Entertainment Computing 43 (2022) 100516

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: www.elsevier.com/locate/entcom

A modeling environment for reinforcement learning in games

Gilzamir Gomes [a], Creto A. Vidal [b], Joaquim B. Cavalcante-Neto [b], Yuri L.B. Nogueira [b]

[a] State University of Acaraú Valey, Department of Science & Technology, Sobral, Brazil
[b] Federal University of Ceara, Computer Department, Fortaleza-CE, Brazil

ARTICLE INFO

Keywords:
Reinforcement Learning
Games
Non-Player Characters
Navigation Problem
2021 MSC:
00–01
99–00

ABSTRACT

Developing Non-Player Characters, i.e., game characters that interact with the game's environment autonomously with flexibility to experiment different behavior configurations is not a trivial task. Traditionally, this has been done with techniques that limit the complexity of the behavior of Non-Player Characters (NPCs), as in the case of using Navigation Mesh (NavMesh) for navigation behaviors. For this problem, it has been shown that reinforcement learning can be more efficient and flexible than traditional techniques. However, integrating reinforcement learning into current game development tools is laborious, given that a great deal of experimentation and coding is required. For that, we have developed a modeling environment that integrates with a game development tool and allows the direct specification of reward functions and NPC agent components with maximum code reuse and automatic code generation.

1. Introduction

In the early days of Artificial Intelligence (AI), games were used as interesting problems that would demonstrate the ability of machines to solve them. In his pioneering work, Arthur Samuel [1] used Reinforcement Learning (RL) to solve checkers board games. Games like *Starcraft II* keep challenging state-of-the-art AI algorithms [2]. The progress of AI, on the other hand, brings benefits to the video game industry. Yannakakis and Togelius [3] present a non-exhaustive list of AI methods applied in games, such as the first popular application of neural networks in the game *Creatures* and the use of behavior trees in the game *Halo 2*.

Nowadays, there is a demand for game environments in which not only AI techniques can be explored, but also well developed AI techniques can be applied in video games. There are several examples of recent research that use games as interesting and complex problems for AI [4–7]. Regarding the use of AI in games, several studies show how AI can improve *gameplay* in video games [8,3,9,10].

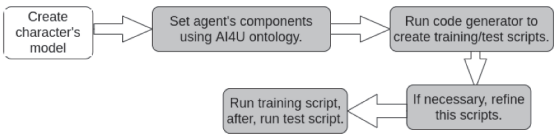
In AI for playing games, an open problem is the obtaining of complex behaviors for Non-Player Characters (NPCs) using end-to-end machine learning techniques. Deep Reinforcement Learning (DRL) is a promising approach for NPCs development. Alonso et al. [10] show that DRL supplants traditional search-based approaches for navigation in games. Several other works show that RL is promising in obtaining human-like behaviors in video games or in applications with autonomous virtual characters [11–13]. Therefore, experimenting with RL approaches in games is a relevant problem.

Despite the growing importance of modeling autonomous agents using DRL, current game development tools do not provide a intuitive agent abstraction and formalism to model reinforcement learning agents to control NPCs. Juliani et al. [14] come closest to this through an open source tool called *MLAgents* (Machine Learning Agents) that make it possible to model game environments for Artificial Intelligence experiments based on machine learning. However, that tool does not provide a visual abstraction for modeling autonomous agents. Furthermore, MLAgents is a tool focused only in Unity game engine.

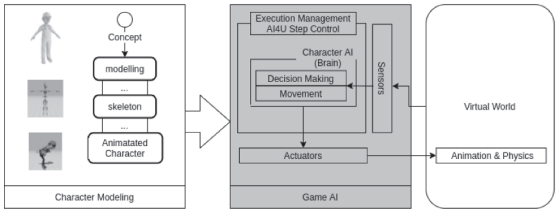
The attempt to provide a formalism for modeling NPC behavior is not new, as highlighted by Gino [15], who uses the concept of *Statecharts* for this purpose. Therefore, in this work, our modeling environment AI4U (*Artificial Intelligence for Unity*) [16] is enhanced to provide an abstraction to model NPCs like agents based on concept of task environment [17]. Thus, AI4U was specifically refined to the preparation of reinforcement learning experiments in games. In general, the high-level goals of this environment are: to enable the reproducibility of methods and results; to simplify the way of designing reinforcement learning algorithms in games; and to enhance the readability of RL agents. Furthermore, our approach can be adapted to different game development platforms. Specifically, we have implemented prototypes of our approach to Unity [14] and Godot [18], two of the most used game development tools. Unity and Godot were chosen because they have rich documentation, an active development community, and can run on different levels of hardware and software architectures. In Fig. 1, we summarize the techniques proposed in this paper and places them in the game character creation pipeline.

This work is an extension of paper [19] with the following extensions:

https://doi.org/10.1016/j.entcom.2022.100516
Received 4 May 2021; Received in revised form 20 July 2022; Accepted 27 July 2022
Available online 3 August 2022



(a) Steps required to train a character controller using AI4U.



(b) Summary of steps required to create a game character.

Fig. 1. The methods proposed in this paper are located in these highlighted stages of the game character creation pipeline using artificial intelligence for games. Consider a animated character model without behaviour controller. Character Artificial Intelligence (AI) is defined using AI4U's ontology of environment task. After that, code generator creates training and test scripts then programmer refine (if necessary) and run these scripts. Highlighted In (b), we can see final character AI architecture.

1. Corrections in Section 2, regarding the return definition in reinforcement learning.
2. Inclusion of a new work in Section 3, which presents a new tool available in the academic community.
3. Rewriting of Section 4, which describes the AI4U environment, in order to provide a more didactic presentation of the ideas, and to describe extended features of AI4U. Now, instead of generating static code simply based on a predefined framework, AI4U was modified to generate code in accordance with the agent model created by the user. In addition, almost all of the figures in this section were replaced. Only two figures remained unchanged.
4. Rewriting of Section 5 with modification of figures, modification of one example (MazeWorldBasic) and inclusion of two new examples (MemoryV2 and NavBoy).
5. A discussion section (Section 6) has been added after the results section (Section 5).
6. Rewriting of Section 6 (now, it is the Section 7) to better represent the goal of the current version of the AI4U environment.

2. Background

For understanding the methods used in the design of the modeling environment presented in this work, we give an overview of reinforcement learning, and of formal specification of reward functions.

2.1. 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 return R_t from time t is a measure of the agent's performance written as

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

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 the expected return (Eq. 2).

$$V_\pi(s) = \mathbb{E}_\pi[R(t)|s_t = s]. \tag{2}$$

In Approximate Reinforcement Learning, value functions, approximated by parameters θ, can be represented, using non-linear model, through several types of neural networks. Mnih et al. [4] show that reinforcement learning with deep neural networks gets a superhuman performance in several Atari games. In that approach, the agent selects actions based only on high dimensional input representations, such 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 (2). Therefore, in DRL, the state value function is $V_\pi(s;\theta)$, i.e., the expected value from state s using the policy π based on parameters θ.

2.2. 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) [20]. This component allows the specification of reward functions as the Finite Deterministic Automata (FDA). A RM 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 2.1. Markovian Decision Process (MDP) with an initial state is a tuple $\mathscr{M} = (S, A, s_0, T, r, \gamma)$ 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 2.2. Vocabulary and Labeling Function is a set $\mathscr{P}$ of propositional symbols. A labeling function is a function $L : S \times A \times S \to 2^{\mathscr{P}}$ that maps experiences to truth assignments over the vocabulary $\mathscr{P}$.

Definition 2.3. 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 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 2.4. 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, $\delta : Q \times \Sigma \to Q$ is a transition function, and $\rho : Q \times \Sigma \to R$ is the output function.

Definition 2.5. Setting is a tuple (S, A, P, L) where S, A, P, and L are defined as in Definitions 2.1, 2.2, 2.3, and 2.4.

Definition 2.6. Reward Machine (RM) for setting (S, A, P, L) is a Mealy Machine $(Q, q_0, \Sigma, R, \delta, \rho)$ where the input alphabet is $\Sigma = 2^\rho$, and R is a finite set where each $R \in \mathscr{R}$ is a reward function from $S \times A \times S$ to $\mathbb{R}$.

Definition 2.7. Deterministic Finite Automata (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 a Reward Machine, a sequence $(S \times A)^+ \times S$ is mapped into value assignments to propositional symbols in $\mathscr{P}$. 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. [20] propose DFA as an intermediate language for specifying reward functions in NMRDPs. Therefore, any language that can be translated into a DFA can be used to specify non-Markovian reward functions.

3. Related Works

Reinforcement learning researchers use different approaches to simulating environments and games. Recently, several simulation environments and platforms have been made available to evaluate the performance of reinforcement learning algorithms. Juliani et al. [14] 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* [21]; and the environment, Obstacle Tower [14].

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 [21], DMLab-30, Hard Eight [22], AI2Thor [23], OpenAI Retro [24], DMControl [25], and ProcGite [26].

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 [27], VizDoom [28], Habitat [29], DeepMind Lab [30], Acme [31], PyBullet [32] and GVGAI [33].

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 [14]. Juliani et al. [14] argue that modern video game engines are strong candidates for this category. Therefore, game engines like Unity [14], Unreal Engine [34,35] and Godot [36] can be used to define complex AI game environments. Although any general game development platform can be used as a general platform, Unity has been widely used because it provides an open-source *framework* of agents with machine learning, the ML-Agent [14]. 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) [37] and SAC (Soft Actor-Critic) [38] algorithms; and a low-level programming interface that allows Unity environment control from Python.

To extend the ecosystem of existing tools, we developed a modeling environment as a game engine plugin focused on simplicity, abstraction of intelligent agents, flexibility of choosing algorithms, and ease of specifying reward functions. The goal is to facilitate the investigation of reinforcement learning applications in games. Although we implemented this tool for Unity and Godot game engines, our approach is not limited to any specific game engine. As already pointed out, Unity and Godot were chosen because they have rich documentation, an active development community, and can run on different levels of hardware and software architectures.

4. AI4U: A Modeling Environment for Game Reinforcement Learning Experiments

Our modeling environment allows the prototyping of autonomous NPCs for video games based on the concept of task environment [17]. We named this modeling environment *Artificial Intelligence for Unity* (AI4U), as it was designed initially for the Unity game engine. AI4U is based on three pillars: simplicity, flexibility of choosing different implementations of artificial intelligence algorithms, and ease of specifying agent-environment interaction. Simplicity is achieved through integration with game engines, taking advantage of their scripting systems to provide built-in functionalities that allow the creation of agents and environments through modules that help to define sensors, actuators, reward functions and objects. 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* [39]. Thus, based on the provided task's environment, AI4U automatically generates a basic training and testing loop. The reward functions are easily specified by means of visual components that are available in game engines like Unity and Godot.

4.1. Game Environment and the Task Environment Abstraction

The current extension of AI4U [19] is a modeling environment for specifying game environments like a task environment. For this, we defined an ontology for describing NPCs as rational agents. The modeling of NPCs is based on abstraction of task environment. According to Russel and Norvig [17], to design a rational agent, one must specify the task environment, which includes the environment, the agent's actuators and sensors, and a performance measure. An environment is a virtual world modeled in a game engine. The agent's actuators and sensors are built-in reusable components of the AI4U or components made by users using the AI4U application programming interface. The performance measure is defined by means of rewarding events. Thus, AI4U defines an ontology that facilitate NPC modeling and supports declarative and visual specification of the environment, of the agent's sensors and actuators, and of the reward functions.

AI4U's ontology includes the concept of a agent's brain as a script that receives the data captured by the agent's sensors and produces an action that runs in the environment. So, we map the game engine's concepts to AI4U's ontology. For example, a reward function, in Unity, is a *prefab* component. In Godot, a reward function is a reusable subtree of the game scene.

In Fig. 2, part of the AI4U's ontology is shown. Each circle represents a concept. The agent is composed of other concepts, such as Sensor, Actuator and Brain. Sensors obtain information from the environment and pass it to the agent. Actuators define what actions the agent can perform in the environment. The agent's brain defines the agent's behavior. It is important to break the agent down into simpler components, as this allows for component reuse. For example, the same ray casting sensor can be shared by several agents, changing only the parameters that are specific to each agent. This allows for code reuse,

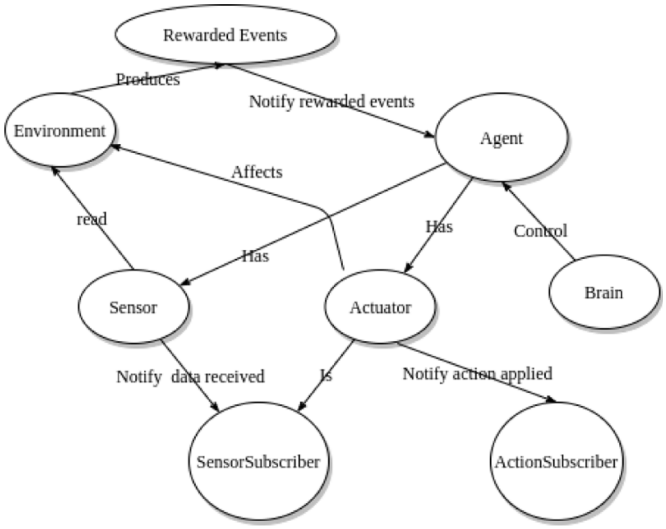


Fig. 2. AI4U's ontology.

avoiding the redeployment of capabilities shared by multiple agents.

AI4U was integrated into the Unity and Godot game development platforms, taking advantage of the flexibility provided by these platforms' scripting systems. Next, the environment specification is introduced.

4.2. Environment's Specification

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

4.2.1. Definition of the Environment's Properties

Every environment modeled with AI4U API (Fig. 3) is automatically associated with a configurable structure, which is compatible with the OpenAI Gym *framework*, a widely used specification for reinforcement learning environments [39]. Gym is very handy because many useful algorithms that are implemented and made available in open-source repositories are based on it. Thus, in order to use those algorithms in new environments that follow Gym's environment format, little adaptation is necessary. In this way, the AI4U's API can be used with little effort for code generation. But for this, only internal components of the AI4U must be used. Next, the AI4U's API is introduced.

4.3. AI4U's Application Programming Interface

Fig. 3 shows the AI4U's API (Application Programming Interface) used to model an agent or an NPC. The modeling can be done in two levels: low-level and high-level.

In low-level modeling, the user needs to code a script with a class that inherits directly from the *Agent* class or the *RLAgent* class, and manually specify the agent's behavior using pre-programmed objects and functions available in the AI4U API.

In high-level modeling, the specification of the agent is done through prefabricated modules (the so-called prefabs of the Unity engine or the stored subtrees of the Godot engine), among which, the main types are sensors, actuators, and reward functions. Each engine has its specifics. Next, we show how these components are made in Unity and in Godot.

4.4. Component-Based Modeling

The main platforms for game development, such as Unreal, Unity and Godot, have visual or non-visual scripting systems for specifying the behavior of game objects. In addition, those tools provide ways of building components that can be reused across different projects.

We looked specifically at an efficient and simple way of developing reusable components for deploying intelligent agents on two of these platforms. Below, we outline how we did this for Unity and Godot.

4.4.1. Component-Based Modeling in Unity

In Unity, the behavior of a game object is defined by the scripts associated with it. Unity also allows the creation of empty objects associated with scripts that perform reusable behavior. Once the association is established, it can be saved in a generic way to be reused or to replicate objects. Unity does that through its prefab objects. When a prefab is created, it can be replicated several times by adapting only a few convenient parameters. Therefore, the prefabs act as generic components that can be associated with several objects and events in the game. In Unity, we use prefabs to implement features that promote the reuse of code and facilitate the specification of environment, agents, and reward functions. For example, the reward function is modeled as a script that can be associated with any object in the scene. In games where an NPC must capture a flag, a reward is produced when the NPC's virtual body collides with the flag. In Unity, the script *TouchRewardFunc* produces a reward when the NPC's body touches other game object specified in a field named *Target*. Thus, *TouchRewardFunc* produces a reward only if the NPC's body touches the specified target game object. Therefore, the script *TouchRewardFunc* is a reusable component of AI4U.

In this way, we were able to model the AI4U's ontology as Unity game objects.

4.4.2. Component-Based Modeling in Godot

In Godot, an object in the scene has only one script associated with it, which determines its behavior and the type of object. Objects are nodes of the scene tree. Godot allows one to store a subtree to be reused in different scenes or in other projects. Thus, subtrees are like reusable components when object scripts are parameterized properly. In Fig. 4, for example, one can see: on the left, the *RLCharacterAgent* node, a subtree that contains an agent's configuration ready to be reused in various projects that use three-dimensional game NPCs; on the right, the agent's parameters that can be changed, the maximum number of steps per episode (*Max Steps*) and the kind of position sensor (*Global Position Sensor*).

Still in Fig. 4, the *RLCharacterAgent* component is an agent composed

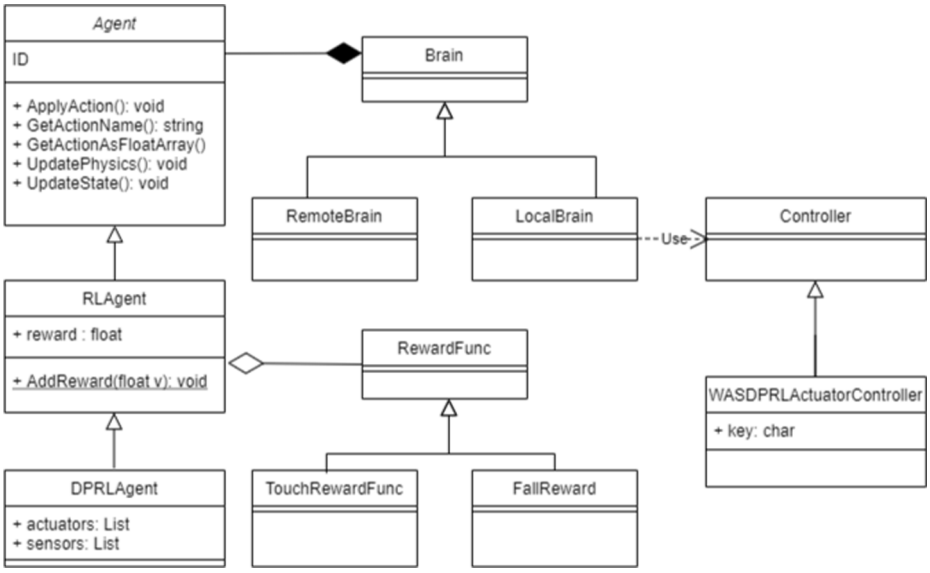


Fig. 3. AI4U's API classes.

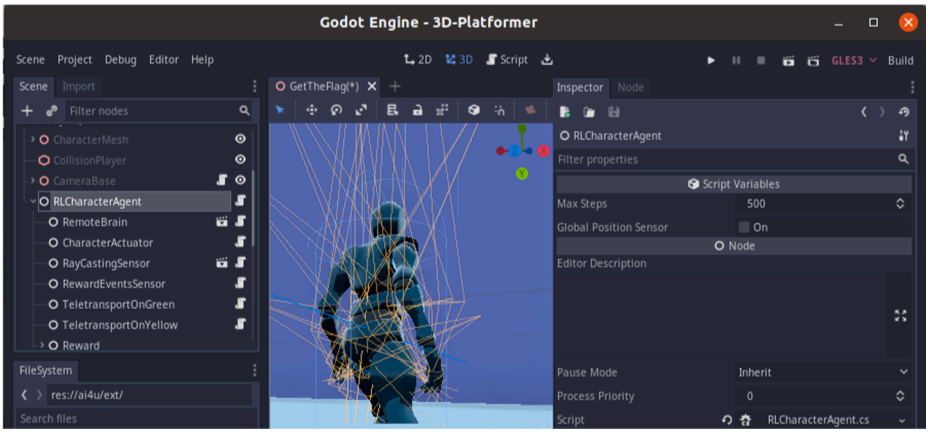


Fig. 4. An agent is a subtree of Godot's scene tree.

of a brain (*RemoteBrain*), sensors (*RayCastingSensor* and *RewardEventsSensor*), an actuator (*CharacterActuator*) and a reward function (*Reward*). The actuator named *CharacterActuator* allows an animated model to perform animated actions such as walking, crouching, running and jumping.

In this way, we were also able to implement the AI4U's ontology as reusable components in Godot.

4.5. Programming the Agents' Behaviors

Game objects can be controlled by objects of the type Brain. 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 the control of a virtual character by a model that was trained using some API, external to AI4U. An *LocalBrain* object allows the agent to be controlled by a decision-making model already trained and compiled to work locally, but it also allows a Controller object to control a game object through external devices, such as mice, keyboards and joysticks. This functionality is important for debugging the game environment during development. For example, in games, it is common to use the W, A, S, and D keys on the alphanumeric keyboard to control the movements of an avatar. So, we provide a controller called *WASDPRLActuatorController* (in Unity version) and *WSDLCharacterController* (in Godot version) for that purpose. This makes it easier to test gameplay, allowing, for example, to check the navigability in the environment before training a navigation agent.

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 *RewardFunc* type. An object of *RewardFunc* type is associated with one or more agents and an event in the environment.

The agent specification in Godot and Unity uses a common API, but the agent specification is tailored to each platform. While Unity favors composition, Godot favors scene structure in a scene tree. Next, we explain the two ways of specifying agents.

4.5.1. Programming the Agent's Behaviors in Unity

In Unity, a game object can use multiple scripts, which define its behavior. In Fig. 5, we show the association of the modules *LocalBrain* and *RemoteBrain* with a game object representing a virtual character. A *script* containing the class *DPRLAgent*, a specialization of the class *RLAgent*, was also associated with this object. So, the object can undergo physical control with application of forces. The point of application of a force can be defined through a controller *WASD* (class *WASDPRLActuatorController*) that is associated with the object receiving the force. This case was based on high-level modeling, therefore, no code was needed to implement the agent's behavior.

The controller *WASDPRLActuatorController* allows the application of forces to the 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 *RemoteBrain* type. In addition, it is necessary to add an actuator of MoveActuator type for the selected object.

4.5.2. Programming the Agent's Behaviors in Godot

In Godot, we can use only one script by node. Node in Godot is the Unity equivalent of game objects. In this case, the behavior of an object is defined both by the script associated with it and by that object's child nodes in the scene tree.

This structuring of objects as subtrees requires another specification

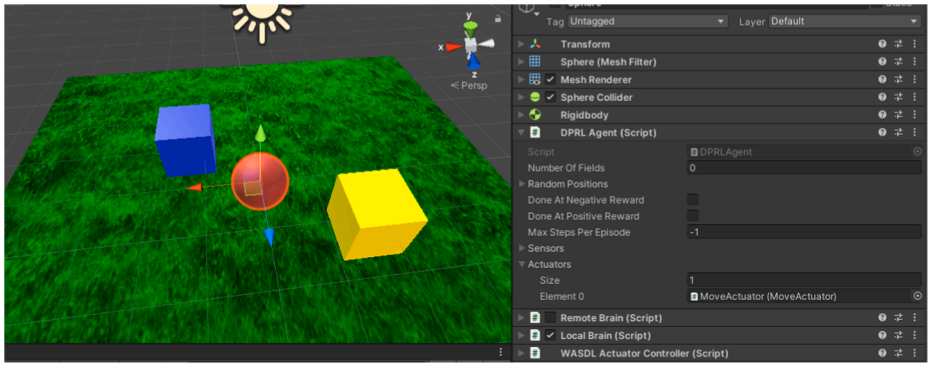


Fig. 5. Configuration of the agent to control a sphere.

approach. For example, in Fig. 4, the *RewardFunc* node is a reward function because its script is from a class that inherits from the *RewardFunc* class. Also, this node is a child of *RLCharacterAgent*, this automatically associates this reward function with the *RLCharacterAgent* agent. Note that this agent is a child node of the Player node, this means that *RLCharacterAgent* defines the behavior of the Player node. These behaviors are linked to the objects the first time the game is started, when the nodes of the scene tree are instantiated and informed through a callback function that they are ready to use.

4.6. Reward Function Specification

Rewarding events are functions that define how the agent's behavior will be evaluated. These functions are associated with an agent and can generate a reward for this agent in two situations: upon evaluation of the environment state or when an environmental event is triggered. In the first case, the function produces a reward in accordance with a certain state of the environment. In the second case, the function is called either to generate a fixed reward or to produce a reward as a result of evaluating the state of the environment.

Besides, rewarding events can be an atomic function modeled as an object type *RewardFunc* or a composite event modeled as a subtree of the scene, which is named logical tree. Nodes in a logical tree can be logical operators or events in the environment. Those two types of nodes are reward functions. Leaf nodes produce reward values when certain events occur in the environment or when the state of the environment equals a certain value. Logical operations nodes receive the evaluation result from their children and apply a logical operation. For example, in the tree shown in Fig. 6, the *Died* node performs a logical OR operation and is fired when one of its child nodes ("Fell in the pit" or "Eaten by the monster") is evaluated. Those nodes have several configuration options. The output of a logical operator can be a predefined value provided by the modeler or a linear combination of the output of the child nodes with the weights defined by the modeler of the reward function. In this way, we obtain an expressive and computationally inexpensive formalism for visual modeling of reward functions.

The reward function's specification tool is equivalent to a Reward Machine (RM). However, visual specification takes advantage of the visual appeal and creative power of modern game engines. That type of specification is usual in game development, for example, the composition of properties in materials through *shaders*, and virtual character animation. In addition, through their properties, these components can be concatenated to specify more complex reward functions. When the game event occurs, a reward is calculated and assigned to agents associated with the corresponding reward function. Thus, in Section 4.7, we present the visual specification of reward functions and their equivalence with the RM concept.

4.6.1. Rewarding Specification In Unity

Component-based modeling is natural in Unity, in the sense that the behavior and functionality of a game object are defined by the components added to the object. These components can be Unity built-in components or scripts developed by third parties. In Fig. 7, we show how the specification of a reward function in Unity looks like in this way of modeling.

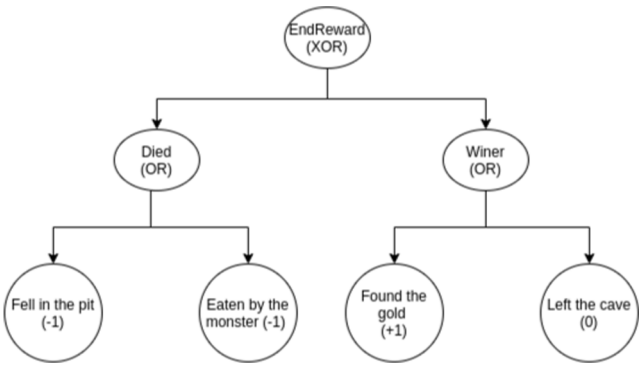


Fig. 6. A reward function modeled as a logical tree.

4.6.2. Rewarding Specification In Godot

In Godot, it is not possible to add multiple scripts (behaviors) directly to an object. To add more than one behavior or feature to an object, we have to package everything in a script or build the object as the root of a subtree of the scene. Building reward functions as scene subtrees seemed more flexible and easier to model, without increasing the computation cost too much, since the depth of the tree is limited. We can add a reward function for an agent by simply adding the node representing the reward as a child of the node representing the agent, as shown in Fig. 8.

4.7. Visual Reward Function Specification

In general, the goals of NPCs in games consist of capturing items, reaching a certain region of the scene, touching objects (levers or crates, for example) etc. So, for every event in a game, it is convenient to assign a reward function R_e, whose generation is subjected to one or more preconditions. The function R_e is represented by an object of *RewardFunc* type. For analysis purposes, the function R_e is associated with a propositional symbol g, which is evaluated either as true, in case the event associated with g occurs, or 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 in 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, for that, we use reusable components. Thus, by associating several reusable components with different elements of the game and with different events, it is possible to concatenate simple reward functions to form more complex ones. Best of all, it can be done visually.

Instead of a propositional symbol g, however, AI4U uses a graphical symbol G, which is associated with: an event in the game, a reward assignment rule, and a list of agents. A reward assignment rule is an object that inherits from the RewardFunc class. Thus, if the event associated with G occurs, the reward generation rule is used to produce a reward that is attributed to the agents linked to G. Besides, component G can be related to other components, producing sequential relationships. This property is essential for specifying non-Markovian reward functions [20].

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 that are scattered randomly in a rectangular region. Consider that a = *touch the yellow object*, b = *touch the green object*, and ab = *touch the yellow object first and the green object next*. Fig. 9 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 occurred yet. 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. 10 partially shows a structure produced using the modeling environment developed in this work. This structure is equivalent to the DFA shown in Fig. 9, 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. 10, objects of the same color are connected by preconditions, indicating that rewards will be given only when the object is touched for the first time (indicated by an attribute not shown) or

Fig. 7. Rewarding event specification in Unity. Here, the agent receives a reward when it touches the red box. For this, we select the red box, then add *TouchRewardFunc* for this object and set the agent that will receive this reward.

Fig. 8. Rewarding event specification in Godot. Here, we define a reward that agent receive a reward when it to touch the red pillar. For this, we select the agent node, then add *RBTouchRewardFunc* node as child of the agent node. Then, we setting in field Target Path the red pillar as a rewarding object.

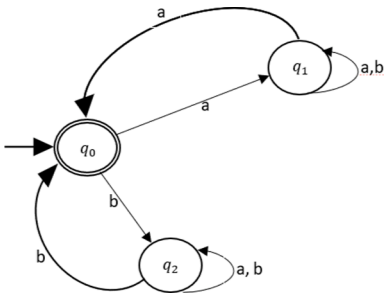


Fig. 9. Deterministic Finite Automata that accepts sequence of identical symbols.

when an object of the same color is touched. 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. 10 shows an example of a partial reward function specification. The set of configurations consists of a disjunction of functions. In a disjunction of the reward functions, the final reward is the sum of the rewards produced by each function. Also, Fig. 10 shows a reward function specification that is equivalent to a logical structure $Q = q : -p_1, p2, \ldots, p_n$ where q, and p_i for $i \in \{1, 2, \ldots, n\}$ are logical symbols. The rule Q defines a conjunction of events for event q to occur. Notice that q is true only if the precondition $p_1, p_2, \ldots, p_n$ is true. The sentence $true : -p_1, p_2, \ldots, p_n$ indicates that the agent only receives a reward if the events $p_1, p_2, \ldots, p_n$ 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. 11 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.

As shown, AI4U's specification of reward functions is equivalent to a 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. [20]. The configuration in Fig. 11 shows that, 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 or not the associated event occurred. These values can be concatenated as observations captured from the environment and sent to the agent. Thus, they expose the temporal relationship in the current state, allowing the agent to decide on the domain of a MDP.

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

Code can be generated automatically when using the high-level API. To do this, the RLAgent script must be associated with one or more objects. So, actuators and sensors can be associated with the agent (see Fig. 12). A code-generating component captures information associated with the agent and with the reward functions, and produces Python code for predefined algorithms. This approach is based on [16], which generated a fixed script based on the OpenAI Gym framework. However, we generate a script dynamically in accordance with the components (actuators and sensors) assigned to the agent.

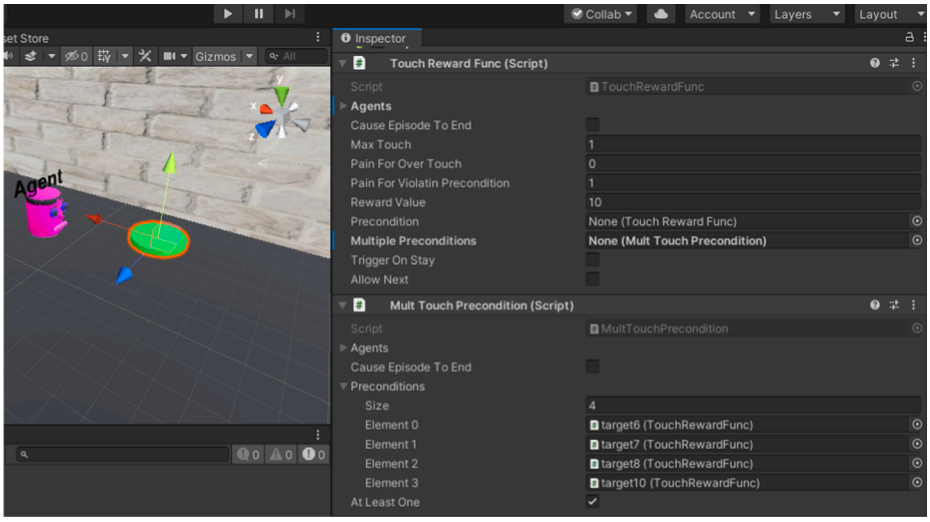
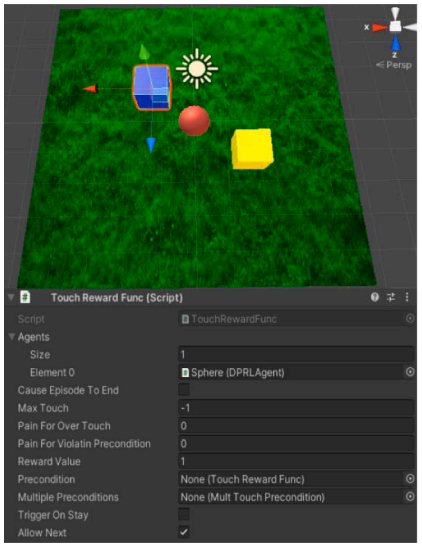
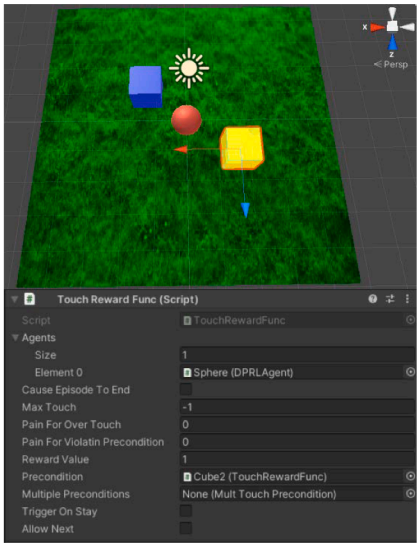


Fig. 10. The selected green cylindrical object only generates a reward if either no green object has been touched before or some other true object has been touched already. For this, we associate a script (shown in the Touch Reward Func panel) to the selected green object that generates rewards when touches to the object occur, however, only if multiple preconditions (shown in the Mult Touch Precondition panel) are satisfied.



(a) Configuration of a reward by tapping the blue cube. The value of the 'Reward Value' field is equal to one (1). Note that the 'Allow Next' field is selected which indicates that the notification of this event will be recorded for the next event, so that the agent has sequential information.



(b) Configuration of a reward by tapping the yellow cube. In this case, the value of the 'Reward Value' field is equal to one (1), but there is a precondition: touch the object 'Cube2', which is the blue cube. Note that, in this case, the 'Allow Next' field is not checked.

Fig. 11. Reward generation setting when two events occur: the sphere receives a reward if it touches the blue cube, but it will only receive a reward if it touches the yellow cube after it touches the blue cube.

The sensors and actuators provide the necessary information for specifying the shape of data, the observation space, and the agent's action space. The specification of the environment is encapsulated in the AI4U's API, giving the modeler the possibility of declarative modeling, but allowing the flexible programmatic use of the specification. The specification of the declared environment is translated into a common Gym environment format, which facilitates the generation of code for common training and execution loops in the implementation of reinforcement learning algorithms. With this, the AI4U can generate execution loops.

For the generation of agent code to occur, it is necessary to provide some extra information to AI4U. That 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. 13, 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 in order to generate the agent's execution loops. Training and test loops are generated with the *stable-baselines* [40] PPO2 [37] algorithm, and the built-in A3C implementation. The script *EnvironmentGenerator* contains pieces of information such as the action shape (attribute *Action Shape*), and other attributes of the environment. Fig. 14 shows the code produced by an *EnvironmentGenerator*.

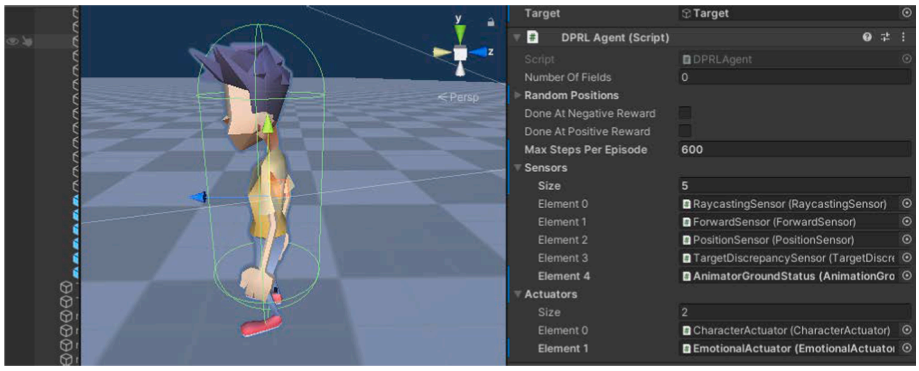


Fig. 12. Agent's configuration for automatic code generation using information about sensors and actuators. The *Max Steps Per Episode*, *Sensors* and *Actuators* fields provide information used to automatically generate the agent code. The *Max Steps Per Episode* field controls the agent's execution loop, determining the maximum number of simulation steps for each episode. In the *Sensors* field, there is a list of five elements, each of which describes a different type of sensor. The *Actuators* field describes a list of actions that the agent can take in the environment.

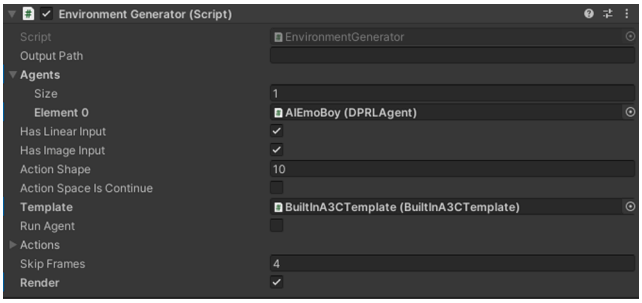


Fig. 13. Specifying environment settings for automatic code generation. The various fields allow you to specify the agents that will be generated (agents field), the types of the inputs (*Has Linear* Input and *Has Image Input* fields), number of actions (*Action Shape* field), the nature (continuous or discrete) of the actions (field *Action Space is Continue*), the algorithm used (Template field) and the number of frames skipped at each time step (*Skip Frames* field).

4.9. Other Features

AI4U offers a virtual humanoid character with several ready animated movements: jumping, turning, walking, running, crouching. In addition, the character comes with several facial expressions of emotion already. The character can easily be replaced by another avatar representation, reusing all the animations and facial expressions available. Fig. 15 shows a virtual character performing some movements and facial expressions.

AI4U integrated this virtual character, and successfully trained it to solve the point-to-point navigation problem, as shown in the results section of this work, presented next.

5. Results

In this section, we present some general aspects of AI4U's implementation followed by some didactic examples for demonstration purposes, and some simple application examples to demonstrate further possibilities of use of this modeling environment. AI4U was used by the authors in a research project for developing autonomous virtual characters with emotions [41], and it is being used in an ongoing research for modeling the environment. The use cases were modeled in Unity and Godot. Unity version and Godot versions share the same client code to training agents.

5.1. Implementation

The two main components of AI4U are the server and the client. The modeling environment on the server side provides information that the client uses for decision making. The server was coded with the C# programming language because it is a common language to both Unity and Godot. The client component, on the other hand, was coded using the Python programming language because it has a comprehensive community of Artificial Intelligence in general, and reinforcement learning in particular, that distributes code and works around an

```
class Agent(BasicAgent):
    def __init__(self):…
    def reset(self, env):…
    def act(self, env, action, info=None):
        for _ in range(4):
            envinfo = env.one_stepfv(action)
            if envinfo['done']:
                break
        state = get_state_from_fields(envinfo)
        return state, envinfo['reward'], envinfo['done'], envinfo
def make_env_def():…
make_env_def()
#END::GENERATED CODE :: DON'T CHANGE
def train():
    env = make_vec_env('AI4U-v0', n_envs=1) #Make the environment
    model = PPO2(MlpPolicy, env,verbose=1, n_steps = 128, learning_rate=0.00003, nminibatches=64, noptepochs=10,
    model.learn(total_timesteps=500000) #Training loop
    model.save('ppo2model') #Save trained model.
    del model # remove to demonstrate saving and loading
def test():
    env = make_vec_env('AI4U-v0', n_envs=1) #Make the environment
    model = PPO2.load('ppo2model')
    # Enjoy test loop with trained agent
    obs = env.reset()
    while True:
        action, _states = model.predict(obs)
        env.step(action)
        #env.render()
```

Fig. 14. Automatically generated test and training loop.

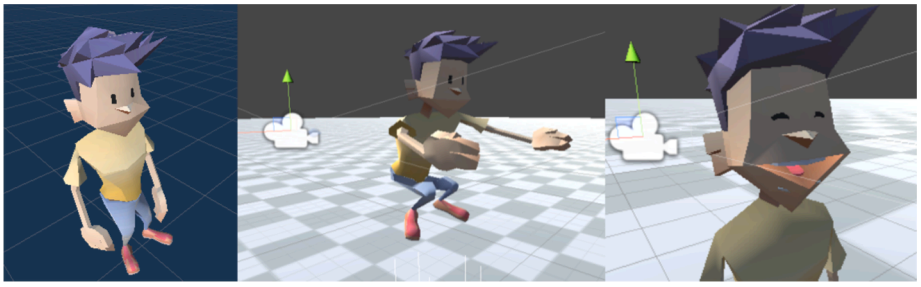


Fig. 15. Virtual character available for use and integrated with the AI4U API.

ecosystem based on the Python programming language. After training the agent, one can implement a controller that makes use of the trained model without needing the client code in Python. This is relevant if the model is put into production after being tested and approved. To show the potential of the developed modeling environment, five examples, based on AI4U, are presented in the next subsection.

5.2. Application Examples

AI4U allows for reinforcement and Artificial Intelligence learning experiments using the Unity game development platform. Thus, we modeled five scenarios with different degrees of difficulty and environment settings to show AI4U's versatility. All experiments were performed on the same computer, under the same conditions. Environments and agents were modeled by reusing components as much as possible. The five scenarios used in this article were: **BallCaseBox**, **MemoryV1**, **MemoryV2**, **MazeWorldBasic** and **NavBoy**.

Initially, a scene was visually created for each simulation scenario. Then, physical simulation was added to the scene elements. Sensors are added to the character's body so that the agent controlling the character can sense the physical elements of the scene. Then, actuators were added to the agent, so that it can interact with elements of the virtual scene. Another visually added component was the reward function, which defines the agent's goal.

Character control must also be added in order to coordinate information between sensors and actuators so that the agent maximizes the expected return. In addition, character control is defined in two possible ways: 1) via a remote Python script and 2) via a local controller usually programmed in C#. We use Python's remote script to train the agent, and local script to debug the scene during its construction.

A code generator component is added to the scene, and it is triggered when the scene is executed. The generator reads the properties of environment, sensors and actuators from the agent, and then generates code to train the agent according to number and type of sensors and actuators. The following two factors may indicate the necessity for the programmer to change this code: the support of the DRL framework used for the types and quantities of sensors and actuators; and the use of algorithms not supported by the adopted frameworks.

5.2.1. Scenario 1: BallCaseBox

The first scenario is the **BallCaseBox** environment. The ball is a character whose objective is to touch the box. So, we created a simple scenario containing a plane that represents the ground where both the ball and the target box are. We added two sensors for the ball: a ball position sensor and a box position sensor. We also added a motion actuator that lets one move the ball forward, backward, and sideways. In order to model the ball's ability to learn how to chase the box, we added a reward function to the box, which generates a $+1$ reward when the ball collides with the box. Also, we do not want the ball to fall off the plane. Thus, we added a reward function to the agent, which generates a reward of -1 whenever the agent is below the plane. All these components were added visually, without typing a single line of code while modeling the environment and the agent. Finally, we added a code generator component to the scene, which produced the client code in Python. The client code uses a reinforcement learning algorithm to train a neural network according to components added to the agent at modeling time. The **BallCaseBox** agent's neural network is a Multi-Layer Perceptron (MLP) network. The net receives six real values as input, which represent the positions of both the ball and the box. The output of the net is the probability of executing each of the four physical movements: forward, backward, left, and right. An image of the modeled environment and the result of training the agent using PPO2 [23], and A3C [4] algorithms (implementation of stable-baselines) are shown in Fig. 16.

5.2.2. Scenario 2: MazeWorldBasic

The second scenario is a maze exploration environment named **MazeWorldBasic**. This environment consists of a three-dimensional maze from which the agent has to find its way out before running out

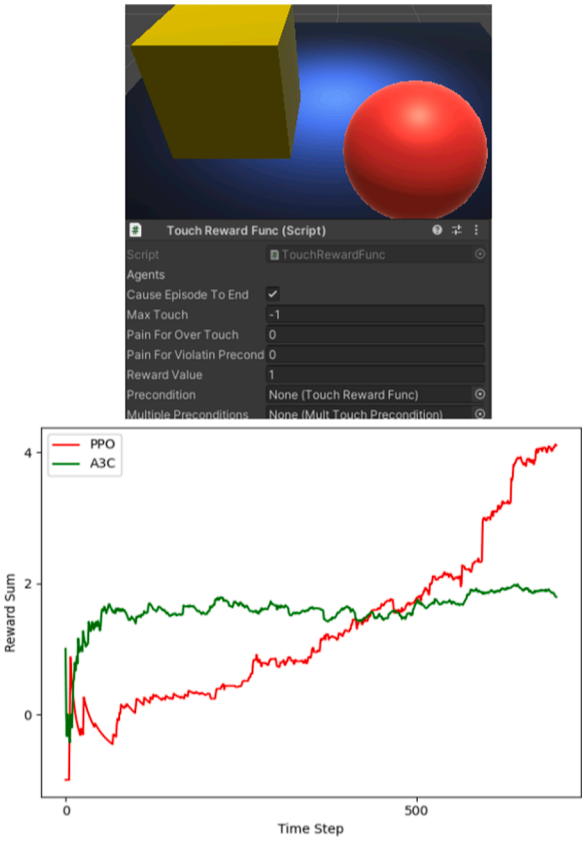


Fig. 16. The sphere controller is based on a neural network. The controller can apply force along the axes of the xz-plane. The result obtained over 800 episodes is shown. The graphs are automatically generated by the client-side API. On the left, it is shown the touch configuration of the target box. On the right, it is shown the training result of the A3C and PPO2 algorithms.

of energy. The maze contains red and green spheres scattered on the ground. 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. On the other hand, an energy gain occurs whenever a green sphere is touched. We modeled the agent by adding two sensors: a sensor based on ray casting and a touch sensor. The sensor based on ray casting provides the agent with a rudimentary first-person view of the environment, i.e., a 20×20-image produced by ray casting using perspective projection. The touch sensor, on the other hand, returns the code of the object that collided with the agent's body or returns 0 if there was no collision. The agent's virtual model is an animated character, so we had to add an actuator of the *CharacterActuator* type, which is capable of controlling the characters with animation actions for the agent's possible actions: *walking*, *running*, *jumping*, *turning left*, and *turning right*. The agent's objective is to find one of the exits. Thus, to define a reward function for this purpose, we placed boxes as outputs, and, for each box, we added a reward function that produces a value of $+100$ whenever the agent collides with the box. Then we added a code generation component, which produces code in Python to train the agent using the A3C and PPO2 algorithms. In this case, the neural network contains convolutional layers to receive the sequence of images captured by the agent's sensors. In addition, the neural network outputs the probability distribution of animated actions that the character can perform. Since the adopted frameworks support both MLP-type neural networks and networks with three-dimensional inputs, the generated code in this example did not have to be adapted. Fig. 17 shows how the environment looked like after modeling. In this environment, we tested the A3C and PPO2 algorithms.

5.2.3. Scenario 3: MemoryV1

The third scenario is a memory test named **MemoryV1**. In **MemoryV1** 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 $+10$. To model this goal, we add reward functions connected to objects of the same color, such that a reward is only generated if all objects of the same color are tapped at least once. The observations are similar to those obtained in the **MazeWorldBasic** environment. The discrete set of actions are: to rotate one-degree around the vertical axis of the agent, and to walk. For this environment, it was necessary to use the non-Markovian reward function specification, as shown partially in Fig. 18. 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.

5.2.4. Scenario 4: MemoryV2

The fourth experiment was designed to assess the possibility of sequential interactions with temporal dependencies. So, we design an environment in which the agent has to remember a past choice while making a choice in the present. In this environment, named **MemorvyV2**, we created a physical simulation of three rooms separated by two doors. In the first room, the agent is trapped by a door that must be opened by selecting a button of a certain color. There are two possible colors for the buttons: yellow and red. An image of this scene is shown in Fig. 19. To model the agent's behavior, we added a ray casting-based vision sensor and rigid body motion actuators (applying force in the agent's motion directions). Thus, the agent is able to detect and to approach objects.

Besides the image that is captured by the vision sensor, the agent has

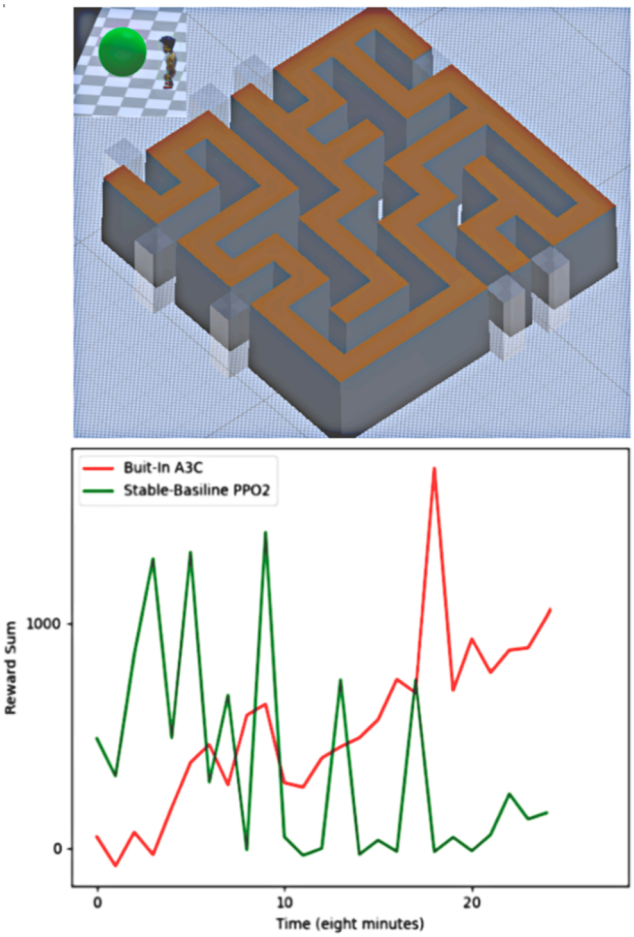


Fig. 17. Example of a maze exploration application using the built-in implementation of the A3C algorithm and the stable baseline implementation of the PPO algorithm. The figure shows: the maze and the character models (Top); and the training results of the A3C and PPO2 algorithms (Bottom).

access to the history of its interactions with the environment, described as a sequence of pairs, (a_t, r_{t+1}), where a_t is the action performed at time t and r_{t+1} is the received reward just after the action is performed. During an episode, only the pairs (a_t, r_{t+1}) with non-null rewards are recorded. So, at the outset of an episode, a button color is chosen at random to indicate the buttons that will produce rewards. To all those buttons, we assign a reward function component of the type *TouchRewardFunc* with a value of $+1$. To all the other buttons, we assign reward functions *TouchRewardFunc* with a value of -1. That function registers a node that represents the event in list of nodes of rewarded events. At every time step t before the end of the episode, the agent receives a partial list of rewarded events, $E = [(a_1, r_1), (a_2, r_2), \ldots, (a_t, r_t)]$. Those intermediate rewards are not used by the learning algorithm, but they are used as input to the agent's decision-making model. However, when the list of rewarded events is completed at the end of an episode, all the rewards in that list are used by the learning algorithm. In this experiment, the end of an episode occurs when the agent enters the target room. In order to record a sequence of button touches, a reward function of type *MultiTouchPrecondition* was assigned to the agent. That function has to be registered as a callback function that is triggered by the touch events generated by the *TouchRewardFunc* attached to the agent. For that, we implemented a protocol of type *publish-subscribe* [42].

MemoryV2 environment was designed to test learning with memory. The agent needs to remember past choices while solving a navigation problem between two rooms. In this case, we used the A3C algorithm with LSTM [43] networks to try to solve the problem. We

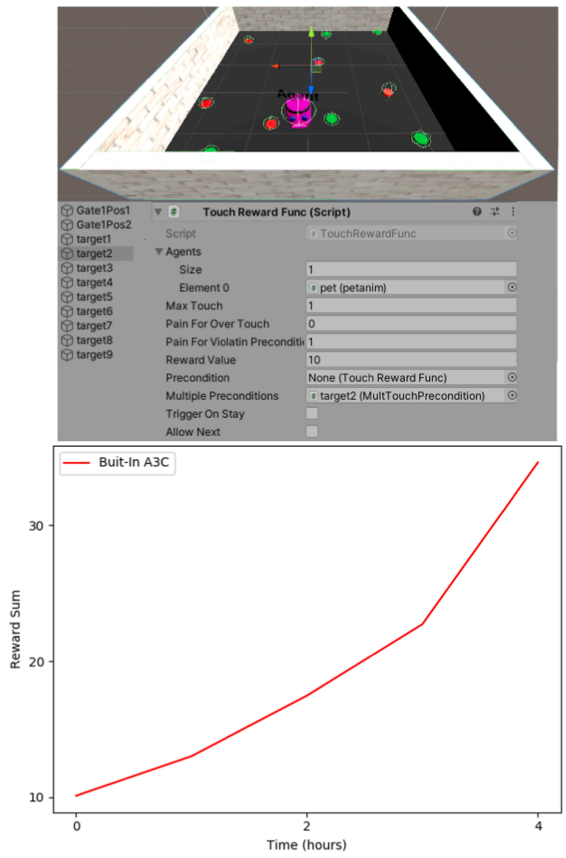


Fig. 18. Example of an application using the built-in implementation of the A3C algorithm. On the left, it is shown the top view of the environment, and on the right, it is shown the training result of the A3C algorithm.

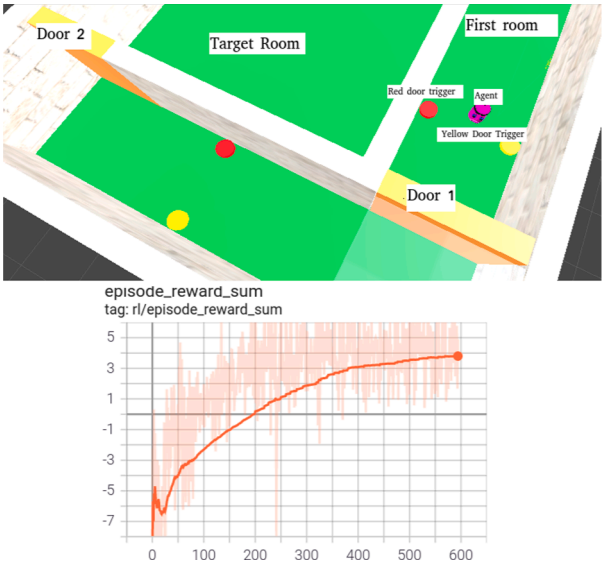


Fig. 19. Example of an application using the built-in implementation of the A3C algorithm. Above, part of the environment and the virtual character are shown. On the left, the environment and the agent are shown. On the right, the sum of reward per episode is shown over several simulation steps.

show the result of this experiment in Fig. 19.

5.2.5. Scenario 5: NavBoy

NavBoy is an experiment to solve the problem of navigation between two points in a complex scenario containing ramps and stairs. In addition, we used a humanoid character available in the AI4U modeling environment. This experiment shows the maturity of the AI4U for complex problems, in which the agent's training takes place in two different time steps, combining low and high level control. Fig. 20 shows the results obtained with the A3C algorithm. In this case, the problem of navigation in games using reinforcement learning was solved. The agent uses visual sensors and information regarding the arrival (target) position to solve the problem of navigating from one point to another in a scenario. To solve this problem, we used the A3C algorithm adapted with curriculum learning. First, the learning system learns how to solve simpler instances of the problem in which the target position is close to the agent. After the agent reaches a success rate of more than 50% in those simple instances, it continues the learning process with more complex instances, in which the target is located on a building further away from the agent. With this configuration, a success rate of more than 90% was obtained. This result is compatible with that obtained by Alonso [10], but we went further, as we obtained this result with a humanoid character who performs plausible animations of walking, jumping and turning.

6. Discussions

The environments modeled in the application examples show the viability of the developed modeling environment. The graphs in Figs. 16–20 show the results obtained with each algorithm used in the experiments. The algorithms were used with their respective default parameters. The goal here was to show the flexibility of the presented modeling environment 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 a trivial change of just a few lines of code. In addition to the presented environments, AI4U base code was used in the development of autonomous and emotional characters in [41]. A complete description of the navigation environment (**NavBoy**) was published by [44]. In addition to those examples, AI4U can be used in any game that is developed with the game engines Unity and Godot. For that, the game can be adapted to AI4U or use AI4U's API from the beginning.

Use case modeling was done with reusable components. For example, all agents in the **MazeWorldBasic**, **MemoryV1**, **MemoryV2** and **NavBoy** environments use a ray casting type sensor.

The **BallCaseBox**, **MemoryV1** and **MemoryV2** environments use characters that are represented by a single rigid body. The character moves as a result of direct application of forces to its body. In that case, the output of the neural network that controls the character is interpreted as the application of forces to the character's directions of movement. **MazeWorldBasic** and **NavBoy** environments have characters with animated and complex avatars. In that case, the output of the neural network that controls the character is interpreted as high-level actions that are performed by the character's animation controller, as described by [44].

The environments generated in all case studies can be used with any other reinforcement learning algorithms besides the ones tested and shown in this work. The generated code is easy to understand and read by anyone able to program in Python, as shown in Fig. 14.

Each application programmed using AI4U generated two components: the client and the server. The client is the code generated in Python to train and test the agent, which is a controller based on neural networks. The server is the simulation of the compiled environment that interacts with the client. A client can interact with any other compatible server. For example, a server created in Godot can interact with a client generated for a server of the same type created in Unity. For this, it is enough that the name of the sensors, actuators and input and output identifiers are the same names that the client code expects.

Thus, the task environment abstraction does more than to facilitate the description of agents, as it creates a common language that allows

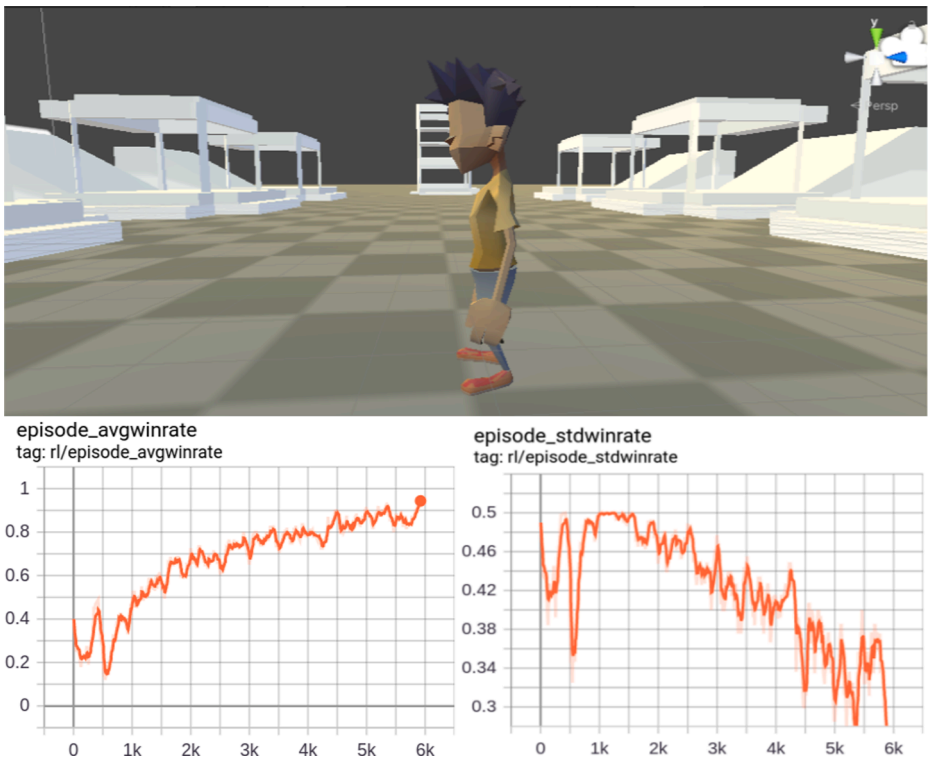


Fig. 20. Example of an application using the built-in implementation of the A3C algorithm. Above, part of the environment and the virtual character are shown. Below, the results obtained (success rate and standard deviation of the success rate) are shown.

interoperability between game programs and the reuse of components between different artifacts. Given that game development requires strict practices to deal with the complexity of this type of application, this feature facilitates the incorporation of artificial intelligence into game platforms.

7. Concluding Remarks

In this work, we enhanced our modeling environment AI4U to facilitate the preparation of reinforcement learning experiments in games. AI4U proved to be viable for modeling NPCs that learn based on reinforcement learning with neural networks. We were able to model several simple game experiments, but we also showed that it was possible to model an environment that simulates a more complex game scenario, with ramps and stairs, in which an NPC has to navigate between two random points. Thus, we were able to use AI4U in a relevant video-game problem.

Moreover, we showed that learning is a promising approach to obtain NPCs autonomous behaviors in video games. 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. AI4U helps to overcome this barrier by promoting the visual specification of reward functions, environments and agents, in addition to automatic code generation. Besides, the integration of AI4U with third-party algorithms is almost straightforward, as long as they follow Gym's environment specification. The use of AI4U, though, is not limited to environments that follow Gym's environment format. However, without following that specification, the automatic code generation does not work. This modeling environment proved to be flexible in the development of five case studies with different levels of complexity, integrating them with both AI4U's internal algorithms and algorithms implemented by third parties. In addition, our modeling environment integrates with two popular game development tools: Unity and Godot.

We shown that AI4U provide a set of benefits. It allows the specification of a reward function intuitively and interactively, which reduces the problem to make autonomous characters perform complex behaviors using a process in which the modeler can control only one reward function that produces a scalar value [45]. It allows the incorporation of visual specification of a reward function, which is compatible with the formal specification. It permits the integration with a game development platform. It permits the specification of agents' reusable components as sensors and actuators, facilitating reuse and code generation. Thus, AI4U contributes to facilitate the experimentation of reinforcement learning in games and to the reproducibility of experiments carried out in this domain.

The next steps in this research include a search for hyper-parameters of training algorithms for a deeper analysis of the experiments performed in this work and the modeling of more complex reinforcement learning experiments aimed at obtaining autonomous virtual characters.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Supplementary material

Supplementary data associated with this article can be found, in the online version, at https://doi.org/10.1016/j.entcom.2022.100516.

References

[1] A.L. Samuel, Some studies in machine learning using the game of checkers, IBM J. Res. Dev. 3 (3) (1959) 210–229, https://doi.org/10.1147/rd.33.0210.
[2] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A.S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al., Starcraft ii: A new challenge for reinforcement learning, arXiv preprint arXiv:1708.04782.
[3] G.N. Yannakakis, J. Togelius, Artificial intelligence and games, Vol. 2, Springer, 2018.
[4] 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, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (2015) 529–533, https://doi.org/10.1038/nature14236.
[5] J. Beck, K. Ciosek, S. Devlin, S. Tschiatschek, C. Zhang, K. Hofmann, Amrl: aggregated memory for reinforcement learning, in: International Conference on Learning Representations, ICLR, 2019.
[6] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R.H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, et al., Model-based reinforcement learning for atari, in: Eighth International Conference on Learning Representations, IEEE, 2020.
[7] B. Neyshabur, S. Tu, X. Song, Y. Jiang, Y. Du, Observational overfitting in reinforcement learning, in: International Conference on Learning Representations, NeurIPS, 2020.
[8] J. Forgette, M. Katchabaw, Learned behavior: Enabling believable virtual characters through reinforcement, in: Integrating Cognitive Architectures into Virtual Character Design, IGI Global, 2016, pp. 94–123.
[9] P.B.S. Serafim, Y.L.B. Nogueira, C.A. Vidal, 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. 117–11709.
[10] E. Alonso, M. Peter, D. Goumard, J. Romoff, Deep reinforcement learning for navigation in aaa video games, arXiv preprint arXiv:2011.04764.
[11] I. Makarov, P. Zyuzin, P. Polyakov, M. Tokmakov, O. Gerasimova, I. Guschenko-Cheverda, M. Uriev, Modelling Human-like Behavior through Reward-based Approach in a First-Person Shooter Game, MPRA Paper 82878, University Library of Munich, Germany (Jul. 2016). https://ideas.repec.org/p/pra/mprapa/82878.html.
[12] S. Feng, A.-H. Tan, Towards autonomous behavior learning of non-player characters in games, Expert Syst. Appl. 56 (2016) 89–99, https://doi.org/10.1016/j.eswa.2016.02.043, https://www.sciencedirect.com/science/article/pii/S0957417416300768.
[13] R.F. Julien Dossa, X. Lian, H. Nomoto, T. Matsubara, K. Uehara, A human-like agent based on a hybrid of reinforcement and imitation learning, in: 2019 International Joint Conference on Neural Networks (IJCNN), 2019, pp. 1–8. doi: 10.1109/IJCNN.2019.8852026.
[14] A. Juliani, V.-P. Berges, E. Vckay, Y. Gao, H. Henry, M. Mattar, D. Lange, Unity: A general platform for intelligent agents, arXiv preprint arXiv:1809.02627.
[15] G. Wuytjens, Modelling and transformation of non-player characters' behaviour in modern computer games, Ph.D. thesis, Universiteit Antwerpen (2012).
[16] O. for Blind Review, Omitted for blind review, in: Omitted for Blind Review, IEEE, 2020, pp. 19–28.
[17] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Prentice Hall, 2010.
[18] A. Thorn, Moving from Unity to Godot, Springer, 2020.
[19] Omitted for Blind Review, https://omitted_for_blind_review, accessed:2020-07-20.
[20] A. Camacho, R.T. Icarte, T.Q. Klassen, R.A. Valenzano, S.A. McIlraith, Ltl and beyond: Formal languages for reward function specification in reinforcement learning., in: IJCAI, Vol. 19, 2019, pp. 6065–6073.
[21] M.G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: An evaluation platform for general agents, Journal of Artificial Intelligence Research 47 (2013) 253–279.
[22] 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.
[23] E. Kolve, R. Mottaghi, W. Han, E. VanderBilt, L. Weihs, A. Herrasti, D. Gordon, Y. Zhu, A. Gupta, A. Farhadi, Ai2-thor: An interactive 3d environment for visual ai, arXiv preprint arXiv:1712.05474.
[24] T. Beysolow II, Custom openai reinforcement learning environments, in: Applied Reinforcement Learning with Python, Springer, 2019, pp. 95–112.
[25] 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.
[26] K. Cobbe, C. Hesse, J. Hilton, J. Schulman, Leveraging procedural generation to benchmark reinforcement learning, arXiv preprint arXiv:1912.01588.
[27] M. Johnson, K. Hofmann, T. Hutton, D. Bignell, The malmo platform for artificial intelligence experimentation., in: IJCAI, 2016, pp. 4246–4247.
[28] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, 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.
[29] 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.
[30] 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.
[31] M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, et al., Acme: A research framework for distributed reinforcement learning, arXiv preprint arXiv:2006.00979.
[32] Pybullet, a python module for physics simulation for games, robotics and machine learning, https://github.com/bulletphysics/bullet3, accessed:2020-07-20.
[33] R.D. Gaina, A. Couëtoux, D.J. Soemers, M.H. Winands, T. Vodopivec, F. Kirchgeßner, J. Liu, S.M. Lucas, D. Perez-Liebana, The 2016 two-player gvgai competition, IEEE Transactions on Games 10 (2) (2017) 209–220.
[34] N. Avradinis, S. Vosinakis, T. Panayiotopoulos, A. Belesiotis, I. Giannakas, R. Koutsiamanis, K. Tilelis, An unreal based platform for developing intelligent virtual agents, WSEAS Transactions on Information Science and Applications (2004) 752–756.
[35] P. Gestwicki, Unreal engine 4 for computer scientists, Journal of Computing Sciences in Colleges 35 (5) (2019) 109–110.
[36] M. Toftedahl, 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.
[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347.
[38] 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.
[39] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, CoRR abs/1606.01540. arXiv:1606.01540. http://arxiv.org/abs/1606.01540.
[40] 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, Y. Wu, Stable baselines, https://github.com/hill-a/stable-baselines (2018).
[41] O. for Blind Review, Omitted for blind review, in: Omitted for Blind Review, 2019, pp. 223–231.
[42] L. Roffia, F. Morandi, J. Kiljander, A. D'Elia, F. Vergari, F. Viola, L. Bononi, T. S. Cinotti, A semantic publish-subscribe architecture for the internet of things, IEEE Internet of Things Journal 3 (6) (2016) 1274–1296.
[43] S. Hochreiter, J. Schmidhuber, Lstm can solve hard long time lag problems, Adv. Neural Inform. Process. Syst. (1997) 473–479.
[44] O. for Blind Review, Omitted for blind review, in: Omitted for Blind Review, IEEE, 2021, pp. 19–28.
[45] S. Hamdy, 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.