Two Level Control of Non-Player Characters for Navigation in 3D Games Scenes: A Deep Reinforcement Learning Approach

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*—This paper presents a deep reinforcement learning approach for navigation problem coupled with traditional animation processes in games. Deep Reinforcement Learning (DRL) is a promising approach for this problem. So, we design a Non-Player Character (NPC) as an autonomous agent guided by a neural controller. Our approach works with the control of virtual game characters at two different levels of abstraction. A neural controller produces high-level actions, which are performed by both a character controller and an animation controller. We test our approach on a three-dimensional game scene. We find that our approach achieves promising results in navigation of animated characters in a game scene.**

***Index Terms*—non-player characters, reinforcement learning, navigation problem.**

I. Introduction

Generating navigation behaviors is a typical problem when dealing with current Non-Player Characters (NPCs). A first approach to game navigation was the Waypoint graph [1], which consists of a set of points of interest connected to each other. This approach has several main drawbacks: its manual construction is prone to human errors, it can not control dynamic objects, it is expensive to build since it needs to check all $n(n\text{-}1)$ combinations of paths, and the paths tend to not look realistic since all agents follow the same set of constrained paths [2].

NavMesh is the currently dominant approach for navigation problem and it has overcome the main waypoint graph problems. NavMesh divides the game map into a set of convex regions, witch can each be trivially navigated within [2]. Once the polygons have been placed, a graph is created by using the polygons as nodes and by connecting adjacent polygons with edges [2]. In each polygon, navigation is trivial. So, search algorithms (such as A* and Dijkstra's algorithm) find the shortest path between any two nodes. The found path is typically smoothed to look more realistic to the player.

However, NavMesh lack the scalability and the flexibility necessary to provide more versatile navigation behaviors that enable the character to follow a diverse variation of paths. This is because adding new abilities to a preprogrammed agent requires adding new edges to the navigable polygon graph. While this is not an issue for simple skills, adding complex skills often requires a redraw of the navigation map prone to failure and rework.

Alonso and Peter [2] show that reinforcement learning can be more flexible than NavMesh in generating complex navigation behaviors for NPCs. They also argue that reinforcement learning is a promising approach to point-to-point navigation in three-dimensional mesh environments. Reinforcement learning works for the navigation problem usually use very simple characters, without character part animations for skills such as walking or jumping (see reference [2] and Mirowski's classic work [3]). Furthermore, if we consider the success rate of hitting the target, for predefined environments, NavMesh and classic search (A*) achieved maximum success rate. But that approach is not flexible in the sense that new agent scenarios and skills would require manual readaptation of the agent. Thus, their results were obtained using only simple avatars (often a box with a texture) and basic movements (without articulation of limbs, for example). Training a reinforcement learning model to learn from scratch how to produce animations (e.g, articulated limbs for walking) is computationally expensive. However, that feature is essential to viable NPCs in modern games. So, without an appropriate decision-making architecture, even the most complex behaviors of NPCs lack animation expressiveness in their actions and are visually cumbersome.

In this work, we use a model-free reinforcement learning since it does not force us to make strong assumptions about the environment, and, therefore, it helps to reduce the modeling time of the problem. We tried an approach for learning navigation behavior in three-dimensional games in terms of computational resources, which was easier to integrate with game animation modeling tools, and minimizes memory consumption. Thus, we use A3C algorithm for training because we do not make assumptions about heavy use of hardware. In fact, we want that our approach is able to run in many different types of hardware. In addition, we adopted curriculum learning and an informed reward function according to the approach presented in [2].

Thus, our approach train a humanoid character through two levels of control. In the first level, a neural network sends an abstract action (for example, walking) to a programmed character controller; and, in the second level, the character controller interprets the received action in terms of velocity, and triggers an animation produced by an animation controller in a synchronized manner. This procedure makes it possible to obtain movement with convincing animations, which are designed by human modelers. We show how to encapsulate the state of the animation to preserve the property of the Markovian state necessary for an agent to learn to navigate in a three-dimensional environment, while exhibiting convincing animation of its body.

II. Related Works

The use of reinforcement learning for training NPCs has aroused the interest of the academic community for about a decade [4], [5], [6], [7], [8], but only recently, robust results for AAA games have been achieved [2]. Still, only recently game development tools have started to support agent training using environments built with the tool, even though this support is primarily for testing algorithms, techniques and machine learning approaches using games as test beds for artificial intelligence [9], [10]. In 2020, it was shown that the navigation behaviors, in 3D videogames, obtained with reinforcement learning are superior to those obtained with traditional approaches [2]. The training of an NPC with reinforcement learning capable of expressing emotions and varying the NPC's behavior according to its emotional state is shown by Gomes *et al.* [11].

Navigation is what takes an NPC from one point to another on a game map, so it is an essential component of an NPC. NavMesh is the game industry's preferred approach to deal with the navigation problem. Unfortunately, complex navigation skills that extend the character's movement ability increase NavMesh's complexity, making it unmanageable in many practical settings [2]. Game designers are therefore restricted to only adding skills that can be manipulated by NavMesh. Thus, Alonso *et al.* [2] proposed the use of Deep Reinforcement Learning (Deep RL) model-free to learn navigation through 3D maps using any navigation skill. They further tested that approach in complex 3D environments using the Unity Game Engine [9], and maps that were one order of magnitude greater than those normally used in the Deep RL literature. The authors reported success rates above 90% in all tested scenes, including a real game scene.

Previous work used model-free RL for navigation, either in simple 2D environments [12], [13] or in complex navigation problems [2]. Recent works, to overcome the absence of planning in model-free RL, used a hierarchical architecture, where intermediate goals are given to a controller by a high-level planner [14]–[16]. As navigation in a visually complex environment is usually modeled as a partially observable Markov decision process, the importance of memory use has been previously recognized [3]. Although unstructured memory like LSTM (Long-Short Term Memmory) [17] can be used, architectures involving spatially structured memory have also been explored [18], [19]. The use of auxiliary tasks to accelerate the learning of RL problems based on challenging goals was also a subject of study [3], [20], [21]. Alonso *et al.* [2] used the SAC (Soft Actor Critic) algorithm as an end-to-end solution to the navigation problem over a 3D map. The agent is controlled by a neural network whose input consists of a 3D occupation map, a 2D depth map and a linear input containing information such as the agent's position and the target position in the game's environment. The agent uses a neural network with an LSTM layer to handle partial information.

Therefore, our work differs from those in the literature because it combines reinforcement learning with a neural controller specially designed to interact with a character's controller. So, the agent has two levels of control in which a higher level controller selects abstract actions (e.g., walk) that are effectively performed by a preprogrammed lower level controller. Moreover, for the reinforcement learning problem, we combine navigation learning using the A3C algorithm combined with curriculum learning and an informed reward function (which is the opposite of a sparse function). For the control of the NPC, we use a neural network architecture that acts according to environment state perceived by the agent.

III. Methods

Here, we shown our Deep Reinforcement Learning (DRL) System for character control in navigation problem in games. The main objective of the agent is to reach a given target position randomly placed in the game's scene. For this, the agent's architecture is endowed with four main components: (1) a neural controller, (2) an agent body with actuators and sensors, (3) a controller that receives actions from agent's actuators and translates those actions into agent velocity that are applied to the agent in a manner that is consistent with the agent's body animations, and (4) an animation controller that runs the agent's body animation. The environment interaction interface consists of sensors and actuators, which are modeled with the AI4U tool [10], a public tool available to facilitate the development of game environments with support for reinforcement learning algorithms through the Unity Game Engine [9]. We also detail the components of the agent's architecture and the way to integrate them to generate navigation behaviors

with expressive animations. The overall DRL agent is shown in Fig. 1.

A. Neural Network Controller

The neural network controller implements the agent's policy. The neural network (NN) receives perceptions of virtual sensors, and, in response to those perceptions, it outputs a discrete probability distribution $p = p_{a_1}, p_{a_2}, ..., p_{a_k}$, where p_{a_i} is the probability of the action a_i. The agent take an observation o_t at time step t, and selects the action a_i in a probabilistic manner according to the probability distribution p.

B. A Neural Network Architecture

The agent's neural network has a *feedforward* architecture with two input layers, two hidden layers, and two output layers. The amount of neurons in the input and output layers depend on the type of problem you want to solve. To determine the amount of input neurons, it was taken into account the agent's perception system. In addition, one of the output layers produces the actions' probabilities, and the other output layer estimates the state value. The agent uses the actions' probability distribution to select the current action. Estimated state values are used during the agent's training.

The network's inputs constitute an observation o_t at time step t, which contains two groups of perceptions: a visual data, and a linear data. The visual data is a sequence of bidimensional matrices R_k shaping a three-dimensional signal $[I_3, I_2, I_1, I_0]$, where I_k is a visual signal perceived by the agent at time step $t - k$. Hence, to represent the visual data, we use frame stacking strategies based on [22]. The image I_k is a matrix with the codes of the seen objects in each cell, in this case. The linear input consists of the agent's forward direction, orientation relative to the target, distance to target, jump status, and touch signals. In the next section, we describe this inputs in more details.

To process those two types of input data, the network has two feature extractors: a convolutional bidimensional feature extractor, and a linear feature extractor. The convolutional feature extractor has two layers that use the *ReLu* [23] activation function: the first layer consists of 128 (4×4) filters with stride equals to 2, and the second layer consists of 64 (2×2) filters with stride equals to 1. The linear feature extractor has two dense *ReLu* layers containing 30 neurons each. The input features are concatenated to be processed by the deeper layers (see Fig. 2).

We use two configurations of the deeper layers. The simpler one is shown in Fig. 3. The more complex configuration, shown in Fig. 4, uses two stacked LSTM (Long-Short Term Memory) [17] layers. The purpose of the latter configuration is to deal with aspects of the environment that are not directly observable. Thus, the size of the visual input has also been increased to a sequence of size eight so that the recurrent neural network LSTM can learn from sequential data. Experimentally, we found that the use of larger sequences in the input increases the performance of networks that learn sequential decisions, but at a higher computational cost.

C. The Agent's Interface with the Environment

The agent has sensors to perceive the environment and actuators to control the NPC. The NPC is controlled from a player's perspective, however, autonomously, because instead of being controlled by a human, the character is controlled by a script that makes decisions based on an underlying neural network model. Thus, the NPC runs on the Unity Engine architecture and in the AI4U framework, which allows control of the NPC through neural network. The agent's neural network selects actions based on the current state. The interdependence of the components of this architecture is shown in Fig. 5.

The actions generated by the neural network of the agent are applied to a physical controller and to an animation controller based on unity's animation mechanisms. More specifically, the neural network outputs an abstract action, such as "walk", to the physical controller, which moves the agent's avatar, say, forward, while the animation controller produces coherent avatar's movements, for example, the avatar's legs' animation. That two-level control is essential to apply reinforcement learning to games, because it integrates reinforcement learning with animations produced by artists.

Two kinds of sensors were provided to the agent: a linear sensor, and a two-dimensional visual sensor. The linear sensor captures global features, such as the agent's world orientation, relative orientation, distance to target, and touch on environment's objects. The two-dimensional visual sensor returns an array representing the local image seen by the agent at a given time step. While the linear sensor provides information more directly related to the goal of the agent's navigation, the two-dimensional visual sensor provides an input channel that allows the agent to perceive nearby objects, allowing the agent to learn how to interact with objects in a manner that is consistent with the agent's view.

The visual sensor uses ray-casting to generate an image I_t at time step t. In our model, a 30×30-pixel image is generated by casting rays that define a symmetrical frustum with a 90 degree field of view, as shown in Fig. 6. Therefore, as a result of its artificial vision, at each instant, the agent receives a matrix of 30×30 float-point numbers representing the visual information.

In turn, at each time step t, the agent's linear sensor captures:

- $\mathbf{d}_t = (G - P_t)/||(G - P_t)||$, the unit vector from the agent's current position, P_t, to the target's position, G, which does not change during an episode;
- $\mathbf{f}_t$, the agent's view direction (a unit vector);
- $\delta_t = \mathbf{f}_t \cdot \mathbf{d}_t$, the agent's orientation relative to the target orientation;
- $S_t = (g_t, h_t)$, the animation controller's status, where g_t is a Boolean variable, which is 0 if the agent is on a flight mode, and it is 1 if the agent is in contact with a floor, h_t is the absolute height of the agent with respect to the ground floor; and

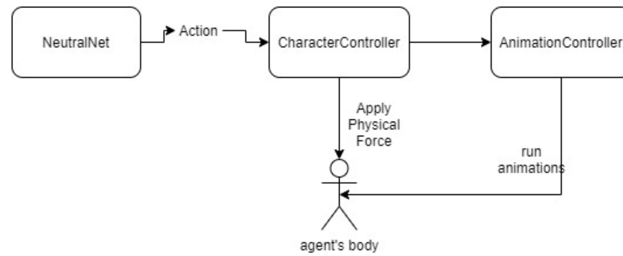

Fig. 1. An overall description of our DRL System.

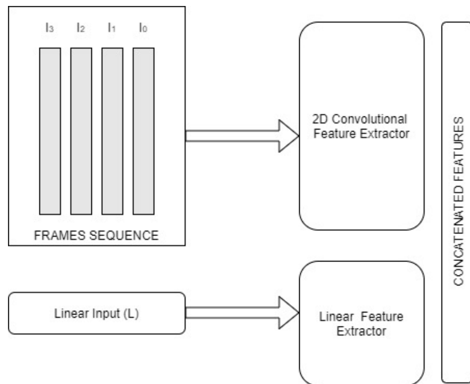

Fig. 2. Convolutional layers for extracting features from the agent's visual input.

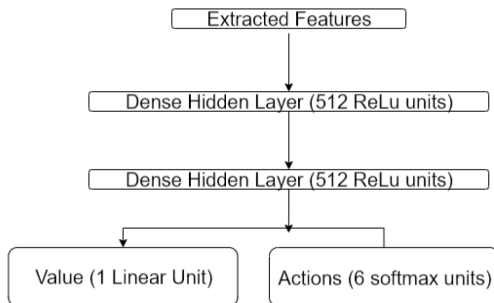

Fig. 3. Deeper layers of the network. In this configuration, we use traditional fully connected feedforward layers.

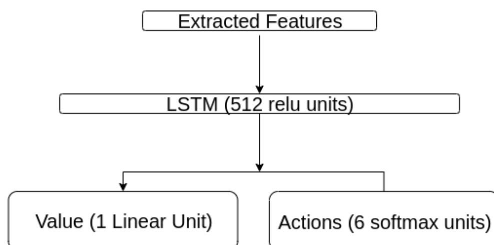

Fig. 4. Deeper layers of the network. In this configuration, we use two stacked LSTM layers.

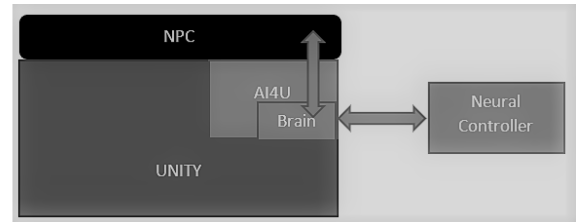

Fig. 5. Map of Interrelations between the components of the approach used in this work. The NPC implements its functions based on AI4U and Unity. AI4U uses the functions of Unity and provides a specific view or form of use of Unity for those who use AI4U.

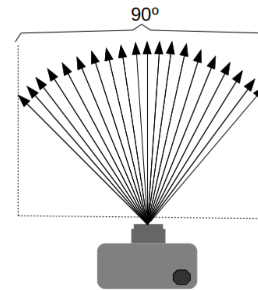

Fig. 6. Ray casting with an aperture angle of 90 degrees and symmetrical perspective projection.

- $T_t = (T_w, T_g)$, the touch sensor's signal, where the variables $T_w = T_g = 0$ if no touch is detected, $T_w = 1$ when the agent touches a wall, and $T_g = 1$ when the agent reaches the target.

In addition to information captured by the linear sensors, the model receives the pair, (a_{t-1}, r_t), where a_{t-1} is the previous action performed by the agent, and r_t is the reward the agent received for that action. Therefore, the linear input is represented by the tuple $L_t = (\mathbf{d}_t, \mathbf{f}_t, \delta_t, S_t, T_t, a_{t-1}, r_t)$.

The actual input to the agent's neural network at time t consists of the last four sequences of data from both sensors, $O_t = (L_{t-3}, L_{t-2}, L_{t-1}, L_t, I_{t-3}, I_{t-2}, I_{t-1}, I_t)$.

To select actions, the agent has a virtual actuator that physically controls the NPC in the three-dimensional environment of the game. Virtual actuator has two levels of control. In first level, the agent's neural controller sends a high-level command $action \in [0, 1, 2, 3, 4, 5]$ to the second level. Then, the second level translates the received action into an action encoding

(numerical array), and sends it to the character's controller. The character's controller applies a velocity proportional to the value of the sent command, and activates an animation controller to produce a corresponding animation based on the received action. Specifically, at every time step t, the character's controller translates the command $action$ into a vector $a = (fb, lr, ju, jf)$, where fb is a real number that is used in Equation (1) to compute the agent's forward and backward velocity; lr is a real number that represents the agent's *left* or (*right*) rotation; $ju = 1$ if the agent is supposed to perform an upward jump and $ju = 0$ otherwise; and $jf = 1$ if the agent needs to perform a forward jump and $jf = 0$ otherwise. The command list supported by the agent's neural controller is defined as:

- Forward (command 0): $[f, 0, 0, 0]$, where v is a fixed positive real number;
- Backward (command 1): $[-b, 0, 0, 0]$, where b is a fixed positive real number;
- Turn Left (command 2): $[0, l, 0, 0]$, where l is a fixed positive real number;
- Turn Right (command 3): $[0, -r, 0, 0]$ where r is a fixed positive real number;
- Jump (command 4): $[0, 0, 1, 0]$; and
- Jump Forward (command 5): $[0, 0, 0, 1]$.

For example, if the actuator controller receives a command code 0 (Forward movement), it translates it into an action vector $a = (f, 0, 0, 0)$. The character controller translates this action into the agent's velocity

$$\mathbf{v} = (\mathbf{f}_t \times f \times \mu)/\Delta t, \quad (1)$$

where Δt is the time step between two successive frames, and μ is a multiplier factor to adjust speed to different time scales. Then, the animation controller produces expressive movements synchronized with the agent's velocity.

D. Deep Reinforcement Learning in Two Time Steps

In real games, the human-like character's movements are expected to be animated to resemble human movements. Instead of training the character to learn how to solve its main navigation objective while learning to articulate its limbs in a realistic manner, we train the character to learn how to solve its main objective while being animated by traditional animation mechanisms, which provide a rich library of ready-made animations designed by professional modelers and artists.

Joint training of the agent to obtain its main behavior while managing an animation controller (which affects the character's movement) adds a level of extra time to the result of the agent's actions. This is because the actions generate effects with different duration (see Fig. 7), especially when using an animation controller. For example, when the agent performs a jumping action, no other movement actions will produce any effect when the agent is in the air, except the rotation of certain body parts, such as the turning of the head. We simply expand the state of the environment with information from the animation controller to allow the reinforcement learning algorithm to learn how to solve the problem with a high success rate. The necessary information is the height of the agent with respect to the ground reference plane and the feet contact status with the floor.

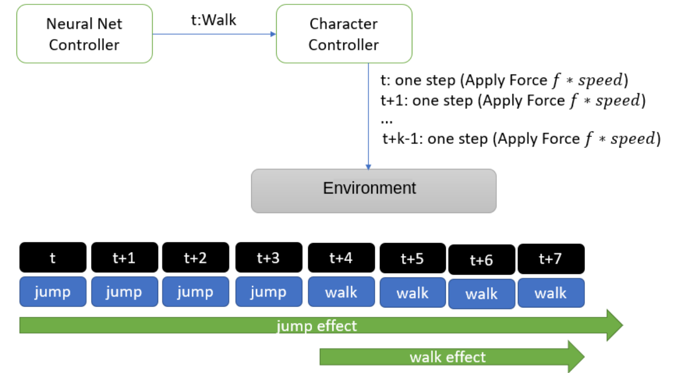

Fig. 7. The boxes labeled with a white or gray background indicate the components of our reinforcement learning system. The blue arrow indicates the sending of commands or actions. When the character controller receives an action, it repeats that action k times in the environment. The black boxes indicate the time steps in which the actions were applied. The blue boxes indicate the actions applied in their corresponding time steps. The green arrows indicate the duration of the effects of each action. Note that the jump action affects the effects of the subsequent actions.

E. The Reward Function

We use the AI4U [10] tool's description of reward functions to annotate events in the environment with functions that generate reward. More specifically, each agent's action was noted with a reward function based on [2] and shown in Equation 2:

$$R_t = max(min_{\forall i \in [0,t-1]} E(t,i), goal), 0) - \alpha + 100_{touch(agent,goal)}, \quad (2)$$

where $E(t,i) = D_i(agent, goal) - D_t(agent)$, D_t is the Euclidean distance between the positions of its arguments at time t, α is a positive value that represents a penalty for each performed action (the agent is expected to perform the least amount of actions possible for reach its goal), and $touch(agent, goal)$ is a predicate that is true when the agent touches the object that represents the position it has to reach, and false otherwise. Note that $100_{touch(agent,goal)}$ is equal to one hundred (100) only if the predicate $touch(agent, goal)$ is true, otherwise it produces a 0 (zero).

We use the AI4U tool to implement Equation (2) in order to visually associate events in the environment with the generation of annotated rewards. So, AI4U manages the agent's reward at each time step t. At the beginning of a time step t, that is, before the environment receives the agent's next action, the value of r is equal to zero.

Predefined objects (which in Unity are called prefabs) are associated with events in the environment and increases or decreases r as those events occur. We set up an event that is fired whenever the environment receives an action sent by the agent. In this case, the event adds the first part of Equation (2) to r, that is,

$$R_t^{(1)} = max(min_{\forall i \in [0,t-1]}(E(t,i), goal), 0) - \alpha. \quad (3)$$

Then, we associate the collision event with the target object, which indicates that the agent reaches the the main goal. The reward generation function produces

$$R_t^{(2)} = 100_{touch(agent,goal)}. \quad (4)$$

In practice, this means that r is equal to the result shown in Equation 2, that is, $r = R_t = R_t^{(1)} + R_t^{(2)}$.

F. Agent's Training

Our approach was partially based on the work of Alonso *et al.* [2], with necessary adaptations for achieving efficiency and openness. To achieve efficiency we make as few assumptions as possible about the hardware requirements when using the training algorithm and the action execution model. To achieve openness we consider a training algorithm with public implementations adaptable to the demands of our work. Thus, since the A3C algorithm has widely used and tested public implementations, we adapted it to run on low-cost hardware, as described in section III-B

We use the A3C algorithm for training in a point-to-point navigation scenario. The agent learns in two steps. In the first step, the agent learns an easy problem, and in the second step, a more difficult problem. In the performed experiments, a distance greater than 4.5 units of distance (ud) substantially increased the convergence time of the algorithm, so when the target is placed at a maximum distance of 4.5ud, the problem is considered easy to solve by RL. The target position is chosen to be placed randomly in one of the buildings in the scene. Then, the agent trains on an easy problem first until it reaches a 50% success rate. Next, the agent trains on a difficult problem, until it reaches a 90% success rate or up to approximately 6,000 training episodes.

IV. Experiments

To demonstrate the capacity of our DRL system, we used a map with dimensions of $400\text{m} \times 400\text{m} \times 35\text{m}$ shown in Fig. 8. The evaluated configurations are found in Table I. To show that our reinforcement learning system is capable of learning how to solve the problem of navigation in games when we have two different levels of control, we tested a base case and three variations of our system. In the first configuration, we used a feedforward neural network model (neural network architecture shown in Fig. 3), providing each required input to that model. In the second configuration, we used a feedforward neural network model, but we do not use information from the animation controller (jump status and NPC height). The purpose of this configuration is to verify the hypothesis that the agent only learns with a state description that takes into account information from the animation controller. Next, we tested two configurations with LSTM networks. One with a single layer and one with two stacked layers. The two stacked layers contain 128 neurons each. The single-layer LSTM network is shown in Fig. 4. Based in [3], LSTM hidden units output LSTM hidden activation signals.

During the training phase, each configuration of our DRL system was trained for 6,000 episodes. Each configuration shown in Table I was tested in the simulated environment with the same set of initial seeds for the different configurations. We obtained results with a confidence level of over 95%.

TABLE I
REINFORCEMENT LEARNING SYSTEM CONFIGURATIONS. FF IS THE NEURAL NETWORK ARCHITECTURE SHOWN IN FIG. 3 AND LSTM IS THE ARCHITECTURE SHOWN IN FIG. 4

Configuration	Model Architecture	State
Base	FF	It includes animation status
VAR1	FF	It does not include animation status
VAR2	LSTM (One Layer)	It includes animation status
VAR3	LSTM (Two Layers)	It includes animation status

V. Results and Discussions

The test results show that the proposed approach obtained NPC navigation behaviors with animated avatar. The graph in Fig. 9 shows the agents' success rate's evolution per episode over $6,000$ episodes during training. Fig. 10 shows the success rate's moving average per episode for each approach after the training of the agent over 100 episodes. In this case, we assess whether the agent maintains performance after training, that is, in a testing phase (in analogy to what is done in supervised learning). In the test phase, the agent has already been trained and it is no longer learning. In addition, the environment is initialized differently from the training phase, such that the position of obstacles is randomly placed in the scene. And in this case, the pseudo-random number generator is initialized with different seeds in relation to the training phase. During the test phase, it is natural that the success rate at the beginning is high and decreases with time until it stabilizes (see Fig. 10), given that the probability of the agent making a mistake increases with time.

The results show that the agent was able to learn in a scene with obstacles. The scene's topology contains ramps, walls and stairs arranged in a way that looks like a city street, but containing a wide open space that could cause the agent to wander in the empty region. Fig. 11 shows, qualitatively, the the agent's behavior of avoiding obstacles, jumping stairs, going up a ramp and successfully reaching the target.

The experiments showed that all variations of the proposed approach were able to improve their success rates during training. The Base configuration with a feedforward neural network obtained the best performance, both qualitatively and quantitatively. The comparison between *Base* and *VAR1* models tells us that the information from the animation controller and the height of the agent are two essential pieces of information for the agent to learn a refined control of its actions.

The agents with LSTM network (VAR3 and VAR4) obtained a much worse result than the agents Base and VAR1. Note that the number of episodes was stipulated *a priori*, considering a configuration of experiments with few computational resources. The problem is modeled as a partially observable problem, because at each moment, the neural network has the global information of the target's location and the local

Fig. 8. Map used in the experiments (a three-dimensional map with dimensions of 400m × 400m × 35m, containing ramps, obstacles and towers).

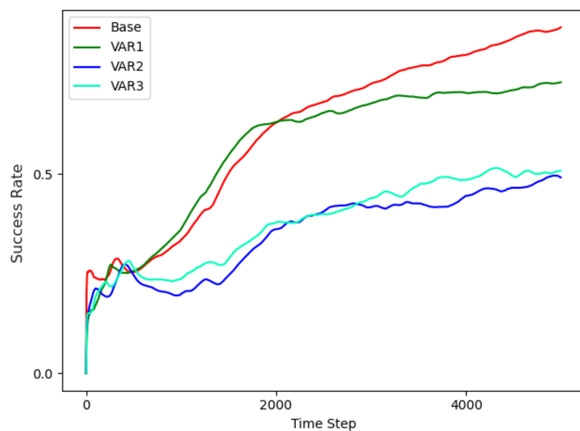

Fig. 9. The success rate's moving average of the last hundred episodes during training of the agents.

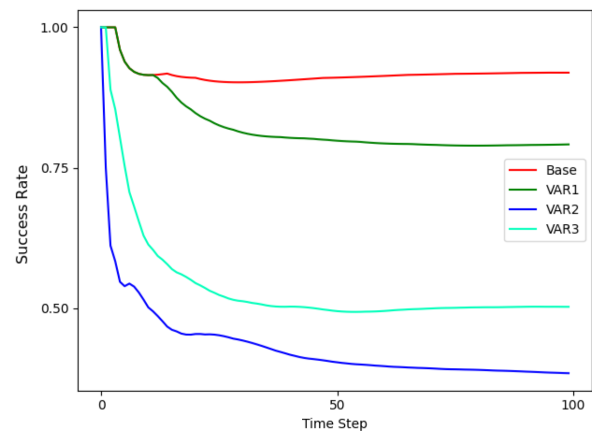

Fig. 10. The success rate's moving average during testing of the agents.

information of the obstacles, including the ground touch sensors and the agent's height sensor. Thus, at each instant, the agent has all the relevant information to reach the target position. In this scenario, LSTM networks are not essential. Thus, given the same amount of training, it is natural for a feedforward network to converge faster than a more complex model like a LSTM network.

VI. Conclusion

The reinforcement learning approach presented in this work proved to be effective in solving the navigation problem of animated NPCs in games, being able to combine global information, directly related to the goal, with visual information to promote navigation behavior that avoids different types of obstacles. We also used a direct and effective training strategy that combines easy and difficult episodes. This training strategy using the A3C algorithm proved to be efficient in obtaining policies with a higher entropy, which is necessary for environments with different scenes.

We also showed that managing an animation control, while the agent navigates the environments, adds extra complexity, since an animation controller can invalidate the Markovian state's assumption, hindering the learning convergence. This was solved simply by adding information from the animation controller to the state perceived by the agent. This approach achieved greater performance in the tests.

Finally, although we have shown that reinforcement learning systems are efficient for the point-to-point navigation problem of animated virtual characters, there is still a number of problems that need to be solved if reinforcement learning systems are viable candidates to replace the approaches of the

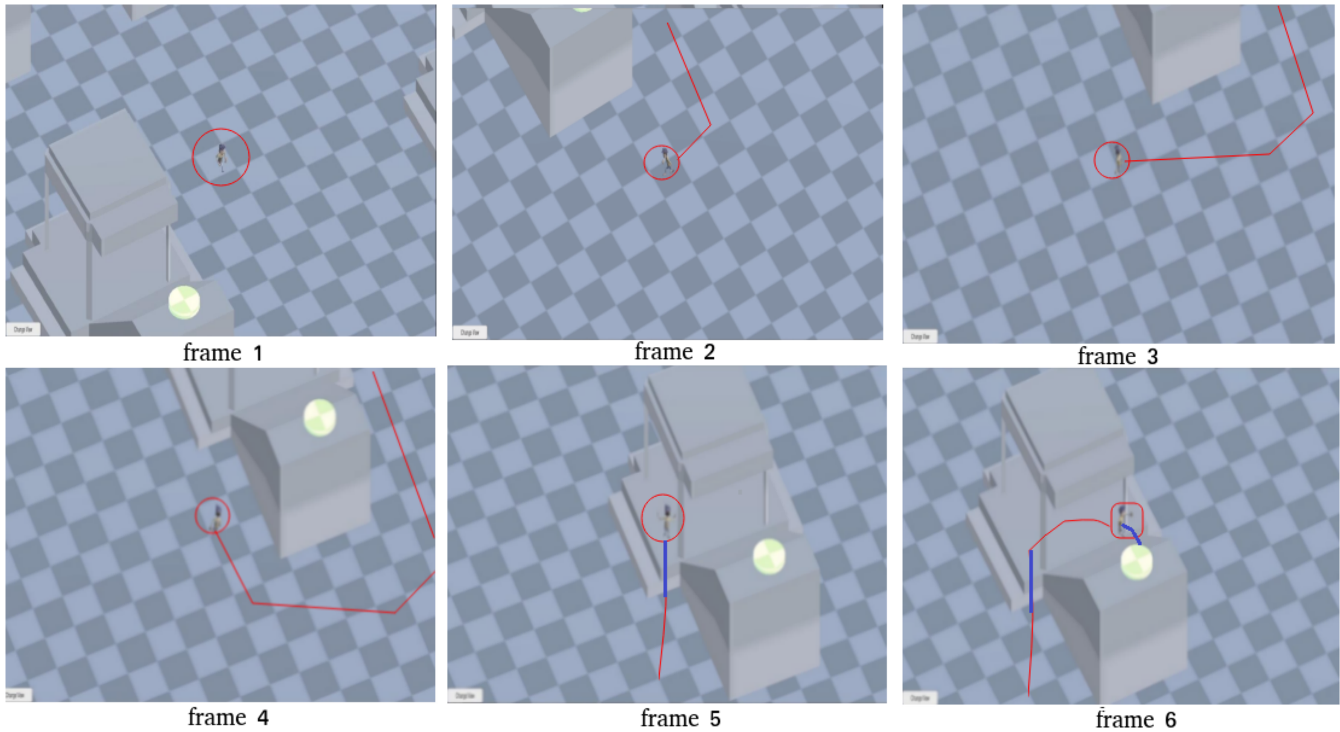

Fig. 11. The agent's behavior evading obstacles, jumping stairs and going up a ramp. The captured frames are in a temporal sequence, but with variable capture time between them. The red line indicates the agent's trajectory. The blue line indicates a jump and the straight line indicates that the agent performed the walk action. In this case, the agent reached the target (greenish cylinder).

classic NavMesh. Among those problems, we can mention: generalization of scene (adapting agents that are trained in one scene to function in other scenes), behavior variability, navigation problem with sub-problems along the way, and navigation in open worlds games.

References

[1] L. Lidén, "Strategic and tactical reasoning with waypoints," in *AI Game Programming Wisdom*, S. Rabin, Ed. Hingham, MA, USA: Charles River Media, 2002, pp. 211–220.

[2] E. Alonso, M. Peter, D. Goumard, and J. Romoff, "Deep reinforcement learning for navigation in aaa video games," *arXiv preprint arXiv:2011.04764*, 2020.

[3] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu *et al.*, "Learning to navigate in complex environments," *arXiv preprint arXiv:1611.03673*, 2016.

[4] S. Phon-Amnuaisuk, "Learning chasing behaviours of non-player characters in games using sarsa," in *Applications of Evolutionary Computation*, C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcázar, J. J. Merelo, F. Neri, M. Preuss, H. Richter, J. Togelius, and G. N. Yannakakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 133–142.

[5] T. Barron, M. Whitehead, and A. Yeung, "Deep reinforcement learning in a 3-d blockworld environment," *Deep Reinforcement Learning: Frontiers and Challenges, IJCAI*, vol. 2016, p. 16, 2016.

[6] F. G. Glavin and M. G. Madden, "Learning to shoot in first person shooter games by stabilizing actions and clustering rewards for reinforcement learning," in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, Aug 2015, pp. 344–351.

[7] J. Beck, K. Ciosek, S. Devlin, S. Tschiatschek, C. Zhang, and K. Hofmann, "Amrl: Aggregated memory for reinforcement learning," in *International Conference on Learning Representations*, 2019.

[8] A. Dobrovsky, U. Borghoff, and M. Hofmann, "Applying and augmenting deep reinforcement learning in serious games through interaction," *Periodica Polytechnica Electrical Engineering and Computer Science*, vol. 61, no. 2, pp. 198–208, 2017. [Online]. Available: https://pp.bme.hu/eecs/article/view/10313

[9] 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.

[10] G. Gomes, C. A. Vidal, J. B. Cavalcante-Neto, and Y. L. Nogueira, "Ai4u: A tool for game reinforcement learning experiments," in *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2020, pp. 19–28.

[11] 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)*, Oct 2019, pp. 223–231.

[12] P. Mirowski, "Learning to navigate," in *1st International Workshop on Multimodal Understanding and Learning for Embodied Applications*, 2019, pp. 25–25.

[13] E. Wijmans, A. Kadian, A. Morcos, S. Lee, I. Essa, D. Parikh, M. Savva, and D. Batra, "Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames," in *International Conference on Learning Representations*, 2019.

[14] S. Bansal, V. Tolani, S. Gupta, J. Malik, and C. Tomlin, "Combining optimal control and learning for visual navigation in novel environments," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100. PMLR, 30 Oct–01 Nov 2020, pp. 420–429.

[15] B. Eysenbach, R. R. Salakhutdinov, and S. Levine, "Search on the replay buffer: Bridging planning and reinforcement learning," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 246–15 257.

[16] X. Meng, N. Ratliff, Y. Xiang, and D. Fox, "Scaling local control to large-scale topological navigation," *arXiv preprint arXiv:1909.12329*, 2019.

[17] S. Hochreiter and J. Schmidhuber, "Lstm can solve hard long time lag problems," in *Advances in neural information processing systems*, 1997,

pp. 473–479.

[18] E. Parisotto and R. Salakhutdinov, "Neural map: Structured memory for deep reinforcement learning," *arXiv preprint arXiv:1702.08360*, 2017.

[19] E. Beeching, J. Dibangoye, O. Simonin, and C. Wolf, "Egomap: Projective mapping and structured egocentric memory for deep rl," in *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, 2020.

[20] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 5055–5065.

[21] D. Ghosh, A. Gupta, J. Fu, A. Reddy, C. Devin, B. Eysenbach, and S. Levine, "Learning to reach goals without reinforcement learning," *arXiv preprint arXiv:1912.06088*, 2019.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[23] H. Ide and T. Kurita, "Improvement of learning for cnn with relu activation by sparse regularization," 05 2017, pp. 2684–2691.