AALBORG UNIVERSITY COPENHAGEN

M.SC. THESIS

Motion Matching with Proximal Optimization Policy

Author:
Yousif A. H. NAJIM

Supervisor:
George PALAMAS

A thesis submitted in fulfillment of the requirements
for the degree of M.Sc. in Medialogy

in the

Department of Architecture, Design and Media Technology

September 11, 2019

AALBORG UNIVERSITY COPENHAGEN

Abstract

Department of Architecture, Design and Media Technology

M.Sc. in Medialogy

Motion Matching with Proximal Optimization Policy

by Yousif A. H. NAJIM

The aim of this thesis was to build a motion matching system, augmented with reinforcement learning using the learning algorithm Proximal Policy Optimization. Extensive background research was done, and the system was implemented in Unity. Results show that the motion matching system has descent accuracy, however performance with reinforcement learning is only great when dealing with large amounts of agents. As such, more training and testing is needed to investigate this, in order to increase the robustness of the system.

Acknowledgements

Special thanks to

George Palamas, Project Supervisor, Aalborg University, Copenhagen, DK

Kenneth Berle, Replay Institute, Copenhagen, DK

Daniel Bordig, Replay Institute, Copenhagen, DK

Pirate Lurifax Holtmann, Replay Institute, Copenhagen, DK

Phillip Hargett, Replay Institute, Copenhagen, DK

Tim Monrad Larsen, Replay Institute, Copenhagen, DK

Tobias Nordvig Møller, Aalborg University, Copenhagen, DK

Jonas Aksel Billeskov, Aalborg University, Copenhagen, DK

Contents

Chapter 1

Introduction

The videogame industry has for several decades been pushing the boundaries of what we thought was possible in real-time graphics and simulations (Engel, 2019). Their pursuit in trying to overcome the uncanny valley has yielded technologies that have given birth to breathtaking virtual worlds, available from the comfort of your home. However, some technologies have advanced further than others, in particular the graphical fidelity in modern systems (Eric Haines, 2019).

This is in stark contrast to the fidelity of real-time animations, which seemingly has not made as large of an advancement (Cooper, 2019). That is not to say that there has been no improvement; on the contrary, animations look better than ever thanks to the advent of motion-capture solutions.

Physics simulations have also advanced significantly over the years, though many of these improvements have yet to make it into real-time systems due to performance constraints, and are therefore mostly utilized in the realm of film-rendering (Matt Pharr, 2016; Najim, Triantafyllidis, and Palamas, 2018). Similar performance heavy advancements can also be found in the realm of graphical fidelity, most notably ray-tracing (Eric Haines, 2019).

As such, offline systems have always been a step ahead as everything is pre-rendered, offloading virtually all performance-cost for the end-user. This is an affordance that real-time systems do not have, even if certain aspects can be faked such as pre-baked lights and global illumination (Lengyel, 2019).

As a matter of fact, real-time systems have a trend of faking things due to it being performance friendly, as long as they appear like the genuine article (Engel, 2016).

This mentality of fake-it-till-you-make-it has been around since forever, and can even be seen in modern technology: NVIDIA have achieved real-time raytracing (Eric Haines, 2019), which is revered as being the holy grail in real-time visual systems, despite the fact that it is not "traditional" real-tracing, but an effect that heavily resembles it at a fraction of the performance cost. But that does not matter; if it looks real then no one will be able to tell the difference.

Animation systems on the other hand have always been a mixed bag in comparison (Cooper, 2019). Incredibly fidelity can be achieved, although it relies heavily on the quality of the animation clips, as well as the people who painstakingly has to make all the clips connect in the correct order. While cheaper motion capture systems or alternatives have started to show up, making it more accessible than ever to achieve high-fidelity animation clips, the systems in which we handle them has remained largely the same. This is problematic as it is not scalable nor easily modifiable, especially when introducing new animations to a pre-existing animation tree. It also relies heavily on manually interconnecting every animation, making it increasingly difficult to keep an overview of the entire system, both visually and programmatically. The interconnectivity of the animations also plays a key role in making the animation system appear believable, making it even more difficult when manually connecting them.

In collaboration with Replay Institute, a sports-technology firm specializing in data tools, this thesis investigates whether it is possible to find a solution that mitigates the problems present in traditional animation systems, and how such a solution would be implemented. More specifically, the presented solution must be developed in Unity, which is the development environment of Replay Institute, whilst also being performance-friendly.

Chapter 2

Analysis

2.1 Background work

Some of the earliest examples of digital computer animation systems are several decades old (Noll, 1968; Baecker, 1969). However, they were made for specific purposes like mathematical simulations of higher dimensions (Noll, 1968), or used for animated visual displays (Baecker, 1969). (Burtnyk and Wein, 1971) proposed a computer-generated key frame animation program, capable of interpolating in-between frames, and (Csuri et al., 1979) presented an all-in-one modelling and animation program, reminiscent of modern software solutions like Blender and Maya. Realistic simulations were also being investigated (Hahn, 1988), were they build a model for physical simulation of rigid bodies that took into account various physical characteristics, like deformity and elasticity.

More recently, machine learning has been used to produce animations (Holden, Komura, and Saito, 2017; Zhang et al., 2018; Lee et al., 2019). Using a novel neural network architecture, (Holden, Komura, and Saito, 2017) developed a real-time character controller which could automatically produce animations depending on their goal and surrounding environment, making it adaptable to various kinds of situations. Their model was trained on gigabytes of locomotion data, however the system itself is compact, measuring at only a few megabytes. It is also extremely fast, requiring only milliseconds of execution time. (Zhang et al., 2018) build a system that would mitigate the need for data preprocessing when dealing with Character-driven Quadruped motion controllers. The system essentially works by analyzing a frame

window coupled with user-control signals to produce motion. They trained a model using a novel neural network architecture called Mode-Adaptive Neural Networks, which is capable of adapting the weights of its prediction network dynamically, in tandem with what they refer to as their expert weights. Each of these expert weights specialize is a particular movement, which allows the system to blend seamlessly between the desired motions. (Lee et al., 2019) build a comprehensive skeletal model consisting of over 300 bones, that could generate a wide variety of human motion ranging from typical to highly stylistic. The simulation of anatomical features and motor system where trained with deep reinforcement learning, and is capable of dynamic motor skills such as bone deformity and muscle weakness.

Facial animations have also been an active field of research for decades (Parke, 1972; Waters, 1987; Williams, 1990). Creating a face consisting of 250 polygons, (Parke, 1972) managed to animate it with cosine interpolation. They used 3D data to describe the facial expressions which was obtained using photographs. Years later, (Waters, 1987) would propose a new process for muscle modeling, that would allow for the animation of facial expressions using only a few parameters, whilst also being able to work on virtually any facial topology. (Williams, 1990) were one the first to create system for facial tracking, which they could then map to a digital face, allowing digital facial expressions to be driven by professional actors.

More recently, (Karras et al., 2017) developed a machine learning algorithm that could drive 3D facial animations in real-time with low latency using audio input. It is aided by a model that is used to make sense of the different variations in facial expressions, which simply cannot be derived from audio alone. In regards to head-mounted displays (HMD), (Olszewski et al., 2016) proposed a facial expressive and speech animation system, where a small monocular camera is mounted inside the HMD. Test subjects would then wear the headset and perform various emotional expressions and speak phonetically-balanced sentences. These recordings would then be coupled with animation-data and used to train a convolutional neural network. Their results allowed them to produce a system in which HMD-users could control digital avatars in real-time using their facial expressions, whilst also producing

speech animations and emotional expressions. (Cong, Bhat, and Fedkiw, 2016) proposed a new framework for facial animations, where a blend shape system for the facial muscles is used to drive the underlying anatomical and biomechanical simulation, whilst retaining a high degree of controllability. Their framework can produce animations of high quality, which they argue can already be integrated in high-end production pipelines.

Traditional animation systems that are employed in modern game engines have largely remained the same for years (Cooper, 2019). However, (Buttner, 2015; Clavet, 2016) recently introduced a new form of animation system that would largely eliminating the need for manually interconnecting every piece of animation, Motion Matching. It essentially works by computing a cost function, and using it to determine the best piece of animation to use in a given context. It is argued to be one the next steps towards next-generation animation quality (Cooper, 2019; Buttner, 2015; Clavet, 2016).

It is apparent that the animation field of research is more active than ever, especially with the advent of machine learning. In the context of this thesis, Motion Matching appears to be a solution that would mitigate many of the issues found in traditional animation systems. The robustness of such a solution is also backed by the fact that major productions have begun to employ it, more specially in videogames (Clavet, 2016). As Unity is also a game engine, I will proceed with it as the chosen solution and analyze it in-depth in the following section.

2.2 Motion matching explained

Motion matching is a next-generation animation system that works by jumping between different animations based on an algorithm at runtime (Cooper, 2019; Buttner, 2015; Clavet, 2016), in order to reach a specific target position. It automatically handles transitions between them, leading to a greater visual fidelity.

2.2.1 Animation preprocessing

Every piece of animation needs to be split into what is referred to as a pose. A pose is essentially a bookmark in an animation clip, where various information regarding said animation at that particular time is saved. The saved information includes local velocity, feet positons and trajectory of the pose. By preprocessing all the animations, you build a pose database that can be saved on disk and used later at runtime. The only human intervention in generating these poses is labelling specific animations, for later weighting during computation.

2.2.2 Algorithm

The motion matching algorithm consist of two parts; the first part goes through all the poses and computes a cost function for each of them, and the second part blends to the pose with the lowest cost. The computation of the cost consists of two functions: the current cost and the future cost.

Current cost is where the pose-matching against the current pose of the agent happens. Each candidate pose is compared to the current pose in terms of velocities and bone positions. The bigger the discrepancy, the higher the penalty to the cost.

Then, the candidate pose is matched against the goal that it needs to reach, which is the future cost. This is done by comparing the past and future trajectory of the pose against the trajectory to the goal, as well as comparing its velocity against the goal's direction.

After all the poses have been compared against the current pose, the pose with the lowest cost is selected. The winning pose is then compared against the current animation, in order to see if the pose is part of the same animation. This is done in order to prevent jerky blending, as there is no need to blend to a pose that will occur momentarily if the animation is allowed to keep playing.

2.2.3 Strengths and weaknesses

Motion matching eliminates the need for manually interconnecting every animation, as well as the need to hardcode every call to said animations. It is also easily scalable and extremely flexible when it comes to adding new animations, as no animation tree will need to be redone. Also, the cost function ensures that the winning pose matches the current pose, whilst bringing you closer to the goal position. This effectively creates far greater transitions between animations, making them appear even more believable. The robustness of the system is backed by major productions in videgames (Clavet, 2016).

However, there are two points in which motion matching falters: Animation data and performance. The animation fidelity of motion matching heavily relies on the data from which the poses where constructed. (Clavet, 2016) uses raw mocap data that was captured in a professional grade studio, which is not something that is accessible to most people. Not only that, mocap data has to consider a wide variety of angles which increases the amount of animations needed, as well as total animation time. In other words, motion matching works best when coupled with large amounts of high-quality animation data.

In terms of performance, it would appear that the algorithm linearly goes through all the stored poses. The bigger the amount of poses stored, the greater the impact on performance it will have. (Clavet, 2016) argues that using K-D tress would mitigate some of the performance issues, however, using more complex machine learning algorithms, such as neural networks and reinforcement learning, could potentially yield far greater results (Cooper, 2019; Sutton and Barto, 2018).

As such, it would be interesting to investigate the possibility of incorporating machine learning with motion matching. Unity has also recently added support for reinforcement learning (Juliani et al., 2018), allowing for intelligent agents to be trained. Training an agent to perform motion matching could lead to a substantial performance increase. Therefore, the next section examines reinforcement learning and investigates how it works.

2.3 Reinforcement Learning

Reinforcement learning is a computational approach to learning from interaction (Sutton and Barto, 2018), which mimics the way we learn in real life. Rather than exploring and mapping out the psychology of the human mind, in hopes extracting a learning model that can be implemented computationally, reinforcement learning adopts a simpler approach that is focused on goal-directed learning from interaction. It is a machine learning approach that works by mapping actions with situations in order to maximize a reward signal. As with other machine learning algorithms, when a model is trained, it can be extremely fast and efficient with low performance cost.

The learner, or agent, is not told which actions yield the most reward and is thus forced to learn by trying everything that it can. This allows the agent to discover the most optimal actions given a situation. However, in some cases, a given action might not only affect the current reward but also the reward of the next situation. The learning outcome of this procedure is referred to as a control policy, and it is the goal of the agent to learn the best policy.

It is important to note that reinforcement learning is different from other types of machine learning, specially supervised and unsupervised learning. Both these methods rely on large amounts of data in order to find specific or hidden patterns, whilst reinforcement learning relies on figuring out the best approach given a goal and a set of actions.

2.3.1 Problem and solution formulation

Reinforcement learning can be thought of as field of study encompassing two topics: One that pertains to the problem itself, and the class of solution methods that can be used in solving said problem.

The problem and solutions are traditionally formulized using the formal framework of Markov decision processes. The idea behind the Markov framework is to capture

the most critical components of a problem the agent is facing, without trivializing any of them:

- The ability to perceive its environment (sensation)
- The ability to affect its current state (action)
- The state of the environment must also be related to the desired target (goal)

These three components define the interaction between the agent and its environment, and is a simple framework intended to represent the essential features in artificial intelligence problems.

2.3.2 Reward and Value

The goal of the agent is to maximize the amount of Reward it gets for each step, whereas Value is the cumulative reward in the long run. This is where future planning in important. An agent might take a step that gives less reward in the current state, but will help yield more reward in the long run. This balance of taking steps that cumulatively yield a greater reward is arguably the most critical component in reinforcement learning. The estimation of the value is one of the key discoveries made in this field, as its behavior closely mimics that of humans. For instance, imagine if you were given a small crash prize of a 100USD, but are told that if you refrain from accepting it, you would be given an even bigger prize a week from now. Common sense would dictate that you wait a week, as this yields the greater reward – this type of behavior is what we want our learning agent to exhibit.

2.3.3 Related work

The earliest examples of machine learning in games can be traced back several decades. (Samuel, 1967) investigated two machine learning procedures using the game of checkers. Similar to modern reinforcement learning procedures, they gave their computer the rules of the game (learning environment), a sense of direction (goal), and a list of parameters with different weights (actions). Their results verified that a computer is indeed capable of learning, provided it has been programmed to do

so, and that it can do so in a reasonable amount of time (8-10 hours of training). In a sense, they laid the principles that would shape the future field of reinforcement learning.

(Laird and VanLent, 2001) were one of the first to propose using modern computer games as training environments to further the research of artificial intelligence. Their reasoning is that games are increasingly providing realistic, rich and diverse environments, many of which mimic our own world, as well as increasingly complex AI agents that inhabit them. (Bengio et al., 2009) presented a new framework for training AI agents that roots itself our systematic approach to teaching humans. In teaching methods around the world, students are always presented with problems in an organized manner. As students become more proficient, the problems will increase in complexity as well as introduce new concepts. (Bengio et al., 2009) formulized this approach to teaching in the context of machine learning under the name curriculum learning. They implemented this new training strategy and their experiments showed that significant improvements in generalization can be achieved.

The first documented learning model that successfully learned control policies from high-dimensional data using reinforcement learning was made by (Mnih et al., 2013). They successfully trained a model that exceeded previous approaches across a variety of 2D games, and even exceeding a few human experts. (Silver et al., 2017) achieved similar results when they managed to train an AI that mastered the game of GO. 3D shooters have also been tested and mastered by an AI (Lample and Chaplot, 2017; Wu and Tian, 2016).

Reinforcement learning can be used to train intelligent agents, and as such, we need intelligent environments in which agents can be trained. In an effort to standardize the evaluation of AI development, (Bellemare et al., 2013) developed The Arcade Learning Environment (ALE), a publicly available environment that serves as a platform for both problems, methodologies and evaluation of AI technology. Its contains several hundreds of Atari 2600 game environments, making it possible to report empirical results from different machine learning approaches. Similar in some regards to ALE, (Johnson et al., 2016) presented Project Malmo which is an AI experimentation platform built on top of the massively popular computer game Minecraft. Their

platform facilitates the implementation of AI technologies that can be easily coupled with the game's infinitely varied gameplay, which might lead to newer frontiers being discovered and explored within the field. The platform is currently open source, in support of a shared collaborative future in AI research. Similarly, The House of Interactions (THOR) is a framework that pertains to visual AI research (Kolve et al., 2017), which consists of its own photorealistic 3D learning environment.

Unity recently added support for deep reinforcement learning, called Unity ML-agents (Juliani et al., 2018). This allows the popular engine to be used as a powerful learning environment for a wide variety of scenarios. In the next section, we will examine ML-agents closer and investigate whether it can be used to train a motion matching system.

2.3.4 ML-agents explained

Ml-agents is an open source plugin for Unity (Juliani et al., 2018), that interfaces with Python. It employs several state of the art techniques, most notably Proximal Policy Optimization (PPO), which is an algorithm used for optimizing control policies (Schulman et al., 2017). It also uses curriculum learning for better learning. Most of the implementation is based on TensorFlow, one of the most widely used software libraries in machine learning.

Setting up for training

There are four key components when setting up a training environment with ML-Agents: The agent, the brain, the area and the academy.

The agent is where all the actions and observations are defined. Reward settings for each step is also set here, alongside a multitude of other training specific variables. The brain is the neural network that is trained using the agent, and can be connected to multiple agents at a given time. The area is the environment in which the agent is trained, and is reset every time the max amount for a given "episode" is reached. The academy is the manager of the training session, and is responsible for interfacing

with Python, as well as ending the training session when maximum steps allowed in total has been reached.

Reinforcement learning can be used to achieve great results whilst keeping performance cost low. If motion matching is combined with the state-of-art techniques found in ML-agents, it could potentially result in an animation system that is both extremely fast and efficient. Of the few training algorithms implemented, PPO is the most recent and it would be interesting to see how it would interplay with an equally new solution.

2.4 Problem statement

We reviewed relevant background work of animation systems, which ended with having Motion Matching as the desired animation system to implement. Motion matching was then explained and issues with its performance were highlighted. As Unity is the development environment, ML-agents, with its state-of the-art reinforcement learning techniques, presented itself as an opportunity to fix performance constraints. Reinforcement learning was then investigated, and after reviewing relevant literature, it was concluded that it could indeed help with reducing computational cost. Lastly, Ml-agents was briefly reviewed and PPO was chosen as the training algorithm for motion matching, as it appears to be one of the newest algorithms in the field.

As such, we arrive at the following problem statement:

Is it possible to implement Motion Matching in Unity, and increase its performance using Proximal Policy Optimization?

From the problem statement, we can derive the following design requirements:

- Implement a pose-generation system (section 2.2.1)
- Implement a motion matching system, capable of using the generated poses (section 2.2.2)
- Implement and set up a training environment for the motion matching system, using PPO as the training algorithm (section 2.3.4)

2.4.1 Success criteria

It is important to note that this is a one-student project and as such, the scope of the project should not greatly exceed what is feasible by myself. Motion matching requires large amounts of animations to work properly, and as documented in section x, mocap data seems to be ideal for it. As such, based on the limited resources I have available, the accuracy and animation fidelity of the motion matching system will not be a critical priority; however, if there is time to spare, this will be reconsidered.

The success criteria for this project is ultimately a working motion matching system, where the performance is significantly improved using PPO.

Chapter 3

Implementation

The following system was primarily programmed in C-Sharp and implemented using the Unity game engine, and will consist of three sections: The creation of the poses, the motion matching algorithm itself, and the integration of ML-Agents and PPO.

3.1 Creating a pose

There are two scripts at work here, namely Pose and PoseManager. As both of their names indicate, Pose pertains to the actual pose itself i.e. it is a container for all the pose information, whilst PoseManager handles the creation and saving of the poses.

3.1.1 Pose Script

The pose script contains a multitude of variables, each used in the motion matching algorithm. It is important to note that all created poses are stored in a dictionary, where each key represents an animation, and each value of a key represent an actual pose. This allows for easier debugging, visual inspection and better general coherence of the system. Below is a table that briefly explains each variable and its purpose:

animationName	The name of the animation that the pose belongs to
animationIndex	The key in the dictionary that represents the animation
poseIndex	The value linked with the key that contains the actual pose. The poses are ordered in a sequential manner, where the value of each pose indicates when it "plays" in the animation
animationTag	The tag of the animation, which is used for weighting purposes during computation
animationTime	From 0 to 1, indicates at what time in the animation the pose was sampled
animationLengthInSec	The length of the animation in seconds
amountOfPosesInAnim	The amount of poses for the animation, which also indicates the length of the key pertaining to the animation
poseVelocity	The velocity of the pose in local space
poseRotation	The rotation of the pose in world space
futurePoseIndecies	An array containing the values of the 10 poses that follow
futureTrajectory	Using the poses that follow, it contains a 10-point trajectory that stretches one second into the future
futureVelocities	Contains the pose velocities for each of the poses in the future trajectory
pastPoseIndecies	An array containing the values of the 10 poses that precede it
pastTrajectory	Using the poses that precede it, it contains a 10-point trajectory that stretches one second into the past
pastVelocities	Contains the pose velocities for each of the poses in the past trajectory
feetPositions	The feet positions of the pose in local space
feetVelocities	The feet velocity of the pose in world space

3.1.2 Pose Manager

The PoseManager handles the creation of the poses and stores them in a local JSON file. Due to the nature in which Unity handles animations, a multitude of workarounds were implemented in order to facilitate the pose creation process.

Accessing animations and tags

In order to create the poses, we need to be able sample an animation at different animation times. However, Unity does not provide any known way of accessing an animation clip programmatically. At most, it only gives you access to tinker with the settings of the clip and even these are inaccessible from code, as seen on figure 3.1. This leaves us with one option only: The Animator component.

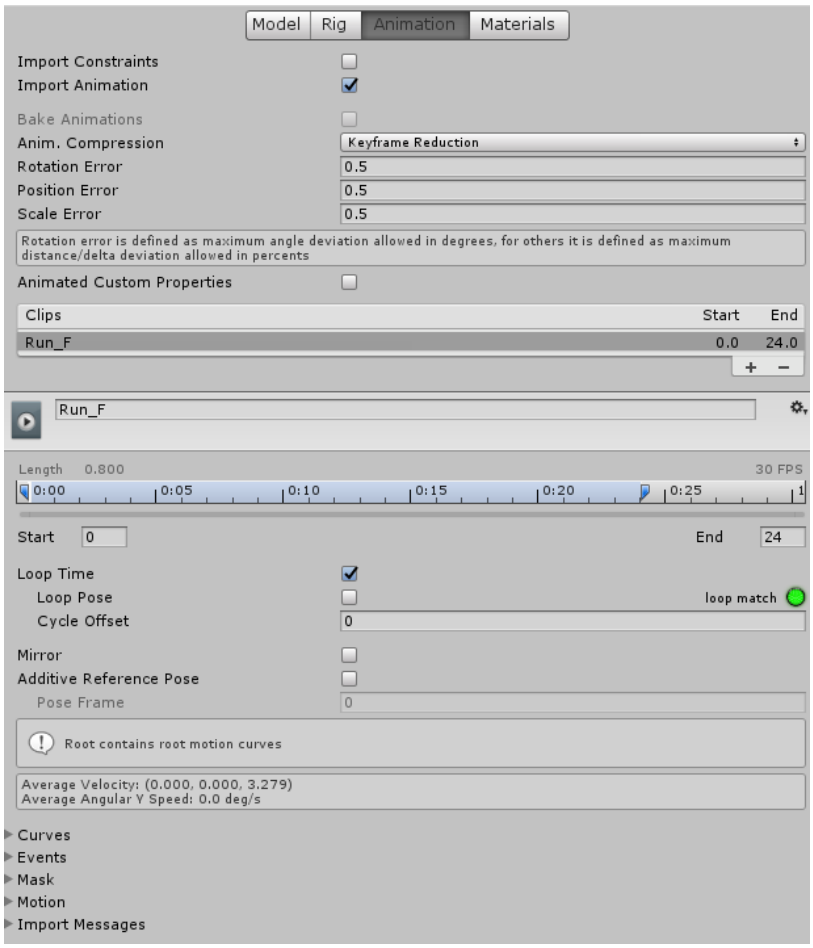

FIGURE 3.1: Animation settings available

The Animator is a component that handles the animation states of a given agent. In order to work with it, one has to give it an Animation Controller which is where all the animation states are setup.

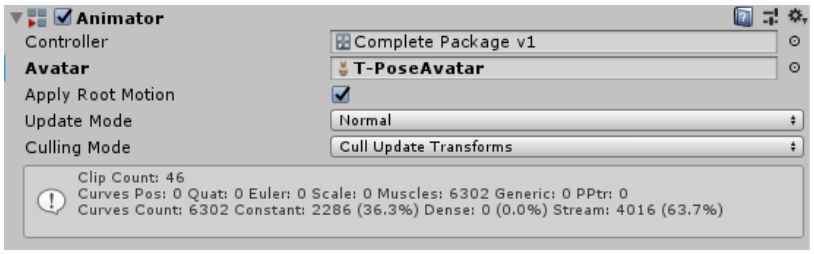

FIGURE 3.2: Animator component on agent

Using an animation controller, we can gain access, albeit limited, to the animations directly. Being able to access the animations programmatically is a step in the right direction, however accessing them programmatically over simulated time is another problem in itself.

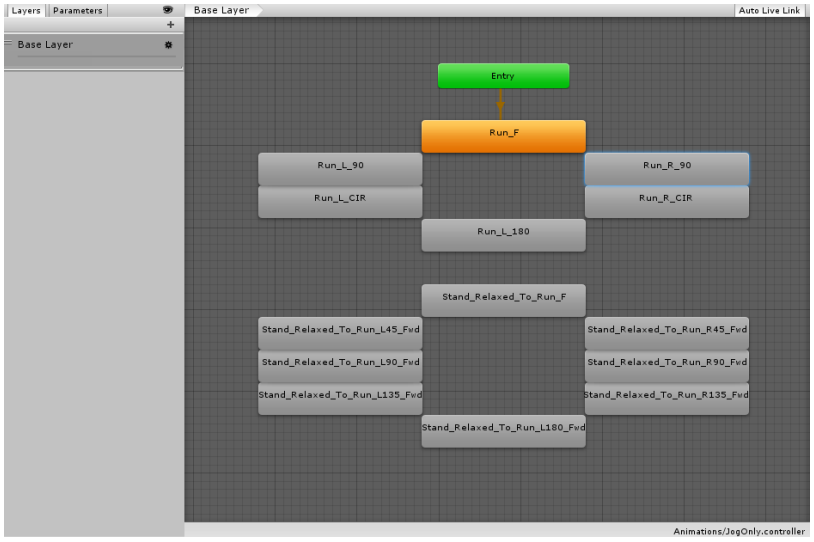

FIGURE 3.3: Example of Animation Controller

Unity has a couple of update cycles, most notably its normal update and fixedUpdate. Normal update is called once per frame, whilst fixedUpdate is frame-independent and is tied to the physics calculations of the engine. As such, it is highly advisable to use fixedUpdate for physics related calculations, as its not tied to the performance of the system like update is.

Using fixedUpdate, I tried to quickly loop through an animation and sample it, however, it yielded disappointing results: each sample's root positon, feet positions and

velocities where all null, however the animation times were correctly logged. Other variations of programmatically speeding up the sampling process were attempted but with similar results. This all seemed to point in the same direction: The animation system in Unity is seemingly using a hidden update cycle, separate from the fixedUpdate cycle. This unfortunately forces me to loop through an animation at normal speed while sampling it. As such, the sampling time equals the cumulative length in seconds of all the animation clips.

Another issue lies in the tagging of animations. In the animator controller, you can tag each animation state with a string, however, they are stored as hashtags and accessed as so. The problem with hashtags in Unity is that they are generated differently based on the method, as well as the instance in which it was done. This effectively means that storing the hashtags for later weighting during motion matching will not work, as there will be no way for us to know what string each hashtag was derived from. Even if we generated new hashtags from the animation tags at runtime, using the same exact hash-conversion method in order to compare against the hashtags stored in the poses, they will be completely different since they were not generated in the same instance.

In order to work around this issue, a string array containing all the animation tags was prepared then converted to a new hashtag array, using the hash-conversion method of the animator component during the same instance in which the hashtags of the animations were accessed. This ensured that the hashtags were identical. During pose creation, the accessed hashtags are compared against the stored hashtags, and if there is a match, we use the index of the stored hashtag to directly access the animation tags array we prepared previously, as both arrays are ordered identically. This allows us to store the tags as they are directly in the generated poses, and is far more convenient when we later do tag-weighting.

Processing the animations

The poses are created using the ProcessAnimations function, which is called a fixed amount of times every second (covered in a later section). It consists of three main

steps in the form of if statements:

```
private void ProcessAnimations()
{
    if (finishedFinalAnimation)...
    else if (finishedCurrentAnimation)...
    else if (animator.GetCurrentAnimatorStateInfo(0).normalizedTime >=
        0.01f)...
}
```

- First we check if all animations in the animation controller has finished processing. This statement primarily deals with the saving of the generated poses (saving all poses)
- If the above statement is false, we check whether the current animation has finished playing. This statement calculates trajectories and other remaining information for the poses created from the current animation (finishing the current poses)
- If both statements are false, we proceed with processing the current animation and generating the poses (creating the current poses)

Generating the poses

The procedure in which the pose is created and populated with relevant information is relatively straight forward. The pose is created and by using the animator component, we gain access to various data.

```
Pose currentFramePose = new Pose();
AnimatorClipInfo[] animClip = animator.GetCurrentAnimatorClipInfo(0);
currentFramePose.animationName = animClip[0].clip.name;
currentFramePose.animationIndex = animatorIndex;
currentFramePose.animationTime =
    animator.GetCurrentAnimatorStateInfo(0).normalizedTime;
currentFramePose.poseVelocity = animator.velocity;
currentFramePose.animationLengthInSeconds = animClip[0].clip.length;
```

```
currentFramePose.feetPositions[0] =
    animator.GetBoneTransform(HumanBodyBones.LeftFoot).position -
    this.transform.position;
currentFramePose.feetPositions[1] =
    animator.GetBoneTransform(HumanBodyBones.RightFoot).position -
    this.transform.position;
```

As covered earlier, we compare hashtags to assign the correct animation tag to the pose.

```
int currentAnimationHashTag =
    animator.GetCurrentAnimatorStateInfo(0).tagHash;
for (int i = 0; i < animationHashTags.Length; i++)
{
    if (currentAnimationHashTag == animationHashTags[i])
    {
        currentFramePose.animationTag = animationTags[i];
        break;
    }
}
```

In regards to feet velocities, we simply use the pose that preceded the current pose to calculate them. However, the first pose created cannot have a pose that precedes, therefore we postpone its feet velocities calculation until we have created all the poses for the current animation.

```
if (currentAnimationPoses.Count > 0)
{
    currentFramePose.positionOffset =
        currentAnimationPositions[currentAnimationPositions.Count - 1]
        - currentAnimationPositions[currentAnimationPositions.Count - 2];
    Vector3 leftFeetVelocity = currentFramePose.feetPositions[0]
        - currentAnimationPoses[currentAnimationPoses.Count -
            1].feetPositions[0];
    Vector3 rightFeetVelocity = currentFramePose.feetPositions[1]
```

```
        - currentAnimationPoses[currentAnimationPoses.Count -
            1].feetPositions[1];

    currentFramePose.feetVelocities[0] = leftFeetVelocity;
    currentFramePose.feetVelocities[1] = rightFeetVelocity;
}
```

When the animation reaches its end, we allow it to loop for what is equivalent to one pose-length, as we need the loop offset to properly calculate the trajectories later.

```
if (animator.GetCurrentAnimatorStateInfo(0).normalizedTime >= 1)
{
    finishedCurrentAnimation = true;
    startPoseLoopOffset =
        currentAnimationPositions[currentAnimationPositions.Count - 1]
        - currentAnimationPositions[currentAnimationPositions.Count - 2];
}
```

Calculating the trajectories

After all the poses of the current animation has been created, we calculate the missing feet velocities of the first pose, before we proceed with calculating the past and future trajectories for the poses.

```
//calculate feet velocities for the first pose
currentAnimationPoses[0].feetVelocities[0] =
    currentAnimationPoses[0].feetPositions[0] -
    currentAnimationPoses[currentAnimationPoses.Count - 1].feetPositions[0];
currentAnimationPoses[0].feetVelocities[1] =
    currentAnimationPoses[0].feetPositions[1] -
    currentAnimationPoses[currentAnimationPoses.Count - 1].feetPositions[1];
```

For each pose, we loop through the created poses and fetch the 10 poses that precede it and the 10 poses that follow it. We then use modulus to ensure that we loop back to the beginning of the poses in case we reach the end of the list.

```
//Here we calulcate the 10 poses that follow the current pose
for (int j = 0; j < 10; j++) {
    int poseIndex = (currentAnimationPoses[i].poseIndex + j + 1) %
        currentAnimationPoses.Count;
    Pose futurePose = currentAnimationPoses[poseIndex];
    currentAnimationPoses[i].futurePoseIndecies[j] = futurePose.poseIndex;
}
//Here we calulcate the 10 poses that precceded the current pose
for (int j = 0; j < 10; j++) {
    int poseIndex = mod((currentAnimationPoses[i].poseIndex - j - 1),
        currentAnimationPoses.Count);
    Pose pastPose = currentAnimationPoses[poseIndex];
    currentAnimationPoses[i].pastPoseIndecies[j] = pastPose.poseIndex;
}
```

This is done as animations normally loop back to the beginning, and therefore the last pose will naturally be followed by the first pose. During this process, we also make sure we store their indices, which is particularly useful for later debugging. Afterwards, we start calculating the trajectories, starting with the future followed by the past. The principal in calculating both trajectories for a given pose is the same:

- We create a trajectory point and offset its location based on the 10 stored poses

```
Vector3 futureTrajectoryPoint = Vector3.zero;
float futureRotation = 0;
int startKFuture = currentAnimationPoses[i].futurePoseIndecies[0];
int endKFuture = startKFuture + 10;
for (int k = startKFuture; k < endKFuture; k++) {
    int currentIndex = mod(k, currentAnimationPoses.Count);
    int previousIndex = mod(k - 1, currentAnimationPoses.Count);
    futureTrajectoryPoint += Quaternion.Euler(0, futureRotation, 0) *
        currentAnimationPoses[currentIndex].positionOffset;
```

```
    futureVelocity = (Quaternion.Euler(0, futureRotation, 0) *
        currentAnimationPoses[currentIndex].poseVelocity) +
        futureTrajectoryPoint;
```

- For each offset, we log its position and velocity in their respective arrays inside the pose

```
currentAnimationPoses[i].futureTrajectory[futureIndex] =
    futureTrajectoryPoint;
currentAnimationPoses[i].futureVelocities[futureIndex] =
    futureVelocity;
```

- If one of the stored poses equals the first pose created for the animation, we rotate the remaining pose velocities. This is because animations usually have a different start rotation with each consecutive loop, which we will have to account for to properly map out trajectories for the poses that occur at the end of the animation

```
if (currentIndex == 0) {
    float newRot = futureRotation +
        currentAnimationPoses[currentIndex].poseRotation.y +
        currentAnimationPoses[previousIndex].poseRotation.y;
    newRot = newRot > 360 ? newRot - 360 : newRot;
    newRot = newRot < 0 ? newRot + 360 : newRot;
    futureRotation = newRot;
}
```

When the trajectories have been calculated, we proceed to store the new poses in a dictionary, where the key is the animation, and the values are the poses themselves. We then clear our all the temporary variables in order to reuse them for the next animation.

```
newPoseDatabase[animatorIndex] = new List<Pose>(currentAnimationPoses);
currentAnimationPoses.Clear();
currentAnimationPositions.Clear();
currentAnimationRotations.Clear();
```

At the very end we check whether we have gone through all the animations in the animation controller, and if so, use a boolean value to signify that all animations have completed processing. If not, we merely proceed to the next animation.

```
if (animatorIndex < animationClips.Count - 1){
    Debug.Log("Next State");
    animatorIndex++;
    animator.Play(animationClips[animatorIndex]);
} else {
    finishedFinalAnimation = true;
}
```

Saving the Poses

When all animations have finished processing and all the poses are neatly arranged in a dictionary, we proceed with saving said dictionary to a local JSON file using the SavePosesToJSON function.

```
private void SavePosesToJSON()
{
    string[] valuesArray = new string[newPoseDatabase.Count];
    for (int i = 0; i < newPoseDatabase.Count; i++) {
        valuesArray[i] = JsonHelper.ToJson(newPoseDatabase[i].ToArray());
    }
    string keys = JsonHelper.ToJson(newPoseDatabase.Keys.ToArray());
    string values = JsonHelper.ToJson(valuesArray);
    string[] dictionary = { keys, values };
    string finalString = JsonHelper.ToJson(dictionary);
    string path = Application.persistentDataPath + "/" + jsonSaveName +
        ".json";
    File.WriteAllText(path, finalString);
}
```

First, we convert the keys and values into two separate JSON strings, which we then combine in a string array. The string array is then converted into a JSON string,

which is then saved to disk.

Custom Update

The poses are sampled with a 10hz frequency, or simply put, 10 times per second. In order to achieve this desired sampling rate, we use an enumerator.

```
IEnumerator CustomUpdate(float frameRate) {
    beginProcessing = false;
    float waitTime = 1 / frameRate;
    for (int i = 0; i < frameRate; i++){
        yield return new WaitForSeconds(waitTime);
        if (!finishedProcessing) {
            animator.enabled = true;
            ProcessAnimations();
        }
    }
    beginProcessing = true;
}
```

3.2 Motion matching

Motion matching is an animation system that works by computing a cost and jumping to the pose with the lowest cost. The computation of the cost is split into two parts: The current cost, which compares the current pose with the candidate pose, and the future cost, which compares the candidate cost against the goal that the system, or agent, is trying to reach.

```
    private float ComputeCost(Pose currentPose, Pose candidatePose)
    {
        float cost = 0;
        cost += ComputeCurrentCost(currentPose, candidatePose);
        cost += ComputeFutureCost(candidatePose);
        return cost;
    }
```

3.2.1 Current Cost

We compute the current cost by comparing the difference in local velocities, feet positions and feet velocities. The bigger the difference, the higher the cost. We use lambda weights for each calculation in order to control their influence on the final cost.

```
private float ComputeCurrentCost(Pose currentPose, Pose candidatePose)
{
    float distanceCost = 0;
    //difference in local velocity
    distanceCost += velocityLambda *
        Vector3.Distance(currentPose.poseVelocity.normalized,
        candidatePose.poseVelocity.normalized);
    //difference in local feet positions
    distanceCost += feetPosLambda *
        Vector3.Distance(currentPose.feetPositions[0],
        candidatePose.feetPositions[0]);
    distanceCost += feetPosLambda *
        Vector3.Distance(currentPose.feetPositions[1],
        candidatePose.feetPositions[1]);
    //difference in local feet velocity
    distanceCost += feetVelLambda *
        Vector3.Distance(currentPose.feetVelocities[0].normalized,
        candidatePose.feetVelocities[0].normalized);
    distanceCost += feetVelLambda *
        Vector3.Distance(currentPose.feetVelocities[1].normalized,
        candidatePose.feetVelocities[1].normalized);
    distanceCost *= currentCostWeight;
    return distanceCost;
}
```

3.2.2 Future Cost

We compute the future cost by first comparing the direction of the goal against the local velocity of the pose.

```
Vector3 goalDirection = goalTrajectory[9] - posePosition;
Vector3 candVelocity = Quaternion.Euler(0, currentRot, 0) *
    candidatePose.poseVelocity;
```

Afterwards, we do a for loop that runs 10 times which we use to go through both the past- and future trajectory stored in the candidate pose. We compare them against the goal trajectory and the path traversed by the agent, both of which also consist of 10 points. For each loop, we compare a goal trajectory point against a future trajectory point in terms of velocity and direction, followed by a comparison in terms of distance. We then compare a past trajectory point against a path traversed point in terms of distance.

```
for (int j = 0; j < 10; j++){
    //Goal trajectory matched against candidate future trajectory in terms
        of velocity
    Vector3 rotatedVelocity = Quaternion.Euler(0, currentRot, 0) *
        candidatePose.futureVelocities[j];
    float trajectoryVelCost = trajectoryVelLambda *
        Vector3.Distance(goalTrajectoryDirection.normalized,
        rotatedVelocity.normalized);
    futureCost += trajectoryVelCost;
    //Goal trajectory matched against candidate future trajectory in terms
        of position
    Vector3 trajectoryPoint = Quaternion.Euler(0, currentRot, 0) *
        candidatePose.futureTrajectory[j];
    trajectoryPoint += posePosition;
    float trajectoryPosCost = trajectoryPosLambda *
        Vector3.Distance(goalTrajectory[j], trajectoryPoint);
    candTrajectoryPos.Add(trajectoryPosCost);
    futureCost += trajectoryPosCost;
```

```
    //Agent past trajectory matched against candidate past trajectory in
        terms of position
    Vector3 pastTrajectoryPoint = Quaternion.Euler(0, currentRot, 0) *
        candidatePose.pastTrajectory[j];
    pastTrajectoryPoint += posePosition;
    float pastTrajectoryPosCost = pastTrajectoryPosLambda *
        Vector3.Distance(previousTrajectory[9 - j], pastTrajectoryPoint);
    futureCost += pastTrajectoryPosCost;
}
```

What we are ultimately doing here is trajectory matching the different graphs, and the more similar they are, the lesser the cost and vice versa. As with the current cost, we use weighting lambdas to influence each calculation's effect on the final cost.

3.2.3 AmoUpdate

The function that ties everything together is AmoUpdate, which is called 30 times per second using an enumerator. Higher framerates have been tested, but yielded significantly less accurate results whilst also being more performance heavy. We first start by creating the variable costForBestPose and set it to the maximum possible float value. This variable will be used to store the calculated cost for the best candidate pose. We then calculate our current pose using the previous pose from the frame earlier. We do this because the animation will have progressed slightly from the last frame, and therefore our current pose will have changed.

```
float costForBestPose = float.MaxValue;
previousBestAnimTime += customDeltaTime;
if (previousBestAnimTime >= 1) {
    previousBestAnimTime -= 1;
}
currentPoseLerped = EvaluateLerpedPoseFromData(previousBestAnimIndex,
    previousBestAnimTime);
```

We then loop through all the poses, computing the cost for each of them. Every time we calculate a candidate cost that is lower than costForBestPose, we overwrite it with the calculated cost and store the candidate pose as the best pose. As such, the pose with the lowest possible cost is selected.

```
for (int i = 0; i < poseDatabase.Count; i++) {
    for (int j = 0; j < poseDatabase[i].Count; j++) {
        Pose candidatePose = poseDatabase[i][j];
        float candidateCost = ComputeCost(currentPoseLerped, candidatePose);
        if (candidatePose.animationTag !=
            TrajectoryController.desiredMovementTag) {
            if (candidatePose.animationTag != null) {
                candidateCost *= 30f;
            }
        }
        if (candidateCost < costForBestPose) {
            bestPoseMatchedAgainstCurrentPose = candidatePose;
            costForBestPose = candidateCost;
        }
    }
}
```

We also use the tag of the pose as a weighting cost, comparing it against the desired tag of the movement system. The reason we do this is simply because it increases the accuracy during tricky movement actions at runtime. Afterwards, we check whether the selected pose belongs to the same animation and if it is too close in terms of animation time.

```
isSameAnimationAndVeryClose =
    previousBestAnimIndex ==
        bestPoseMatchedAgainstCurrentPose.animationIndex &&
    Mathf.Abs(previousBestAnimTime -
        bestPoseMatchedAgainstCurrentPose.animationTime) < .4f;
```

We do this as there is no need to blend to a pose that will come up shortly after. It also prevents the animation from appearing jerky. If the check is passed, and the

pose is save to blend to, we call the PlayAnimation function.

```
if (!isSameAnimationAndVeryClose) {
    PlayAnimation(bestPoseMatchedAgainstCurrentPose);
    previousBestAnimIndex = bestPoseMatchedAgainstCurrentPose.animationIndex;
    previousBestAnimTime = bestPoseMatchedAgainstCurrentPose.animationTime;
    previousBestPoses.Add(bestPoseMatchedAgainstCurrentPose);
}
```

3.3 Proximal Policy Optimization

The training setup for reinforcement learning with ml-agents is straightforward, and consists of five different parts: Agent, Brain, Training Area, Academy and Python.

Agent is the learner in our system. He has a set of discrete actions that equal the length of the pose-database. Each action corresponds to a pose, circumventing the need for linearly sorting through all the poses. The agent is connected to a learning brain, which is essentially the neural network, or model, that is trained. As training goes on, the model will slowly learn the best policies for maximising reward.

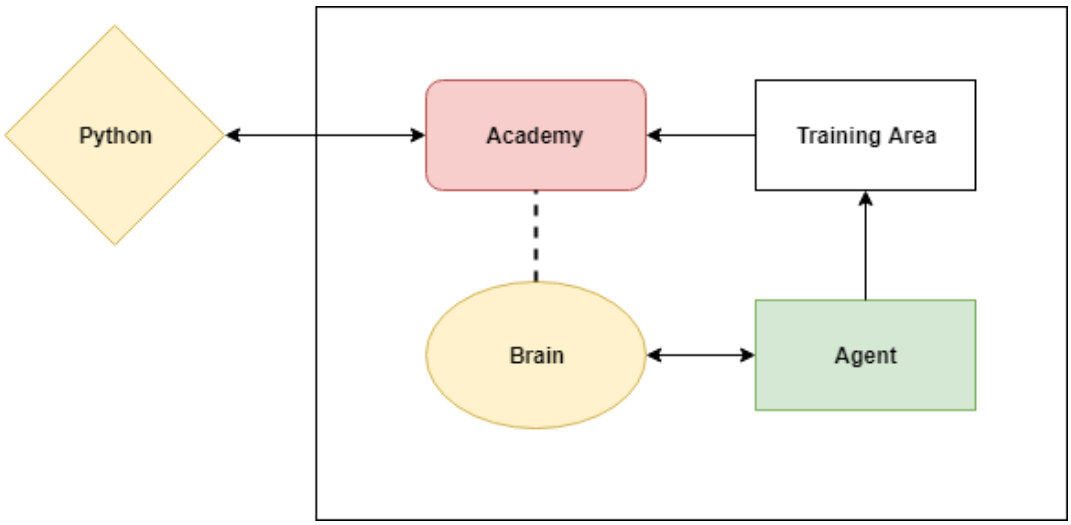

FIGURE 3.4: Overview of the ml-agent workflow

Training area is the area in which the agent is trained. It consists of a simple plane with invisible walls, in order to train basic obstacle avoidance as well. In this area, a goal is randomly positioned, and it is the job of the agent to reach said goal. When

that happens, the agent is rewarded a point, and the goal is randomly positioned somewhere else, repeating the loop.

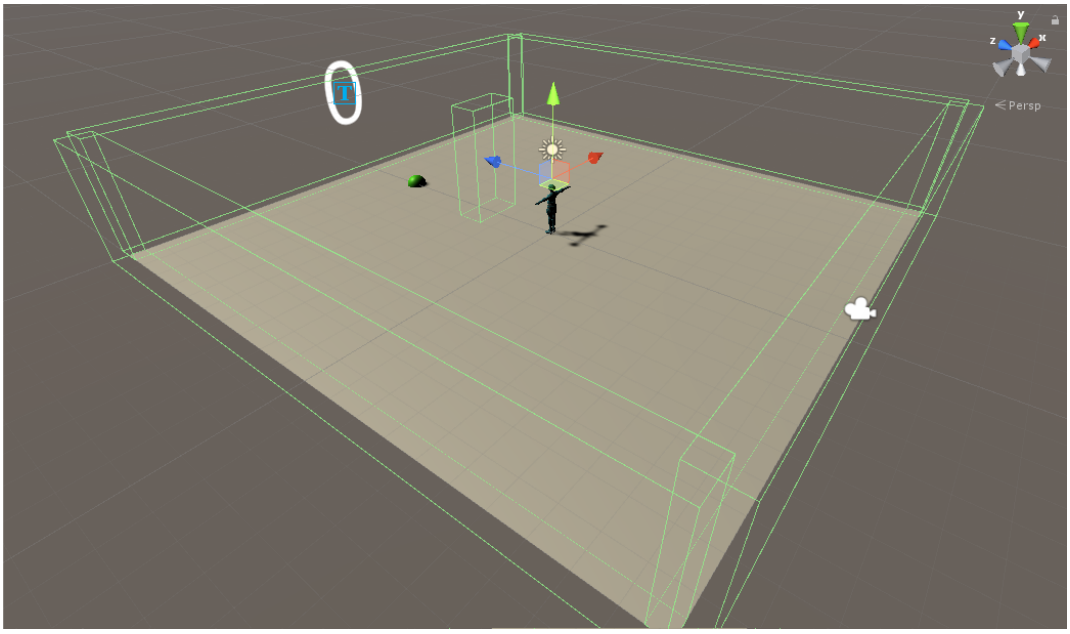

FIGURE 3.5: The training area. The green lines are the invisible walls and the green ball is the goal marker

The academy can be seen as the overseer of all training. It manages all training area instances, and handles the reset parameters for each of them, including those pertaining to curriculum training; this and other hyperparameters are set externally with python. It also acts as the gateway between Unity and python.

Python essentially handles the training process. All that is required here is setting the necessary hyperparameters and curriculum settings. It is also here we specify which algorithm we want to use, which in this case is Proxial Policy Optimization (PPO).

```
AgentLearningParameters:
    trainer: ppo
    use_curiosity: true
    summary_freq: 2000
    curiosity_strength: 0.01
    curiosity_enc_size: 256
    time_horizon: 128
    batch_size: 128
    buffer_size: 2048
```

```
    hidden_units: 512
    num_layers: 2
    beta: 1.0e-2
    max_steps: 5.0e5
    num_epoch: 3
```

These parameters are mostly random, as a wide variety was tested until the model started learning correctly.

The curriculum consists of 7 increasingly difficult lessons. The reset parameter, which is the parameter that the academy uses to reset the training area instances, is the goal radius; in the first lesson it is 2.5 meters, but at the end, it reaches 0, meaning that the agent has to collide with it physically instead of it being a mere euclidean distance measurement.

```
{
    "measure": "reward",
    "thresholds": [ -0.1, 0.7, 1.7, 1.7, 1.7, 2.7, 2.7 ],
    "min_lesson_length": 80,
    "signal_smoothing": true,
    "parameters": {
        "goal_radius": [2.5, 2.0, 1.0, 0.5, 0.2, 0.1, 0, 0]
    }
}
```

The threshold for each lesson is linked with the cumulative reward. If the cumulative reward for an episode exceeds the threshold of the current lesson, we proceed to the next one. An episode can be thought of as time-frames in which we evaluate the current progress of the training, after which we reset the training areas and cumulative reward. However, the policies learned transfer over to the next episode, and it is in this fashion that the model is trained.

Chapter 4

Methodology

4.1 Accuracy of Motion Matching

In order to determine whether our Motion Matching system works as intended, we will have to examine its accuracy. The accuracy will be measured by logging the cost of the best matched pose, every single frame that the system is called. The closer the cost is to 0, the more accurate it is. We do this over a span of five minutes. The testing setup for this will consist of a single agent, that is randomly assigned a new goal position every two seconds.

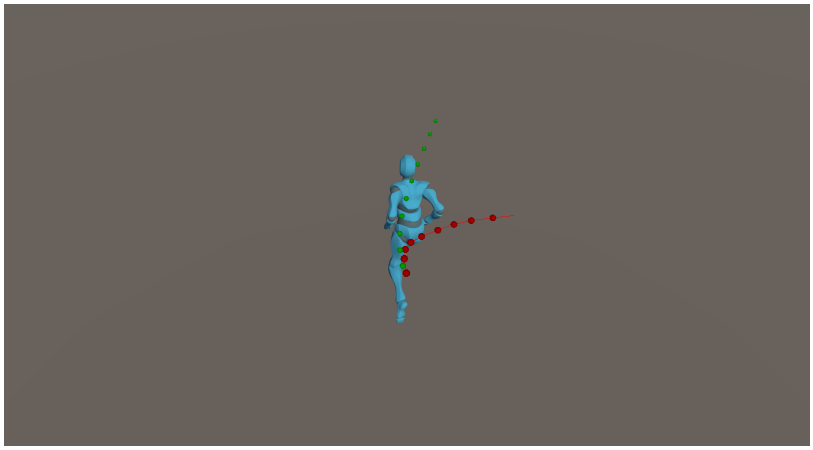

FIGURE 4.1: Accuracy Testing Setup

4.2 Reinforcement Learning

The model is trained using multi-agent parallelization, for faster learning and efficiency. The training setup will consist of nine agents in total, each having their own

area with obstacles and goals. In order to determine whether a model is training correctly, we examine the following:

- Cumulative Reward, in order to observe whether the agents manage to increase their reward over time.
- Curriculum Lesson, to see if they manage to reach increasingly harder lessons.
- Entropy, to observe the randomness of the model. Should decrease over time if learning is successful.
- Policy Loss, to observe the change in policies. Should decrease over time as the best policies are learned.
- Value Estimate, to observe the agent's ability to predict future reward. Should increase over time as well.

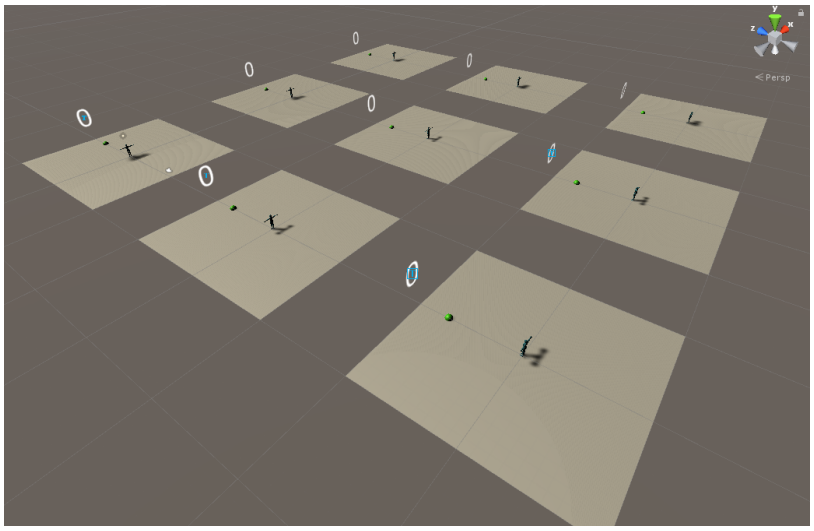

FIGURE 4.2: Training setup

4.3 Performance

Performance of motion matching without reinforcement learning will be determined by logging the framerate across seven testing conditions.

Each test condition differ in the amount of agents running: 1, 4, 8, 16, 32, 64 and 128. The same test conditions also apply to motion matching with reinforcement

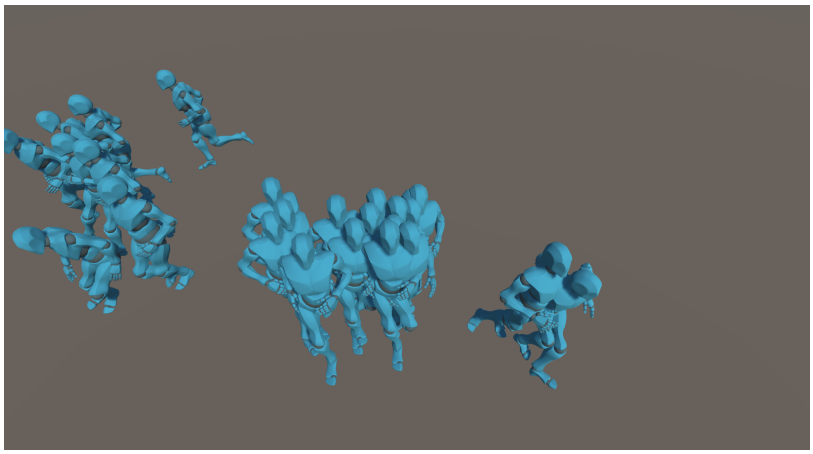

FIGURE 4.3: Performance setup

learning, totalling 14 tests. All testing conditions have their own Unity build, in order to mitigate bias to performance.

Chapter 5

Results

The following section will consists of the following charts, as covered in section 4.2:

Best cost chart 5.1, which shows what the cost for each best pose was during a period of frames. Best cost Histogram 5.2, which shows the distribution of best costs. Cumulative reward 5.3, which shows the reward gained over training time. Curriculum lesson 5.4, which shows the progression of the model in terms of increasing difficulty. Entropy 5.5, which shows the tendency of randomness in the model over training time. Policy Loss 5.6, which shows how often the policy is changed in the model. Value estimate 5.7, which shows the the ability of the model to predict reward for future steps. Performance chart 5.8, which shows the performance of motion matching with and without reinforcement learning. And finally, figure 5.9 is a chart that shows average framerate across all testing conditions (see section 4.3).

The motion matching system consists of 16 animations, with 241 poses created. Total animation length is 25.6 seconds. Training was done with a timescale of 20, as higher timescales did not work optimally. All training and testing was done on a computer with the following specifications:

CPU	Intel Core i7-8750H @ 2.2Ghz
GPU	16GB Dual-Channel DDR4-2667MHz
RAM	NVIDIA GeForce GTX 1060 Max-Q Design 6GB
STORAGE	Samsung 970 EVO Plus 1TB SSD

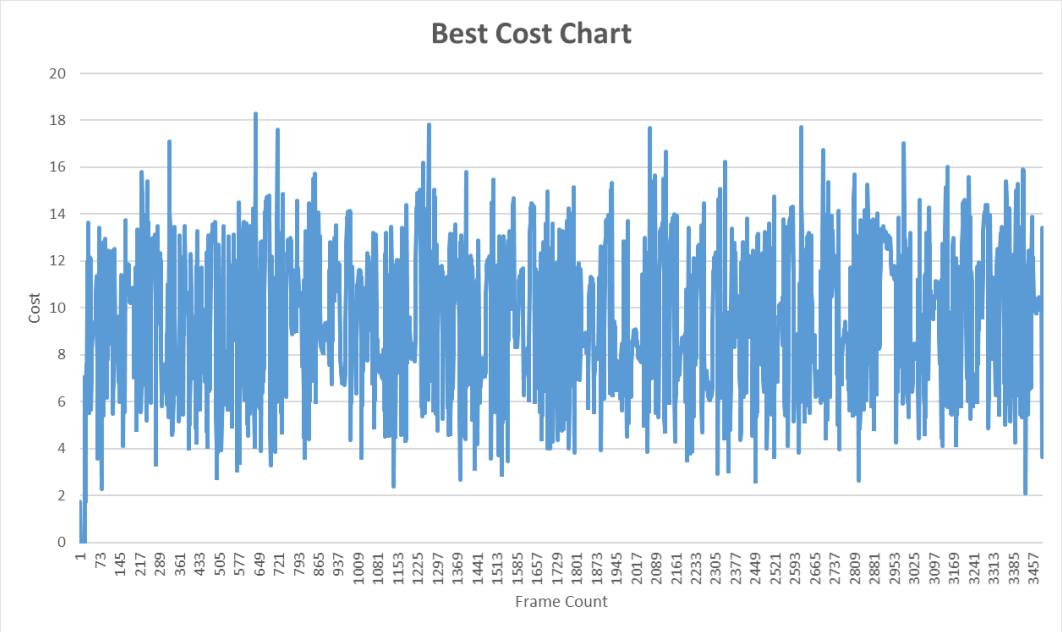

FIGURE 5.1: The average cost was 9.44, with the highest cost being 18.27, and lowest cost 0.

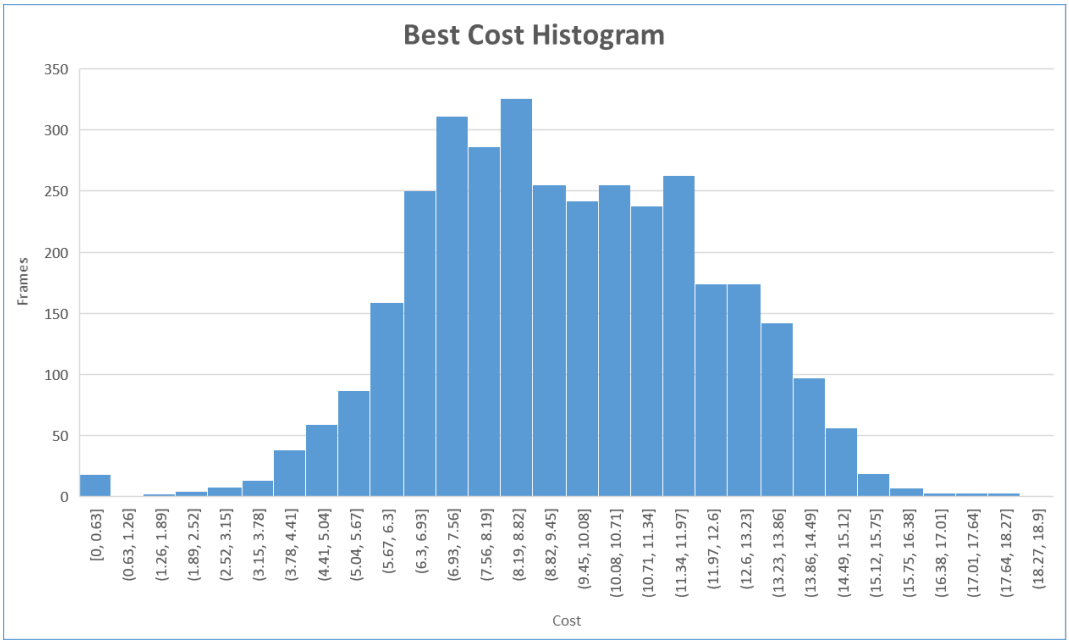

FIGURE 5.2: The best costs show a normal distribution.

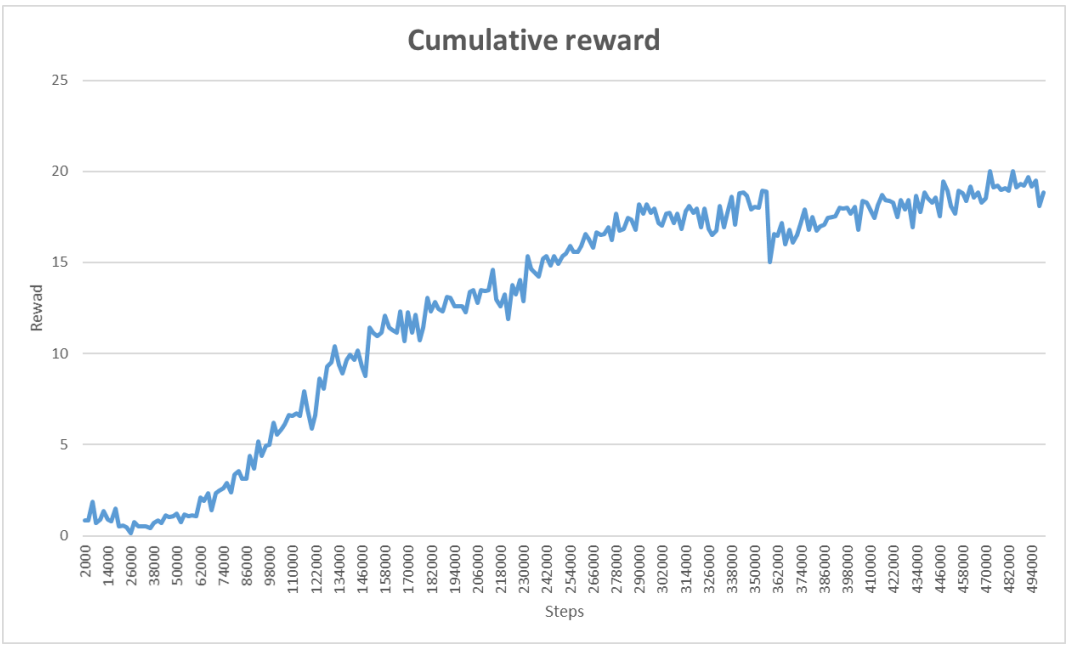

FIGURE 5.3: Highest reward achieved was 20

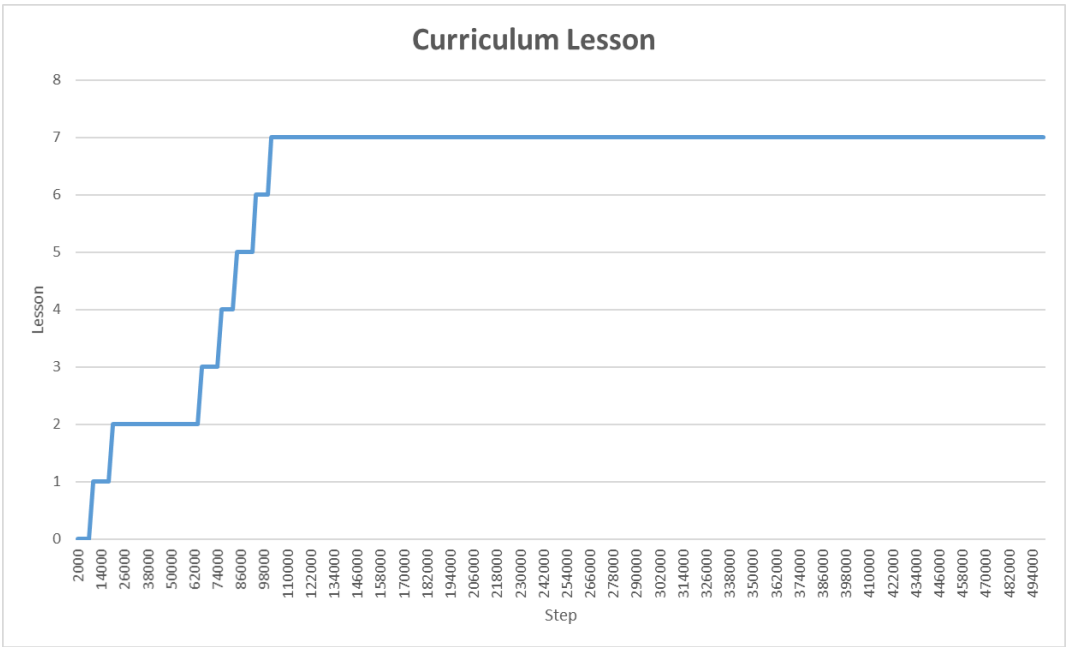

FIGURE 5.4: Final lesson reached at step 102000, a fifth of the overall training time

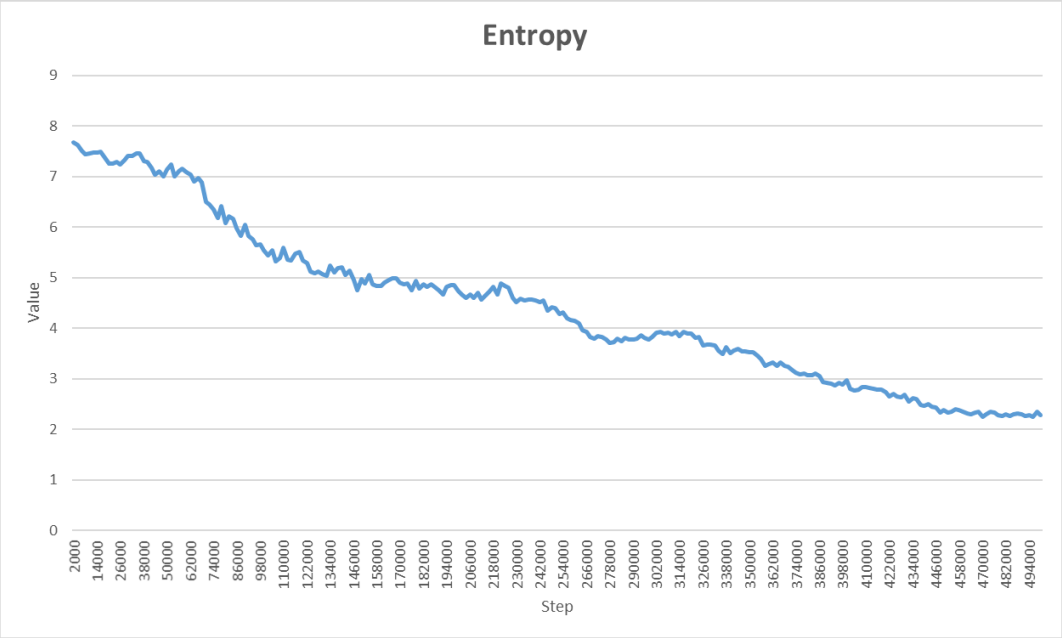

FIGURE 5.5: A steady decrease in randomness of the model

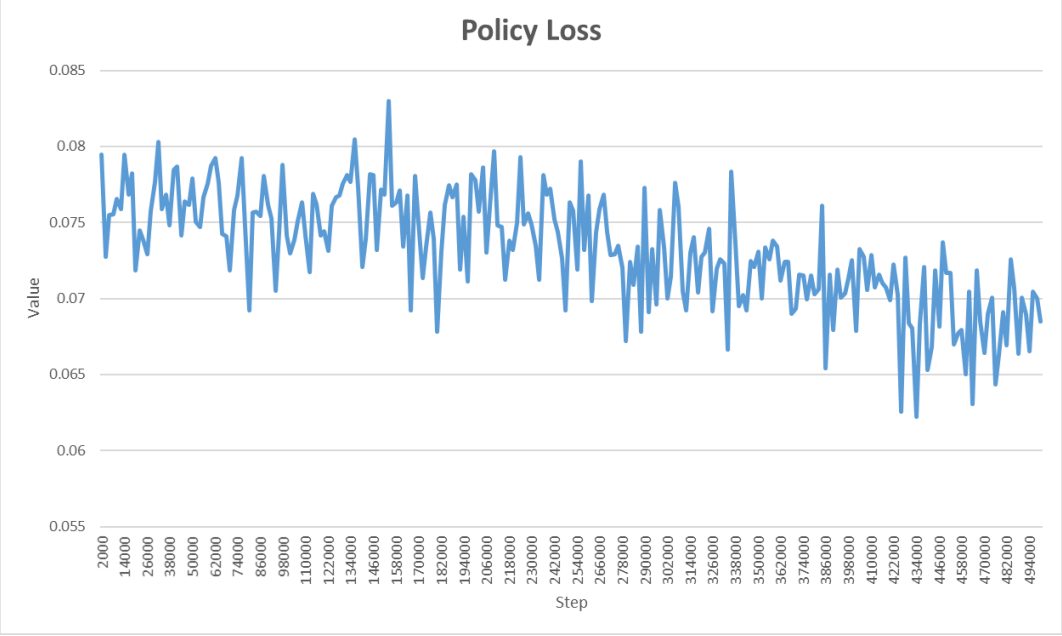

FIGURE 5.6: Policy change is slowly decreasing, but fluctuates wildly

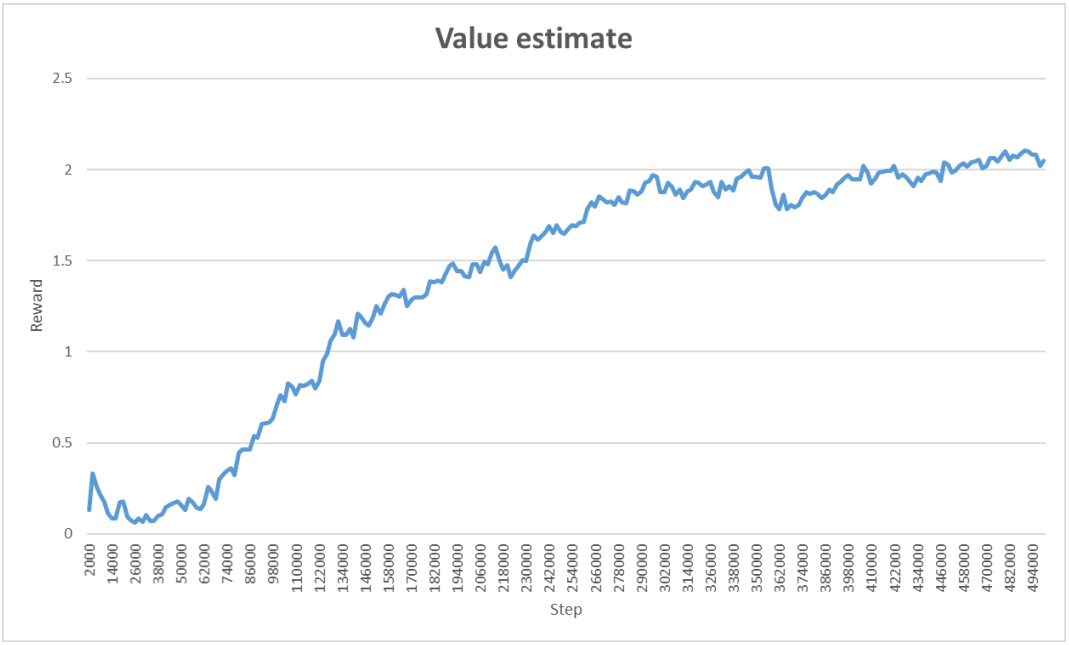

FIGURE 5.7: Model is getting better at value prediction, with the highest being 2.1

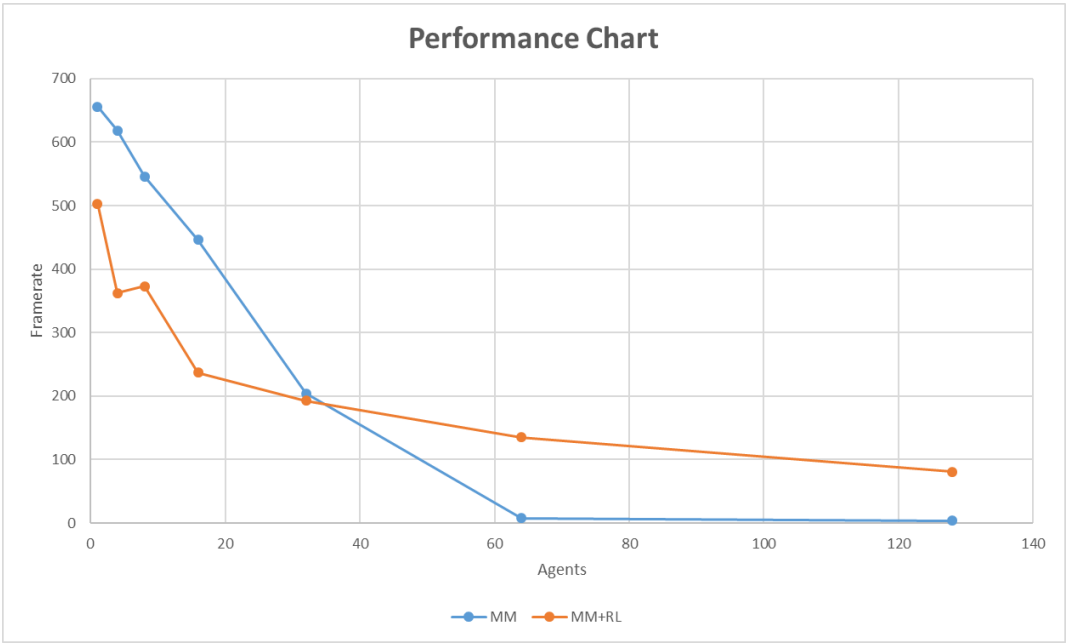

FIGURE 5.8: Performance is in favour of normal motion matching in the beginning, but drastically falls behind when agents increase in number

Agents	MM	MM+RL
1	656	503
4	618	362
8	546	373
16	446	237
32	204	193
64	7.5	135
128	3.7	81

FIGURE 5.9: The framerate in each test condition. After 32 agents, motion matching without RL slows down considerably.

Chapter 6

Discussion

The average cost of the best pose matched against the current pose was 9.44, resulting in decent accuracy of the system 5.1. However, the cost is still too high and minor mistakes do happen in the form of wrong poses being selected. This can be attributed to several factors like animation data and implementation. Only a handful of animation clips were used in the system, which drastically reduces accuracy and adaptability to a variety of situations. The cost functions are also not optimal, and have been rewritten and rethought a multitude of times. On the other hand, given the limited amount of animation, the accuracy is impressive as it is.

The training of the model was largely successful; the cumulative reward kept on increasing 5.3, which means that the agents were learning how to reach their target in the shortest time possible. The curriculum was also cleared relatively fast 5.4, however this means that it was not adequate enough i.e. it might have been too easy for the agents. The whole point of curriculum learning is to gradually introduce more difficulty, in order to increase the intelligence of the agents. If they clear it in a short amount of time, then for the rest of the training period they will remain on the same difficulty, considerably slowing down their learning as they keep learning the same thing over and over. A simple solution would have been to add more lessons with increasing difficulty; the goal radius is off-limits in this regard, as it cannot become smaller, but the introduction of new obstacles might work.

In terms of randomness, it would appear that the model slowly started to settle down over time 5.5, which is always a good indication of successful learning. Value estimation also kept on getting better as training went on 5.7, however the policy

kept on fluctuating, despite an overall decrease in change. This could mean that while that overall policies were getting better, it would fluctuate between them to figure out the best approach. Ultimately, a decrease is observed which is a good thing.

The biggest surprise is performance 5.8. It would appear that motion matching without reinforcement learning is faster when dealing with a select amount of agents. After a certain threshold, more specifically 32 agents, performance practically dies 5.9. In contrast, performance with reinforcement learning is still really good, even at higher agent counts. And even if vanilla motion matching is faster at lower agent counts, reinforcement learning is still fast in its own right. Theoretically, however, this does not make much sense. Reinforcement learning should be faster than linearly going though every pose in the pose database. This could be because of the lack of animation data. Motion matching works best with lots of animation clips and poses, however the implemented system is considerably below what is expected. The 241 poses used is far too little, but it could explain why the system is fast as there is far lesser poses to work with.

The performance differential could also be attributed to the fact the ML-agents is still very much in beta state. I also experienced great distress in getting it to work, a testament to its current state. As such, this will need to be revisited in the future. It is also worth it to point out, that vanilla motion matching runs impressively well, making it viable for single-character controllers in games. With a little more training, the reinforcement solution could also be used for background NPC-behaviour or something similar to that.

In regards to Replay Institute, they want the accuracy to be better before using it in their products.

6.1 Future works

The system as it stands now needs more training; Whilst improvement is observed, more training time with more powerful computers is needed to bring it even further.

The curriculum should also be adjusted accordingly. The accuracy of the system is good, but in order to bring it closer to the ideal average cost of 0, more animations need to be added.

The system could be extended to include swarm A.I behaviour, which would drastically reduce computations when dealing with large amounts of agents. Different types of loco-motions sets could be tested as well. Generative animation combined with motion matching could also be an interesting venue, in which estimated animations are split into poses.

Chapter 7

Conclusion

The aim of this thesis was to build an animation system that mitigated many of the drawbacks found in traditional systems, whilst still being performance-friendly. After researching relevant areas, we settled on motion matching trained with proximal policy optimization and formulated the following problem statement:

Is it possible to implement Motion Matching in Unity, and increase its performance using Proximal Policy Optimization?

After implementing, training and testing the system, we can conclude that this problem statement was only partially answered: While motion matching was successfully implemented, the performance gains of Proximal Policy Optimization was only noticeable when using large amounts of agents. As it stands, the system needs to become more robust and requires further training and testing before it can be used by Replay Institute (collaboration company).

Bibliography

Baecker, Ronald Michael (1969). *Interactive computer-mediated animation.* Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

Bellemare, Marc G et al. (2013). "The arcade learning environment: An evaluation platform for general agents". In: *Journal of Artificial Intelligence Research* 47, pp. 253–279.

Bengio, Yoshua et al. (2009). "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 41–48.

Burtnyk, Nester and Marceli Wein (1971). "Computer-generated key-frame animation". In: *Journal of the SMPTE* 80.3, pp. 149–153.

Buttner, Michael (2015). *Motion Matching - The Road to Next Gen Animation*. URL: https://www.youtube.com/watch?v=z_wpgHFSWss&t=658s (visited on 09/11/2019).

Clavet, Simon (2016). *Motion Matching and The Road to Next-Gen Animation*. URL: https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road (visited on 09/11/2019).

Cong, Matthew, Kiran S Bhat, and Ronald Fedkiw (2016). "Art-directed muscle simulation for high-end facial animation". In: *Symposium on Computer Animation*, pp. 119–127.

Cooper, Jonathan (2019). *Game Anim: Video Game Animation Explained*. A K Peters/CRC Press; 1 edition.

Csuri, Charles et al. (1979). "Towards an interactive high visual complexity animation system". In: *Acm Siggraph Computer Graphics*. Vol. 13. 2. ACM, pp. 289–299.

Engel, Wolfgang (2016). *GPU Pro 7: Advanced Rendering Techniques*. A K Peters/CRC Press.

– (2019). *GPU Zen 2: Advanced Rendering Techniques*. Independently published.

Eric Haines, Tomas Akenine-Möller (2019). *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress.

Hahn, James K (1988). "Realistic animation of rigid bodies". In: *Acm Siggraph Computer Graphics*. Vol. 22. 4. ACM, pp. 299–308.

Holden, Daniel, Taku Komura, and Jun Saito (2017). "Phase-functioned neural networks for character control". In: *ACM Transactions on Graphics (TOG)* 36.4, p. 42.

Johnson, Matthew et al. (2016). "The Malmo Platform for Artificial Intelligence Experimentation." In: *IJCAI*, pp. 4246–4247.

Juliani, Arthur et al. (2018). "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627*.

Karras, Tero et al. (2017). "Audio-driven facial animation by joint end-to-end learning of pose and emotion". In: *ACM Transactions on Graphics (TOG)* 36.4, p. 94.

Kolve, Eric et al. (2017). "Ai2-thor: An interactive 3d environment for visual ai". In: *arXiv preprint arXiv:1712.05474*.

Laird, John and Michael VanLent (2001). "Human-level AI's killer application: Interactive computer games". In: *AI magazine* 22.2, pp. 15–15.

Lample, Guillaume and Devendra Singh Chaplot (2017). "Playing FPS games with deep reinforcement learning". In: *Thirty-First AAAI Conference on Artificial Intelligence*.

Lee, Seunghwan et al. (2019). "Scalable muscle-actuated human simulation and control". In: *ACM Transactions on Graphics (TOG)* 38.4, p. 73.

Lengyel, Eric (2019). *Foundations of Game Engine Development, Volume 2: Rendering*. Terathon Software LLC.

Matt Pharr Greg Humphreys, Wenzel Jakob (2016). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers In.

Mnih, Volodymyr et al. (2013). "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602*.

Najim, Yousif Ali Hassan, Georgios Triantafyllidis, and George Palamas (2018). "Dynamic Fracturing of 3D Models for Real Time Computer Graphics". In: *2018-3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON)*. IEEE, pp. 1–4.

Noll, A Michael (1968). "Computer animation and the fourth dimension". In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part II*. ACM, pp. 1279–1283.

Olszewski, Kyle et al. (2016). "High-fidelity facial and speech animation for VR HMDs". In: *ACM Transactions on Graphics (TOG)* 35.6, p. 221.

Parke, Frederick I (1972). "Computer generated animation of faces". In: *Proceedings of the ACM annual conference-Volume 1*. ACM, pp. 451–457.

Samuel, Arthur L (1967). "Some studies in machine learning using the game of checkers. II—Recent progress". In: *IBM Journal of research and development* 11.6, pp. 601–617.

Schulman, John et al. (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Silver, David et al. (2017). "Mastering the game of go without human knowledge". In: *Nature* 550.7676, p. 354.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning, Second Edition*. MIT Press Ltd.

Waters, Keith (1987). "A muscle model for animation three-dimensional facial expression". In: *Acm siggraph computer graphics* 21.4, pp. 17–24.

Williams, Lance (1990). "Performance-driven facial animation". In: *ACM SIGGRAPH Computer Graphics*. Vol. 24. 4. ACM, pp. 235–242.

Wu, Yuxin and Yuandong Tian (2016). "Training agent for first-person shooter game with actor-critic curriculum learning". In:

Zhang, He et al. (2018). "Mode-adaptive neural networks for quadruped motion control". In: *ACM Transactions on Graphics (TOG)* 37.4, p. 145.