that will specify how many float and integers are needed and how to interpret those floats and integers to perform the corresponding actions. When testing the Game Plugin Subsystem, one common point of feedback was that this was too confusing and time consuming for the users to setup.

Users often had to manually create code that transforms keyboard, mouse, and joystick inputs (which are in the form of booleans and 2D float vectors) into a list of floats and integers. To address this, an alternative design was proposed, where the `RealmActuator` accepts functions that can interpret lists of floats and integers, but will also accepts functions that interpret booleans, 2D float vectors, or 3D float vectors. In this alternative, `RealmActuator` internally transforms all of these data types into lists of float and integers, but this process is invisible to the user. After testing both designs, this new alternative, shown in Figure 6, was favored over the previous design since it allows users to work with more natural data types like booleans and vectors. Additionally, it abstracts away some details of working with the RL algorithm, which reduces the amount of knowledge of RL that is required of the user.

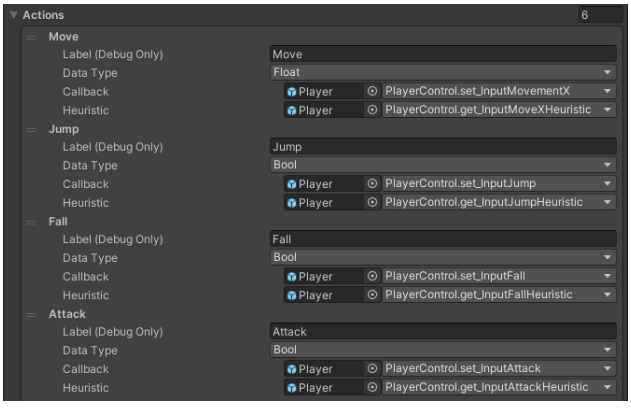



Figure 6: Interface in Unity for configuring the actions that players can take. For each action, users can select the data type (boolean, integer, float, 2D vector, or 3D vector) using a dropdown menu.

**Implementation of Rewards** In the original design, `RealmScore` requires the user to provide a function that returns the current gamescore (score achieved by the player) as a float, and sends it to ML-Agents as the reward signal. Alternative methods were also proposed, such as:

- allowing the user to draw regions on a map and rewarding the agent for visiting those regions.
- allowing the user to define specific tasks through code and rewarding the agent for completing those tasks.

These alternatives were proposed as potential extension features that would give users more options to create more diverse agent behaviours and generate more useful data. However, after testing, and with feedback from the project's consultant, the alternative of "allowing the user to draw regions on a map and rewarding the agent for visiting those regions", shown in Figure 7, was heavily favoured and included into the design. The benefit of this alternative is that it does not require familiarity with RL to use and directly rewards the agent for performing a common task in platforming games – navigating to a certain area – while also serving as an alternative reward function if the gamescore is not available.

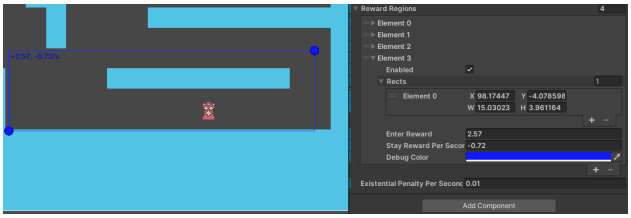



Figure 7: Interface for configuring the rewards using the alternative design. Users can specify regions in the game world and associated rewards for entering or staying in the region, which can be done precisely using the input fields on the right, or visually by dragging the circular handles on the left.

### 3.1.4 Satisfying Specifications

With the classes `RealmAgent`, `RealmSensor`, `RealmActuator`, `RealmScore`, and the RL algorithms provided by the ML-Agents library, the Game Plugin Subsystem provides the functionality to automatically play a game without human input, satisfying specification FS1. The class `DataRecorder` provides the functionality to collect data that can be used to generate heatmaps and graphs of the gamescore over time, satisfying specification FS2. The class `VideoRecorder` provides the functionality to record video replays of the game being played automatically, satisfying specifications FS3 and FS4. The total size of the git repository containing this subsystem is 2.91MB (reported by GitHub) which is less than 100MB, satisfying NFS5.

## 3.2 Reinforcement Learning Subsystem

The main purpose of the RL Subsystem is to perform RL training. However, modern RL algorithms commonly contain many hyperparameters (i.e., parameters that control the learning process, and

```yaml
default_settings:
  trainer_type: ppo
  hyperparameters:
    batch_size: 64
    buffer_size: 12000
    learning_rate: 0.0003
    beta: 0.001
    epsilon: 0.2
    lambd: 0.99
    num_epoch: 3
    learning_rate_schedule: linear
  network_settings:
    normalize: true
    hidden_units: 128
    num_layers: 2
    vis_encode_type: simple
  reward_signals:
    extrinsic:
      gamma: 0.99
      strength: 1.0
  keep_checkpoints: 5
  max_steps: 10000
  time_horizon: 1000
  summary_freq: 5000
```

```yaml
default_settings:
  trainer_type: ppo
  hyperparameters:
    batch_size: [64, 128, 256] # Categorical values
    buffer_size: log_unif(2000, 12000) # Continuous ranges (int)
    learning_rate: log_unif(0.0003, 0.01) # Continuous ranges (float)
    beta: 0.001
    epsilon: 0.2
    lambd: 0.99
    num_epoch: unif(1, 15)
    learning_rate_schedule: linear
  network_settings:
    normalize: true
    hidden_units: 128
    num_layers: 2
    vis_encode_type: simple
  reward_signals:
    extrinsic:
      gamma: 0.99
      strength: 1.0
  keep_checkpoints: 5
  max_steps: 10000
  time_horizon: 1000
  summary_freq: 5000
```

Figure 8: (Left) Configuration for ML-Agents (Right) Configuration for RealmAI

are therefore not learned during training). Furthermore, performance of these RL algorithms are heavily dependent on the choice of hyperparameter values [5]. Understanding appropriate hyperparameter values to use often takes experience, which is an unreasonable assumption for game developers, who often do not have RL domain knowledge. Therefore, RealmAI's RL Subsystem offers hyperparameter tuning capabilities with recommended ranges, so that appropriate hyperparameter values can be found automatically. Throughout this entire process, various gameplay data, training metrics and gameplay footage are automatically recorded, so that analysis can be performed on the data to provide insights (more of this in later sections).

### 3.2.1 Overall Design

The RL Subsystem is comprised of 2 main submodules, Unity's ML-Agents Python package and the hyperparameter tuner. Unity's ML-Agents Python package is made by Unity [6] and provides two main functionalities: implementations of several RL algorithms and the communication protocol that enables the RL algorithms to play compatible games (i.e., those integrated with ML-Agents Unity plugin, such as the Game Plugin Subsystem). On the other hand, the hyperparameter tuner is a Python package fully-designed by the team and its main purpose is to perform hyperparameter tuning for algorithms provided by ML-Agents. Implementations of hyperparameter tuning algorithms were provided by the Optuna library [7]. The hyperparameter tuner was designed to be fully compatible with ML-Agents, so using the hyperparameter tuner does not limit users to any specific RL algorithm.

**Hyperparameter Tuning Configuration** A flexible syntax structure for the configuration of hyperparameters is designed such that it looks similar to the structure of ML-Agents' RL training configuration file. However, it has the added functionality of allowing users to customize the hyperparameters that they would like to tune and the ranges of values to tune over (Figure 8).

### 3.2.2 Design Alternatives and Iterations

**Training Manager** As seen in the block diagram (Figure 2), the Game Plugin Subsystem interacts with the RL Subsystem through a training manager. In other words, from the Unity IDE there needs to be a mechanism that starts the training process.

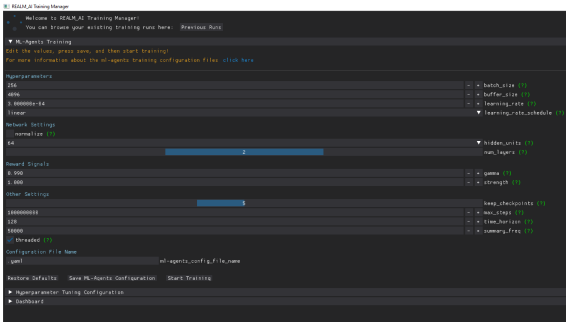
The initial design used a drop-down menu in Unity to start the training process and the hyperparameter tuning. By clicking these menu options, preconfigured terminal commands would run. However, there are many problems with this approach. Firstly, it doesn't provide any flexibility and customization to users with deeper levels of RL knowledge. Secondly, it is not user-friendly since after the training process starts, there is no indication of the training progress.

The proposed solution was to create a Graphical User Interface (GUI), shown in Figure 9. This allows users to have full control of the entire training process. As the training manager, this GUI has the following responsibilities:

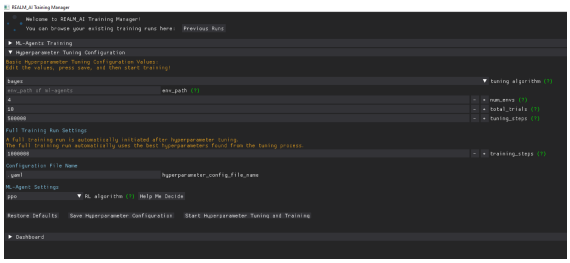
- Initiate the training for the RL agent (shown in Figure 9a). Users will have the ability to modify and adjust settings that are used by the ML-Agents Library. The ML-Agents library controls the training process.
- Initiate hyperparameter tuning and then run the training (shown in Figure 9b). Users will have the ability to modify and adjust settings that are used by the tuning process.
- Pause and resume training and hyperparameter tuning runs.
- Monitor logs that are created during training and hyperparameter tuning (shown in Figure 9c).
- Browse existing training runs.
- View the results through a local web app.

**Multiple Levels of Abstraction** In the earlier iterations of the project, the only method to interact with the RL Subsystem was through editing the configuration file. However, the team quickly realized that editing a configuration file is clunky and can be overwhelming to novice RL users. As a result, the RL Subsystem exposes multiple levels of abstraction to cater to users with various levels of familiarity with RL:

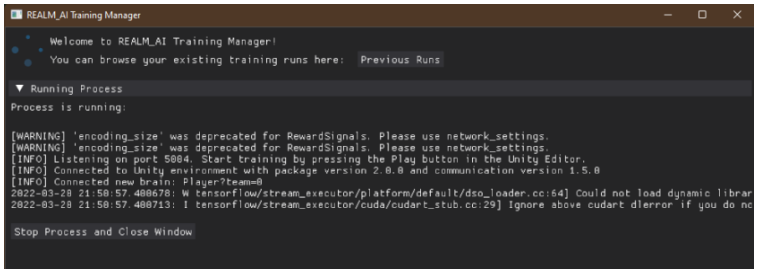
- The training interface caters to those who are new to RL and would like to be guided through a GUI (Figure 9b).
- Command line interface (CLI) arguments provides users with finer-grained control over the tuning process (Figure 10a).
- The `.yaml` configuration file caters to experienced users who prefer full control over all aspects of the tuning process (Figure 10b).



(a) Training Interface



(b) Hyperparameter Tuning Interface




(c) Training Logs

Figure 9: Training Manager functionalities

A difficulty that the team ran into was deciding what to do if users interact with the RL Subsystem through both the CLI arguments and the configuration file simultaneously with conflicting settings. At the end, it was decided that the CLI arguments takes precedence over the configuration file.

```
usage: realm-tune [-h] [--config-path CONFIG_PATH] [--output-path OUTPUT_PATH]
 [--behavior-name BEHAVIOR_NAME] [--algorithm {bayes,random,grid}] [--total-tr
ials TOTAL_TRIALS]
                  [--warmup-trials WARMUP_TRIALS] [--eval-window-size EVAL_WIN
DOW_SIZE] [--env-path ENV_PATH] [--use-wandb] [--wandb-project WANDB_PROJECT]
                  [--wandb-entity WANDB_ENTITY] [--wandb-offline] [--wandb-gro
up WANDB_GROUP] [--wandb-jobtype WANDB_JOBTYPE] [--full-run] [--full-run-max-s
teps FULL_RUN_MAX_STEPS]

RealmAI hyperparameter optimization tool

optional arguments:
  -h, --help            show this help message and exit
  --config-path CONFIG_PATH
                        Path of configuration file
  --output-path OUTPUT_PATH
                        Specify path where data will be stored
  --behavior-name BEHAVIOR_NAME
                        Name of behaviour. This can be found under the agent's
 "Behavior Parameters" component in the inspector of Unity
  --algorithm {bayes,random,grid}
                        Algorithm for hyperparameter tuning
  --total-trials TOTAL_TRIALS
                        Number of hyperparameter tuning trials
```

(a) CLI Arguments

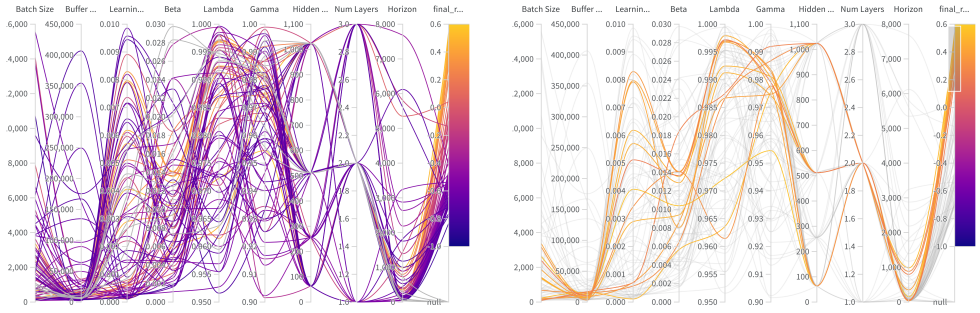
```yaml
realm_ai:
  behavior_name: 3DBallHard
  algorithm: bayes # or random
  total_trials: 21 # total number of trials (inclu
  warmup_trials: 5 # number of "warmup" trials whe
  eval_window_size: 3 # optional, training run is
  run_id: test # optional, specify to manually spe
  full_run_after_tuning: # optional, if specified,
  should be specified so that we know the name of
    max_steps: 20000 # number of steps to run for
    run_id: full_run # name of the complete traini
  ...
```

(b) `.yaml` Configuration File

Figure 10: Different ways of interacting with the RL Subsystem

**Integrations with Weights and Biases** Another key design decision made over the course of development of the RL Subsystem is the integration of Weights and Biases (W&B) [8], an online machine learning platform that focuses on tracking machine learning experiments. W&B provides various tools to visualize hyperparameter tuning runs, which can help users learn the intuition of the various hyperparameters, and their relationship with each other. For instance, it can help users find patterns in hyperparameter values for well-performed trials (Figure 11a), and provides tools to show which hyperparameters are significant, as well as their correlation to the training objective

(i.e., final reward in this case) (Figure 11b). The end goal is that this intuition can help users pick better hyperparameter values over time, reducing their need to rely on large hyperparameter sweeps for every RL training run they perform, which can be very expensive, especially for independent or smaller game studios. However, since using W&B requires data to be uploaded to their servers, the team made the decision to make it optional, so that users can retain full control over where their data is stored.




(a) (Left) A visualization of the relationship between hyperparameters and the final metric. (Right) The same visualization, but with poor-performing runs filtered out.


(b) A table sorted by significance of hyperparameter with respect to the final metric. On the right-most column, green represents positive correlation, red represents negative correlation.

Figure 11: Various tools provided by Weights and Biases

### 3.2.3 Quantitative Analysis

**Hyperparameter Tuning Ablation Analysis** To study the effect of hyperparameter tuning, an ablation study was performed. A large-scale hyperparameter sweep containing 80 training runs (each with a unique set of hyperparameters) was performed on a custom-made 2D game (Figure 4). After that, two agents were trained for $10^7$ steps, one using default hyperparameters provided by ML-Agent, and another using the best set of hyperparameters found from the hyperparameter sweep. From Figure 12, it is evident that the agent trained using tuned hyperparameters performed significantly better than the baseline agent.

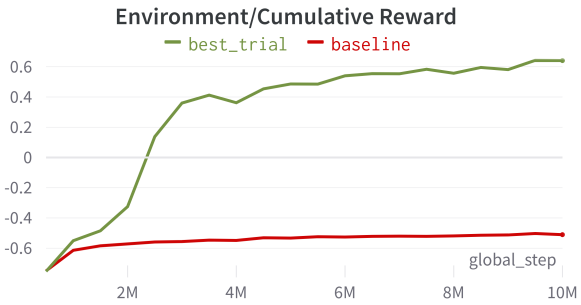



Figure 12: A cumulative reward vs training steps graph comparing performance between 2 agents. The green line shows performance of the agent trained using tuned hyperparameters, the line in red shows performance of the agent trained using default ML-Agents hyperparameters.

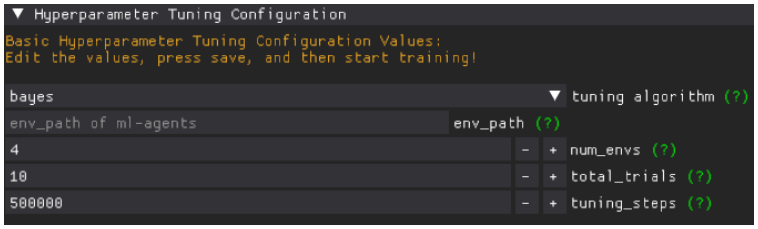



Figure 13: A closeup screenshot of the Training Manager. `total_trials` denote the number of hyperparameter trials, `tuning_steps` denotes the number of training step per trial.

### 3.2.4 Satisfying Specifications

Since the hyperparameter tuner is algorithm-agnostic and fully configurable in terms of hyperparameters to tune, and their ranges, this satisfies FS5. On the other hand, from Figure 13, it can be seen that the number of hyperparameter tuning trials and the number of training steps can be configured through the Training Manager GUI, satisfying FS6. The RL Subsystem uses Unity ML-Agents to perform RL training, which logs various metrics, including Gamescore vs Timestep, to TensorBoard[4], satisfying FS9. The RL Subsystem also has the feature to pause and resume training and hyperparameter tuning, satisfying NFS3.

## 3.3 Data Storage Subsystem

### 3.3.1 Overall Design

The Data Storage Subsystem is used to store and persist data collected from the training scripts of the game plugin which include agent positional data and video recordings of the agent. The data is stored in a particular file system structure shown below in Figure 14. Every training run has its data stored in a similar file system structure. The RealmAI directory stores the agent's data per episode in DAT files as well as the recorded gameplay videos. The schema of each DAT file includes the episode number, duration of the episode, agent reward, list of the $x$ positions, and list

[4]Metrics logged to TensorBoard from the ablation analysis in Section 3.2.3 is publicly available at `https://wandb.ai/kenminglee/realm_tune/groups/Comparison/workspace?workspace=user-kenminglee`

of the $y$ positions. The other directories and files are generated by ml-agents and includes things like TensorBoard data as well as the agent's training checkpoint.

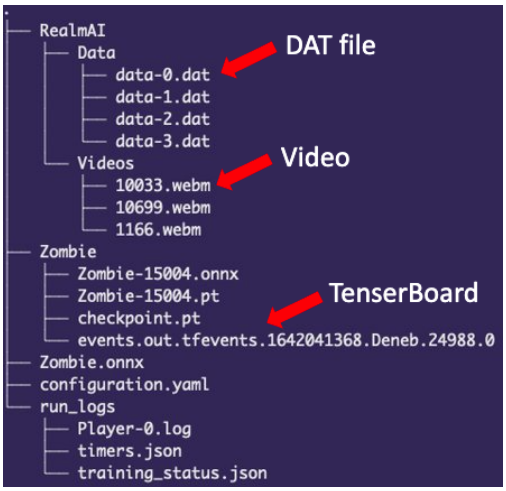



Figure 14: File system structure of RealmAI training run

### 3.3.2 Design Alternatives and Iterations

**On-Premise vs. Cloud** First there was a decision made to choose on-premise storage over cloud storage. This was because storing video data was more efficient on-premise compared to on the cloud. It is also less costly to use on-premise storage compared to paying for a cloud provider.

**Relational Database vs Non-Relational Database vs File System** There was also a design decision made in how to support data persistency. The alternatives were a relational database, non-relational database, and local file system. It was determined that the local file system offered the best support for storing video data in native video file formats. It also provides easy access for the users, as they do not need to learn SQL or a query language to get data from the relational or non-relational database.

**File System Structure** A design iteration was applied to standardize the file system structure. Since generating heatmaps requires users to specify a directory of heatmap data, there needs to be a consistent file system structure for every training run. This includes having the `RealmAI/Data` and `RealmAI/Videos` directories inside the root directory of the training run.

**Data File Format** Furthermore, another design iteration involved changing the file format of the agent data in the `RealmAI/Data` directory. Initially, the JSON format was used as it provided a human-readable format. However, throughout the development process, it was determined that the DAT file format was more efficient. The portion of the data that takes the most space is the raw positional coordinates, which does not provide the users with helpful information unless those coordinates are plotted onto a map. It was determined that using the DAT file format would reduce the size of the `RealmAI/Data` directory and still provide a format that can be used by the Report Subsystem. Table 5 shows the modified comparison between different formats after considering the DAT file format and it's properties, as well as adjusting some weighting criteria (more emphasis on efficiency of storing data). It is shown that the DAT format provides the most efficiency for storing

positional data per episodes and for storing single decimal values for metrics like gamescore and duration per episode. These two factors have the highest weighting as well in the selection criteria.

Table 5: Comparison between CSV, JSON, XML, and DAT formats

| **Criteria (weighting)** | **1. CSV** | **2. JSON** | **3. XML** | **4. DAT** |
|---|---|---|---|---|
| Efficiency for storing series of positional data per episode (0.55) | 7 | 7 | 6 | 10 |
| Efficiency for storing a single decimal value for the gamescore and duration per episode (0.3) | 1 | 9 | 9 | 10 |
| Human Readability (0.1) | 8 | 9 | 8 | 1 |
| Flexibility (forwards and backwards compatibility) (0.05) | 5 | 10 | 10 | 1 |
| Total Score | 0.52 | 0.80 | 0.73 | **0.87** |

#### 3.3.3 Satisfying Specifications

FS7 states that the training metrics and gameplay data need to be stored in a persistent file or database. This is satisfied by how all data from the training scripts are stored on the user's local file system. FS11 states that all raw collected data can be retrieved by the user. This was satisfied by users' ability to access the data stored on their local file system.

### 3.4 Analysis Subsystem

#### 3.4.1 Overall Design

The Analysis Subsystem was implemented using a Python Flask server as the backend infrastructure. The Flask server was used to serve requests from the frontend Report Subsystem for generating heatmaps using the standard HTTP protocol (discussed more thoroughly in the Report Subsystem section). The Report Subsystem can send an asynchronous HTTP request to the Flask server to initiate heatmap generation. The Flask server actively waits for requests and creates a new heatmap according to the parameters specified in the request body. It then issues an HTTP response with the image encoded in Base64. The Report Subsystem receives this response and decodes the encoded image and display it on the dashboard.

The 5 types of heatmaps generated by the Analysis Subsystem show various aspects of the training results:

- **Naive Heatmap** Shows the density of paths taken by the agent during training. This is useful to highlight the most common regions that the agent traverses through the entire training session.

- **Heatmaps by Reward** Shows paths taken by agents in episodes that achieved top 10% highest and lowest gamescore. Showing the most common paths of unsuccessful and successful attempts can be helpful for game developers to make game design choices, such as the layout of the game.
- **Heatmaps by Episode Length** Shows paths taken by agents whose episode lasted the longest or shortest. This could highlight potential exploits or bugs in the game.
- **Heatmaps by Episode** Shows paths taken by agents by episode range. This highlights the agent's learning progression between different episode intervals.
- **Heatmaps by Last Position** Shows the last location of the agent before the game ends. This can be a very useful indicator for the game developers to gauge the difficulty of various parts of the game.

### 3.4.2 Design Alternatives and Iterations

**Usage of Heatmaps** An anonymous survey was sent to game industry professionals with varying levels of experience to gather information regarding features that could improve their workflow. As seen in Figure 15, it was clear that heatmaps were the most highly requested feature. Due to the versatility of heatmaps in showing different insights to developers, heatmaps were selected over other alternatives.

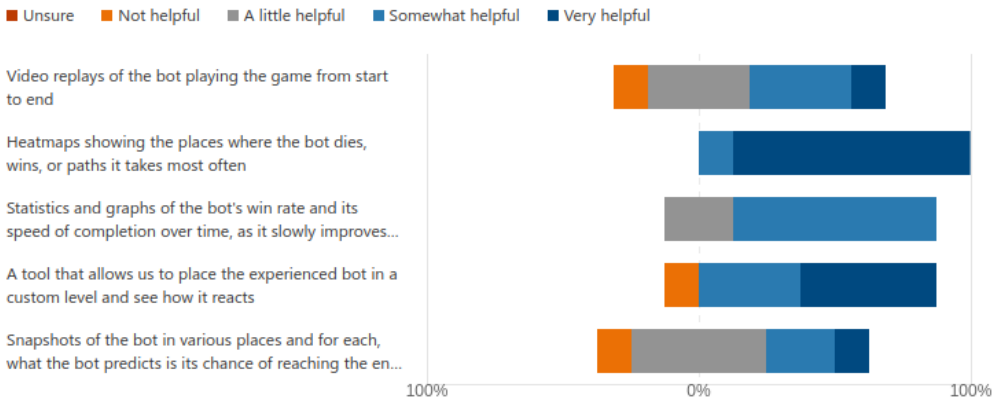



Figure 15: A divergent stacked bar chart showing ratings for every feature proposed

**Backend Server Selection** Another design iteration involved the selection of Flask for the backend server. This suited the overall design of the system as it decoupled the frontend Report Subsystem from the Analysis Subsystem. As a result, with a separate backend server, users could integrate the Analysis Subsystem into their ecosystem. For example, if game developers already have existing dashboards, the Python Flask server can send heatmaps to them instead of the Report Subsystem. Providing a Flask server also allowed for users to specify customizable parameters in their requests. Without the support of a backend server, users would only have access to a set amount of heatmap templates. Lastly, Flask was selected over other alternatives such as Express.js, since many team members were already familiar with the framework or had extensive prior Python

experience. This significantly reduced on-boarding time and allowed for greater focus on feature development.

### 3.4.3 Quantitative Analysis

**Heatmaps Usefulness** To evaluate if the generated heatmaps are actually useful to game developers, an interview was held for a number of game developers with varying experience in the field. The results from the interview showed the usefulness of the heatmaps. Readers are referred to Section 4, where the study methodology, alongside results and findings of the interview are thoroughly investigated.

### 3.4.4 Satisfying Specifications

FS8 states that the system must be able to generate at least 3 types of heatmaps. The system satisfies this specification by allowing the users to generate 5 heatmap types (naïve, by reward, by episode length, by last position, and by range of episodes).

## 3.5 Report Subsystem

### 3.5.1 Overall Design

The Report Subsystem has three main purposes: enabling generation and display of heatmaps with customizable parameters, providing an interface for developers to browse video replay files, and providing an interface for developers to view RL (TensorBoard and W & B) metrics. This was achieved using a custom web application implemented in React.js. This modern frontend framework was selected as it allowed the team to create a user-friendly, flexible interface. Multiple team members were also already familiar with using this framework, which greatly aided in development velocity.

The Report Subsystem leverages a modular design that takes advantage of the React.js design framework. The final design features a directory selection module that allows users to select their desired directory location for generated heatmaps. After selecting a directory, users can interact with the dashboard module which can generate and display the 5 heatmap categories discussed in the Analysis Subsystem section. This module receives user input and sends an asynchronous request to the Analysis Subsystem. Users are able to input custom parameters using form input and custom sliders, as demonstrated in Figure 16. Users also have the ability to delete existing heatmaps. Additionally, users can view videos of saved gameplay data as seen in Figure 17.

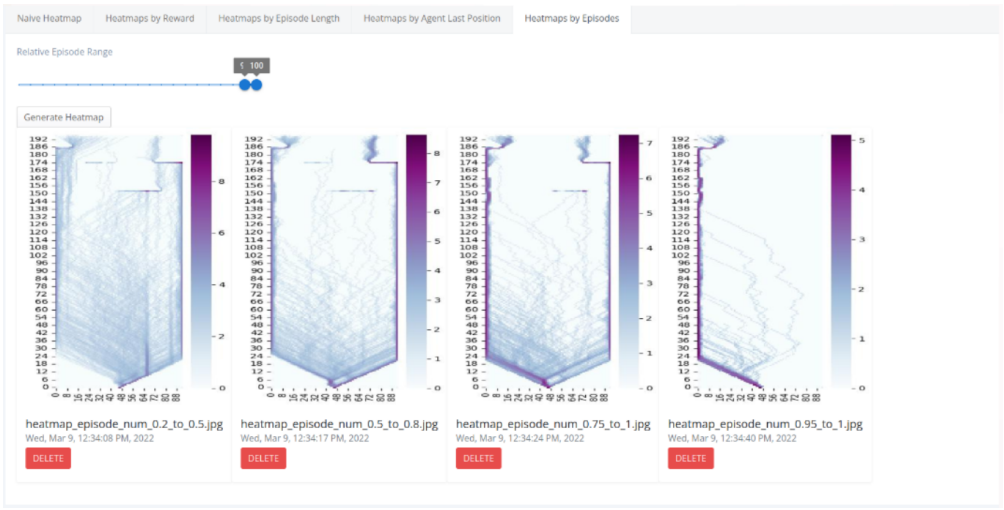



Figure 16: Heatmap generation dashboard with slider for episode range.

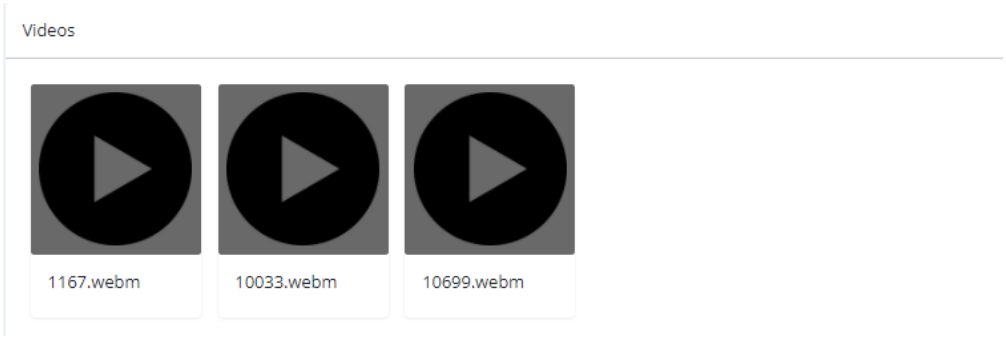



Figure 17: Users can view recorded videos in dashboard.

### 3.5.2 Design Alternatives and Iterations

A custom local web application was chosen, but two alternatives were considered - a local desktop application and embedding heatmaps within TensorBoard itself.

**Local Desktop Application** A local desktop application would be implemented in some desktop application framework such as Electron. It was determined that this option would add support for customizable heatmap generation and recorded video viewing. However, multiple team members were already familiar with frontend web frameworks, so learning a new development framework would have added unnecessary on-boarding time. A custom web application also integrates well with the web-based Flask server from the Analysis System.

**Embedding Images and Videos in TensorBoard** TensorBoard supports the display of multimedia related to machine learning but lacks the ability to host custom images and videos specified by the user. A separate custom web application provides more low-level control over heatmap generation. Integrating a custom web application with TensorBoard is also friction-less due to the fact that TensorBoard is also a web-based application. Thus, it is more feasible to extend the capabilities of TensorBoard using React.js. Lastly, using a custom web app provides greater

flexibility compared to being constrained by TensorBoard. It allows the integration of any other cloud-based resources such as Weights and Biases. Thus, the custom web application allowed for the addition of essential features while still incorporating many of the useful advantages of TensorBoard.

### 3.5.3 Quantitative Analysis

The NFS2 non-functional specifications states that heatmap generation should not exceed 2 hours. In an experiment with increasing data file sizes, it was estimated that an input file with a size of 19GB would result in a 2 hour run-time. It was also determined that on average, 3 days of training produced 70MB of data, which represents an average workload. Therefore, for a majority of workloads the 2 hour limit will not be exceeded. Results from varying input file sizes are represented in Figure 18:

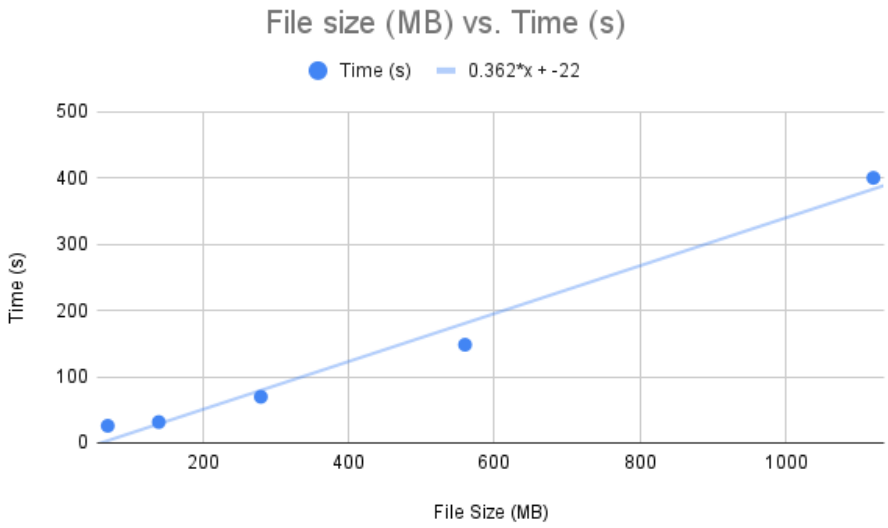


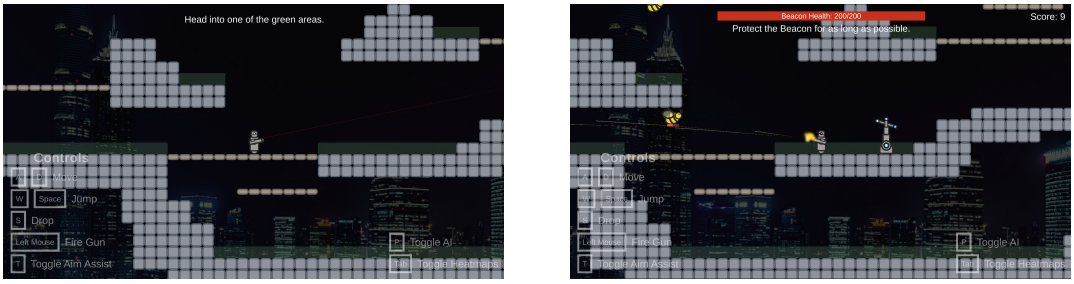

Figure 18: File Size (MB) vs Time (s) demonstrating heatmap generation time

Thus, from the above figure it is clear that the Reporting System satisfies NFS2. The solution is scalable as well and has the capacity to deal with ever increasing input file sizes in the future if necessary.

### 3.5.4 Satisfying Specifications

FS10 states that the system must have the ability to display heatmaps and generated training curve. This is satisfied by the user interface which allows generation and display of heatmaps. The TensorBoard integration also provides the ability to view generated line graphs. FS12 specifies that users can view recorded gameplay videos. This is satisfied by the ability of users to view and play recorded gameplay videos through the dashboard interface. NFS2 specifies that the heatmap generation should take less than 2 hours. In the previous quantitative analysis subsection, it is

determined that this requirement was satisfied.




(a) The player standing still (b) The player shooting incoming enemies

Figure 19: The 2D platformer game created for studying the effectiveness of the system.

# 4 Study on the Effectiveness of the System

## 4.1 Study Methodology

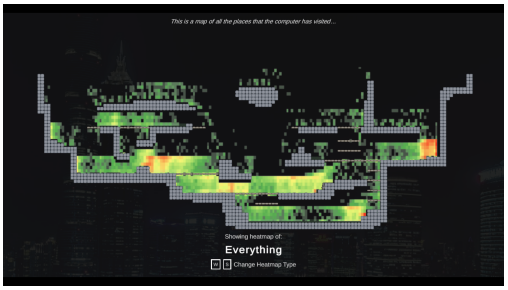
To verify the effectiveness of the heatmaps and video replays generated by the whole system, a 2D platformer game was created and integrated with the system to produce an agent that can play the game and generate heatmaps. The game, shown in Figure 19, involves controlling a character, navigating around some platforms, placing down a "Beacon", and then defending the Beacon from approaching waves of enemies by shooting them with a gun. Players are scored based on how long they can defend the Beacon without dying or letting the Beacon get destroyed. The game takes place inside a medium sized arena, where there are many different locations to place the Beacon. This makes the game interesting as the locations have different terrain, which makes defending the Beacon easier at some locations and harder at others.

Four game developers with various amounts of game development experience were interviewed. They were asked to play the game, view the generated heatmaps (shown in Figure 20), view the video replays of the agent, and finally to playing the game again. Afterwards, they were asked to rate from 1 to 5 about whether they learned anything from the replays or the heatmaps, and how useful they thought the replays or the heatmaps were for game design work.

## 4.2 Findings

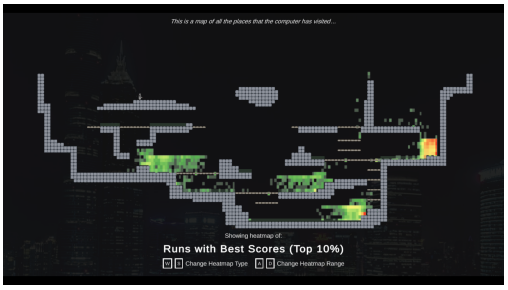
In this section, findings from a mixed-method analysis of the interviews are presented. The quantitative analysis is presented first, followed by the qualitative analysis.

Table 6 shows the quantitative data from the interviews, which includes the number of years of game development of each participant, their responses to the questions relating to the usefulness of the replays and heatmaps, and a list of the scores they achieved playing the game both before and after they saw the replays and heatmaps.

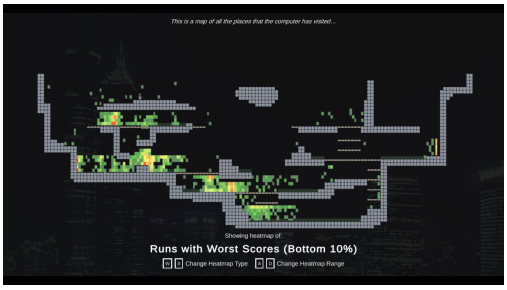



(a) Basic heatmap with no filters


(b) Heatmap showing the final positions of the agent




(c) Heatmap showing the best performing locations




(d) Heatmap showing the worst performing locations

Figure 20: Examples of some heatmaps shown to the interview participants. These heatmaps are shown using a custom in-game user interface.

Table 6: Results from the Interviews with Game Developers

| | **Participant 1** | **Participant 2** | **Participant 3** | **Participant 4** |
|---|---|---|---|---|
| Years of Game Development Experience | less than 1 | 1 | 1 | 3-4 |
| Did you learn from the replays or heatmaps? (Rate 1-5) | 5 | 4.5 | 1 | 1 for heatmaps, 5 for replays |
| How useful are replays or heatmaps for game design work? (Rate 1-5) | 4 | 5 for heatmaps, 2 for replays | 5 | 5 |
| Scores before seeing replays and heatmaps | 89, 67, 35, 65, 124, 88 | 145, 48, 131, 34, 25, 144, 115 | 126, 36, 190 | 34, 176, 86, 137 |
| Scores after seeing replays and heatmaps | 520, 108 | 88, 185, 295, 120, 292 | 178 | 73, 123 |

### 4.2.1 Quantitative Analysis

A one-way paired-samples t-test found, with marginal significance, that after looking at RealmAI's heatmaps and trained agent, participants performed much better in the game ($M$=196.5, $SD$=89.2) than before using RealmAI ($M$=98.81, $SD$=17.5), $t(3)$=-1.89, $p$=.078, $d$=0.94. The low $p$ value means that the probability that the improvement in participants' performance was a result of random occurrence is low, while the high Cohen's $d$ value represents a large effect size, indicating that the difference between two groups are not negligible.

Although the increase in performance might be caused by increased familiarity with the game, the large Cohen's $d$ value provides a strong indication that this difference in score is largely due to viewing the heatmaps and agents. This is further supported by how participants rated the usefulness of the heatmaps and agents. Particularly, three out of four participants rated the heatmaps and agents as useful for learning something new ($M$=3.38, $SD$=1.80) and all four rated the heatmaps or agent valuable for game design or balancing purposes ($M$=4.38, $SD$=0.75).

Moreover, a strongly negative and significant correlation between participants' average initial score before using RealmAI and how much they learned from using the tool was observed ($r$=-0.95, $p$=0.052). In other words, participants who played worse initially, and arguably had lower levels of understanding of the game initially, learned more new things by using RealmAI. As such, this provides possible hints that RealmAI is especially useful in real-world game development scenarios during the early stages when developers understand their games less.

### 4.2.2 Qualitative Findings

A qualitative analysis of the open-ended questions also reveals several interesting insights that further support the quantitative findings above. For convinience, participants are referred to by indexes (e.g., P1 for the first participant).

First, observing the agent's behavior led P1 to revise their views on the aim assist feature in the game. While P1 initially disliked the feature, observing the agent convinced P1 that while it makes the game simpler, it allows players to focus more on movement and dodging. On the other hand, P3 and P4, who were interviewed as a group, realized that jumping could actually help the player move around the map more efficiently than walking. As such, observing the agent "opened [the game] up for more strategies [they] have never considered." In other words, having agents could be helpful for game developers to re-evaluate the impacts of certain features and mechanics by having an additional perspective.

The heatmaps and agents were also useful for generating insights into game balance. P1 noted that observing agents could be helpful in making adjusts to areas that are overperforming or underperforming compared to the other areas. Other participants also found the heatmaps and agents to be helpful for level design. For instance, based on a heatmap, P2 commented on one particular area of the map that, although accessible, is unused by agents. Similarly, they pointed out how some areas are more popular than other areas, and that while certain areas has "really interesting terrain that [they] would want players to explore", those areas are not popular at all with the agents. Moreover, P2 did not realize that the far right of the map has "good enemy

spawns" until seeing the heatmaps, which provided better insights into overall gameplay trends than observing an agent play the game. These comments provide good examples of how heatmaps and agents could be useful for level designing by hinting at changes that might be needed for particular areas of a map.

One surprising finding is that heatmaps and agents could be helpful not just by providing new insights, but also by confirming developers' intuitions about the game. Particularly, although P3 stated that they knew of the agent's strategy even before observing it, P4 remarked that "having some sort of proof [for their thoughts] is nice."

Interestingly, P1 observed that some aspects of how agents were optimizing their gameplay were unintuitive for human players. An example is how the agents jumped around while shooting the gun, which makes aiming harder and hence unrealistic for usual players. P4 also commented on this, stating that while the agent's strategy seemed more refined than their own, it was too precise and difficult for them to replicate. However, this observation has an interesting implication, which is that this is precisely what allows RealmAI's agents to be useful for catching hard-to-find bugs or exploits in a game that would otherwise be difficult to find for human playtesters.

# 5 Discussion and Conclusions

## 5.1 Evaluation of Final Design

### 5.1.1 Project Objective

The project achieved its objective of creating an automated reinforcement learning tool that generates statistics to help developers understand and balance their games before release.

### 5.1.2 Design Specifications

**Game Plugin Subsystem** The Game Plugin Subsystem contains the classes `RealmAgent`, `RealmSensor`, `RealmActuator`, `RealmScore`, which enables the RL algorithms to play the game automatically, and the classes `DataRecorder` and `VideoRecorder`, which collects the required data while the game is being played. These classes satisfy FS1 to FS4. Using GitHub, the total size of the subsystem is confirmed to be less than 100B, satisfying NFS5).

**Reinforcement Learning Subsystem** The RL Subsystem performs hyperparameter tuning, and is fully user-configurable. This includes customizing the number of hyperparameter tuning trials and number of training steps per trial, satisfying FS5 and FS6. The training can be paused and resumed, and training curves are generated by ML-Agents, satisfying FS9 and NFS3.

**Data Storage Subsystem** The Data Storage Subsystem enables all training data to be stored on the user's local file system, which is easily accessible, satisfying FS7 and FS11.

**Analysis Subsystem** The Analysis Subsystem can generate 5 different types of heatmaps, satisfying FS8.

**Report Subsystem** The dashboard enables users to generate and view heatmaps, satisfying

FS10. The ability to view recorded gameplay videos satisfies FS12. Lastly, heatmap generation is far below the 2 hour limit, satisfying NFS2.

**Overall System** The project code is available to users for free which satisfies NFS1. The documentation can be found within 2 clicks on the Report Subsystem which satisfies NFS4. Furthermore, Weights and Biases, which is the only online component, is optional. This satisfies NFS6. In addition, an end-to-end test of running all subsystems was conducted on Windows, macOS, and Ubuntu, which satisfies NFS7.

## 5.2 Use of Advanced Knowledge

This project uses upper-year ECE knowledge from several subject areas, ECE 457B - Fundamentals of Computational Intelligence, ECE 457C - Reinforcement Learning, ECE 452 - Software Design and Architecture, ECE 356 - Database Systems, and CS 349 - User Interfaces.

RL concepts and the different training algorithms are essential to this project. The Reinforcement Learning Subsystem requires upper year knowledge in ECE 457B (Fundamentals of Computational Intelligence) and ECE 457C (Reinforcement Learning) for the RL training algorithms: Proximal Policy Optimization [9] and Soft Actor Critic [10]. This knowledge is also used for the hyperparameter tuner.

Knowledge in ECE 452 (Software Design and Architecture) is used to dictate the overall software architecture of the project. This project follows the Pipe and Filter architecture where gameplay data is first recorded and sent to storage, then the data is retrieved when needed and goes through a series of filters (heatmap generation, processing, etc). Finally it is displayed in the Report Subsystem.

A data storage system is also used to store video replay files, training checkpoints, and any gameplay data recorded from the agents. Knowledge about how to record, format, and store data efficiently from ECE 356 (Database Systems) is applied to this project.

For the reporting system and the training manager, a user interface was designed that enhances ease of use and responds effectively to user interaction. Knowledge of design elements and frontend frameworks was acquired in CS 349 (User Interfaces). This was instrumental for understanding how to efficiently organize data such as generated heatmaps and recorded training videos.

## 5.3 Creativity, Novelty, Elegance

To summarize, playtesting is a very expensive process and is prone to mistakes. The iterative and sequential nature of game testing makes it a good problem to solve with RL. Currently, existing solutions are either proprietary, too generic, require the user to have expertise in RL, etc.

The novelty of this project stems from the idea of using agents to gather data through playing the game similar to how any play-tester would. This data is then available to the developers in different forms. For example, heatmaps can express the most common paths and video replay files to show how an agent would interact with a level. The generated data can help with finding bugs, exploits, game balancing, level tweaking, and helping developers visualize different play-styles and

strategies.

This project also features a novel use of RL in the game industry: to emulate player behaviour for the purpose of generating data for analysis. While RL has been relevant in the game industry, most projects involving RL and games focus explicitly on bug-finding or research in RL methods instead of generating gameplay data.

The elegance of this design is the simplicity and easy to integrate plugin. Game developers would add on an additional plugin to their Unity workflow and configure a few parameters in the Reinforcement Learning Subsystem. With the click of a few buttons, a game developer is granted useful statistics generated from agents playing the game millions of times.

## 5.4 Quality of Risk Assessment

Within the Project Specifications and Risk Assessment Document submitted in ECE 498A, several possible risks were identified. The team was able to successfully address these risks, as detailed below.

**User Risks** There were no physical components in the project, so there were no physical safety hazards that could endanger users. The only potential non-physical risk was the safeguarding of sensitive user data. However, this was mitigated by enabling the whole system to be utilized offline without reliance on cloud infrastructure. Thus, by adhering to secure data storage standards, any catastrophic risks to users, such as data loss, were averted.

**Technical Difficulty** The team was able to identify this risk as high impact and of moderate probability. The team was able to effectively plan deliverable tasks according to their technical capabilities and allocated time to educate themselves on the relevant required subject areas. Furthermore, the scope of the project was kept focused on a specific game genre and the different subsystems were designed to be loosely coupled.

**Insufficient Time** The team identified this risk as high impact and of moderate probability. Despite these challenging and uncertain times, the team was able to make adjustments accordingly to any unforeseen roadblocks. Adhering to a gantt chart to plan deadlines enabled the team to complete deliverables on time and ensure team members could become unblocked quickly.

## 5.5 Student Workload

The workload of each student was tracked beginning in ECE 498A, as well as throughout ECE 498B. Overall, all group members contributed an approximately equal amount of work across the entirety of the project. Ken Ming Lee (20%), Ethan Lee (20%), Cliff Li (20%), Matthew Tang (20%), and Michael Meng (20%). Overall, the total number of hours of work contributed from all team members was approximately 1310 hours.

# References


[1] W. Luton, *Making better games through iteration*, Oct. 2009. [Online]. Available: https://www.gamasutra.com/view/feature/132554/making_better_games_through_.php?print=1.

[2] P. Mirza-Babaei, N. Moosajee, and B. Drenikow, "Playtesting for indie studios," in *Proceedings of the 20th International Academic Mindtrek Conference*, ser. AcademicMindtrek '16, Tampere, Finland: Association for Computing Machinery, 2016, pp. 366–374, ISBN: 9781450343671. DOI: 10.1145/2994310.2994364. [Online]. Available: https://doi-org.proxy.lib.uwaterloo.ca/10.1145/2994310.2994364.

[3] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, *Unity: A general platform for intelligent agents*, 2020. arXiv: 1809.02627 [cs.LG].

[4] T. Minkkinen, "Basics of platform games," 2016.

[5] B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra, "On the importance of hyperparameter optimization for model-based reinforcement learning," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2021, pp. 4015–4023.

[6] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, *et al.*, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2018.

[7] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[8] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: https://www.wandb.com/.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[10] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018. arXiv: 1801.01290 [cs.LG].

[11] M. Toftedahl, *Which are the most commonly used game engines?* https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/Which_are_the_most_commonly_used_Game_Engines.php, 2019.

[12] *About vector graphics*, https://docs.unity3d.com/Packages/com.unity.vectorgraphics@2.0/manual/index.html, 2020.

# Appendices

## Appendix A Choice of Game Engine and Unity's Package System

A significant choice for game developers when starting a new project is the choice of the game engine. A game engine can be described as a game production software that generally provides:

- Code libraries to handle complex tasks often required in games such as physics and rendering
- GUI tools to help human developers create game content, such as tools to create user interfaces or to arrange objects in a game level.

In general, modern popular game engines can be seen as level editors that allow game developers to arrange objects to create game levels and write code to control the objects to behave in specific ways. A screenshot of a game engine called Unity is shown in Figure 21.

Currently, there are many different game engines to choose from, with popular choices being Unity Engine by Unity Technologies and Unreal Engine by Epic Games. A 2019 survey of games published on Steam and Itch.io found that games most often use the Unity and Unreal game engines. In addition, it concludes that Unity is "very popular among hobbyist and indie developers" [11].

Additionally, Unity has a package system that allows third party's to extend the game engine with custom functionalities. A Unity "package" is a container for code libraries or GUI tools that can be imported into Unity through its package system. For example, Unity cannot render vector graphics (like SVG files) by default, but a Vector Graphics package can be imported into Unity to add additional functionalities for creating, editing, and rendering SVG files [12].

Given the popularity of Unity, it is a clear choice to develop the Game Plugin Subsystem as a single Unity package that a game developer can import into Unity using its packing system.

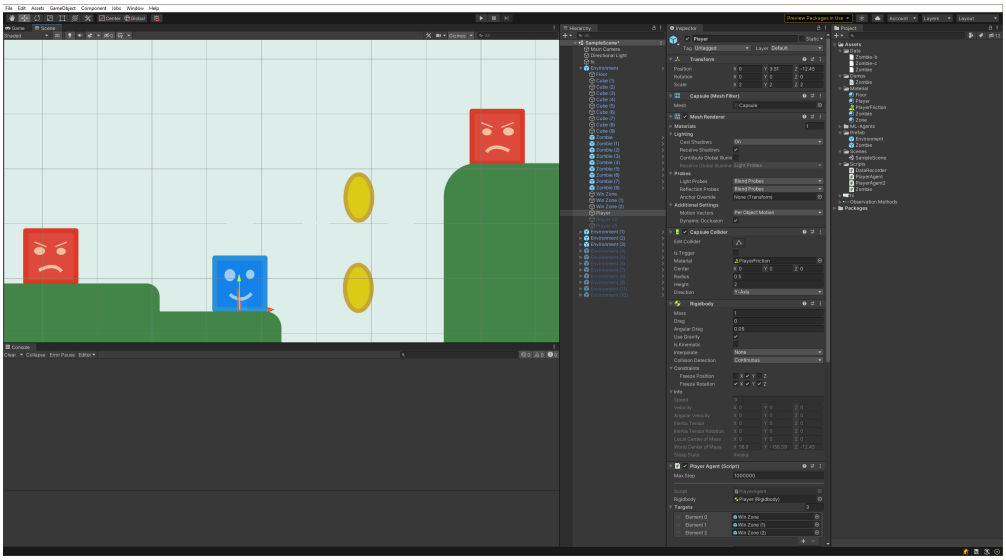


Figure 21: Screenshot of a game being built within the Unity editor