RocketDRL: a Novel 3D Environment for Training Reinforcement Learning Agents

Hyuan P. Farrapo
Virtual UFC Institute
Federal University of Ceará
Fortaleza, Brazil
hyuan007@alu.ufc.br

José G. R. Maia
Virtual UFC Institute
Federal University of Ceará
Fortaleza, Brazil
gilvanmaia@virtual.ufc.br

Paulo B. S. Serafim
Instituto Atlântico
Fortaleza, Brazil
paulo_serafim@atlantico.com.br

Rômulo F. Filho
Teleinformatics Engineering Department (DETI)
Federal University of Ceará
Fortaleza, Brazil
romulofffufc@gmail.com

ABSTRACT

The development of autonomous agents capable of presenting more human-like behavior is currently driven by Deep Reinforcement Learning techniques. Deep Reinforcement Learning is an active field of research that is fueled by virtual environments usually inspired or borrowed from video games. Several works in the field are limited to playing classical control tasks, 2D environments, or outdated games. Therefore, most environments used in research are significantly different from those available in current trending 3D games. This paper introduces RocketDRL, a novel Deep Reinforcement Learning environment which supports mechanics for 3D games inspired by the popular "car football" game Rocket League. Besides the classical gameplay, we implemented three challenging minigames based on the mechanics from this title with an advanced simulation of physics with fine-grained car control: penalty shoot, free kick, and aerial shoot. Moreover, we also provide promising baseline results using Unity's ML-Agents Toolkit, which is an easy way to train and evaluate the agents.

CCS CONCEPTS

• **Computing methodologies → Computer vision**; **Intelligent agents**.

KEYWORDS

virtual environments, autonomous agents, reinforcement learning, simulation

1 INTRODUCTION

In the history of virtual environments development, the search for autonomous controlled behavior, that would mimic human-like interactions in complex layers, is driving the industry for years until this day [10]. This kind of agency can greatly improve the relevance of evaluations within its dynamics. Even in environments with simple mechanics, the uniqueness in the interactions between pre-programmed entities and autonomous agents is valuable [1].

Although works on the field tend to train agents for common control tasks [7] or even classical 2D games [12], in recent years, we have seen advances in the development of autonomous agents for 3D environments. At this moment, the works converged to use Deep Reinforcement Learning (DRL), a technique that combines classical Reinforcement Learning algorithms and the progress in Deep Learning research [14]. Widely used in the construction of autonomous robots [15], most DRL research uses digital games as a benchmark environment [10].

One of the first video games to aim for unique complex interactions through the environment was Pac-Man[1], where the ghosts had their own unique heuristic behaviors. With the adoption of simple rules, this method created a challenging gameplay loop, improving the player experience throughout the levels. As the games were evolving, the variables through the mechanics and the environments developed became richer and bigger. Thus, the traditional scripted mode of implementing behaviors became harder.

With the progress in computational power, available for end-users, the cost of developing, training, and evaluating Machine Learning techniques decreased. It was in this context that DRL arises, with the seminal work of [13]. At the time, creating an autonomous agent capable of learning how to play Atari games by only seeing the pixels of the screen was a big milestone. Since then, DRL has emerged as an important research area.

This technique is being used with real-life situations, such as automated driven systems [5], financial trade systems [21], and recommendation systems [22]. However, the most common virtual environments used to train the agents are video games. In this case, the agent act as a player and has to learn suitable behaviors accordingly. Therefore, we can have more immersive experiences, with challenging rivals or more natural partners driven by artificial intelligence.

In this way, a game that summarizes some real-life applications and stimulates the dynamics of complex interactions is Rocket League[2]. It presents a virtual 3D environment composed of a car football match with advanced rules and skills that makes the player deal with unusual tasks. These characteristics make Rocket League a potential strong virtual environment for DRL development.

This work presents RocketDRL, a novel fully 3D environment inspired by Rocket League and developed under Unity ML-Agents [8]. RocketDRL allows single or multiple agents and presents all of the basic mechanics from Rocket League. Besides the full game,

[1] https://pacman.com/en/
[2] https://www.rocketleague.com/

we also created three minigames to enable agents' training under different complexities. The first minigame, called Penalty, expects the agent to kick the ball inside the goal. The Free Kick minigame puts a barrier between the player and the goal. Finally, the Aerial minigame expects the player to shoot the ball in the air. RocketDRL is an environment suitable for DRL training, capable of empowering the agents to learn unusual behaviors in different tasks. A resume of the interaction dynamics in the environment is showed in Figure 1.

This paper is organized as follows. In Section 2, we present a brief description of Deep Reinforcement Learning and the methods used in this work. In Section 3, we present works that use digital games as a testbed for autonomous agents. In Section 4, we present the RocketDRL environment, and its configurations. In Section 5, we evaluate the proposed environment. Finally, in Section 6, we present closing remarks and future works.

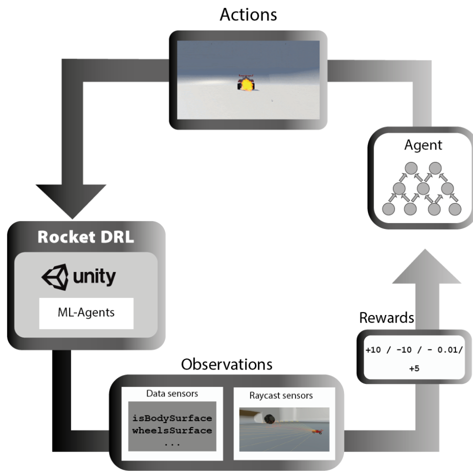

Figure 1: RocketDRL is a fully 3D environment developed with Unity and the Unity Machine Learning Agents Toolkit, featuring custom integration with DRL agents. The implemented agent can perform any of the actions available in the environment, that get processed by the environment, providing new observations and rewards of the corresponding action to the agent.

2 BACKGROUND

In this section we present a brief description of Reinforcement Learning, its modern combination with Deep Learning, called Deep Reinforcement Learning, and Proximal Policy Optimization, the algorithm used to train the agents presented in this work.

2.1 Reinforcement Learning

.

Reinforcement Learning is a learning paradigm that deals with learning through the interaction of an agent with the environment [18]. Traditionally, for each timestep, t, the environment presents the current state observation, o_t, to the agent, which executes one action, a_t, and receives a corresponding scalar signal, called reward, r_{t+1}. The ultimate goal is to maximize the sum of discounted rewards, G_t, in which

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \, , \tag{1}$$

and γ, $0 \leq \gamma < 1$, is a discount factor to ensure that the above sum always converges.

To achieve this goal, the agent has to learn the optimal probability distribution of actions for each observation. This probability is called policy, π, and defines the behavior of an agent. In some cases, there is only one action with probability of 1.0 for each state, which defines a deterministic policy. However, the most challenging problems are stochastic, which means that a policy returns a probability of several possible actions.

One way to define the policy is computing the gradient of a performance function, J, with parameters θ, and update it using gradient ascent. Intuitively, the performance function indicates if the current policy is performing well. Formally, it can be defined by the sum of the expected rewards by following the current policy, such that

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_t R(\tau) \right] \, , \tag{2}$$

where τ is the trajectory, i.e., the sequence of observations and actions followed by the agent, and $R(\tau)$ are the rewards received by following this trajectory. Therefore, we can update the values of θ using

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid o_t) R(\tau) \right] \, . \tag{3}$$

When the observation space or action space grow, computing the performance and its gradient becomes much harder.

2.2 Deep Reinforcement Learning

The first method of Deep Reinforcement Learning (DRL) was presented by [13] and effectively started a new research area. In the paper, the authors presented a combination of Convolutional Neural Networks and Q-Learning [20], a value-based Reinforcement Learning algorithm, which was capable of beating seven Atari games receiving the raw pixels of the screen.

Most modern DRL methods are policy-based, i.e., they use policy gradient algorithms to find the optimal policy. The gradients are obtained from automatic differentiation, backpropagated through the Deep Neural Network, then the weights, θ, are updated.

There are several methods of policy gradients in the literature. In this work, we use Proximal Policy Optimization (PPO) [16]. It is an algorithm already present in the ML-Agents toolkit, which facilitates the reproduction of our work. PPO uses both value functions and policy loss to update the weights, using an advantage function, A_π, in which

$$A_\pi(a \mid o) = Q_\pi(a \mid o) - V_\pi(a \mid o) \, , \tag{4}$$

where

$$Q_\pi(a \mid o) = \sum_t \mathbb{E}_{\pi_\theta} \left[R(a_t \mid o_t) \mid o, a \right] \tag{5}$$

and

$$V_\pi(a \mid o) = \sum_t \mathbb{E}_{\pi_\theta} \left[R(a_t \mid o_t) \mid o \right] \, . \tag{6}$$

The loss function, $L(\theta)$, is defined such that PPO maximizes the surrogate objective function

$$L(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t \right) \right] , \quad (7)$$

where $\hat{\mathbb{E}}$ and $\hat{A}$ are, respectively, the expectation and advantage estimates obtained empirically, $r(\theta)$ is the ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid o_t)}{\pi_{\theta_{old}}(a_t \mid o_t)}, \quad (8)$$

and the $clip$ function limits the lower and upper values of $r_t(\theta)$ to $1-\varepsilon$ and $1+\varepsilon$, respectively.

With the updated weights, the current observation is passed to the Neural Network, which selects one of the possible actions. The agent executes this action and receives a reward. This is the main dynamic of agent interaction with the environment.

3 RELATED WORKS

In this section, we describe three important DRL Toolkits, which allow researchers to train and evaluate agents in several games. We also present learning environments and discuss why Rocket League is a good DRL task.

3.1 DRL Toolkits

There are many virtual environments available for research, including games [11], the stock market [21], and physics simulations [4]. Some of them are wrapped in a toolkit, that may contain more environments and other development tools. Here, we present three important DRL toolkits, chosen because of their relevance in the research field.

3.1.1 Arcade Learning Environment. The Arcade Learning Environment (ALE) [2] was one of the first sets of environments used to train DRL agents. It was built using the Atari 2600 emulator Stella, which allows the user to interact with the game, capture the screen buffer, and share metadata, transforming each of its 60 games in a different RL task. However, since the Atari 2600 did not have support for 3D games, ALE also does not allow training and evaluating 3D agents.

3.1.2 OpenAI Gym. The OpenAI Gym [3] was presented as a set of tools for building, customizing, and distributing environments for Reinforcement Learning research. It includes a collection of benchmark problems and also proposes a standard interface for learning environments. It is highly focused on the environment instead of the agent. One of the main features of Gym is that almost any existing environment, whether 2D or 3D, can be adapted to work according to Gym patterns through the use of a Python API. By doing this, Gym allows for easy customization of environments and comparison between different agents.

3.1.3 Unity Machine Learning Agents Toolkit (ML-Agents). Presented by Unity 3D, the Machine Learning Agents Toolkit (ML-Agents) [8] provides an interface to train Reinforcement Learning agents in the games and simulations built with the platform, essentially making them Learning Environments. The researchers can take advantage of a Python API to communicate with the engine, allowing the implementation of agents using the standard Machine Learning Python libraries, such as TensorFlow.

The toolkit has been made available with a set of pre-built Deep Reinforcement Learning techniques and scenarios, making it easy to quickly test environments and agents. Since Unity 3D is a very popular free engine, used both in small projects and professional games, and is also capable of working with 2D, 3D, Augmented, and Virtual Reality applications, it was chosen as the main platform for developing this work.

3.2 DRL Environments

As the interest in Reinforcement Learning researches grew, many environments were developed [17], covering different game genres, and were made public for use. The Google Research Football [10] is a 3D environment that simulates a football video game. There are environments focused on first-person shooter (FPS) games, such as ViZDoom [9] and Deep Mind Lab [1]. Modern mainstream games are also implemented as game environments for DRL agents training, like MineRL [6], an environment on top of Minecraft, and PySC2 [19], based on the Real-Time Strategy game StarCraft II.

Our work found RoboLeague [3] as starting inspiration, a Rocket League clone proposed to work as a Reinforcement Learning Environment, with some features like physics interactions and approximate visuals of the base game.

However Roboleague did not implement some key features of Rocket League, such as essential car controls (like Double Jump, and flipping the car), and base rules of the main game (like goal detection, division by teams, timers). It also presents complex reutilization of developed components, resulting in a simplistic way of using the ML-Agents toolkit, not reflecting the complex base game behaviors.

Addressing these points, we deliver RocketDRL, a fully new 3D environment, presenting the main key features of Rocket League, with three minigames, ready to be used as a benchmark for Deep Reinforcement Learning agents.

4 PROPOSED ENVIRONMENT

In this section, we describe the RocketDRL features, reward distribution, observation format, actions, and minigames tasks.

4.1 Game Overview

Inspired by the mechanics of Rocket League, our game environment openly available[4] presents a match of football, with cars as participants. The players of the match are divided by teams (going in the range of 1v1, 2v2 until 4v4), and have a total of five minutes to play the game, which goes on a real-time scale. During the time of the match, if someone scores a goal, the timer is paused, and the players are reset to the default positions. The game restarts with a score point being attributed to the team of the scoring player. After time out, the team with the most score points is the winner.

The player controls the car, which model is presented in Figure 2. The possible actions are to accelerate, turn the car, and do other special controls, like jumping with the car, in which the player can regulate the height of the jump according to the duration of the input. If the user activates the input one more time, the car can execute a double jump, which adds an impulse another time while

[3]https://github.com/roboserg/RoboLeague
[4]https://github.com/Hyuan02/RocketDRL

in the air. In case the car is in the air, it is possible to skew the body of the car and perform stunts, with pitch and row axis.

Figure 2: The car model used in the environment, in its front and side view, inspired on Rocket League cars. The model was originally made by RoboSerg.

It is also possible to use a special boost that adds a forward impulse to the car, interfering with his acceleration. In the case of the car being in the air, it adds an impulse on the car that makes it possible to land more quickly, or to plane in the air by more time, depending on the impulse direction.

The scale of elements in the game is more for fun, being the ball bigger than the car. This causes a less complex interaction between the player and the ball, with more ease to perform stunts and tricks instigating the player to execute football movements with little effort. The model of the ball [5] is presented in Figure 3.

Figure 3: The ball model used in the environment.

In the part of the physics simulation, aiming for a more arcade gameplay, the environment behaves in a non-realistic manner, with common world simulation values, like gravity, being different from their real counterparts. The car and the ball have their own frictional models, composed of rigid bodies. Moreover, the ball has a particular way to react to bounces and collisions within the environment.

The game utilizes simple shapes in the collision models to improve performance. The car is composed of an Oriented Bounded Box (OBB) for the body and spherical shapes for the wheels. The spherical mesh is utilized for the ball too. Furthermore, the stadium has its own collision mesh.

Taking RoboLeague as a starting point, the implementation was remade and improved for less coupling, easing the integration of Unity ML-Agents into the agents of the game. The key assets, like the car and stadium models, were also reused, making use of the meshes for the graphics. For the agent mechanics, every core aspect of the gameplay was turned into a component, splitting key controls, like the ground control of the car, the jump mechanic, boosting, and stunting.

[5]https://www.cgtrader.com/free-3d-models/car/sport/free-3d-model-rocket-league-ball

The properties of the car are managed by a wrapper and centralized in one component that stores physical properties, like the forward velocity of the car and state data, like the car jumping state and if it is boosting, making it easier to read. For the controls aspects, an interface was implemented, increasing modularization, dividing the inputs into signals that are utilized by the manager and distributed for the components.

The physics behavior was implemented using Unity‘s default system, with objects making use of native rigid bodies components and their signals, and adjusting some global variables like gravity for the whole simulation. The objects were composed of collision and trigger shapes, splitting the responsibilities between interaction functions and reaction behaviors. To receive these signals, custom behaviors were implemented, like the ball’s friction model.

To maintain compatibility with different types of game rules, the match logic was implemented deriving from a base class, increasing code reuse. The main game flow was developed adding core mechanics like the timer, team allocations, and scores, but for simplified development, the scene is presented with only one player, but it can be increased if needed.

The objects in the scene, like the ball and the car, have customized parameters, starting with default values according to the Rocket League reference values[6]. Parameters like the weight, friction, and scale of objects can be customized utilizing Unity’s native components, which modifies the mechanics of the environment. Abstract parameters that influences directly the mechanics of the game can also be customized, like the boost impulse or cost per second.

4.2 Reward Distribution

In the minigames routine, similar rewards were dfined because of their general purposes. Being the main objective of the minigames to score a goal, the agent receives a positive reward of 10 points in case of success. To encourage the agent to perform the correct actions fast, a negative reward of -0.01 was added at every step of an episode. If the time of a minigame runs out or if the agent fails to achieve the goal, it receives a negative reward of -10, which ends the episode.

For the aerial minigame, an intermediate positive reward of 5 points is added if the agent does the entitled stunt while touching the ball one time during the episode, being above 4 meters of the ground. That reward was implemented after previous training with non-satisfactory behaviors, in a form of encouraging the agent into doing the advanced objective. After that, if it scores a goal, 10 points are also given. The stunt in the aerial minigame is mandatory for the agent to score the maximum reward.

4.3 Observations

For the state observations passed to the agent, we used a combination of data sensors, with normalized values of the importance of the game environment, and 3D raycast sensors, that come natively with the ML-Agents toolkit.

The sensors were implemented in an independent manner from the start, taking key considerations from the game that are important for a human-based player, and then refined throughout

[6]https://github.com/RLBot/RLBot/wiki/Useful-Game-Values

the training sessions, aiming to maximize the performance of the autonomous agent. The specifications for the minigames are below.

4.3.1 *General Data Sensors.*

For the general sensors used in the three minigames, we have:

- boolean sensor, informing if the car is ready to drive or not.
- boolean sensor, informing if the body of the is on the ground or not.
- integer sensor, informing how many wheels of the car are on the ground.
- boolean sensor, informing if the car is jumping.
- float sensor, informing the boost quantity ready to use.
- boolean sensor, informing the car is using the boost or not.
- integer sensor, informing the direction of the car velocity through the sign.
- float sensor, informing the speed of the car.
- float sensor, informing the steer angle of the car.
- 3D vector sensor, informing the rotation of the car on Euler angles.
- 3D vector sensor, informing the related distance vector of the car and the ball.
- 3D vector sensor, informing the related distance vector of the ball and the goalpost.

4.3.2 *Barrier Data Sensors.*

For the barrier minigame, we have one specific data sensor:

- 3D vector sensor, informing the related distance between the barrier and the ball.

4.3.3 *Aerial Data Sensors.*

For the aerial minigame, we have three specific data sensors:

- enumerator state sensor, informing the if the ball is frozen, or in movement;
- boolean sensor, informing the aerial stunt was made;
- enumerator state sensor, informing the ball was touched, or is waiting for game reset.

4.3.4 *General Raycast Sensors.*

For a proper orientation and additional data about the environment, we added a raycast sensor (Figure 4 and Figure 5) component in the agent, that casts rays at a maximum angle of 70 degrees in the front part of the car, detecting the ball and also the barrier in the case of the barrier free kick minigame. The rays have a reach of 40 units and are splitted in 8, with a difference of 8.75 degrees each in the Y direction.

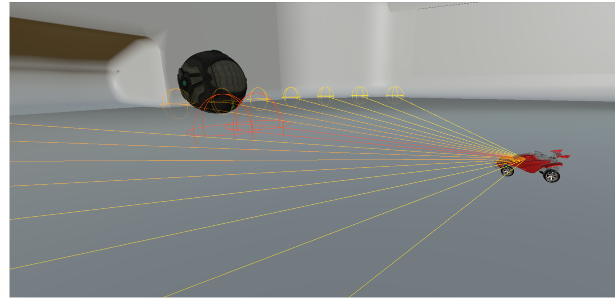

Figure 4: The rays casted from the native ML Agents component integrated on the agent, viewed from the side, colliding with the ball.

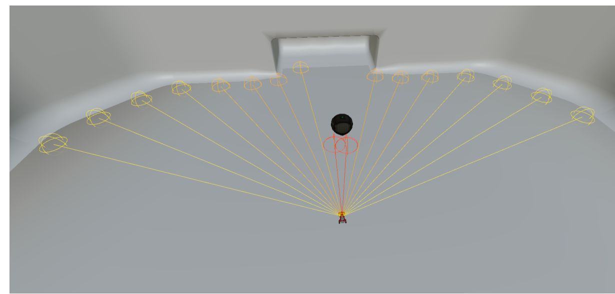

Figure 5: A top-down view of the rays casted through the environment by the agent.

4.4 Actions

For the actions of the agent, we implemented an interface, simulating the input of player controls, like button presses and axis handling, with discrete and continuous actions. The process of request an action from the agent was made every 5 steps of the episode. The agent was free to do actions between these intervals.

4.4.1 *Continuous actions.*

There are two continuous actions:

- float action, varying -1 to 1, simulating the acceleration, brake and rear input; and
- float action, varying -1 to 1, simulating the turning axis of the car.

4.4.2 *Discrete actions.*

There are three discrete actions:

- binary action, varying 0 to 1, simulating the jumping input;
- binary action, varying 0 to 1, simulating the boosting input; and
- binary action, varying 0 to 1, simulating the drifting input.

4.5 Minigames Task Description

To escalate the process of training through the developed environment, we implemented three minigames that the agent has to utilize the core mechanics of the game to achieve their goals.

The first one, is a classical of football games, based on the rules of a penalty kick (Figure 6). The agent and the ball have a random initial position on the Z-axis. After that, it marks the start of the minigame, that the agent has to kick the ball to the preferred goalpost. In case it scores a goal, the agent wins the round. If not, the agent loses and scores nothing. The minigame repeats on cycles after the result of the kick, going cumulative if the agent scores more goals.

Barrier free kick (Figure 7) is the next minigame developed. Also inspired by football games, its goal is to mark a score avoiding the barrier in front of the goalpost. The car, the ball, and the barrier position are randomized, and the barrier can be sometimes above the ground, force the agent to do a low kick with the car.

Finally, we developed a harder minigame, stimulating the use of an advanced technique learned by more experienced players (Figure 8). It consists in making a goal by driving the car into the air, and aim at the ball, which is dropped after three seconds of the

start of the game, by using the steering commands into the air, the agent adjusts the car to do the right effect that makes the ball go towards into the goalpost. The name of this stunt is called Aerial. The goal is only valid if the referred stunt is done.

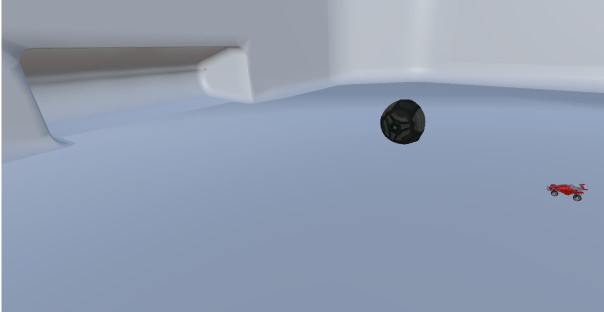

Figure 6: A screenshot of the environment on the penalty minigame, at the start of the episode.

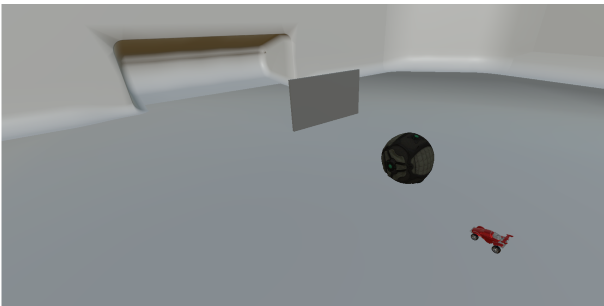

Figure 7: A screenshot of the environment on the barrier minigame, at the start of the episode.

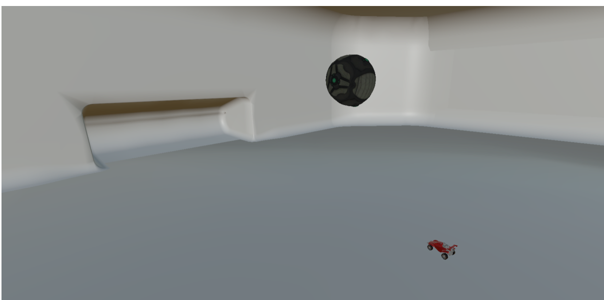

Figure 8: A screenshot of the environment on the aerial minigame, at the start of the episode.

5 EVALUATION

This section presents the experiments made to evaluate the use of the environment as a Reinforcement Learning task. The behavior of the agents when playing the minigames can be viewed in the accompanying videos.

5.1 Unity ML-Agents

The ML-Agents toolkit features some DRL algorithms, that could be integrated and adapted to the environment. The DRL algorithm used for the training of the agents was PPO [16], chosen for being a popular algorithm used for agent training of DRL tasks, and serving for the accomplishment of validating the existence of a positive learning rate throughout the training process. Utilizing this model, it is possible to customize some hyperparameters, which are stored in a configuration file used by the trainer. For each minigame, a custom configuration file was written, where the parameters were refined, tested in a portion of the training sessions, and in case of promising results, used on the definitive session length. To facilitate the reproduction of our experiments, we present the configuration files used in Appendix A.

The models were trained with the use of concurrent instances in the scenes. Since the proposition was to validate a learning rate progress and present initial complex behaviors by autonomous agents, aspects about the setup used or training performance were not considered in the evaluation process. Below are presented the fundamental parameters used to customize the trainer in the process, beginning with the hyperparameters.

- batch_size: Number of experiences that are collected before a new iteration of the gradient descent. It does not have a limit range, but the typical range in PPO varies from 512 to 5,120.
- buffer_size: Number of experiences that are stored for the process of learning and updating the policy model. While it does not have a limit range, typically the PPO range used varies from 2,048 to 409,600.
- learning_rate: The coefficient that influences the model update strength. The typical range varies in 0.00001 to 0.01.
- beta: Being a PPO-specific hyperparameter, it controls the possibility of the agent taking random actions through the environment, that with a greater value, it can lead to a bigger exploration across the environment by the agent. The typical range is 0.01 to 0.0001.
- epsilon: It controls the velocity of the policy updates during training. The typical range is between 0.1 and 0.3.
- lambd: A coefficient that controls the dependency of the agent on its previous values before updating to a new value. The typical range is 0.9 to 0.95.
- num_epoch: Controls how the policy is updated, with how many passes the gradient descent operation will do through the experience buffer. The typical range is 3 to 10.
- learning_rate_schedule: Controls if the learning rate maintains its value over time or decreases linearly. It can have two values: linear or constant.

It is possible to customize the network used in the training. For this training, we used a simple model, with few layers and hidden units. The customizable settings are described below.

- normalize: a boolean value that determines if the observation inputs should be normalized in the training process. The normalization is based on the running average and variance of the values during the training process.

- hidden_units: The number of units on the hidden layers of the neural network. Typically, it should be in the range of 32 to 512.
- num_layers: Number of layers of neural network. Typically, it varies from 1 to 3.
- vis_encode_type: Encoder type in case if visual observations are used. It is possible to choose from 5 different implementations, varying from a default value, "simple" that is a traditional implementation using two convolutional layers, or the resnet implementation, which is a complex network consisting of three stacked layers.

The parameters for reward signals were based on values of references of default examples of the ML-Agents toolkit. It is possible to choose the type of reward: intrinsic (with custom implementations like RND or Curiosity) or extrinsic. On the extrinsic implementation, it is possible to customize some parameters like:

- gamma: A downsampling factor for future rewards into the environment. The default range for this parameter is 0.8 to 0.995.
- strength: A multiplier factor for the rewards received by the environment. It is common to use a neutral value, like 1.

There are some general parameters of the trainer that need to be configured too:

- max_steps: Parameter that measures a length of a training. A typical range for this parameter is between 500,000 and 10,000,000.
- time_horizon: Defines the number of steps that are collected to be added to the experience buffer. The typical range is from 32 to 2048.

5.2 Results

In this subsection, we present the results of training the PPO configured agent, through the three minigames developed, and explain the performance and the behavior of the agent through the games.

5.2.1 Penalty. In this minigame, that the agent has to shoot the ball to the goalpost with no obstacles, the results were optimal, and after 30 million steps, the model converged with the agent hitting almost the totality of the shoots through the goal, achieving a very precise performance. An image with the results is shown below (Figure 9).

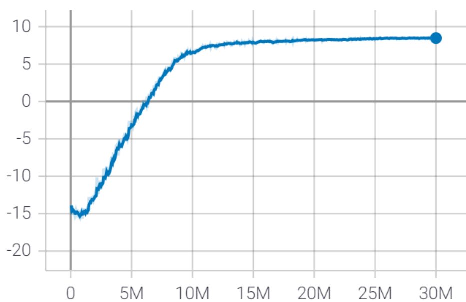

Figure 9: The reward coefficient results of the penalty training.

5.2.2 Barrier. In this minigame, the agent has to kick the ball to the goalpost as before, but with a barrier between them, that appears at different positions every episode. The results in this minigame had moderate performance, with the agent achieving great precision when the barrier had a higher height and a mediocre precision when the barrier had a smaller height. The model converged to this performance after 50 million steps (Figure 10).

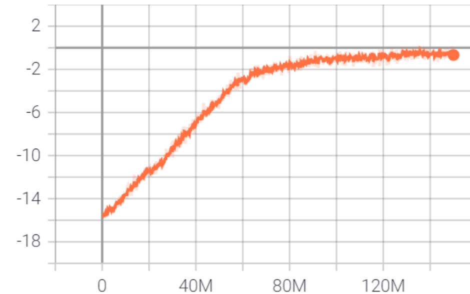

Figure 10: The reward coefficient results of the barrier training.

5.2.3 Aerial. The aerial minigame is an advanced minigame, made to test the performance of the agent, on executing complex skills, in which the agent only has valid goals, when is did an advanced air stunt, that touches the ball in the air, and it scores after. In this situation, the agent had a poor performance, understanding the objective, but with a very small amount of successful episodes. The agent was trained with 120 million steps, hitting its convergence (Figure 11).

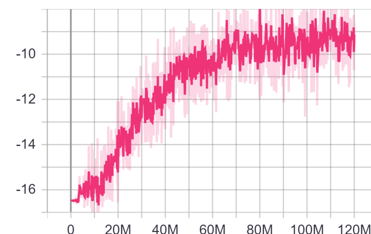

Figure 11: The reward coefficient results of the aerial training.

6 CONCLUSION

In this work, we introduce RocketDRL, a novel 3D learning environment inspired by Rocket League. The purpose of the project was to develop a deeply customized “car football” environment that allows for an intuitive process of development, training, and evaluation of Deep Reinforcement Learning agents. RocketDRL aims to increase the list of available environments to make agents learn complex immersive behaviors focused on a modern mainstream game.

To expand the learning possibilities, RocketDRL features three minigames designed to challenge the agents in the virtual 3D environment. To validate the possibilities of use for learning purposes,

we trained baseline agents, showing initial results and rich possibilities to achieve through the process of implementing autonomous agents. The agent trained to play the Penalty minigame learned the optimal behavior quickly. The other agents, although struggled to achieve optimal behavior, showed increasing performance. This indicates that RocketDRL is a suitable environment for the development of Deep Reinforcement Learning agents.

As future work, new minigames would expand the catalog of challenges, increasing the possibilities of evaluation in RocketDRL. Moreover, in this work, we used the baseline PPO agent present in ML-Agents, which is enough to validate RocketDRL as an adequate environment. However, other agents could be developed and evaluated in RocketDRL to be able of obtaining a satisfactory performance in all minigames.

REFERENCES

[1] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. 2016. DeepMind Lab. arXiv:1612.03801 [cs.AI]

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47 (2013), 253–279.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *ArXiv* abs/1606.01540 (2016). arXiv:1606.01540 http://arxiv.org/abs/1606.01540

[4] Misha Denil, Pulkit Agrawal, Tejas D Kulkarni, Tom Erez, Peter Battaglia, and Nando de Freitas. 2017. Learning to Perform Physics Experiments via Deep Reinforcement Learning. arXiv:1611.01843 [stat.ML]

[5] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An open urban driving simulator. In *Conference on robot learning*. PMLR, 1–16.

[6] William H. Guss, Cayden Codel, Katja Hofmann, Brandon Houghton, Noboru Kuno, Stephanie Milani, Sharada Mohanty, Diego Perez Liebana, Ruslan Salakhutdinov, Nicholay Topin, Manuela Veloso, and Phillip Wang. 2021. The MineRL 2019 Competition on Sample Efficient Reinforcement Learning using Human Priors. arXiv:1904.10079 [cs.LG]

[7] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. 2019. Learning to Walk via Deep Reinforcement Learning. arXiv:1812.11103 [cs.LG]

[8] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, et al. 2018. Unity: A general platform for intelligent agents. *ArXiv* abs/1809.02627 (2018). http://arxiv.org/abs/1809.02627

[9] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. 2016. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. 1–8. https://doi.org/10.1109/CIG.2016.7860433

[10] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. 2020. Google Research Football: A Novel Reinforcement Learning Environment. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 4501–4510. https://doi.org/10.1609/aaai.v34i04.5878

[11] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS Games with Deep Reinforcement Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 2140–2146.

[12] Ilya Makarov, Andrej Kashin, and Alisa Korinevskaya. 2017. Learning to Play Pong Video Game via Deep Reinforcement Learning.. In *AIST (Supplement)*. 236–241.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints* (2013), 1–9. arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236 arXiv:1312.5602

[15] Hai Nguyen and Hung La. 2019. Review of Deep Reinforcement Learning for Robot Manipulation. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 590–595. https://doi.org/10.1109/IRC.2019.00120

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *ArXiv e-prints* (2017), 1–12. arXiv:1707.06347 http://arxiv.org/abs/1312.5602

[17] Paulo Bruno Sousa Serafim, Yuri Lenon Barbosa Nogueira, Joaquim Bento Cavalcante-Neto, and Creto Augusto Vidal. 2020. Deep Reinforcement Learning em Ambientes Virtuais. In *Introdução a Realidade Virtual e Aumentada* (3 ed.), Romero Tori, Marcelo Silva Hounsell, Cléber Gimenez Corrêa, and Eunice Pereira Santos Nunes (Eds.). Sociedade Brasileira de Computação - SBC, Chapter 20, 423–436. http://rvra.esemd.org/

[18] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press.

[19] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. 2017. StarCraft II: A New Challenge for Reinforcement Learning. arXiv:1708.04782 [cs.LG]

[20] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King's College, Cambridge, UK. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

[21] Jia WU, Chen WANG, Lidong XIONG, and Hongyong SUN. 2019. Quantitative Trading on Stock Market Based on Deep Reinforcement Learning. In *2019 International Joint Conference on Neural Networks (IJCNN)*. 1–8. https://doi.org/10.1109/IJCNN.2019.8851831

[22] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. 2018. DRN: A Deep Reinforcement Learning Framework for News Recommendation. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 167–176. https://doi.org/10.1145/3178876.3185994

A CONFIGURATION FILES

A.1 Penalty Kick Minigame

The settings used in the penalty kick minigame are presented in Figure 12.

```
behaviors:
  Penalty:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 204800
      learning_rate: 0.001
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      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: 3
    max_steps: 30000000
    time_horizon: 1024
    summary_freq: 50000
```

Figure 12: Configurations used in the penalty minigame.

A.2 Barrier Free Kick Minigame

The settings used in the barrier free kick minigame are presented in Figure 13.

```
behaviors:
  Barrier:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 204800
      learning_rate: 0.0001
      beta: 0.0001
      epsilon: 0.1
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: constant
    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: 150000000
    time_horizon: 1024
    summary_freq: 100000
```

Figure 13: Configurations used in the barrier minigame.

A.3 Aerial Shoot Minigame

The settings used in the aerial minigame are presented in Figure 14.

```
behaviors:
  Aerial:
    trainer_type: ppo
    hyperparameters:
      batch_size: 3200
      buffer_size: 400000
      learning_rate: 0.001
      beta: 0.001
      epsilon: 0.28
      lambd: 0.93
      num_epoch: 8
      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: 120000000
    time_horizon: 1000
    summary_freq: 12000
    threaded: true
```

Figure 14: Configurations used in the aerial minigame.