AI4U: A Tool for Game Reinforcement Learning Experiments

Gilzamir Gomes
Department of Computing
Federal University of Ceará (UFC)
Fortaleza, Brazil
gilzamir@alu.ufc.br

Creto A. Vidal
Department of Computing
Federal University of Ceará (UFC)
Fortaleza, Brazil
cvidal@dc.ufc.br

Joaquim B. Cavalcante-Neto
Department of Computing
Federal University of Ceará (UFC)
Fortaleza, Brazil
joaquimb@dc.ufc.br

Yuri L. B. Nogueira
Department of Computing
Federal University of Ceará (UFC)
Fortaleza, Brazil
yuri@dc.ufc.br

***Abstract*—Reinforcement Learning is a promising approach to the design of Non-Player Characters (NPCs). It is challenging, however, to design games enabled to support reinforcement learning because, in addition to specifying the environment and the agent that controls the character, there is the challenge of modeling a significant reward function for the expected behavior from a virtual character. To alleviate the challenges of this problem, we have developed a tool that allows one to specify, in an integrated way, the environment, the agent, and the reward functions. The tool provides a visual and declarative specification of the environment, providing a graphic language consistent with game events. Besides, it supports the specification of non-Markovian reward functions and is integrated with a game development platform that makes it possible to specify complex and interesting environments. An environment modeled with this tool supports the implementation of most current state-of-the-art reinforcement learning algorithms, such as *Proximal Policy Optimization* and *Soft Actor-Critic* algorithms. The objective of the developed tool is to facilitate the experimentation of learning in games, taking advantage of the existing ecosystem around modern game development platforms. Applications developed with the support of this tool show the potential for specifying game environments to experiment with reinforcement learning algorithms.**

***Index Terms*—Games, Reinforcement Learning, Autonomous Non-Player Characters**

I. Introduction

The field of Artificial Intelligence (AI) in games is an established research field, which, nevertheless, is still growing and rapidly developing. There are many examples of AI applications in games, such as state/action evaluation, direct action selection, selection between strategies, modeling opponent strategy, content generation, and modeling player experience. AI techniques are commonly used to achieve the level of human performance when playing video games [1]. Several games serve as interesting and complex problems for AI methods, and many applications show how AI can benefit *gameplay* in video games [2]–[4].

One of the open problems in this field is how to obtain complex behaviors of virtual characters using end-to-end machine learning techniques. For this, Deep Reinforcement Learning (DRL) has been explored in several problems involving games and game development environments, and has obtained promising results since it was first proposed. In general, at each step of a DRL process, agents receive high-dimensional data and perform actions in accordance with policies based on deep neural networks. Then, a DRL-based learning mechanism updates the policies to maximize the expected return with an end-to-end method. Markovian Decision Problems (MDP) can be solved by several model-free reinforcement learning methods. The components of a MDP are the environment, the agent (and its actions), and the reward function. Often, defining the environment and the agent is straightforward; however, defining a reward function that expresses the desired behavior of an agent, especially in games, is more challenging, and can be an exhausting trial and error process.

Forgette and Katchabaw [2] proposed the use of RL and the concept of motivation for specifying virtual characters that are autonomous and believable. Developers must specify a list of motivations, which describe particular roles or characteristics in a narrative, and map them to reward functions. Thus, for a character to learn, it needs rewards, which, indirectly, means it needs motivations. Although this is a promising approach, the literature shows that the specification of reward functions in reinforcement learning can be a process subject to many false indications. The most interesting behaviors require the specification of reward functions that usually cannot be converted to a Markovian Reward Function (MRF) easily. So, describing a Non-Markovian Reward Function (NMRF) directly [5] can facilitate the problem specification since it is possible to convert NMRF into MRF automatically.

Another challenging problem is getting autonomous characters to perform complex behaviors using a process in which the modeler can control only one reward function that produces a scalar value [6]. Thus, if describing the behavior through a single reward function is not possible, the problem is broken down into simpler sub-problems, repeatedly, until they can be described by MRFs. Nevertheless, this task is subjected to a tiring process of trial and error. Therefore, incorporating it into game development requires tools that facilitate the specification of relevant reward functions, especially NMRF.

In this work, we present a tool to facilitate the preparation of reinforcement learning experiments in games. It provides the specification of reward functions intuitively and interactively, thus reducing the problem of obtaining behaviors induced by reward functions (especially NMRF). Also, it incorporates the visual specification of reward functions, and it is integrated with a game development platform. We show that the visual specification of the reward function is compatible with the formal specification of reward functions.

II. Background

For a better understanding of the methods used in the design of the tool presented in this work, it is important to highlight reinforcement learning and the formal specification of the reward generation component for the agent to learn effectively.

A. Reinforcement Learning

Mathematically, RL is idealized as a Markov Decision Process (MDP). As the agent interacts with the environment at a given instant t, it perceives a state s_t and executes an action a_t. The state s_t and the action a_t determine the next state s_{t+1} uniquely. For every action a_t, the agent receives a reward $r_t \in \mathbb{R}$. The cycle perception-action-reward progresses with time.

The discounted reward R_t from time t, as a measure of the agent's performance, is written as

$$R_t = \sum_{k=0}^{\infty} \gamma^k \; r_{t+k}, \qquad (1)$$

where r_t is the reward received after the transition from time step t to time step $t+1$, and $\gamma \in [0,1]$ is a discount term that adjusts the importance of the long-term consequences of the agent's actions.

The agent's policy consists in selecting an action, based on a state value function $V : S \to \mathbb{R}$ or on the $(state, action)$ value function $Q : S \times A \to \mathbb{R}$, to maximize R_t.

In Approximate Reinforcement Learning, value functions are approximated by parameters θ. The representation of the value functions can be a non-linear model, as several types of neural networks. Mnih et al. [1] shown that reinforcement learning with deep neural networks gets a superhuman performance in several Atari games. In this approach, the agent selects actions based only on high dimensional input representations (as the pixels of video-game frames). This achievement paved the way for the emergence of Deep Reinforcement Learning (DRL).

In DRL, θ is a set of weights of the neural network and, usually, gradient-based optimization is used to maximize the objective function in (1). Therefore, in DRL, the state value function is $V_\pi(s;\theta)$, which means the expected value from state s using the policy π based on parameters θ.

B. Formal Specifiction of Reward Functions

The component to the visual specification of reward functions presented in this work has the expressive power of a Reward Machine (RM) [5]. This component allows the specification of reward functions as the Finite Deterministic Automata (FDA). An RM [5] is a way to combine Markovian reward functions to shape a non-Markovian reward function. This composition is defined in accordance with a formal specification, using the following definitions.

Definition II.1. *Markovian Decision Process (MDP): An MDP with an initial state is a tuple $\mathcal{M} = (S, A, s_0, T, r, \gamma)$ where S is a finite set of states, A is a finite set of actions, $s_0 \in S$ is the initial state, $T(s_{t+1}|s_t, a_t)$ is the transition probability distribution, $r : S \times A \times S \to R$ is the reward function, and $\gamma \in (0,1]$ is the discount factor.*

Definition II.2. *Vocabulary and Labeling Function: a vocabulary is a set $\mathcal{P}$ of propositional symbols. A labeling function is a function $L : S \times A \times S \to 2^{\mathcal{P}}$ that maps experiences to truth assignments over the vocabulary $\mathcal{P}$.*

Definition II.3. *A Non-Markovian Reward Function: A Non-Markovian Reward Decision Process (NMRDP) is a tuple $(S, A, s_0, T, R, \gamma)$ where S, A, s_0, T and γ are defined as in MDPs, and (unlike in MDPs), $R : (S \times A)^+ \times S \to R$ is a non-Markovian reward function that maps finite state-action histories into a real value.*

Definition II.4. *Mealy Machine: Mealy Machine is a tuple $(Q, q_0, \Sigma, R, \delta, \rho)$ where Q is the finite state set, $q_0 \in Q$ is the initial state, Σ is the input symbols alphabet, R is the finite output alphabet, $\delta : Q \to Q$ is a transition function, and $\rho : Q \times \Sigma \to R$ is the output function.*

Definition II.5. *Setting: a setting is a tuple (S, A, P, L) where S, A, P, and L are defined as in definitions II.1, II.2, II.3, and II.4.*

Definition II.6. *Reward Machine (RM): a reward machine for a setting (S, A, P, L) is a Mealy Machine $(Q, q_0, \Sigma, R, \delta, \rho)$ where the input alphabet is $\Sigma = 2^{P}$, and R is a finite set where each $R \in \mathcal{R}$ is a reward function from $S \times A \times S$ to $\mathbb{R}$.*

Definition II.7. *Deterministic Finite Automata (DFA): A DFA is a tuple $Q, q_0, \Sigma, \delta, F)$ where Q is a finite state set, q_0 is the initial state, Σ is a finite input alphabet, $\Delta : Q \times \Sigma \to Q$ is a transition function, and F is the set of accepting states.*

In an RM, a sequence $(S \times A)^+ \times S$ is mapped in value assignments to propositional symbols in $\mathcal{P}$. The assignment of values for those symbols is input to the Reward Machine, which outputs a symbol from a Markovian Reward Function. Camacho et al. [5] propose DFA as an intermediate language for specifying reward functions in NMRDPs. In this way, any language that can be translated into a DFA can be used to specify non-Markovian reward functions.

III. Related Works

There is a wide range of environments and platforms that serve as simulators to evaluate the performance of Artificial Intelligence algorithms. Juliani et al. [7] organize these tools into four categories: **Environment**, **Environment Suite**, **Domain Specific Platform** and **General Platform**.

The **Environment** category consists of fixed, unique environments that are black boxes from the agent's perspective. Examples in this category are the games: *Pitfall* and *Space Invaders* [8]; and the environment, Obstacle Tower [7].

The Environment Suite category consists of a set of environments organized in a single package, which, in general, is used to test the performance of an algorithm or method in some dimensions of interest. Most environments in a suite share similarities in their state spaces and action spaces. Besides, they require similar (but not necessarily identical) skills to be resolved. Examples of these environments include: ALE [8], DMLab-30, Hard Eight [9], AI2Thor [10], OpenAI Retro [11], DMControl [12], and ProcGite [13].

The Domain Specific Platform consists of tools that allow the creation of sets of tasks in a specific domain, such as locomotion or first-person navigation. These platforms are differentiated from the final category by their narrow focus on the types of environments. This may include limitations on the physical properties of the environment or on the nature of possible interactions and tasks in the environment. Examples in this category include: Project Malmo [14], VizDoom [15], Habitat [16], DeepMind Lab [17], PyBullet [18] and GVGAI [19].

The General Platform includes tools that can create environments with arbitrarily complex visual, physical, and social interaction tasks. The set of environments that can be created by platforms in this category is a superset of those that can be created by (or are in) the other three categories. In principle, General Platforms serve to define any AI research environment of potential interest [7]. Juliani et al. [7] argue that modern video game engines are strong candidates for this category. Therefore, game engines like Unity [7], Unreal Engine [20], [21] and Godot [22] can be used to define complex AI game environments. Although any general game development platform can be used as a general platform, Unity took the lead by providing an open-source *framework* of agents with machine learning, the ML-Agent [7]. ML-Agent provides support for reinforcement learning, imitation learning, curriculum learning, and other approaches related to machine learning. It has a high-level programming interface with standard support for PPO (Proximal Policy Optimization) [23] and SAC (Soft Actor-Critic) [24] algorithms; and a low-level programming interface that allows Unity environment control from Python.

To extend the ecosystem of existing tools, we developed a Unity plugin with an emphasis on principles of simplicity, flexibility of choosing algorithms, and ease of specifying reward functions. The purpose of the tool is to facilitate the investigation of reinforcement learning applications in games.

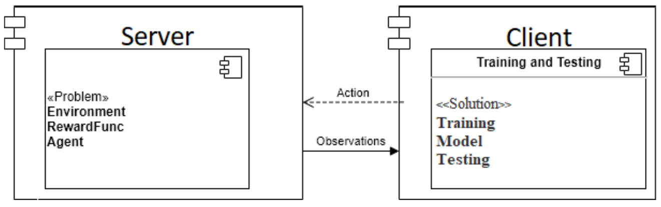


Fig. 1. AI4U main components.

Unity was chosen as an experimentation platform because it has rich documentation, an active development community, and the possibility to run on different levels of hardware and software architectures.

IV. AI4U: A Tool for Game Reinforcement Learning Experiments

In this paper, we expand the ecosystem of tools for reinforcement learning based on the General Platform with an emphasis on game creation tools. Thus, we named this tool *Artificial Intelligence for Unity*, as it was designed on the Unity platform. As mentioned before, AI4U is based on three pillars: simplicity, flexibility of choosing different implementations of RL algorithms, and ease of specifying reward functions. Simplicity is achieved through integration with Unity, taking advantage of its *scripting* system to provide built-in functionality, such as modules that allow one to define a reward function. The flexibility to choose implementations of RL algorithms is obtained through the automatic generation of code in a standard structure provided by Gym's *framework* [25]. Thus, based on provided environment specification, AI4U automatically generates a basic training and testing loop. The reward functions are easily specified through a visual modeling tool, which uses blocks that associate events in the environment with rewards assigned to agents.

Therefore, AI4U [26] is a tool for specifying game environments with features that easily integrate with reinforcement learning algorithms. It supports declarative and visual specification of the environment and reward functions. For this, AI4U has two main components (see Fig. 1): the server and the client. The server allows the specification of the environment, the agent, and the reward function. The client uses those definitions to obtain a decision-making model capable of leading the agent to present the behavior triggered by the reward function.

AI4U was integrated into the Unity game development platform, taking advantage of the flexibility provided by the platform's *scripting* system. For a better understanding of how AI4U can be used, next, we present the environment's specification components, a visual component for specifying reward functions, and the code generation to ease the flexible implementation of training and testing loops of reinforcement learning agents.

A. Environment's Specification Component

The environment's specification consists in determining the environment's properties, programming the objects' behaviors with which the agents interact, and defining the reward functions.

1) Definition of the Environment's Properties: A widely used specification for environments is given by *Gym's framework* [25]. Gym is very handy because many useful algorithms that are implemented and made available in open-source repositories are based on it. Thus, little adaptation is necessary to use those algorithms in new environments that follow Gym's environment format. Every environment modeled with AI4U is automatically associated with a configurable structure, which is compatible with that *framework*. For this, the developer only needs to use AI4U's Application Programming Interface (API), whose class diagram is partially shown in Fig. 2.

2) Programming Objects' Behaviors: There are two types of *Brain* objects available to each agent: an object that inherits from the *RemoteBrain* class and an object that inherits from the *LocalBrain* class. A *RemoteBrain* object allows control of the virtual character by a model being trained using some API (Application Programming Interface) external to AI4U. An object of the *LocalBrain* type allows the agent to be controlled by a decision-making model already trained and compiled to work locally, but it also allows control of the agent through external devices, such as mice, keyboards and joysticks. This functionality is important for debugging the game environment during development when the developer can test the game mechanics manually.

Agents implementing the *RLAgent* class have an associated reward that is affected by programming commands or events that occur automatically during the game and that are associated with objects of the type *RewardFunc*. An object of type *RewardFunc* is associated with one or more agents and an event in the environment. Fig. 3 shows the association of the modules *LocalBrain* and *RemoteBrain* with an object governed by the Unity physics engine.

A *script* containing the class *DPRLAgent* was also associated with the object. The class *DPRLAgent* is a specialization of the class *RLAgent*. With an associated *DPRLAgent*, an object can undergo physical control (application of force). To define the origin of the physical force, a controller *WASD* (class *WASDPRLController*) was associated with the object receiving the force.

The controller *WASDPRLController* allows applying physical force on target object employing the keyboard's W, A, S, and D keys. For the *WASD* controller to work, the object must be associated with a script named *LocalBrain*. For external remote control to work, the object must be associated with a script of the type *RemoteBrain*.

3) Reward Function Specification Component: When the game event occurs, a reward is calculated and assigned to agents associated with the corresponding reward function. The reward function specification tool allows the specification of Markovian and non-Markovian reward functions and is equivalent to a Reward Machine (RM). However, the visual specification takes advantage of the visual appeal and creative power of the modern game engines. The visual specification is usual practice in game development, for example, the composition of properties in materials through *shaders*, and virtual characters animation. Thus, in section IV-B, we present the visual specification of reward functions and their equivalence with the RM concept.

B. Visual Reward Function Specification

In general, the goals of NPCs in games consist of capturing items, reaching a certain region or area, touching objects (levers or crates, for example), and other game events. So, it is possible to assign rewards associated to these events. In this way, collisions and *ray casting* can be captured to produce rewards. For every event in a game, there is an associated reward function R_e. The reward generation is subject to one or more preconditions. The function R_e is represented by an object of type *RewardFunc*. For analysis purposes, the function R_e is associated with a propositional symbol g, which is evaluated as true, if the event associated with the symbol occurs; and it is evaluated as false, otherwise. Considering a set of logical functions S, it is possible to define propositions such as: g_j is true only if any function of S is true. Thus, it is possible to associate the occurrence of logical combinations of events as if they were combinations of propositional symbols. However, we use the visual specification instead of supplying a logical language for specifying reward functions.

Therefore, instead of a propositional symbol g, AI4U uses a graphical symbol G, which is associated with: an event in the game, a reward assignment rule, and a list of agents. Thus, if the event associated with G occurs, the reward generation rule is used, then the reward produced is attributed to agents linked to G. Besides, the component G can be related to other components, making up relationships that last over time. This property is essential for specifying non-Markovian reward functions [5]. There is an equivalence between formal specification and visual specification of reward functions. The state of a DFA can be associated with a sequence of events. Thus, each state in the set of states represents the occurrence of events in a sequence. For example, consider the problem of an agent who must touch all objects of the same color scattered randomly in a rectangular region. Consider that a = *touch the red object*, b = *touch the blue object*, and *ab* means that the agent first must touch the red object, then the blue object. Fig. 5 shows DFA for this problem. Since the agent must touch a sequence of objects of the same color, applying DFA to the input symbols results in $\delta(b, \delta(a, q_0)) = q_1$. The initial state represents the start of the game if no event has occurred. The transition function determined that a given event (represented by the input symbol) combined with the sequence that has occurred so far (current state) results in a state. When the machine is applied to all symbols in the sequence and the final state produced is not in F, the produced sequence does not result in a reward. Fig. 6 partially shows a structure produced using the tool developed in this work. This structure is equivalent to the DFA shown in Fig. 5, however instead of a sequence of symbols that represent states, what is being evaluated are sequences of events of the same nature (touch objects of the same color). In Fig. 6, objects of the same

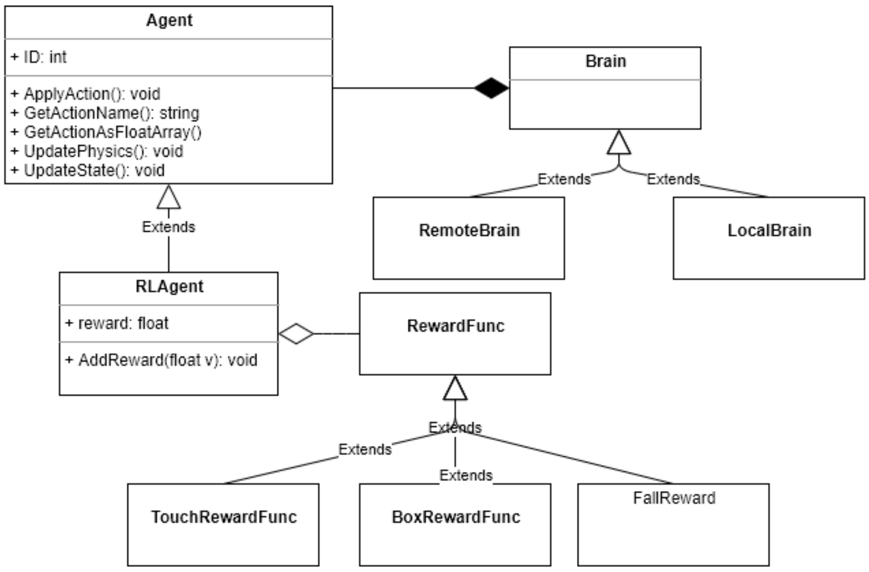


Fig. 2. AI4U API classes.

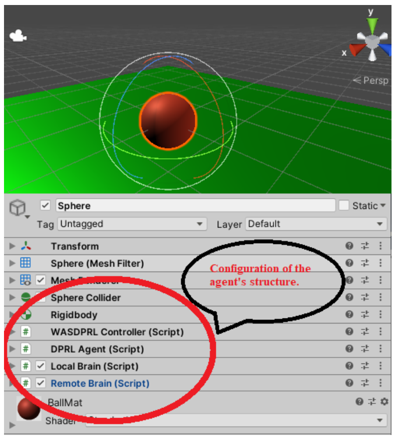


Fig. 3. Configuration of the agent to control a sphere.

```
import gym
from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common import make_vec_env
from stable_baselines import PPO2
from ai4u.utils import environment_definitions
import AI4UGym
from AI4UGym import BasicAgent
import numpy as np

OUTPUT_SHAPE = (7, )
STATE_SHAPE = (6, )
#BEGIN::GENERATED CODE :: DON'T CHANGE
def get_state_from_fields(fields):
    return np.array([fields['tx'], fields['tz'], fields['vx'], fields['vz'],
                    fields['x'],  fields['z']])
def make_env_def():
    environment_definitions['state_shape'] = (6,) #observation space shape
    environment_definitions['action_shape'] = (5,) #action state shape
    environment_definitions['actions'] = [('fx', 0.5), ('fx', -0.5), ('fz', 0.5),
                                          ('fz', -0.5), ('noop', 0.0)]
    environment_definitions['input_port'] = 8080 #agent port
    environment_definitions['output_port'] = 7070 #environment port
    environment_definitions['host'] = '127.0.0.1' #environment host
    BasicAgent.environment_definitions = environment_definitions #configuration
make_env_def()
#END::GENERATED CODE :: DON'T CHANGE
env = gym.make("AI4U") #make a single environment.
env.configure(environment_definitions)
obs = env.reset() #reset environment to get first observation.
done = False #flag to end of the episode
while not done:
    action = np.random.choice(np.arange(6)) #random selection of the actions
    obs, reward, done, info = env.step(action) #get a new observation, and reward
```

Fig. 4. Code of a random Agent generated from the specification given to AI4U.

color are connected by lines, indicating that rewards will only be given when the object is touched for the first time (indicated by an attribute not shown) or when a touch on an object of the same color has occurred. The line coming out of the open circle imposes restrictions to the reward given for touching the related object. The line arriving at a closed circle indicates that touching the arrival object is a prerequisite or one of the prerequisites for awarding a reward to the agent who touches the exit object. An attribute called "at least one" indicates whether the reward is released when at least one of the preconditions is met. Various specifications of this type can be given to various objects in the environment.

The final modeling of a reward function produces several configurations. Fig. 6 shows an example of a single reward function specification. The set of these configurations consists of a disjunction of functions. In a disjunction of the reward

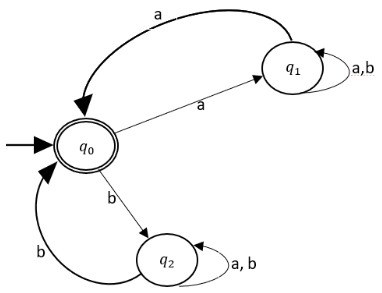
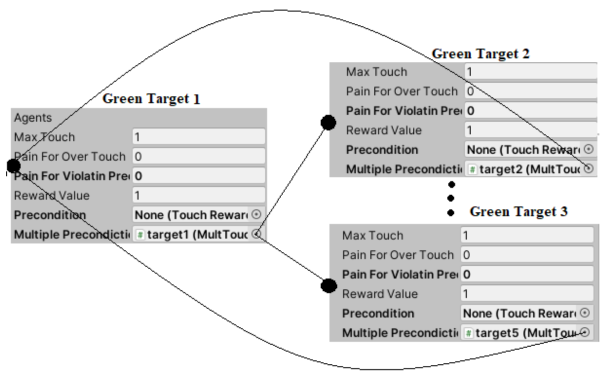


Fig. 5. Deterministic Finite Automata that accepts sequence of identical symbols.



Green Target 1

Agents	
Max Touch	1
Pain For Over Touch	0
Pain For Violatin Pre	0
Reward Value	1
Precondition	None (Touch Rewar
Multiple Precondicti	target1 (MultTouc

Green Target 2

Max Touch	1
Pain For Over Touch	0
Pain For Violatin Pre	0
Reward Value	1
Precondition	None (Touch Rewar
Multiple Precondicti	target2 (MultTouc

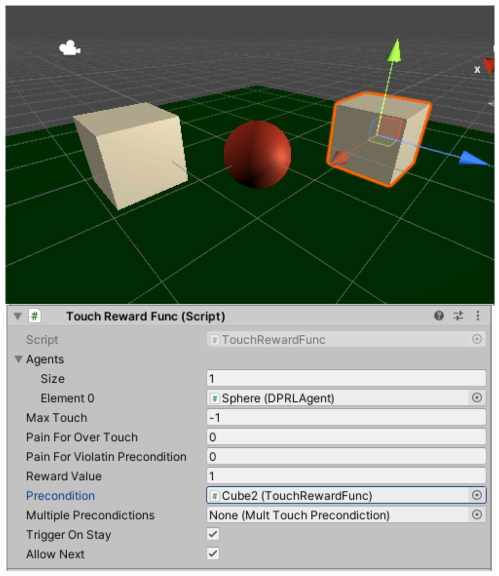
Green Target 3

Max Touch	1
Pain For Over Touch	0
Pain For Violatin Pre	0
Reward Value	1
Precondition	None (Touch Rewar
Multiple Precondicti	target5 (MultTou

Fig. 6. Modeling so that a sequence of two or more touches on different green objects generates a reward. Lines coming out of the open circle indicate that the event associated with that box only generates a reward if one of the events represented by the connected boxes in the closed circle occurs.

functions, the final reward is the sum of the rewards produced by each function. Fig. 6 shows a reward function specification equivalent to a logical $Q = q$:- $p_1, p2, ..., p_n$ where q, and p_i for $i \in \{1, 2, ..., n\}$ are logical symbols. The rule Q defines a conjunction of events for event q to occur. Observe that q is true only if the precondition $p_1, p_2, ..., p_n$ is true. The sentence $true$:-$p_1, p_2, ..., p_n$ indicates that the agent only receives a reward if the events $p_1, p_2, ..., p_n$ occur. AI4U specification uses graphical modules to indicate events in the environment. A graphical module has attributes that allow one to define preconditions and requirements for the produced reward. For example, Fig. 7 shows a setting that defines a reward of value 1 (attribute *Reward Value*) every time the agent touches the selected cube (cube with orange edges). The field *Element 0* of the attribute *Agents* defines the agent that will receive the reward. The result of a rewarding event can be attributed to one or more agents. The field *Precondition* defines a necessary precondition for the reward to be produced. A disjunction of reward events has no precondition for the events assigned to the object.

In this way, we show that the AI4U's specification of reward functions is equivalent to an RM since each modeled reward function is equivalent to a DFA. Therefore, one can specify both Markovian and non-Markovian reward functions. A contribution of this tool is that each occurrence of an event produces an output value concatenated with other output values, showing the sequence of events that occurred. Thus, the reward function is exposed as an input to the decision-making model, which can take advantage of this structure to maximize the agent's learning, as shown by Camacho et al. [5]. In the configuration shown in Fig. 7, during an episode, a sequence $(e_i : v_i, e_j : v_j)$ will be produced, indicating the sequence of events (in this case, touches) that occurred, where e_i and e_j are symbols that identify the events, and v_i and v_j are values that indicate whether the associated event occurred or not. These values can be concatenated as observations captured from the environment and sent to the agent. Thus, exposing the temporal relationship in the current state, allowing the agent



Touch Reward Func (Script)

Script	TouchRewardFunc
Agents	
Size	1
Element 0	Sphere (DPRLAgent)
Max Touch	-1
Pain For Over Touch	0
Pain For Violatin Precondition	0
Reward Value	1
Precondition	Cube2 (TouchRewardFunc)
Multiple Precondictions	None (Mult Touch Precondiction)
Trigger On Stay	✓
Allow Next	✓

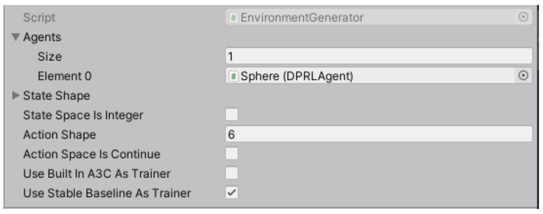
Fig. 7. Reward generation setting when two events occur: touch the two cubes next to the sphere.

to decide on the domain of a MDP.

C. Code Generation of Training and Testing Loops for Reinforced Learning Agents

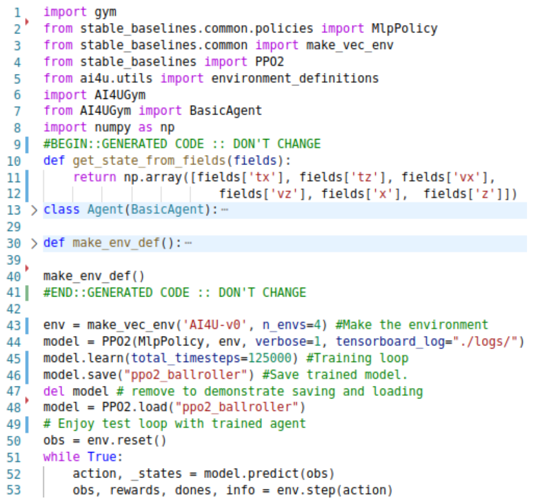
An essential configuration of the environment is the definition of its data shape, the observation space, and action space. The environment specification is encapsulated in the tool's programming interface, giving the modeler the possibility of declarative modeling, but allowing the flexible programmatic use of the specification. The declared environment's specification is translated into a common Gym's environment format, which facilitates the generation of code for training and execution loops common in the implementation of reinforcement learning algorithms. With this, the AI4U tool can generate execution loops, as shown in Fig. 4 or be integrated with tools that generate this loop.

For the generation of agent code to occur, it is necessary to provide some extra information to AI4U. This can be done by associating a *script EnvironmentGenerator* with one of the game objects. And so, when the game is run once, the training and test loop is automatically generated. In Fig. 8, the *script* EnvironmentGenerator was associated with one of the game objects. The configuration of *script* will determine the characteristics of Gym's environment format to be generated. AI4U captures this and other data to generate the agent's execution loops. Training and test loops are generated with the *stable-baselines* [27] PPO2 [23] algorithm, and the built-in A3C implementation. The script *EnvironmentGenerator* contains pieces of information such as the observation's shape (attribute *State Shape*), the type of data (attribute *State Shape is Integer*) that represent the state of the environment, the action shape (attribute *Action Shape*), and other attributes of the environment. Fig. 9 shows the code produced by an *EnvironmentGenerator*.



Script	EnvironmentGenerator
▼ Agents	
Size	1
Element 0	Sphere (DPRLAgent)
▶ State Shape	
State Space Is Integer	☐
Action Shape	6
Action Space Is Continue	☐
Use Built In A3C As Trainer	☐
Use Stable Baseline As Trainer	✓

Fig. 8. Specifying environment settings for automatic code generation.



```
import gym
from stable_baselines.common.policies import MlpPolicy
from stable_baselines.common import make_vec_env
from stable_baselines import PPO2
from ai4u.utils import environment_definitions
import AI4UGym
from AI4UGym import BasicAgent
import numpy as np
#BEGIN::GENERATED CODE :: DON'T CHANGE
def get_state_from_fields(fields):
    return np.array([fields['tx'], fields['tz'], fields['vx'],
                     fields['vz'], fields['x'],  fields['z']])
class Agent(BasicAgent):⋯

def make_env_def():⋯

make_env_def()
#END::GENERATED CODE :: DON'T CHANGE

env = make_vec_env('AI4U-v0', n_envs=4) #Make the environment
model = PPO2(MlpPolicy, env, verbose=1, tensorboard_log="./logs/")
model.learn(total_timesteps=125000) #Training loop
model.save("ppo2_ballroller") #Save trained model.
del model # remove to demonstrate saving and loading
model = PPO2.load("ppo2_ballroller")
# Enjoy test loop with trained agent
obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
```

Fig. 9. Automatically generated test and training loop.

In the next section, results of the AI4U tool implementation are presented.

V. Results

We used AI4U in a research project for developing autonomous virtual characters with emotions [28]. In addition, we show some simple application examples to demonstrate some possibilities of use of this tool. Next, some general aspects of the tool's implementation are presented followed by some didactic examples for demonstration purpose.

A. Implementation

As mentioned before, the two main components of AI4U are: the server and the client components. The server collects pieces of information from the environment, and sends them to the client, which processes them for use on the agent's training and test loops.

The server was coded with C# programming language, and the client component was coded using Python programming language. On the server-side, we chose C# as the main language because it is the main language used to code Unity's game logic. On the client-side, Python was chosen because it has a comprehensive community of Artificial Intelligence in general, and reinforcement learning, in particular, that distributes code and works around an ecosystem based on the Python programming language. Part of the code to manage client-side communication and infrastructure is generated from the environment specification given at the server-side.

To show the potential of the developed tool, three examples based on AI4U are presented in the next section.

B. Application Examples

AI4U allows for reinforcement and Artificial Intelligence learning experiments using the Unity game development platform. Thus, we modeled 3 scenarios with different degrees of difficulty and environment settings to show their possibilities:

- **BoxChaseBall** [29]: this is a simple environment whose input for decision making is a vector of real numbers, and the action is a discrete set. The agent must move a sphere on the plane to make it collide with a box. If the sphere goes off-bounds, the agent receives a -1 penalty; and, if the sphere touches the target, the agent wins a +1 reward. An episode ends when the sphere touches the target or goes off-bounds. The input vector of real numbers encodes the position of the target and the position of the sphere. The actions that can be performed by the agent are: to apply a constant force to the sphere along the z axis; and to apply a constant force to the sphere along the x axis. Fig. 10 shows an image of the modeled environment (left), the result of training the agent using the A3C [1], and PPO2 [10] algorithms (implementation of stable-baselines).
- **MazeWorldBasic** [30]: this environment consists of a three-dimensional maze from which the agent has to find its way out before running out of energy. The maze contains red and green spheres scattered on the ground. The agent controls a three-dimensional avatar with a rudimentary first-person view of the environment. The view is a 20×20 image produced by a ray casting perspective projection. The agent's observation of the environment at a given instant consists of the last four captured images. At the outset of an episode, the agent has 30 units of energy. During the agent's exploration of the environment, energy is lost. Extra loss of energy occurs whenever a red sphere is touched. However, an energy gain occurs whenever a green sphere is touched. The agent's possible actions are: *walk*, *run*, *turn left*, *turn right*, *stay still*. Fig. 11 shows how the environment looked like after modeling. In this environment, we tried the A3C and PPO2 algorithms.
- **MemoryV1** [31]: in this environment, the agent is in a room with several red and green cylinders. The agent has to touch all cylinders of the same color, producing a coherent sequence of touches. At the outset of an episode, the touch color is chosen randomly, and the agent must act to find the next cylinder in the coherent sequence. The episode ends if coherence is broken, in which case, the agent receives a negative "reward" of -10. For each element of a coherent sequence, the agent receives a positive reward of +10. The observations are similar to those obtained in the **MazeWorldBasic** environment. The set of actions is discrete and consists of two actions: one-degree rotation around the vertical axis of the agent, and walk. For this environment, it was necessary to use the

non-Markovian reward function specification, as shown partially in Fig. 12. In this case, it was not possible to use the PPO2 algorithm because the implementation of *stable-baseline* does not provide a standard way of creating a neural network model with two groups of inputs. This is necessary to facilitate the integration of the non-Markovian reward functions into the agent's decision making mechanism.

These environments show the viability of the developed tool. The graphs on the right in Fig. 10, 11 and 12 show the results obtained with each algorithm used in the experiments. The differences in the result of the algorithms, however, do not indicate a better performance of one or the other in these environments, since the algorithms were applied without the refinement of the parameters, that is, with the standard parameter values of the implementations used. The goal is to show the flexibility of the presented tool in adapting to different implementations. Any other algorithm whose implementation is already adapted to Gym's environment format can be evaluated with this tool by changing a few trivial lines of code. In addition to the presented environments, AI4U base code was used in the development of autonomous and emotional characters in [28].

In addition to those didactic examples, AI4U can be used in any game developed with the Unity game engine. A prerequisite for this is that the game be adapted or use the AI4U API from the beginning. An example of this is the adaptation of a game based on Unity's TowerDefense template, shown in Fig. 13, which we are adopting to use with AI4U.

VI. Conclusions, Limitations and Future Work

In this work, we presented the AI4U tool for specifying and developing environments for reinforcement learning experiments in games. Reinforcement Learning is a promising approach to obtaining autonomous Non-Player Characters (NPCs) in video games. However, one of the barriers to researching new approaches to RL applied to games is the layer of complexity added to integrate state-of-the-art RL algorithms into the game environment. New tools have been developed to deal with this problem, such as ML-Agents, but the direct visual specification of non-Markovian reward functions and their translation into a composition of Markovian functions is a contribution of the AI4U. Besides, the integration of this tool with third-party algorithms is almost straightforward, as long as they follow Gym's environment specification. The use of this tool, though, is not limited to environments that follow Gym's environment format. However, without following that specification, the automatic code generation does not work.

Another limitation of AI4U is its strong coupling with Unity, as this makes portability to other game development platforms difficult. Thus, the possible future work is its portability to other game development platforms.

Despite the pointed limitations, the developed tool proved to be flexible in the development of three case studies of environments with different levels of complexity, integrating it with both internal algorithms of the tool itself and with algorithms implemented by third parties.

The next steps in this research include the modeling of more complex reinforcement learning experiments aimed at obtaining autonomous virtual characters with emotion in virtual worlds and the investigation of emotion modeling integrated with the visual description of reward functions.

Acknowledgment

The authors would like to thank CNPq, the State University of Acarau Valley, and the Graduate Program in Computer Science of the Federal University of Ceará MDCC/UFC) for their financial support.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. G Bellemare, A. Graves, M. Riedmiller, A. K Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[2] J. Forgette and M. Katchabaw, "Learned behavior: Enabling believable virtual characters through reinforcement," in *Integrating Cognitive Architectures into Virtual Character Design*. IGI Global, 2016, pp. 94–123.

[3] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[4] P. B. S. Serafim, Y. L. B. Nogueira, C. A. Vidal, and J. B. C. Neto, "Evaluating competition in training of deep reinforcement learning agents in first-person shooter games," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2018, pp. 117–11 709.

[5] A. Camacho, R. T. Icarte, T. Q. Klassen, R. A. Valenzano, and S. A. McIlraith, "Ltl and beyond: Formal languages for reward function specification in reinforcement learning." in *IJCAI*, vol. 19, 2019, pp. 6065–6073.

[6] S. Hamdy and D. King, "Affect and believability in game characters–a review of the use of affective computing in games," in *Proceedings of the 18th Annual Conference on Simulation and AI in Computer Games. EUROSIS*, 2017.

[7] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," *arXiv preprint arXiv:1809.02627*, 2020.

[8] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[9] T. L. Paine, C. Gulcehre, B. Shahriari, M. Denil, M. Hoffman, H. Soyer, R. Tanburn, S. Kapturowski, N. Rabinowitz, D. Williams *et al.*, "Making efficient use of demonstrations to solve hard exploration problems," *arXiv preprint arXiv:1909.01387*, 2019.

[10] E. Kolve, R. Mottaghi, W. Han, E. VanderBilt, L. Weihs, A. Herrasti, D. Gordon, Y. Zhu, A. Gupta, and A. Farhadi, "Ai2-thor: An interactive 3d environment for visual ai," *arXiv preprint arXiv:1712.05474*, 2017.

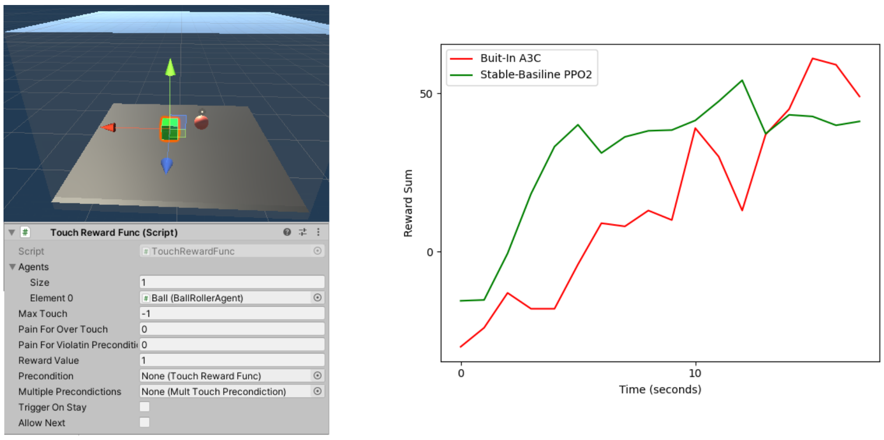
[11] T. Beysolow II, "Custom openai reinforcement learning environments," in *Applied Reinforcement Learning with Python*. Springer, 2019, pp. 95–112.

[12] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq *et al.*, "Deepmind control suite," *arXiv preprint arXiv:1801.00690*, 2018.

[13] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging procedural generation to benchmark reinforcement learning," *arXiv preprint arXiv:1912.01588*, 2019.

[14] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell, "The malmo platform for artificial intelligence experimentation." in *IJCAI*, 2016, pp. 4246–4247.

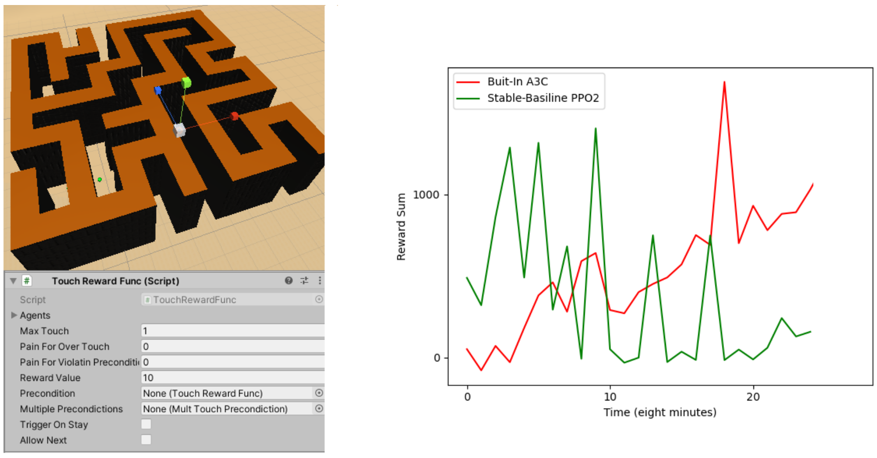
[15] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–8.



Touch Reward Func (Script)

Script	TouchRewardFunc
Agents	
Size	1
Element 0	Ball (BallRollerAgent)
Max Touch	-1
Pain For Over Touch	0
Pain For Violatin Preconditi	0
Reward Value	1
Precondition	None (Touch Reward Func)
Multiple Preconditions	None (Mult Touch Precondiction)
Trigger On Stay	
Allow Next	

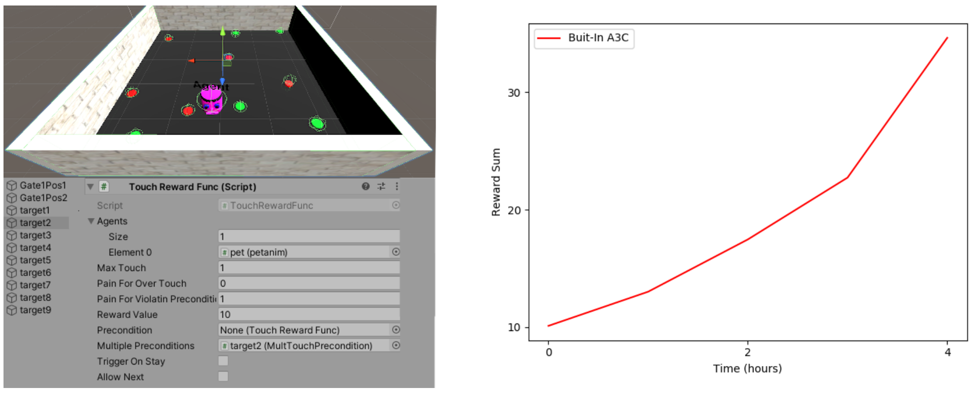
Fig. 10. Example of an application using the built-in implementation of the A3C algorithm and the stable-baseline implementation of the PPO2 algorithm. On the left, it is shown the touch configuration of the target box. On the right, it is shown the training result of the algorithms A3C and PPO2.



Touch Reward Func (Script)

Script	TouchRewardFunc
Agents	
Max Touch	1
Pain For Over Touch	0
Pain For Violatin Preconditi	0
Reward Value	10
Precondition	None (Touch Reward Func)
Multiple Preconditions	None (Mult Touch Precondiction)
Trigger On Stay	
Allow Next	

Fig. 11. Example of a maze exploration application using the built-in implementation of the A3C algorithm and the stable-baseline implementation of the PPO2 algorithm. On the left, it is shown the maze model. On the right, it is shown the training result of the algorithms A3C and PPO2.



Gate1Pos1
Gate1Pos2
target1
target2
target3
target4
target5
target6
target7
target8
target9

Touch Reward Func (Script)

Script	TouchRewardFunc
Agents	
Size	1
Element 0	pet (petanim)
Max Touch	1
Pain For Over Touch	0
Pain For Violatin Preconditi	1
Reward Value	10
Precondition	None (Touch Reward Func)
Multiple Preconditions	target2 (MultTouchPrecondition)
Trigger On Stay	
Allow Next	

Fig. 12. Example of an application using the built-in implementation of the A3C algorithm. On the left, it is shown the top view of the environment, and on the right, it is shown the training result of the A3C algorithm.

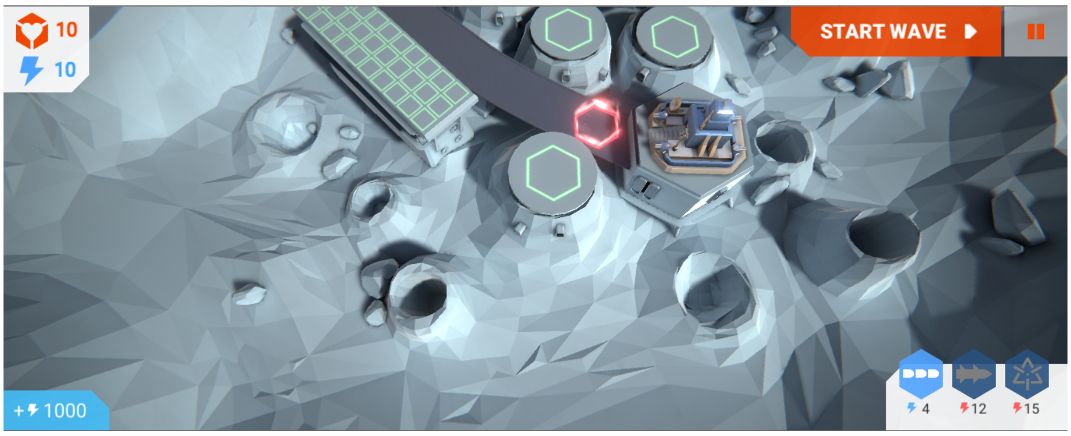


Fig. 13. Adaptation of the TowerDefense game template used with AI4U.

[16] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik *et al.*, “Habitat: A platform for embodied ai research,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9339–9347.

[17] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik *et al.*, “Deepmind lab,” *arXiv preprint arXiv:1612.03801*, 2016.

[18] “Pybullet, a python module for physics simulation for games, robotics and machine learning,” https://github.com/bulletphysics/bullet3, accessed:2020-07-20.

[19] R. D. Gaina, A. Couëtoux, D. J. Soemers, M. H. Winands, T. Vodopivec, F. Kirchgeßner, J. Liu, S. M. Lucas, and D. Perez-Liebana, “The 2016 two-player gvgai competition,” *IEEE Transactions on Games*, vol. 10, no. 2, pp. 209–220, 2017.

[20] N. Avradinis, S. Vosinakis, T. Panayiotopoulos, A. Belesiotis, I. Giannakas, R. Koutsiamanis, and K. Tilelis, “An unreal based platform for developing intelligent virtual agents,” *WSEAS Transactions on Information Science and Applications*, pp. 752–756, 2004.

[21] P. Gestwicki, “Unreal engine 4 for computer scientists,” *Journal of Computing Sciences in Colleges*, vol. 35, no. 5, pp. 109–110, 2019.

[22] M. Toftedahl and H. Engström, “A taxonomy of game engines and the tools that drive the industry,” in *DiGRA 2019, The 12th Digital Games Research Association Conference, Kyoto, Japan, August, 6-10, 2019*. Digital Games Research Association (DiGRA), 2019.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[24] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, “Soft actor-critic algorithms and applications,” *arXiv preprint arXiv:1812.05905*, 2018.

[25] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[26] “AI4U Source Code,” https://github.com/gilcoder/AI4U, accessed:2020-07-20.

[27] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, “Stable baselines,” https://github.com/hill-a/stable-baselines, 2018.

[28] G. Gomes, C. A. Vidal, J. B. Cavalcante Neto, and Y. L. B. Nogueira, “An emotional virtual character: A deep learning approach with reinforcement learning,” in *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, 2019, pp. 223–231.

[29] G. Gomes. (2020, september) Environment boxchaseball. [Online]. Available: https://github.com/gilcoder/BoxChaseBall

[30] ——. (2020, september) Environment mazeworldbasic. [Online]. Available: https://github.com/gilcoder/MazeWorldBasic

[31] ——. (2020, september) Environment memoryv1. [Online]. Available: https://github.com/gilcoder/MemoryV1