Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Learning Behaviors for a Virtual Character

Caio Souza and Luiz Velho (supervisor)

Technical Report TR-21-04 Relatório Técnico

April - 2021 - Abril

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

TECHNICAL REPORT 2021.1

CAIO SOUZA[*], LUIZ VELHO[*]

April 9, 2021

CONTENTS

1 Introduction 1
2 Overview 2
  2.1 Modeling 2
  2.2 Timeline 3
3 Behaviors 3
4 Environment Obstacles & Collision 5
5 Training 7
6 Test Application 8
7 Future Steps 9

1 INTRODUCTION

Artificial Intelligence is a long-standing science goal, and one of the most relatable applications is intelligent agents such as virtual assistants, virtual characters, and robots. Reinforcement Learning, while not new, has been gaining momentum with Deep Learning techniques [1, 5] and is a powerful tool to learn from trial and error instead of the traditional supervised learning paradigm.

Here we intend to model a virtual dog character, called *DogBot*, in a Virtual Reality scene where a player can: call the dog, play fetch and hoop jump. For this task, both traditional animation and state machines are used together with Reinforcement learning in a hierarchical fashion. Our agent abstraction has three levels: *Task Selection* being the top layer, *Task Planning* the mid-level using Reinforcement Learning, and the last level the *Motion Synthesis* responsible for generating the animations of our character.

One key point in our choice of using the traditional animation stack for motion synthesis instead of machine learning and physical simulation is the need for control over the final animation and simplicity, as our agent is intended for interactive media. Physically-based low-level controllers such as [9, 10] are functional. However, they lack naturalness in their final animation result. Other works [4, 6, 7] achieve naturalness using data-driven approaches or more complex muscle-based simulations; it adds the cost of data acquisition and extra computational power.

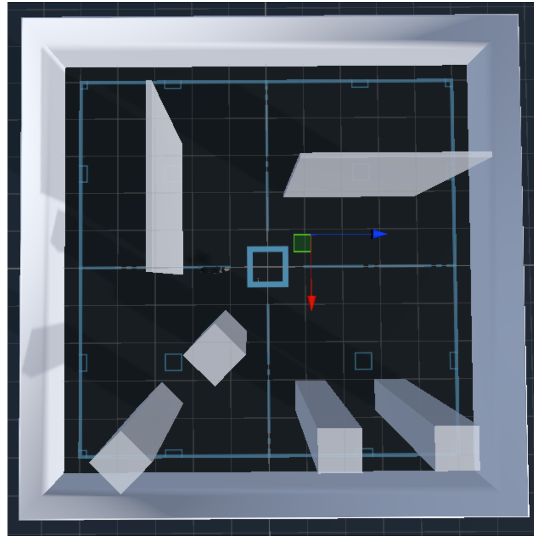
We start the following sections with an overview of our modeling and a development timeline; then, we present more details about the implemented behaviors, the environment, and training. Finally, we conclude with a test application and our intended future steps.

2 OVERVIEW

Currently, the DogBot test scene is on its 4.X version with a 6×6 square area with a few obstacles where our dog agent and the VR player can interact. Its

[*]Instituto Nacional de Matemática Pura e Aplicada, Estrada Dona Castorina 110, 22460-320 Rio de Janeiro - RJ, Brazil.

Figure 1: Top view of test scene.



top view is shown in figure 1. It is an instance of our proposed hierarchical controller modeling, where we mix reinforcement learning with traditional state machines and character controllers to simulate an intelligent agent.

2.1 *Modeling*

Our modeling aims to conceptually splits motion synthesis from the task planning. Although it is not totally independent from the motion synthesis module, the task planner does not generate animations, which brings advantages over the control of the animation's final result.

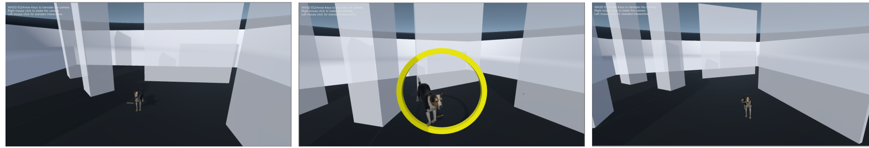
The agent animation controller comprises traditional animation clips, state-machine, and blend-trees, which are handled by the *Animator* in Unity Engine. The animation controller has a few parameters (i.e., speed, turning speed, jump power), allowing control of the output animation. These parameters are not visible by the mid-level task planner which uses reinforcement learning in a per-task fashion, and finally, the top level which switches between tasks and goals based on the user interaction.

This structure effectively makes our approach hierarchical where each level can communicate with other levels, but without knowing its intrinsics. Our framework abstraction is split into the following three levels:

- **High-Level:** The first level of control is a deterministic state machine, which takes user input and environment events to select and set up the appropriate behavior.
- **Mid-Level:** The second level of control is composed of three learned behaviors; they are *Call* (whistle to call the dog), *Fetch* (play fetch with a stick) and *Jump* (play a hoop trick with the dog). In fact, these behaviors are controllers taking input from the agent's sensor and sending lower level actions to the bottom hierarchy.
- **Low-Level:** The last level of control is responsible for the motion synthesis and encompasses the dog character animations, blend-trees, and animator. It works as a traditional character controller playing and mixing the animations, but instead of taking, for example, user input, it takes input from our learned behavior.

Despite machine learning being applied exclusively for the mid-level, it could also be used for the high-level, for example, in voice commands or for a learned low-level motion synthesis as done in works such as [10, 8, 7] and others.

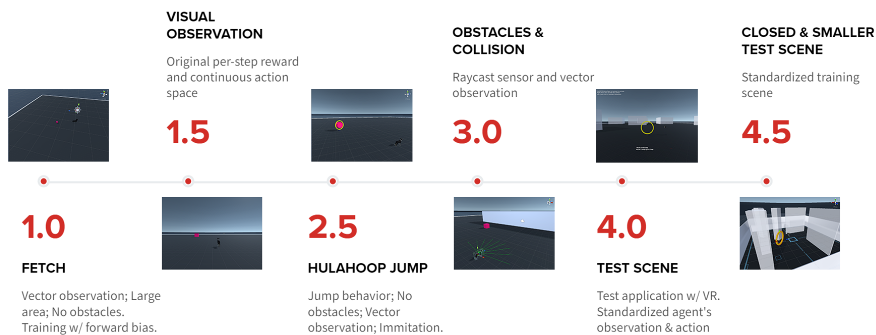
Figure 2: Player view.



2.2 *Timeline*

Our dog agent has evolved during its development, each new addition requiring different modeling or training procedures, which have converged now to a single implementation. In the figure 3 there are some critical points of our agent's development. The timeline steps express our incremental modeling, starting with a simple behavior, grading to a skilled behavior, increasing the environment complexity, and finally reaching a complete playable test scene.

Figure 3: Development timeline.



The first version experimented with aspects of the agent observations and interface between the task planning and motion synthesis. The chosen behavior was the fetch in a large empty environment. (Details and analysis of those experiments were covered in our previous work/report.) The next clear step was implementing another behavior demanding a more refined skill, the hulahoop jump, which became possible by using demonstrations to bootstrap the learning. With the two previous behaviors, it was time to work around the environment by adding obstacles, collisions and upgrading the agent senses accordingly. Finally, all pieces were set, and we could work on complete proof of concept test scene.

In the following sections, we detail the development of these steps.

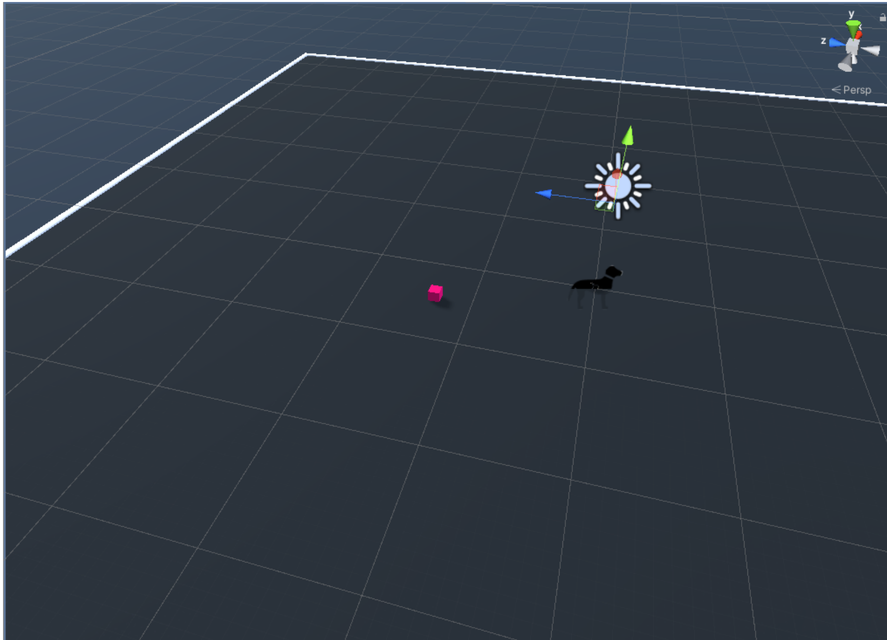
3 BEHAVIORS

Fetch

Our first implementation tackled the fetching behavior in an ample space without obstacles (as shown in figure 4). Its agent's reward modeling was similar to the Unity Puppo demo. The main difference from Puppo is that DogBot's actions have four continuous branches, Y-axis movement (forward/backward speed), rotation direction (left/right), crouch, and jump (which can co-occur), while in contrast, Puppo's actions are forces directly applied to its joints.

The training area is a square $110m \times 110m$ where a randomly collectible cube of $1m$ side spawns. A small reward is given every time step the agent moves towards the collectible, and the final reward when the agent reaches the target, also ending the episode. We experimented with hand-crafted vector features and raw pixels visual observation with both achieving

Figure 4: Fetch training scene for version 1.0



good results cite previous work. Visual observation was slightly better at generalizing, for example, performing in an environment with a variable number of collectibles after training. However, it required much more computational power during its training.

A few challenges emerged during the training; one was the inability to learn the desired behavior initially. We attribute this to the large environment size, the action branches, and small per-step reward. Starting from entirely random behavior was too tricky for the agent to gain a significant reward.

From our inspection, the agent would not commit enough time to a given action (i.e., forward vs. backward) to move and explore the training area. We opted for adding a slight bias in the Y-axis branch, which made the agent favor moving forward instead of backward. Nevertheless, other solutions could also work; a few examples are: using a smaller training area, increasing the per-step reward, or decreasing the agent decision frequency.

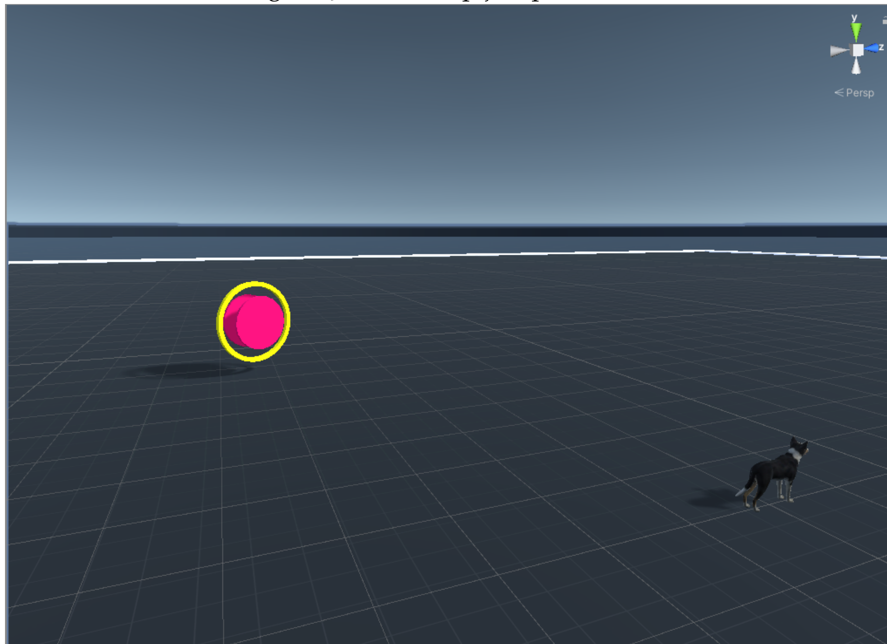
Hulahoop Jump

Hulahoop jump behavior was the next difficulty level, sharing some similarities with the previous fetch behavior but requiring more control of the actions' exact speed and timing.

For this behavior, we dropped the visual observation modeling because the computational power and time required to evolve through experiments became a bottleneck. On the agent sensing side, using the same vector observation of the fetching behavior would be somewhat deficient. Completing this task (i.e., jumping through the hulahoop) would require the agent to enter the correct angle. Hence, we add a new observation: the cosine of the angle between the agent and the hulahoop forward directions.

Sadly, those modeling modifications alone were not enough for training; the agent would not learn the task and would barely learn how to move. Despite our belief that changes to the environment, such as reducing its size, could make the task easier, these types of "facilitation" could prove problematic to extend to other more complex behaviors. The chosen solution was to bootstrap the agent's policy with demonstrations using Behavioral Cloning [11], and Generative Adversarial Imitation Learning [2]. A few minutes of recorded demonstrations (approximately 3 minutes) were enough to bootstrap the learning, which improved with later reinforcement learning; note that these demonstrations were far from perfect or covering all cases.

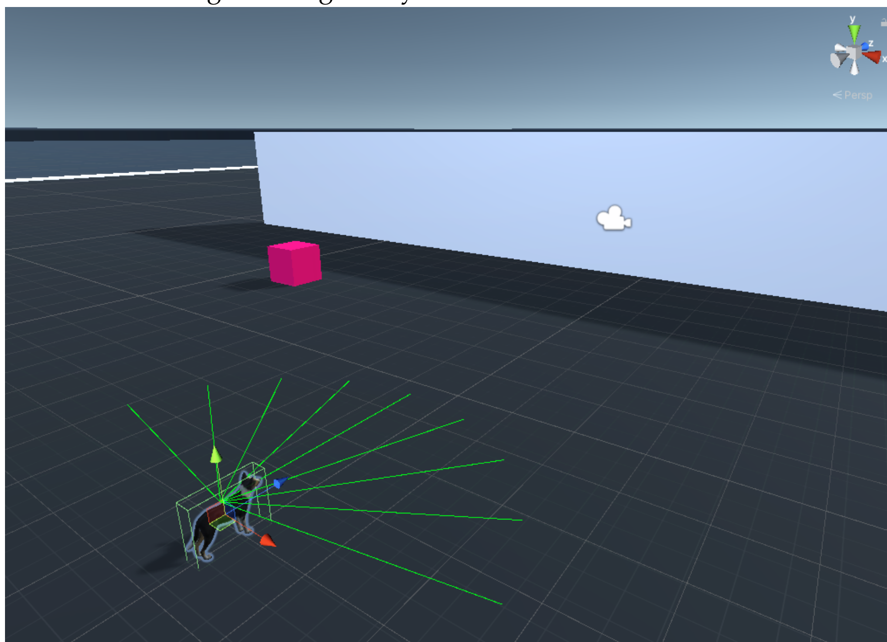
Figure 5: Hulahoop jump behavior



While bootstrapping solved the initial gap in the learning process, it did not prevent the agent from exploiting the reward system's limitations. In its first implementation, a reward was assigned when the agent touched an inner disk in the hulahoop. Given how the game engine handles the collision with this disk, the agent could exploit and get faster rewards by jumping diagonally to the hulahoop without going through it. Using two disks on each side of the hulahoop (as shown in the figure 5) improved but still had some edge cases. The final solution was to check if the jump's initial and final positions were on different sides of the plane in which the hulahoop was. This example demonstrates how the agent can exploit the environment and reward modeling shortcomings in undesired ways.

4 ENVIRONMENT OBSTACLES & COLLISION

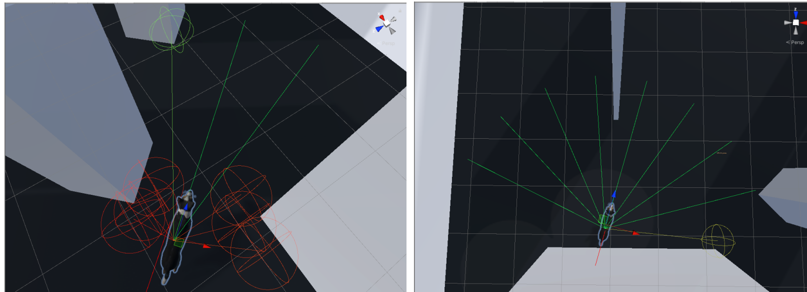
Figure 6: Agent ray-cast sensor and obstacles



In real use cases, the agent would need to perform in spaces of various sizes and deal with obstacles and cluttering, which were lack lusting in our environment. Here, we dealt with integrating obstacles into the scene and the agent's sensing. Visual observation would be an exciting approach, but it would not be computationally viable; hence we chose to use ray-cast sensors (figure 6). They act as a more straightforward and direct vision system detecting object "tags", similar to a vision system with segmentation, but computationally inexpensive compared to raw pixels input.

These sensors also have compromises, having fixed parameters such as ray length, diameter, and the number of rays; therefore, they are subject to superposition and aliasing (figure 7) depending on the distance and shape of their targets. Nevertheless, this sensing approach fits virtual environments well, easing training requirements and the final application's real-time inference.

Figure 7: Accuracy limitations of ray-cast sensor (superposition and aliasing)



The scene objects to be avoided by the agent were tagged as "wall" and would give a slight penalty when hit. Training the fetching behavior with this penalty worked very well, except for the cases where the collectible was exactly behind an acute corner; in these cases, our agent would get stuck, and its sensors would be all overlapping. Other factors besides the limitations of its sensor were responsible for this weakness. First, the penalty of hitting a "wall" was given "on hit" and did not scale with how much time the agent hit would take; and next, the lack of memory would not let the agent remember previous actions or know it was stuck. Adding a "stuck" penalty and setting stuck as an ending state significantly reduced the number of times the agent got stuck. Note that giving a penalty and resetting the agent without ending the episode was much less effective; choosing the right time to end the episode and cut off the reward propagation during the training is crucial in speeding up the learning.

Another worthy note fact that occurred while training the hulahoop jump behavior relates to assigning the "wall" tag to the ring or not; when tagged, the penalty of colliding made the learning process twice as slow. From our inspection, this penalty made the final goal harder by working in opposition to the objective, hiding the positive reward behind a penalty. Of course, in the learning process, the agent would hit the hulahoop ring more often than not, and this penalty would sometimes discourage the agent from getting close to the ring, which is a necessary step to complete the task. Such side effect of a logical but naive choice highlights the complex dynamics of the reward and environment system.

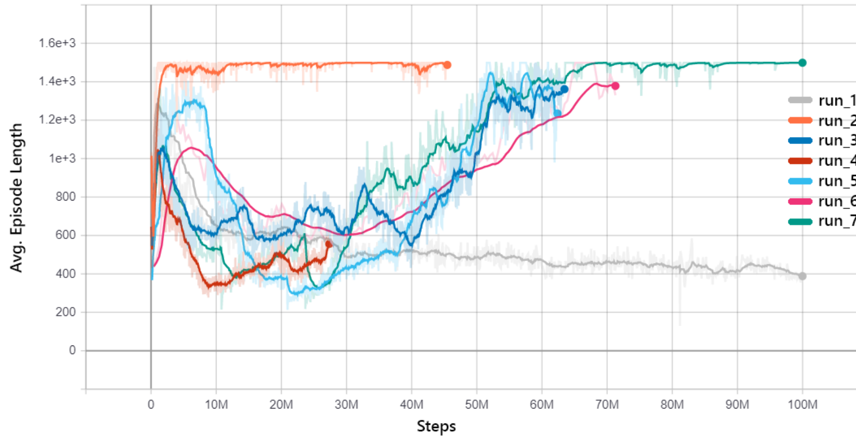
5 TRAINING

We previously described the versions and evolution of our DogBot modeling, citing training aspects only when relevant to the context. Here, we will go deeper on training choices and their impact on our agent. All training was made with ML-Agents 1.0 using the Proximal Policy Optimization algorithm [12], all other parameters and their short description are in the table 1. The only parameter varying between experiments is the *maximum episode length*,

which is different for the large and small scenes. The other parameters were tuned during development together with the training scene and reward.

Figure 8 shows the episode length for various failed training sessions of the hulahoop jump behavior. The expected outcome is a decrease in episode length during training, but it did not happen for various reasons. In the run 1, which appears to behave as expected, the agent was, in fact, not performing the task correctly but exploiting the hulahoop crossing check.

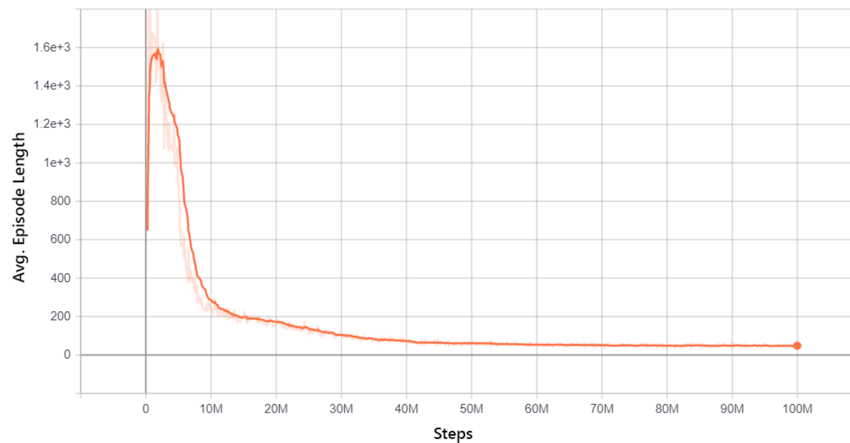
Figure 8: Average episode length failed train cases.



The other runs were steadily decreasing the average episode length until their minimum value around $10M$ to $20M$ steps (note that behavioral cloning took place until $5M$ steps). At that point, the agent would complete the task for a few more straightforward cases, but most of the time, it would get stuck or falsely activating the hulahoop cross-check. This exploit of the final reward and the unbalanced weight of moving toward the target, and the penalty of hitting the outside of the hulahoop ring lead to divergence affecting the GAIL and curiosity reward estimates.

Fixing the environment and lowering the penalty of hitting the outer ring lead to successful convergence, as shown in the figure 9.

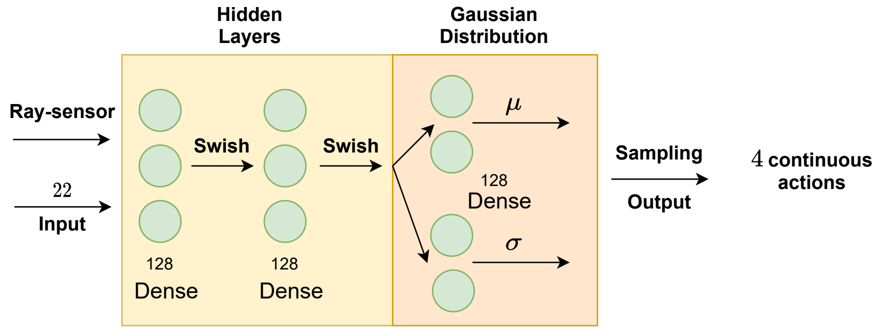
Figure 9: Average episode length of successful trained hulahoop jump behavior.



Network architecture

Our behavior policy is a simple neural network that takes the vector observations and the ray-cast sensor inputs and has four continuous outputs. Figure 10 presents a schema of ML-Agents neural network for our agent. The hidden layers are in yellow and can be configured through *num_layers* and *hidden_units* hyper-parameters.

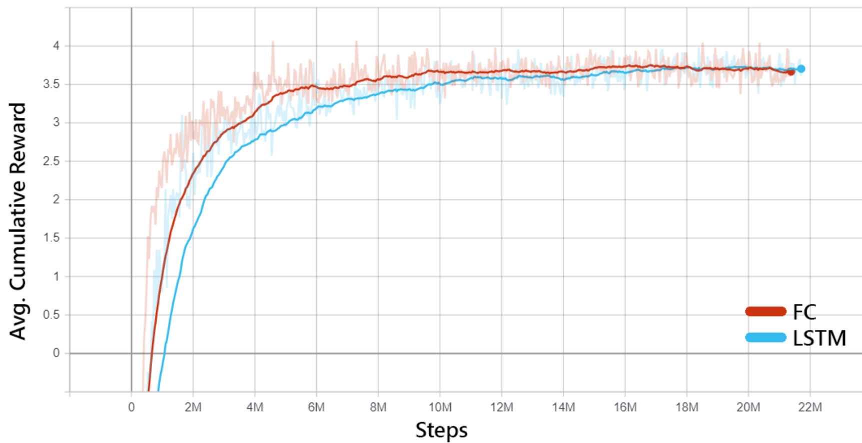
Figure 10: Neural Network layout.



One of the choices to be made on the neural network architecture was to use or not recurrent hidden units, such as Long Short-term Memory (LSTM) [3]. We conducted a simple training and testing procedure using the fetch scene to select between these architectures.

Figure 11 shows a comparison from the cumulative reward during training of traditional fully-connected (FC) versus LSTM, where both behave similarly. Running the two trained models in a test scene verifies their agreement, with scores (number of collectibles fetched during one million steps) being very close to each other (table 2).

Figure 11: Fully connected vs Long Short-Term Memory



Although we mention that memory architectures could improve some of our agent's shortcomings, such as its behaviors on acute corners, our earlier tests did not show any improvement or visual difference in the agent's behavior. Given the increased computational costs of LSTM and no clear advantage without fine-tuning and investigation into its hyper-parameters, the clear choice was to stick with fully connected units.

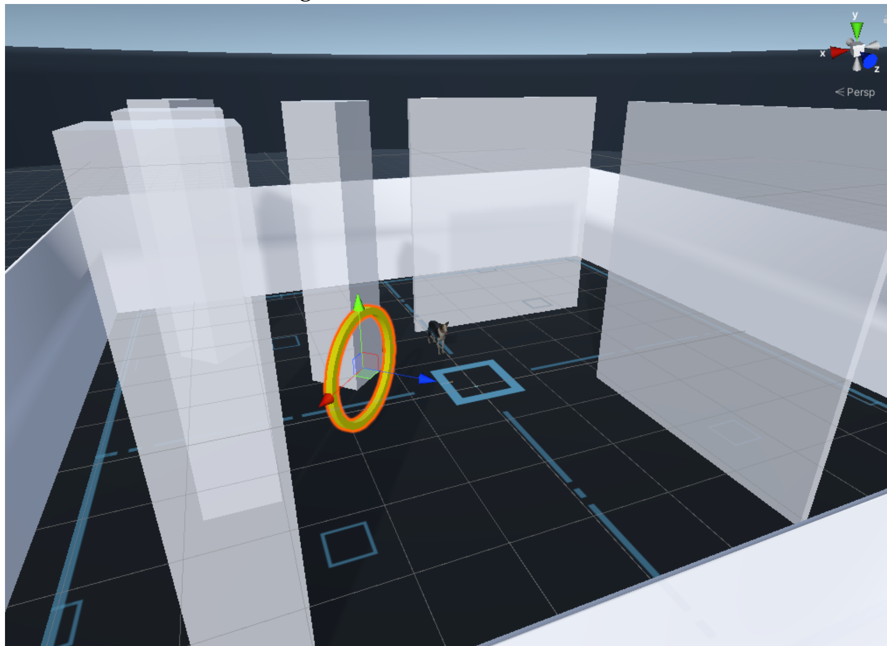
6 TEST APPLICATION

The final step for DogBot was concretely implementing all proposed hierarchical levels of control in a proof of concept test scene. This scene called for various other aspects, such as usability and user experience, and implementing a simple high-level controller.

The test scene is targeted for Virtual Reality (VR), where the player can freely move and interact with DogBot. Three behaviors, split in modes, are available:

- **Call:** The player whistle and the dog will approach the player.
- **Fetch:** The player throw a stick and the dog will fetch it.

Figure 12: Reduced test scene



- **Jump:** The player summons a hulahoop ring and the dog will jump through it.

These behaviors, with a few tweaks, worked well in the large space scene. Most of the needed changes were related to the user perception and the animation controller, such as the agent slowing down when close to the player and fine-tuning the agent's properties (i.e., maximum speed, turning speed, jump power).

In the smaller space, the agent's performance was reasonably when there were no obstacles. However, with a cluttered scene, the agent ultimately failed. This outcome was somewhat expected in the ample space since, most of the time, the ray-cast sensor would not detect anything and sloppily turn left or right when detecting anything. On the other hand, it would frequently be detecting obstacles in the smaller space; choosing the precise direction and turn amount is a must for these situations. Given that, all behaviors had to be re-trained in the smaller scene. In contrast, their learning was faster as completing the task in a reduced space would require fewer time steps, easing the credit assignment for states and actions trajectory.

Interestingly, running these behaviors back in the large scene achieved good results. The only change needed was the normalization of the agent's local position according to the new environment size. Note that our neural network already has a normalization layer, but it would not be sufficient since the training was only conducted in a small environment. In sporadic cases, the agent would not act appropriately, having difficulties contouring long walls when the objective had its position reflected in the other side of the said wall.

7 FUTURE STEPS

While our test scene is fully functional and has some fun behaviors to play with, we feel that to ultimately immerse the user and make the agent believable, it needs to develop *empathy*. Until now, all of the agent's actions are pre-defined and triggered by user commands, which is one side of the coin, with the other side being the agent's autonomous and non-deterministic actions.

Actions such as the Dog agent randomly looking at the user or wandering around wholly change the user experience and make the agent more

appealing. These aspects are not easy to quantify or reward during training because they do not interfere in the final goal but the user perception. Indeed, for robots in a production line, this characteristic is not relevant, but it is fundamental to interactive media agents.

REFERENCES

[1] Yoshua Bengio, Yann LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

[2] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Daniel Holden, Taku Komura, and Jun Saito. Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG)*, 36(4):1–13, 2017.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[6] Seunghwan Lee, Moonseok Park, Kyoungmin Lee, and Jehee Lee. Scalable muscle-actuated human simulation and control. *ACM Transactions on Graphics (TOG)*, 38(4):1–13, 2019.

[7] Hung Yu Ling, Fabio Zinno, George Cheng, and Michiel Van De Panne. Character controllers using motion vaes. *ACM Transactions on Graphics (TOG)*, 39(4):40–1, 2020.

[8] Libin Liu and Jessica Hodgins. Learning basketball dribbling skills using trajectory optimization and deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018.

[9] Josh Merel, Saran Tunyasuvunakool, Arun Ahuja, Yuval Tassa, Leonard Hasenclever, Vu Pham, Tom Erez, Greg Wayne, and Nicolas Heess. Catch & carry: reusable neural controllers for vision-guided whole-body tasks. *ACM Transactions on Graphics (TOG)*, 39(4):39–1, 2020.

[10] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):1–13, 2017.

[11] Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97, 1991.

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Table 1: Training parameters.

Name	Value	Description
batch size	128	Number of samples used for each optimization step.
buffer size	40960	Number of samples collected for each policy update.
hidden units	128	Number of neurons per hidden layer.
num layers	2	Number of hidden layers used for the model.
learning rate	3.0×10^{-4}	Initial learning rate for training.
max steps	1×10^8	Number of total simulation steps (actions) taken for training.
num epochs	3	Number of times each collected observation is used for training.
time horizon	1000	Horizon for learning, it represents how far in time steps one action can influence a past reward.
Extrinsic gamma	0.995	Discount factor for the extrinsic reward
Curiosity strength	0.02	Strength of the curiosity intrinsic reward signal.
Curiosity gamma	0.99	Discount factor for the curiosity reward.
Curiosity encoding size	256	Number of hidden units used to encode the intrinsic curiosity reward. It makes more sense to be smaller than the observation input size, but was left as default.
BC strength	0.5	Strength of behavioral cloning policy update.
BC steps	5×10^6	Number of initial steps where behavioral cloning is active.
GAIL strength	0.01	Strength of GAIL intrinsic reward.
GAIL gamma	0.99	Discount factor for the GAIL reward.
GAIL encoding size	128	Number of hidden units used to encode the intrinsic GAIL reward.
Max episode length	500-5000	Maximum number of steps until an episode ends.
Action repeat	2	Number of frames an action is repeated.
Decision frequency	30Hz	Frequency which the agent take decisions.

Table 2: Scores for FC and LSTM networks.

Network Architecture	Fully-Connected	Long Short-term Memory
Score	1171	1175