HAW HAMBURG

Physics Based Character Animation From Applied Deep Reinforcement Learning

Fynn Braren - 2238797 | Jan Friese - 2117357

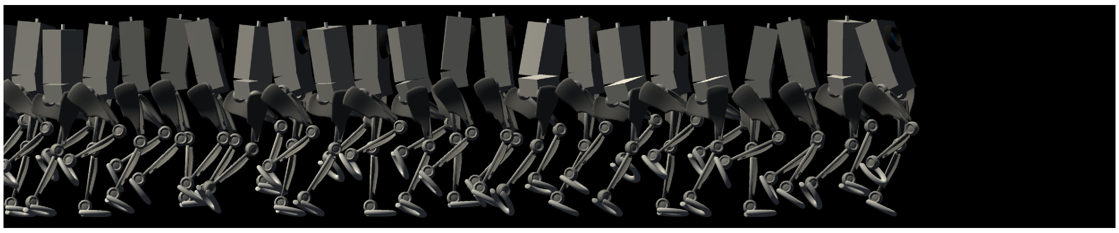
Supervised by: Prof. Dr.-Ing. Sabine Schumann
Submitted on: 28.03.2019

Fakultät Design, Medien, Information
Department Medien Technik

Faculty of Design, Media, Information
Department of Media Technology

Contents

. . . In recent years there have been several demonstrations of how deep Reinforcement Learning methods can be used to solve simulated robotic tasks like locomotion and navigation in complex environments. The ability to generate motion in response to environmental requirements is of special interest, not only in robotics research, but also in digital content production for games and movies, where character animation is an integral part. Our goal with this project is to enable a physics based character to solve various locomotion tasks, whithin a simulated learning environment using the modern game engine Unity and it's machine learning toolkit ML-Agents. To train our Agent to move, we will introduce Curriculum Learning in combination with a Balance Assistant. The balancing forces are reduced in respect to the learning progress. Instead of leveraging the Curriculum Learning implementation shipped with ML-Agents, we implement our own, which is more adapt to locomotion specific problems. Furthermore we will try to exemplary implement a system to teach the Agent multiple skills. To accomplish this we train multiple policys in conjunction within one training session. Our hypothesis is that using multiple policies and switching between them in runtime, will going to return better results than using just one policy and altering the reward function. To avoid problems with the transition between tasks, we will make use of the Balance Assistant to help the Agent with adjusting its velocity or orientation. All our results are tested on a 7-link biped with 14 degrees of freedom. This article summarizes our approach to generating procedural animation in Unity.



Keywords: Reinforcement Learning, Physics Simulation, Character Animation

1 Introduction

Traditional animation workflows in games and movies involve mostly handcrafted motionsets, adjusted to a specific character in a specific environment with a specific range of skills. Both traditional keyframe animation and motioncapture methods require a lot of work and need careful planing ahead. Developers and researchers came up with various solutions to adapt a given animation in realtime to changes in the environment, ie. Retargeting or Inverse Kinematics. Enabling a physics based character with Reinforcement Learning methods to adapt its skills to the environmental requirements results in a more flexible system. Furthermore the physics based character can give responses to perturbations from outside. Training a robust and flexible character involves some challenges, first and foremost the shaping of a good reward function. With growing complexity the reward function becomes hard to tune. To ease the learning process early on we will leverage Curriculum Learning in form of a Balance Assistant. The assistant is inspired by the virtual assistant of Yu et al [1]. Especially introducing multiple tasks can result in the policy failing to learn any of the skills adequately. There have been several approaches to implement multiple locomotion skills for one Agent. Peng et al [2] trained different policies for each required skill and were able to switch between theme in runtime using a controller input in the form of a onehot vector. We will implement a simple version of this by stripping the skillrange. The only highlevel control will be a player input axis as a speed parameter. This results in a standing and a walking skill to be learned. As our character has no arms and thus has no way of balancing its weight, other than using its legs and one upper-body link, the transition between the two skills represents one of the most challenging problems in this project. Our hypothesis is that using two different policies and switching between them in runtime is going to return better results than using just one policy and altering the reward function. Secondly we discuss our Curriculum Learning and evaluate in what way the training benefitted from it. We will first give a short introduction in what has already been achieved in the sphere of procedural character animation with reinforcement learning and than show how we approached clarifying our hypothesis. We took an iterative approach on developing this project. The overall timeframe was set to one month and as this is not a lot of time, we had to carefully plan ahead. With the supplementory material at the end of this article we present the accompanying repsitory and some of the resources used in developement.

2 Related Work

The field of physically based Character Control through Reinforcement Learning gained a lot of interest from various research fields in recent years. 3D locomotion of physics based characters became a standart benchmark for Reinforcement Learning algorithms. Balance and gait control of bipedal or quadruped characters presents a complex continuous control task. As a result there are various frameworks, such as the DeepMind Control Suite [3] or the OpenAI Gym [4] that serve as benchmarks for the ongoing developement of Deep Reinforcement Learning algorithms. These frameworks usually emphasize the robustness and performance of the algorithm and not of the outcoming motion. The reward functions used in the benchmarks are simple and not designed to match realistic gaits. In the recent years, with growing interest from the computer graphics researchers, there have been several approaches to apply DRL algorithms with a focus on realism and applicability for games. One of the most advanced research projects in terms of motion realism is certainly DeepMimic [2] from Peng et al. The realism of the outcoming motion in their work is due to Motioncapture animations presented to the learner. Even though their work targets the use for virtual characters in games and movies, they did not implement it in a Game Engine but in the Bullet Engine, which includes a very accurate Physics System. A great source of inspiration for this project has been the Marathon Environments, that re-implements the classic set of Continuous Control benchmarks typically seen in Deep Reinforcement Learning literature as Unity environments using the ML-Agents toolkit. [5] Another source of inspiration was the work from Yu et al [1], for it is their approach to Curriculum Learning that inspired us to implement the virtual assistant. In general the Machine Learning community is very open and often the results and projects are accessible on Github or aquivalent platforms.

3 Background

This section is dedicated to the technology and terminology involved in this project.

3.1 Deep Reinforcement Learning

Reinorcement Learning (RL) is a Machine Learning method used in sequential decision making tasks or in more detail the task of deciding, from experience, the sequence of actions to perform in an uncertain environment in order to achieve some goals[6]. It can be formalized as an Agent that has to make decisions in an environment to optimize a given notion of cumulative rewards. See figure 1 for a basic overview of the learning process. The Agent is always in one of many possible **states** ($s \in S$) and can take one of many **actions** ($a \in A$), that in turn influence it's state. For every action a **reward** $r \in \mathbb{R}$ is returned by the environment. A **policy** $\pi(s)$ can be trained to return the action that results in the highest cumulative reward for a given state. Following the policy results in a so called **trajectory**, a chain of states, actions and rewards. The Reinforcement Learning process can be modeled as a **Markov Decision Process** (MDP)[7]. See the paragraph Policy Learning for more information on the MDP. **Deep** Reinforcement Learning simply describes the combination of Deep Learning and Reinforcement Learning and brings huge benefits over traditional Reinforcement Learning methods in high dimensional state-spaces.

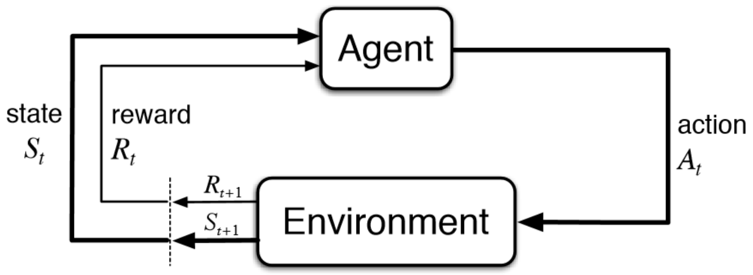


Figure 1: Agent - Environment interaction [7]

3.2 Policy Learning

The following section is dedicated to how we modeled the locomotion learning process. As described above it can be modeled as a Markov Decision Process. Our Learning process is thus represented by the following 6-tuple: $(S, A, r, p_0, P, \gamma)$, where S is the state, A is the action space, $r : S \times A \to \mathbb{R}$ is the reward function, p_0 is the initial state distribution, $P : S \times A \to S$ is the trainsition function and γ is the discount factor $\gamma \in (0, 1)$. The goal now is to find the parameters θ of a policy $\pi_\theta : S \times A \to \mathbb{R}$, that maximizes the expected long-term reward[1]:

$$\pi_\theta * = \underset{\theta}{\text{argmax}}\ \mathbb{E}_{s \sim p_0}[V^\pi(s)], \tag{1}$$

where the value function of a policy, $V^\pi : S \to \mathbb{R}$, is defined as the expected long-term reward of following the policy π_θ from some input state s_t:

$$V^\pi(S_t) = \mathbb{E}_{a_t \sim \pi, s_{t+1} \sim P, \ldots}[\sum_{i=0}^{\infty} \gamma^i r(s_{t+i}, a_{t+i})]. \tag{2}$$

3.3 Curriculum Learning

A **Curriculum Learning** (CL) strategy can optionally be used and may enhance the Reiforcement Learning Process. It presents the agent with a simple version of what is to be learned and gradually increases the complexity of the task. It has been shown that Curriculum Learning can speed up the learning process and positively influence the learned results [8]. See the Curriculum Learning section for details on how we implemented CL.

3.4 Unity Platform

Unity is a game development platform that consists of a **Game Engine** and a graphical user interface called the **Unity Editor**. The flexibility of the underlying engine meets a wide range of requirements, from complex physical simulation to simple 2D games.

Unity Terminology Unity projects consist of one or more **Scenes**, which can be thought of as the virtual environment or level in a game or simulation. A scene usually includes **GameObjects**, which represent the logical and graphical objects and components. The scene is also referred to as the learning environment in this article. Shipped with Unity comes a bunch of built-in GameObjects, such as Cameras, Renderers or Rigidbodies. All scenes, objects and components together are also referred to as the projects **Assets**.

3.5 Unity Machine Learning Agents

The ML-Agents toolkit leverages the open-source library **TensorFlow** and extends the Unity Engine by a machine learning framework. The toolkit enables the development of learning environments which are rich in sensory and physical complexity and support dynamic multi-agent interaction [9]. Included within the ML-Agents toolkit are a variety

of deep Reinforcement Learning algorithms and training scenarious. The relevant Tensorflow logic is abstracted away in the external python code. As we do not implement a new Learning algorithm we do not touch upon the Tensorflow implementation of the ML-Agents in this article.
The ML-Agents toolkit consists of three main components[10]:

- **The Learning Environment**, contains all the unity Assets.
- **The Python API**, which contains all the machine learning algorithms that are used for training (learning a behavior or policy). The API lives outside of Untiy and communicates with the environment through the External Communicator.
- **The External Communicator**, which connects the Learning Environment with the Python API. It lives within the Learning Environment.

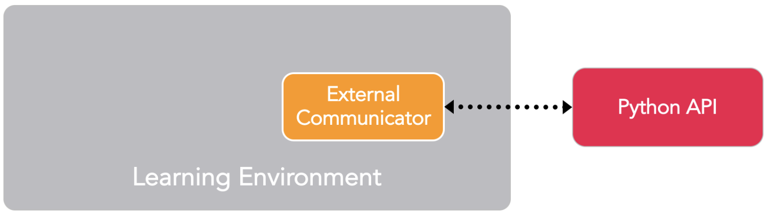


Figure 2: ML-Agents External Communication [10]

The Learning Environment contains three additional components that help organize the Unity scene [10]:

- **Agent** - is the component that, attached to a GameObject in the scene handles observations and performs the actions received by the communicator. From the Agent a reward signal is returned to exactly one connected Brain. It follows the observation-decision-action-reward cycle.
- **Brain** - is the component that receives the observations and rewards from the Agent and returns an action. In essence, the Brain is what holds on to the policy for each Agent and determines which actions the Agent should take at each instance. There are three types of brains. A **Learning Brain**, that receives the Agents observations and returns it's actions. It either holds a fully trained Tensorflow model, or communicates with the Python API, for the decision making. A **Player Brain**, that takes discrete or continuous controller input and passes it to the Agent.

A **Heuristic Brain**, that requires a decision component that holds the decision making logic.

- **Academy** - which orchestrates the observation and decision making process. The External Communicator lives within the Academy.

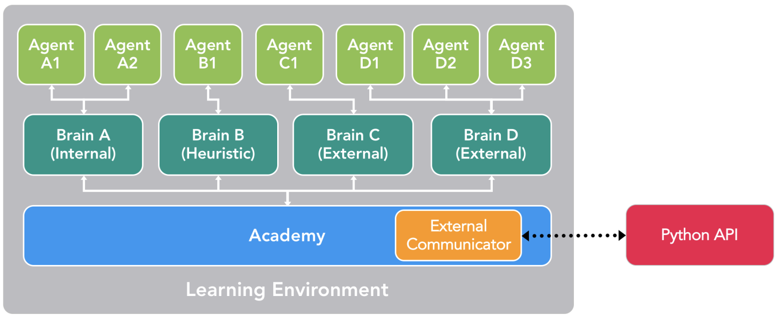


Figure 3: ML-Agents Multi Agents [10]

Training The ML-Agents toolkit supports both a training mode and an inference mode. In training mode the environment requires the external Python API to send the Agents Actions, while in inference mode a Tensorflow model is required. The Tensorflow model needs to be embedded within a Learning Brain. With every completed Training run a Tensorflow model is saved. See the Methods section for details on the Training setup and Action values.

3.6 Physically based Character Control

Interactive character animation using simulated physics, consists of the following three fundamental components[11]:

- **A physics simulator**, which is the heart of any physics based animation system and is responsible for generating the animation, by enforcing physical laws of motion. For this project the Agent itself outputs the appropriate actions from the learned policy.
- **A physics-based character**, which is the actor in the physics simulator. It contains several physics properties. Learn more about how our character is implemented in the Physically Based Character section

- **A motion controller**, which can be regarded as the brain of a physics based character: it attempts to compute the forces and torques required to perform high-level tasks. The policy itself can be regarded as the motion controler.

4 Developement

As stated in the beginning we took an iterative approach on developement. This includes that we had several milestones set in the beginning, which we tried to achieve one by one. As we moved along we discovered better ways of solving already implemented mechanisms and reimplemented them. As expected, most of the time went into shaping and improving the reward function. All relevant settings are set in front of a training session.

5 Methods

All our implemented Mechanics are built with extensibility in mind. It is relatively easy to extend the skillrange, change the curriculum values, interchange the Learning Algorithm, alter the reward function or even switch the character. The following subsections are semantically ordered to give an overview on our individual implementations.

5.1 Environment

The Unity Environment holds a flat plane with a Physics Material attatched to it, which features a slightly higher friction. The scene holds all the relevant objects in a hierarchy. See BILD UNITY INSPECTOR ML-Agents support training with multiple Agents and doing so can significantly speed up the learning process. Each Agent will make its own observations and act independently, but will use the same decision-making logic. Placing too many Agents in a scene can slow down computation. We found that using about 8 Agents yields the best results for our specific setup. Most of the relevant settings are to be set on the Academy object, as it also holds the global Curriculum Controller. See more to the Curriculum in the section Curriculum Learning.

5.2 The Locomotion Agent

The ML-Agents toolkit includes the abstract **Agent** component. It is to be inherited by a custom Agent in order to be implemented with an Agent object in the Environment. We introduce two classes that are holding all the Agents logic:

- **RobotAgent** inherits directly from the Agent component and holds the relevant logic for the Agents observation-decision-action-reward cycle as well as a bunch of Uitility functions involved in calculating the reward signal. See the Reward Utilities section in the Supplementary Materials for a full list of the values that can be used for reward calculation. Additionaly the RobotAgent introcuces early termination of a rollout when the Agent acts after certain criteria (e.g. if the Agent's hip falls below a certain height or a non-foot bodypart touches the ground)
- **MultiSkillAgent** inherits from the RobotAgent class and overrides some of it's methods in order to include the Curriculum Learning and Reward functions for the final Agent.

5.2.1 State Observations

We define a state as $\mathbf{s} = [\mathbf{q}, \dot{\mathbf{q}}, \hat{\mathbf{q}}, \alpha, \bar{\mathbf{v}}, \hat{\mathbf{v}}, \mathbf{G}]$ where $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\hat{\mathbf{q}}$ are the **position**, **velocity** and **angular velocity** of each joint. Additionaly a **normalized rotation** value α in respect to the bodyparts motion range is observed. $\mathbf{v}$ is the current velocity (average over the last 10 simulation steps) and $\hat{\mathbf{v}}$ is the target velocity that is to be matched. $\mathbf{G}$ is a binary vector indicating if the respective joint is in **contact** with the ground. All observation vectors together sum to a total of 92 observations for each Agent.

5.2.2 Reward Function

With the introduction of multiple skills, we tested various reward functions. In this section we will cover the computation of the reward for the walking skill in detail. The reward signal is shaped from the following objectives: move forward, balance, maintain a natural appearance. A reward is computeted on every agent step. This results in the following function:

$$r(s,a) = E_v(s) + E_u(s) + E_p(s) + E_t(s) - D_{limb}(s) - D_a(s) - D_l(s) - D_h(s) \tag{3}$$

where $E_v(s) = -|\bar{v}(s) - \hat{v}|$ encourages the Agent to match the desired velocity. The term $E_u(s)$ encourages the Agent stay upright with its hip and upperbody. $E_p(s)$ and $D_limb(s)$ encourage the Agent to keep a natural gait: setting one feet after the other, while keeping the legs in asyncronuous motion. $E_t(s)$ represents a very small value growing over time. It was introduced with the multiskill implementation to incentivate the Agent to stay alive longer. The term $D_a(s)$ penalizes the Agent for high joint actuation, or in other words it discourages a high effort. This is to avoid shaky movements. The term $D_l(s)$ penalizes the Agent for moving it's joints to the angle limit to avoid unnatural motions. $D_h(s)$ finally penalizes the Agent for letting it's hips fall below a certain height threshold. An extra penalty D_T is send whenever the Agent is terminated before the episode ends.

```
//The step reward for the walk policy
_reward = (
        _velocityReward
    + _uprightBonus
    + _forwardBonus
    + _finalPhaseBonus
    + _timeAliveBonus
    - _limbPenalty
    - _effortPenalty
    - _jointsAtLimitPenalty
    - _heightPenalty
    );
```

```
//The step reward for the Stand policy
_reward = (
    + _uprightBonus
    + _forwardBonus
    + _timeAliveBonus
    + _velocityReward
    - _effortPenalty
    - _heightPenalty
    );
```

5.2.3 Actions

Actions are send from the external communicator to the respective Brain component and from there directly observed by the RobotAgent on every decision step. The actions are presented in form of a float Vector. Each value has a range of $(-1, 1)$. The action values are send to the single BodyParts and mapped to the range $(0, 1)$ to be applied to the Joints.

5.3 Physically Based Character

We gained a lot of inspiration by looking at other implementations of Physics Based Characters in Unity. First and foremost the Walker environment of the ML-Agents example scenes served as a template for our implementation of the Character and it's Motor controls. Similar to them we used Unity **Configurable Joints** (CJ) for setting the motion-type and range of each link. The CJ is a highly customizable components that allows for a lot of different motion and control types[12]. Each CJ requires a Rigidbody. The **Rigidbody** is a Unity Component that helps with Physics simulation[13]. It can be attatched to objects in a scene and allows altering and monitoring of Physics related settings, such as mass or velocity. Through the Rigidbody all forces and torques send by the CJ are applied on the Characters limbs. Inspired by the Walker environment every limb has a **BodyPart** component attatched to it which dircetly recieves the torques from the RobotAgent, in the form of $a \in (0, 1)$. The value a is in turn mapped to the angle α mapped between the min and max range of each limb. α is than sent to the CJ as the new targetrotation. To manage and monitor the current forces of the character, we introduced the **JointDriveController** (JDC) component. The JDC serves basically as an interface for the RobotAgent to collect the individual observations. Maximum allowed force values are set in before training. As stated in the Challenges section we had trouble finding the Physics settings, that match natural looks, while preserving the stabilizing abilities of the agent. There was a lot of trial and error involved with setting the correct forces and masses of the bodyparts.

5.4 Curriculum Learning

Inspired by the work of Yu et al [1] we use a **Balance Assistant**, that operates with propelling and stabalizing forces on the Agent in early learning stages. This ensures that the Agent gets an idea of what is to be achieved in the environment, rather than slowly struggling to find it's way to the optimal policy. It is important to ensure that the curriculum does not assist the Agent for two long, otherwise there is the possibility of the Agent overfitting to one curriculum lesson. Although the ML-Agents toolkit is shipped with a Curriculum Learning implementation, we had to implement our own in

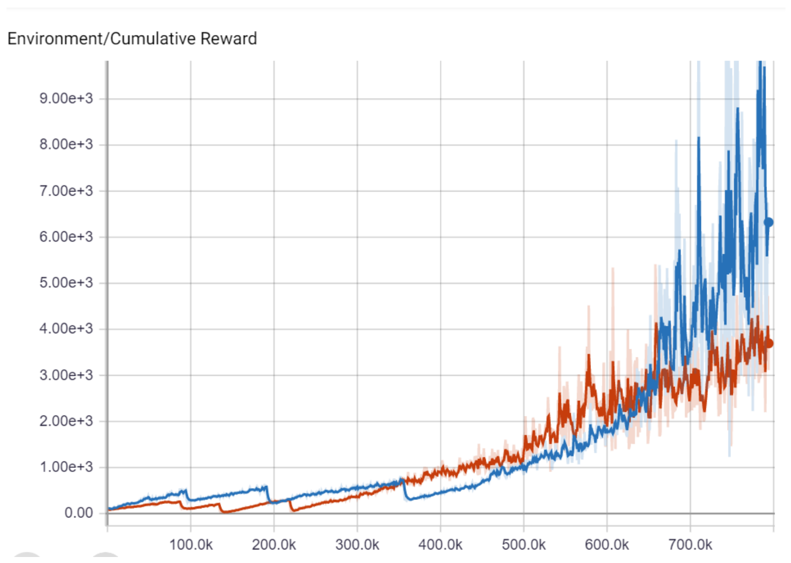


Figure 4: Learning multiple skills with Curriculum active

order to achieve our goals. As the ML-Agents CL implementation lives outside of the Unity Engine it's only measure of learning progress is the overall cumulative reward or the relation between learning steps taken and maximum learning steps (See [14]). While this approach might be sufficient for some environments, one needs a good notion of how far the cumulative reward will rise in respect to the learning progress. With a complex continuous environment like ours this is next to impossible. It makes much more sensce to iterate the CL lessons based on the time the Agent manages to stabilize itself or the distance it travels with the given policy. We used a mix of those values to update the current lesson. We introduced a **MileStone** system, which is dependent on the time t an Agent stays alive. Whenever a Milestone T is reached the assistant forces get reduced, while a **Milestone Counter** k serves as a threshold to be reached in order to update the lesson. If the Agent manages to stay alive for ($t >= zT$) the curriculum gets updatet. Furthermore the Agents cumulative reward R is saved for the given rollout as the **Expertreward** eR, when the Curriculum is updatet. This serves as a comparison to the last successfull policy. The Agents reward R has to reach a certain percentage L of the last expertReward eR. The curriculum is thus updatet when ($R >= L * eR$). The final implementation of our curriculum learning tool consists of the following objects.

- **Curriculum Object**, which is the heart of the curriculum implementation as it holds all the relevant values for the current lesson and the virtual controller

- **Global Curriculum Controller**, which is attatched to the Academy object in the learning environment. The global controller is responsible for keeping the local controllers in sync with each other. See Figure 5 for a view of the editor
- **Agent Curriculum Controller** The Agent controller holds the relevant logic for iterating the curriculum lesson. It is attatched to a single Agent object and receives the curriculum relevant informations from the RobotAgent component. It is responsible for communicating the curriculum values to the Balance Assistant.
- **Balance Assistant** The Balance Assistant applies the balancing, propelling and breaking forces, which it received by the Curriculum Controller.
- **Curriculum Monitor** To monitor the lessonprogression internally we wrote the monitor, which outputs a CSV table that holds: updated Skill, updated Lesson, StepCount at update

5.5 Multiskill Learning

After successfully training working versions for both the standing and the walking policies with Curriculum Learning we started working on a version that should be able to perform multiple motionskills. We had high expectations at the results from just switching the reward function on runtime, in response to external controls and train a single policy which is than able to perform a skill in response to an observed signal. Our next approach was dedicated to training multiple policies, each presented with it's own task and train them in conjunction. There have been several projects where such an approach lead to success, a mentioned example beeing DeepMimic [2]. Additionally we stripped the skill range to a standing and a walking skill, to make the transition comprehensible for the policy. The Actions, Observations and Reward Functions used to train each skill can be observed in the following subsections. To switch between skills we implemented an Inputcontroller, which consists of the following components:

- **ControllerAgent**, which extends the Agent class from the ML-Agents. It holds either a Player Brain or a Heuristic Brain, from which it receives the relevant input signals. This approach has several benefits above implementing the logic from scratch. First and foremost the interchanging of an Agent's brain is easily done, on and off runtime.

- **DecisionHeuristic** The decision component required by the heuristic brain that holds the decision switching skills. Specifically it applies a certain skill with a probabillity.

As we did not know if we are going to succees with multiskill Training we implemented the **ControllerAgent** with the following setups possible:

- training all skills in conjunction from the start. This is the normal approach to teaching multiple skills.
- training one or multiple skills to a certain stepCount to ensure a learning progress, before starting multiskill training.
- training just one skill at a time by setting the stepCount of one skill to the maximum stepCount for the current training session

As the project is kept extensible, it is no problem to implement another skill, by adding a third reward function and a respective brain.

6 Proximal Policy Optimization

In this project we made use of the Proximal Policy Optimization Learning algorithm (PPO), which belongs to the Policy Gradient Learning Methods. PPO has shown great performance and stability in various tasks and environments[15]. As it is one of the shipped algorithms coming with ML-Agents we could implement it without any trouble. The implementation is mostly abstracted away in the python API and the important hyperparameters are set in a configuration file that lives outside of the environment. We used roughly the same hyperparameters for all our training sessions, being inspired by the Marathon Environments [16]. PPO belongs to the policy gradient methods and thus defines an advantage function as $A^{\pi}(s_t, a) = Q^{\pi}(s, a) - V^{\pi}(s)$, where Q^{π} is the stateaction value function that evaluates the return of taking action a at state s and following the policy π_{θ} thereafter. We do not dive deeper into the implementation of PPO as it is beyond the scope of this project.

Hyperparameters The PPO implementation of ML-Agents requires setting a number of hyperparameters. The outcome of training sessions rely to a large part on these parameters. the Table 1. shows our chosen parameters.

Table 1: Hyperparameters

Part		
Parameter	Description	Value
Gamma	discount factor for future rewards	0.995
Lambda	used when calculating the advantage estimate	0.95
Buffer Size	experiences collected before updating the model	20480
Batch Size	experiences collected for updating gradient descent	2048
Number Epochs	passes through the experience buffer during gradient descent	3
Learning Rate	the strength of each gradient descent update step	$3.0e-4$
Time Horizon	maximum experience steps per-agent	1000
Max Steps	simulationsteps run during the training process	$1e6$
Beta	strength of the entropy regularization	$5.0e-3$
Epsilon	threshold of divergence in gradient descent updating	0.2
Normalize	whether vector observation are normalized	True
Number Layers	hidden layers the observation input	3
Hidden Units	units in each fully connected layer of the neural network	512
Recurrent	wether neural network is recurrent	False
Curiosity	wether to use curiosity	False

7 Challenges

To achieve natural looking animation with our systems, we had to get a realistic approximation of natural (real-world) physics. The physics engines in game engines are optimized for stability and performance at the expense of realism [17]. We had to work around that issue, gathering inspiration by other physics based character implementations, like the Marathon Environments [16] or the ML-Agents Example Environments [18]. See more on our Character implemetation in the Methods section. The flexibility of the Unity engine and our implementations is a huge benefit, when it comes to rapid prototyping, but it also introduced some challenges, e.g. when the Agent did not learn as expected, we had to be very accurate with tweaking all the possible values, ranging from simply the reward function or the Curriculum thresholds, to changing the Hyperparameters of the Learning algorithm or introducing new observations to the Agent in the learnign environment. Sometimes this lead to setbacks, where a first successfully trained policy suddenly and unexpected did not work anymore.
One of the most challenging problems with this project was to implement the learning of multiple skills. The first challenge being that we had to alter the architectur of the already designed framework. In this process were a few delicate design decisions involved

that could only be sorted out by implementing and testing. One of the crucial design decisions was the question of wether the Agents in the scene should be presented with learning the same skill at the same time or individual skills. In other words, should we relie on global input control or should each Agent have its own controller. As were not sure if one option or the other would benefit or prevent learning we implemented both and found that there is not destinction.
Another open question was, if the unconstrained way of activating the next skill would'nt leed to too much noise for the Agent to learn a transition. Would it be more comprehensible for the Agent if we would constrain the activation of the next skill to a specific bodypart orientation. For instance, the standing skill would be only than activated, when the character places it's left heel. This is a process used a lot in traditional video game animation, called Motion Matching [19].

8 Results

We implemented a framework for teaching Physically Based Characters to locomote in a simulated environment using the Unity Gameengine. We leveraged the ML-Agents toolkit and successfully trained a policy to balance a bipedal character in running forward. After succeeding in the simple task of learning one skill, we extended the skillrange to standing and walking to be learned in conjunction. This presented a bigger challenge than expected. We managed to train multiple skills to a single Agent, by training multiple policys in conjunction. One policy is trained for each skill, while switching between the skills is handled in runtime and thereby each policy successfully learns to comprehend for the unrelated perturbations from the last policy. We implemented a Curriculum Learning algorithm that gradually introduces the Agent into learning to balance itself. The algorithm helps both in learning a single skill as well as in learning the skill transitions. One of the shortcommings of the project in the current version is the motion, that is still shaky and unnatural in appearance.

9 Outlook

With the framework in place that is adjustable and extensible the project can take many directions in future developement. More skills, a new Agent with more limbs or even a different learning algorithm could be introduced. As the project is implemented with the

Unity Gameengine, it would be a natural conclusion to exploit the functionality in an actual game. There are a number of things that can be improved in the current system. One of them being the transition between the skills. To enhance the transition motion a form of motion matching could be implemented. This could both result in a better learning performance, because the starting state would not be as noisy for the learning agent, as well as in a more natural appearance, because the transition could be supervised by a human.

References

[1] Wenhao Yu, Greg Turk, and C. Karen Liu. Learning symmetry and low-energy locomotion. *CoRR*, abs/1801.08093, 2018.

[2] Xue Bin Peng, Pieter Abbeel, Sergey Levine, and Michiel van de Panne. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *CoRR*, abs/1804.02717, 2018.

[3] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, Timothy P. Lillicrap, and Martin A. Riedmiller. Deepmind control suite. *CoRR*, abs/1801.00690, 2018.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[5] Joe Booth and Jackson Booth. Marathon environments: Multi-agent continuous control benchmarks in a modern video game engine. 2019.

[6] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM.

[9] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018.

[10] Unity Technology. Ml-agents documentation overview. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md. Accessed: 2019-03-25.

[11] T. Geijtenbeek and N. Pronost. Interactive character animation using simulated physics: A state-of-the-art review. *Comput. Graph. Forum*, 31(8):2492–2515, December 2012.

[12] Unity Technology. Unity configurable joint. https://docs.unity3d.com/Manual/class-ConfigurableJoint.html. Accessed: 2019-03-25.

[13] Unity Technology. Unity rigidbody. https://docs.unity3d.com/ScriptReference/Rigidbody.html. Accessed: 2019-03-25.

[14] Unity Technology. Ml-agents documentation overview. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Curriculum-Learning.md. Accessed: 2019-03-25.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[16] Joe Booth Unity Technology. Marathon environments. hhttps://github.com/Unity-Technologies/marathon-envs. Accessed: 2019-03-25.

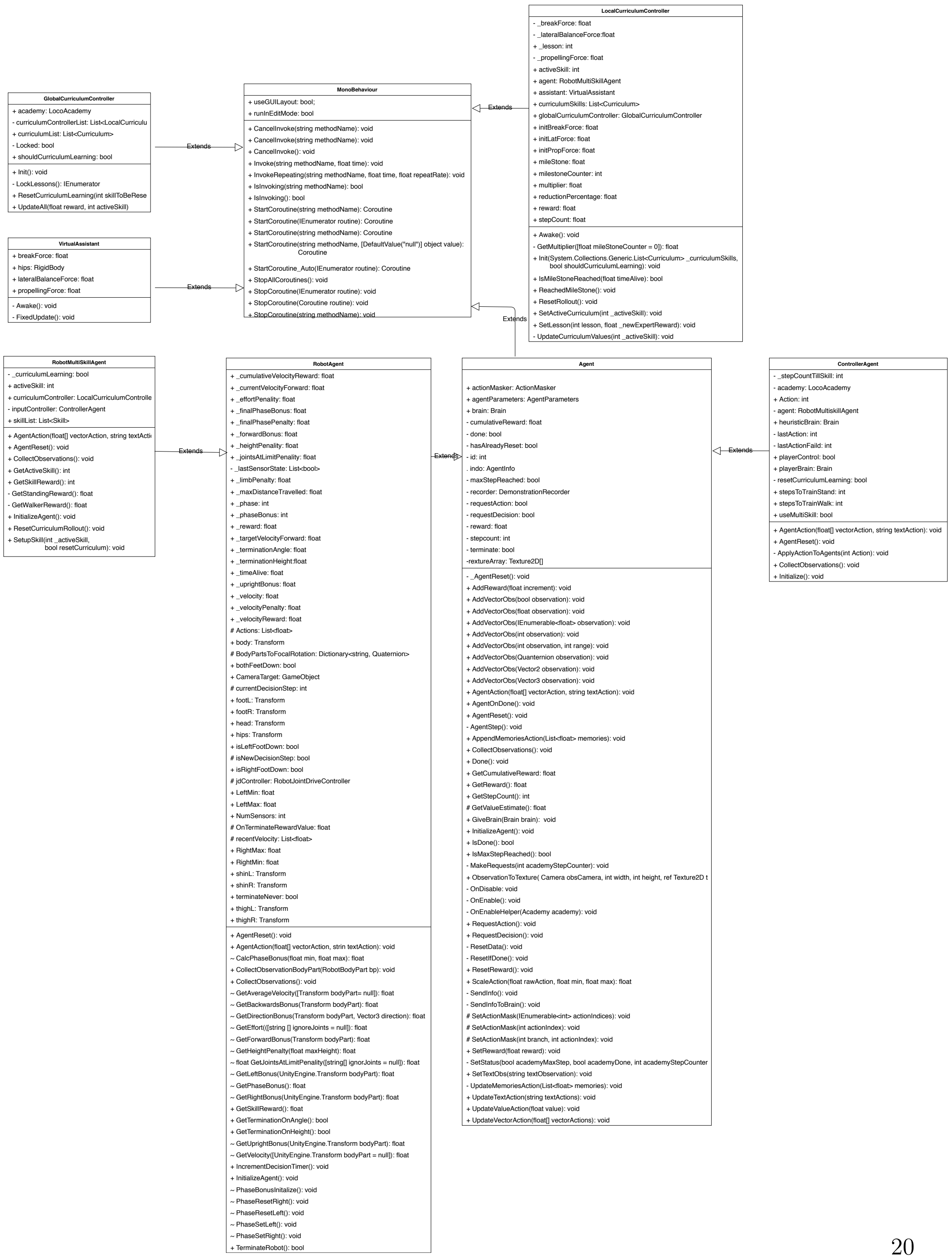
[17] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ODE and physx. In *ICRA*, pages 4397–4404. IEEE, 2015.

[18] Unity Technology. Ml-agents environments. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md. Accessed: 2019-03-25.

[19] Joakim Hagdahl. Realtime animation tips & tricks. http://rockhamstercode.tumblr.com/post/178388643253/motion-matching. Accessed: 2019-03-28.

10 Supplementary Material

10.1 UML Diagram


LocalCurriculumController
- _breakForce: float
- _lateralBalanceForce:float
+ _lesson: int
- _propellingForce: float
+ activeSkill: int
+ agent: RobotMultiSkillAgent
+ assistant: VirtualAssistant
+ curriculumSkills: List<Curriculum>
+ globalCurriculumController: GlobalCurriculumController
+ initBreakForce: float
+ initLatForce: float
+ initPropForce: float
+ mileStone: float
+ milestoneCounter: int
+ multiplier: float
+ reductionPercentage: float
+ reward: float
+ stepCount: float
+ Awake(): void
- GetMultiplier([float mileStoneCounter = 0]): float
+ Init(System.Collections.Generic.List<Curriculum> _curriculumSkills, bool shouldCurriculumLearning): void
+ IsMileStoneReached(float timeAlive): bool
+ ReachedMileStone(): void
+ ResetRollout(): void
+ SetActiveCurriculum(int _activeSkill): void
+ SetLesson(int lesson, float _newExpertReward): void
- UpdateCurriculumValues(int _activeSkill): void
GlobalCurriculumController
+ academy: LocoAcademy
- curriculumControllerList: List<LocalCurriculu
+ curriculumList: List<Curriculum>
- Locked: bool
+ shouldCurriculumLearning: bool
+ Init(): void
- LockLessons(): IEnumerator
+ ResetCurriculumLearning(int skillToBeRese
+ UpdateAll(float reward, int activeSkill)
MonoBehaviour
+ useGUILayout: bool;
+ runInEditMode: bool
+ CancelInvoke(string methodName): void
+ CancelInvoke(string methodName): void
+ CancelInvoke(): void
+ Invoke(string methodName, float time): void
+ InvokeRepeating(string methodName, float time, float repeatRate): void
+ IsInvoking(string methodName): bool
+ IsInvoking(): bool
+ StartCoroutine(string methodName): Coroutine
+ StartCoroutine(IEnumerator routine): Coroutine
+ StartCoroutine(string methodName): Coroutine
+ StartCoroutine(string methodName, [DefaultValue("null")] object value): Coroutine
+ StartCoroutine_Auto(IEnumerator routine): Coroutine
+ StopAllCoroutines(): void
+ StopCoroutine(IEnumerator routine): void
+ StopCoroutine(Coroutine routine): void
+ StopCoroutine(string methodName): void
Extends
Extends
Extends
Extends
VirtualAssistant
+ breakForce: float
+ hips: RigidBody
+ lateralBalanceForce: float
+ propellingForce: float
- Awake(): void
- FixedUpdate(): void
RobotMultiSkillAgent
- _curriculumLearning: bool
+ activeSkill: int
+ curriculumController: LocalCurriculumControlle
- inputController: ControllerAgent
+ skillList: List<Skill>
+ AgentAction(float[] vectorAction, string textActi
+ AgentReset(): void
+ CollectObservations(): void
+ GetActiveSkill(): int
+ GetSkillReward(): int
- GetStandingReward(): float
- GetWalkerReward(): float
+ InitializeAgent(): void
+ ResetCurriculumRollout(): void
+ SetupSkill(int _activeSkill, bool resetCurriculum): void
Extends
RobotAgent
+ _cumulativeVelocityReward: float
+ _currentVelocityForward: float
+ _effortPenality: float
+ _finalPhaseBonus: float
+ _finalPhasePenalty: float
+ _forwardBonus: float
+ _heightPenality: float
+ _jointsAtLimitPenality: float
- _lastSensorState: List<bool>
+ _limbPenalty: float
+ _maxDistanceTravelled: float
+ _phase: int
+ _phaseBonus: int
+ _reward: float
+ _targetVelocityForward: float
+ _terminationAngle: float
+ _terminationHeight:float
+ _timeAlive: float
+ _uprightBonus: float
+ _velocity: float
+ _velocityPenalty: float
+ _velocityReward: float
# Actions: List<float>
+ body: Transform
# BodyPartsToFocalRotation: Dictionary<string, Quaternion>
+ bothFeetDown: bool
+ CameraTarget: GameObject
# currentDecisionStep: int
+ footL: Transform
+ footR: Transform
+ head: Transform
+ hips: Transform
+ isLeftFootDown: bool
# isNewDecisionStep: bool
+ isRightFootDown: bool
# jdController: RobotJointDriveController
+ LeftMin: float
+ LeftMax: float
+ NumSensors: int
# OnTerminateRewardValue: float
# recentVelocity: List<float>
+ RightMax: float
+ RightMin: float
+ shinL: Transform
+ shinR: Transform
+ terminateNever: bool
+ thighL: Transform
+ thighR: Transform
+ AgentReset(): void
+ AgentAction(float[] vectorAction, strin textAction): void
~ CalcPhaseBonus(float min, float max): float
+ CollectObservationBodyPart(RobotBodyPart bp): void
+ CollectObservations(): void
~ GetAverageVelocity([Transform bodyPart= null]): float
~ GetBackwardsBonus(Transform bodyPart): float
~ GetDirectionBonus(Transform bodyPart, Vector3 direction): float
~ GetEffort(([string [] ignoreJoints = null]): float
~ GetForwardBonus(Transform bodyPart): float
~ GetHeightPenalty(float maxHeight): float
~ float GetJointsAtLimitPenality([string[] ignorJoints = null]): float
~ GetLeftBonus(UnityEngine.Transform bodyPart): float
~ GetPhaseBonus(): float
~ GetRightBonus(UnityEngine.Transform bodyPart): float
+ GetSkillReward(): float
+ GetTerminationOnAngle(): bool
+ GetTerminationOnHeight(): bool
~ GetUprightBonus(UnityEngine.Transform bodyPart): float
~ GetVelocity([UnityEngine.Transform bodyPart = null]): float
+ IncrementDecisionTimer(): void
+ InitializeAgent(): void
~ PhaseBonusInitalize(): void
~ PhaseResetRight(): void
~ PhaseResetLeft(): void
~ PhaseSetLeft(): void
~ PhaseSetRight(): void
+ TerminateRobot(): bool
Extends
Agent
+ actionMasker: ActionMasker
+ agentParameters: AgentParameters
+ brain: Brain
- cumulativeReward: float
- done: bool
- hasAlreadyReset: bool
- id: int
. indo: AgentInfo
- maxStepReached: bool
- recorder: DemonstrationRecorder
- requestAction: bool
- requestDecision: bool
- reward: float
- stepcount: int
- terminate: bool
-rextureArray: Texture2D[]
- _AgentReset(): void
+ AddReward(float increment): void
+ AddVectorObs(bool observation): void
+ AddVectorObs(float observation): void
+ AddVectorObs(IEnumerable<float> observation): void
+ AddVectorObs(int observation): void
+ AddVectorObs(int observation, int range): void
+ AddVectorObs(Quanternion observation): void
+ AddVectorObs(Vector2 observation): void
+ AddVectorObs(Vector3 observation): void
+ AgentAction(float[] vectorAction, string textAction): void
+ AgentOnDone(): void
+ AgentReset(): void
- AgentStep(): void
+ AppendMemoriesAction(List<float> memories): void
+ CollectObservations(): void
+ Done(): void
+ GetCumulativeReward: float
+ GetReward(): float
+ GetStepCount(): int
# GetValueEstimate(): float
+ GiveBrain(Brain brain): void
+ InitializeAgent(): void
+ IsDone(): bool
+ IsMaxStepReached(): bool
- MakeRequests(int academyStepCounter): void
+ ObservationToTexture(Camera obsCamera, int width, int height, ref Texture2D t
- OnDisable: void
- OnEnable(): void
- OnEnableHelper(Academy academy): void
+ RequestAction(): void
+ RequestDecision(): void
- ResetData(): void
- ResetIfDone(): void
+ ResetReward(): void
+ ScaleAction(float rawAction, float min, float max): float
- SendInfo(): void
- SendInfoToBrain(): void
# SetActionMask(IEnumerable<int> actionIndices): void
# SetActionMask(int actionIndex): void
# SetActionMask(int branch, int actionIndex): void
+ SetReward(float reward): void
- SetStatus(bool academyMaxStep, bool academyDone, int academyStepCounter
+ SetTextObs(string textObservation): void
- UpdateMemoriesAction(List<float> memories): void
+ UpdateTextAction(string textActions): void
+ UpdateValueAction(float value): void
+ UpdateVectorAction(float[] vectorActions): void
Extends
ControllerAgent
- _stepCountTillSkill: int
- academy: LocoAcademy
+ Action: int
- agent: RobotMultiskillAgent
+ heuristicBrain: Brain
- lastAction: int
- lastActionFaild: int
+ playerControl: bool
+ playerBrain: Brain
- resetCurriculumLearning: bool
+ stepsToTrainStand: int
+ stepsToTrainWalk: int
+ useMultiSkill: bool
+ AgentAction(float[] vectorAction, string textAction): void
+ AgentReset(): void
- ApplyActionToAgents(int Action): void
+ CollectObservations(): void
+ Initialize(): void

10.2 Repository

The project can be downloaded from the accompanying repository, wich can be requested from the author: fynn.braren@outlook.com
The Readme of the project includes the setup instructions.

10.3 Reward Utilities

A list of all the utilities that can be used for computing the reward function and cover the observations. This is an excerpt of the RobotAgent.cs script.

```
/// <summary>
    /// the overall reward
    /// </summary>
    public float _reward;
    /// <summary>
    /// the cumulative velocity reward since agent reset
    /// </summary>
    public float _cumulativeVelocityReward;
    /// <summary>
    /// the time alive since last reset
    /// </summary>
    public float _timeAlive;
    /// <summary>
    /// the bonus for phase regularity
    /// </summary>
    public float _finalPhaseBonus;
    /// <summary>
    /// leverages phase bonus as penalty for both feet on ground
    /// </summary>
    public float _finalPhasePenalty;
    /// <summary>
    /// the forward velocity of given bodypart
    /// </summary>
    public float _velocity;
    /// <summary>
    /// the velocity the agent should match
    /// </summary>
    public float _targetVelocityForward;
    /// <summary>
    /// returns penalty for deviating from forward axis
    /// </summary>
    public float _deviationPenalty;
    /// <summary>
    /// the avarage velocity forward
    /// </summary>
    public float _currentVelocityForward;
    /// <summary>
    /// the forward velocity of given bodypart
    /// </summary>
    public float _velocityPenalty;
    /// <summary>
    /// the final reward for matching the targetVel
    /// </summary>
    public float _velocityReward;
    /// <summary>
    /// the bonus for holding given bodyparts upright
    /// </summary>
    public float _uprightBonus;
    /// <summary>
    /// the bonus for holding given bodypart aligned with forward axis
    /// </summary>
    public float _forwardBonus;
    /// <summary>
    /// penalty for falling below height over feet
    /// </summary>
    public float _heightPenalty;
    /// <summary>
    /// penalty for moving legs irregular - max 0.5
    /// </summary>
    public float _limbPenalty;
    /// <summary>
    /// penalize joint actions at limit
    /// </summary>
    public float _jointsAtLimitPenalty;
    /// <summary>
    /// penalize excessive joint actions
    /// </summary>
    public float _effortPenalty;
    /// <summary>
    /// the maximum reached distance since training began
    /// </summary>
    public float _maxDistanceTravelled;
    /// <summary>
    /// the height used to terminate the agent for given skill
    /// </summary>
    public float _terminationHeight;
    /// <summary>
    /// the angle used to terminate the agent for given skill
    /// </summary>
    public float _terminationAngle;
```

10.4 Recources

Resources used in investigation:
https://docs.google.com/document/d/12KMgMDMvyFujF_SiN1BuCM6WjhSGxXLIkx_KoUPN6p8/edit?usp=sharing