LEIPZIG UNIVERSITY

BACHELOR THESIS

Animation Authoring for Neural Quadruped Controllers

Author:
Paul STARKE

Supervisors:
Prof. Taku KOMURA, University of Edinburgh
Prof. Dr. Gerik SCHEUERMANN, Leipzig University
Dr. Daniel WIEGREFFE, Leipzig University

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Faculty of Mathematics and Informatics
Department of Image and Signal Processing

October 22, 2020

LEIPZIG UNIVERSITY

Abstract

Faculty of Mathematics and Informatics
Department of Image and Signal Processing

Bachelor of Science

Animation Authoring for Neural Quadruped Controllers

by Paul STARKE

Controlling characters via high-level control signal sequences is of high desire for game designers and storytelling tasks. In this thesis, an interactive user-system to author different quadruped character movements, such as locomotion and stylizations thereof, sneaking, eating, and hydrating will be created. The system leverages state-of-the-art technology of AI-driven character controllers and further advances their scope of user applicability for authoring animations. In particular, the thesis continues the recent work on MANN (Mode-Adaptive Neural Networks) for character control by providing the ability to carry out desired motion action types at a specified time or position. The relationship in which the corresponding parameters should be selected to achieve this goal will be shown.

Additionally, this work proposes a new dataset which aims to enhance synthetic motions when trained jointly with motion capture data. The approach is based on manipulating postures using inverse kinematics. The results show that the network is capable of learning the transition of the motion, although it was not captured. Experiments were conducted on the motion quality and path-following accuracy of the articulated character.

Acknowledgements

The writing of a thesis can be an isolated process, especially during COVID-19 lockdown. Therefore I would like to thank especially those people who stood at this time by my side as family, friends, acquaintance, or last but not least as supervisors.

In this sense, I want to greatly thank Prof. Taku Komura, Prof. Dr. Gerik Scheuermann, and Dr. Daniel Wiegreffe for supervising my thesis. I am thankful for the possibility to work on such a state of the art topic and codebase that researchers at the University of Edinburgh developed over the past years.

I further want to give my sincere thanks to Sebastian Starke, not only for his guidance and comments on my work but also for his valuable inspirations.

Along with him, I also want to extend my gratitude to all the lovely people and friends I have made, for making these everyday studies such a great experience and for making me feel home.

Finally, many thanks go to my parents for their everlasting backing and support that brought me through my whole life in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Character animation is a core application in the computer games and films industry, but also in areas of human-computer interaction and virtual reality. Often, motions that appear in nature are used as reference material by animation artists, and the goal is to reproduce those movements as close as possible. Those can then be used by traditional keyframe animation to play back sequences of movements in a user-defined manner. This is a job where professional artists must create a variety of similar movements, especially for the area of locomotion: They must create several realistic key-frames along a trajectory with detailed poses for all possible variations in movement speed, movement style, rotation, and acceleration, where manually synthesizing realistic patterns and transitions between movements can be difficult.

At this point, it is obvious to generalize such complex problems with suitable, intelligent, and efficient computer technology with the help of machine learning. Recently, deep learning has shown a number of successes by applying advanced deep learning models to character control tasks: Functions from high-level control signals that map to low-level character movements are learned by using motion capture data. For example, the user specifies a path as well as desired actions, and the system generates realistic movements that follow the control signals in the best possible way. Learning animations through demanding deep learning processes has gained enormously in relevance within various application areas.

1.2 Starting Point

Over the past years, researchers at the University of Edinburgh have developed a novel technique using neural networks to learn and animate various character movements in high quality from motion capture data [13, 33, 27, 26]. With their research, the computer can learn character movements itself based on motion capture data from quadrupeds and the user can interactively control the character at run-time. This framework [2] provides different character animations, e.g. running, jumping and sitting, which are generated in a responsive and realistic manner in real time.

1.3 Research Objectives

A major issue to generate the inputs for the neural network ultimately consists of mapping control signals during run-time from a keyboard or gamepad to continuous sequences of future control signals. In this work, an animation authoring system is therefore developed which enables to generate such control signals through a suitable user interface, which is particularly relevant for game designers as well as for the creation of film sequences. With that, it can be possible to provide more detailed representations of control signals that can better match the original data.

To learn various of character motions the MANN requires labeled input data, which are often provided via motion capture. This method provides the authors[33] with realistic data, but it's processing procedure is expensive and time consuming. Therefore, this work will append new data to learn new styles by combining the original motion capture data with new synthesized data that is generated by using full-body inverse kinematics. Using that technique, the research objective is to investigate whether such data augmentation can enhance the quality of the synthetic data when trained jointly with the original data, especially during transition movements that can be hard to create via inverse kinematics. Thus, the network shall learn a non-linear function that can regress the missing motion frames, as well as to transform synthetic movements into a more realistic motion.

1.4 Structural Outline

The outline of this thesis is structured as follows:

Chapter 2 discusses relevant artificial neural network architectures and methodologies along with an explanation of their functionality. Then, the technical components of kinematics are introduced, as well as a description of different terms that are relevant in the field of animation.

Chapter 3 continues with a more detailed presentation of related work, with a particular focus on kinematic methods and simulation-based methods for character animation. Those are presented with their benefits and limitations.

Afterward, Chapter 4 presents the developed animation authoring system and shows the method of motion editing with the use of inverse kinematics, the feature extraction, as well as the network training procedure.

Chapter 5 then shows the experimental results during runtime in Unity3D of the system with the newly trained model. In addition, the authoring system is evaluated in comparison to real-time character control.

Finally, Chapter 6 concludes with a summary, a discussion of limitations and some inspirations for future work.

Chapter 2

Background

Animation is a field which is strongly related to advanced knowledge in computer graphics, linear algebra, and computational methods. Since this thesis is based on the framework of a novel neural network and addresses terms for computer animation, the relevant knowledge will be provided in this chapter.

Starting with different architectures (2.1.1) and learning methodologies (2.1.2) for artificial neural networks, some fundamental components and procedures, such as regression and classification (2.1.3), backpropagation (2.1.5) and different activation functions (2.1.4) are described.

The chapter then introduces in the field of kinematics (2.2) which is commonly used for solving challenging character postures in character animation. This includes the term of character hierarchy (2.2.2), and coordinate transformation (2.2.1) between two spaces in which forward kinematics (2.2.3) and inverse kinematics (2.2.4) takes place. Also, a kinematic design of a quadruped geometry for character animation is given in Figure 2.4 which is used throughout this thesis. The particular challenges to this structure for inverse kinematics in animation are described in section 2.3.

Lastly, the fundamental concept of character control (2.4), as well as the benefits of using the three-dimensional engine Unity (2.5) is presented.

2.1 Artificial Neural Networks

ANNs (Artificial Neural Networks) are biologically-inspired models which are able to learn non-linear functions, while its previous version - the Perceptron [24] could only classify data based on a linear function. The network can learn by itself and produce an output that is not limited to the provided input. ANNs are composed of many nodes called neurons, which are connected to each other and function together, by passing information. Each connection is assigned a weight that represents its relative importance. They consist of a number of layers, including one input layer (receives external data), one output layer, and one or more hidden layers. The neurons of each layer calculate a function based on the received data. Their main advantage includes producing the output, even if the information is missing. ANNs are mainly being used

to solve many complex problems in real-time applications, such as image processing, speech recognition, language processing, translation, and time series forecasting.[10, 32]

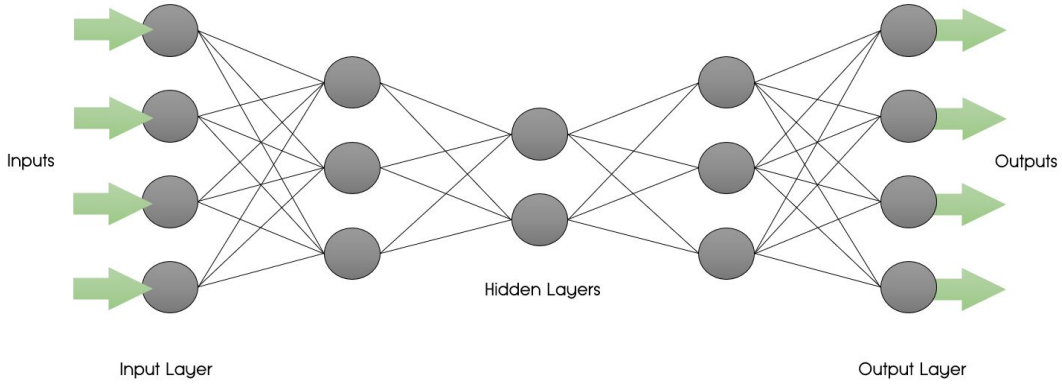


FIGURE 2.1: A fully connected deep feed forward neural network with four input and output neurons.

2.1.1 Architectures

ANNs can be divided into types based on the number of hidden layers they contain and how the neurons are connected. There is a variety of possible connection patterns between two layers:

- ***Fully connected:*** Every neuron in the first layer is connected to every neuron in the next layer, such as shown in Fig. 2.1.
- ***Pooling:*** A set of neurons in the first layer are connected to a single neuron in the next layer. This reduces the number of neurons in that layer. It helps to prevent over-fitting by providing an abstracted form of the representation, but also reduces the computational costs by decreasing the number of parameters for the output.[5]
  There are different types of pooling:
  - Max Pooling: Apply a max filter on a cluster. An example is shown in Fig. 2.2.
  - Average Pooling: Apply an average filter on a cluster.

2.1.1.1 Feed Forward Neural Networks

In a Feed Forward Neural Network (FNN) - as shown in Fig. 2.1, information always moves in only one direction, from the input nodes to the output nodes, without cycling or looping.

- ***Single-layer perceptron:*** It is the simplest type of neural network, where the inputs are fed directly to the outputs.

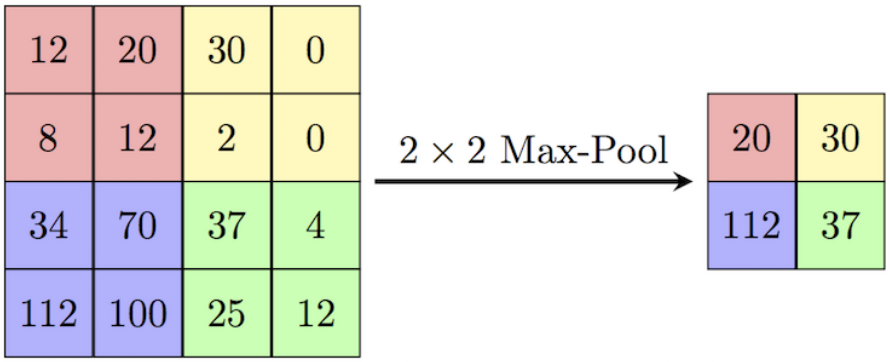


FIGURE 2.2: Max pooling with a 2x2 filter and stride = 2
[17]

- ***Multi-layer perceptron:*** Multiple layer network which requires a learning technique such as a backward propagation.

2.1.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) consists of one or more convolution layers, where convolution takes place instead of general matrix multiplication. Convolution is a special kind of linear operation on two functions that produces a third function. In general, the output of convolution is a sequence where each member of the output is a function of a small number of neighboring members of the input [10]. Therefore, the networks are specialized for processing a grid of values, such as image of variable size. CNN's may include pooling layers to optimize computation and are most commonly used in image and facial recognition applications. The learning process for these applications are surprisingly fast and they achieve high accuracy. [5, 10]

2.1.1.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are specialized in processing a large sequence of values, where each member of the output is a function of the previous members of the output. When applied to data involving time, the network may have connections that go backward in time. This is often represented as cycles in computational graphs to visualize the influence of a present value of a variable on its own value at a future time step. Also, neurons can be connected with neurons of the previous layer. This provides the network the ability to discover time-coded information in the data and make predictions along a time series e.g. predict the next motion.[10, 12]

2.1.2 Learning Methodologies

The main objective of a neural network is to have a model that performs well on both, the data it used for training (e.g. training dataset) and the new data on which the model will be used to make predictions. A model that learns the dataset too well does not generalize well to new data and is overfitted. On the

other hand, a model that is underfitted performs poorly on the training data, as well as on learning the problem in general. While underfitting can easily be addressed by increasing the capacity of the network, reducing overfitting tends to be more complex. Therefore, changing the network's complexity and training the network on more examples as well as adding statistical noise to inputs during training, can help.[10]

In general, machine learning algorithms can be categorized by how much experience they are allowed to have during training. In this work, the method of supervised learning is used.

2.1.2.1 Supervised Learning

Supervised learning requires labeled input data, so the network knows the desired output for each instance. The learning task is to produce the best possible output for unknown instances that rely on the trained dataset. The network predicts an output Y out of an input X and computes an error about the difference of the desired output and the prediction, based on a cost function. A well-known cost is a mean-squared error (MSE) as shown in equation 2.1, which tries to minimize the average squared error between the network's output and the desired output. Here, $\hat{Y}_i$ is the desired output along with the actual output Y_i of the neural network.[10]

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y)^2 \tag{2.1}$$

2.1.2.2 Unsupervised Learning

Unsupervised learning algorithms do not require any instances associated with a label or target. Instead, the input data is given along with a cost function. The method is used for problems when humans can't extract any structure or patterns in the data. A technique for finding such patterns is clustering: Its well known K-Means algorithm divides the dataset into groups of similar instances by forming clusters. [10]

2.1.2.3 Reinforcement Learning

Also, reinforcement learning differs from supervised learning in not needing annotated input and output pairs. Here, the algorithm depends and interacts with an environment to have feedback between the learning system and the experiences it gains. The main goal is to achieve the highest possible reward at any time to optimize the action of an agent, who interacts with the environment. Reinforcement learning is often used in video game applications to win the game.[10]

2.1.3 Regression and Classification

Both models take place in the method of supervised learning, but there is a main difference between them. While in regression the output variable is numerical or continuous, in classification it is categorical or discrete. Although a classification algorithm can predict a continuous value as well, the value is in the form of a probability for a class label. Classification predictions are evaluated by using their accuracy, while in regression a cost function such as the root mean squared error is used in order to analyze the prediction. In this work, the model of regression is considered to predict future motion features and compute them.

2.1.4 Activation Functions

The output of each neuron is calculated by the weighted sum of all inputs with an additional bias added to this term. This sum is called the activation and is passed through a nonlinear activation function. If the value exceeds a given threshold the neuron activates. In Figure 2.3 an example for a single neuron is shown. The bias allows the network to shift the activation function to fit the prediction with the data better.[10]

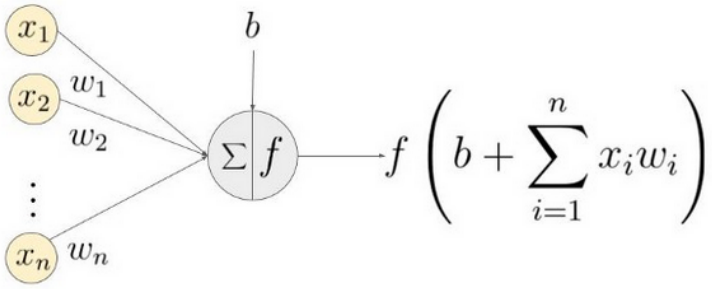


FIGURE 2.3: A single neuron with its corresponding inputs x_1 to x_n and weights w_1 to w_n, a bias b and the activation function f. [19]

Each layer of a neural network can be represented as a matrix

$$X' = A(Wx + b) \tag{2.2}$$

where A is the activation function, W a matrix, x the input vector and b the bias vector.
The table 2.1 discusses three different non-linear activation functions which are most widely used. Also, the choice of the function is heavily dependent on what the ANN is attempting to learn. Note that

$$z = b + \sum_{i=1}^{n} x_i w_i. \tag{2.3}$$

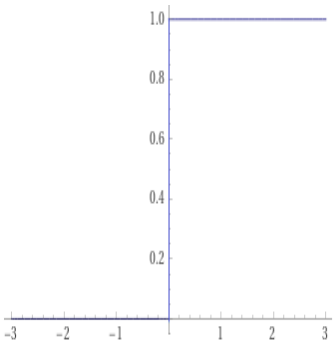
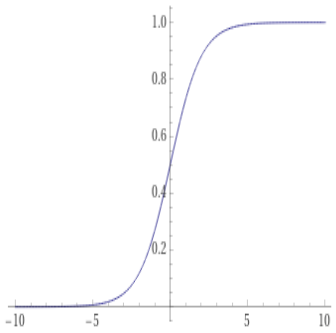
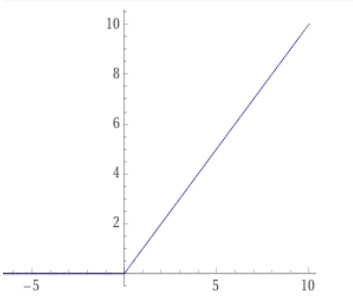
Name	Plot	Advantages	Disadvantages
Binary Step	$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$ 	• Ideal for binary classification	• Not usable for e.g. multi classification with more than two categories
Sigmoid	$sigmoid(z) = \frac{1}{1+e^{-z}}$ 	• Smooth gradient prevents "jumps" of output value • Normalizing the output of each neuron • Enables clear predictions for z above 2 or below -2	• Vanishing gradient - for very high or very low values of z, there is almost no change to the prediction.
ReLU (Rectified Linear Unit)	$relu(z) = max(0, z)$ 	• Computationally efficient • Non-linear, allows backpropagation	• Network cannot learn if input is equal or smaller than zero.

TABLE 2.1: Example's of Activation Functions for ANNs. [1]

2.1.5 Backpropagation

In machine learning, backpropagation is the most commonly used algorithm for training to compute the gradient of the cost function by including the weights of the network. The algorithm searches for the strongest descent in the gradient method and adjusts the connection weights to reduces the error found during learning. The error is "propagated" from the output back to the first layer. Furthermore, the network must be prevented from being overspecified, by introducing a learning rate. This rate defines how many steps the model needs to take, before adjusting the weights again. By increasing the learning rate, the training time decreases, but the model gains a lower ultimate accuracy. [10, 32]

2.2 Kinematics

For designers and artist, it is often easier to define a specific pose by moving parts of the object e.g. arms or legs, rather than directly change joint angles. This can be done by using kinematics. Therefore, kinematics has high relevance for various applications in computer graphics for creating and post-processing animations of virtual characters.[22]

2.2.1 Coordinate Transformations

Kinematics describes the transformation of geometry objects with respect to velocity, position, orientation, and acceleration, but regardless of mass, force, and torque. Coordinate transformation takes place between the space of joint angles and the Cartesian space (see Fig. 2.5), and can be specified using a homogeneous transformation of four-dimensional matrices or quaternion transformation of four-dimensional vectors.

2.2.2 Character Hierarchy

An animated figure as in Fig. 2.4 is modeled with a skeleton of linked rigid segments (bones) connected with joints from the root to the end effectors, called a kinematic chain. The end effector describes the last segment at the end of a kinematic chain. By moving one element of the character it is required to compute the joint angles of the other elements as well, to maintain the join constraints. An inverse kinematic (2.2.4) algorithm [28] can solve this problem in real-time by giving it a set of kinematic chains. [22]

2.2.3 Forward Kinematics

Forward kinematics takes as joint angles as input and calculates the Cartesian position and orientation of the end effector. The mapping from joint space to Cartesian space is straight forward and always returns a closed-form solution. This can be computed for any geometry and is often used for precalculations of inverse kinematic (2.2.4).

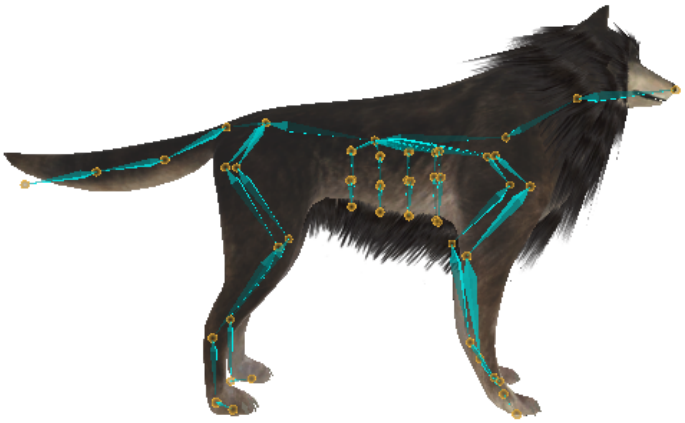


FIGURE 2.4: A visualized set of kinematic chains of a wolf game character. Rigid segments in blue are connected with joints in yellow colour.

2.2.4 Inverse Kinematics

Inverse kinematics describes the opposite of forward kinematics and hence it takes the Cartesian end effector position as well as orientation as input and calculates joint angles. This problem is not trivial and may not have a unique solution, especially when the system is redundant. For instance, given positions of the hands and feet of a character, there are many possible character poses that satisfy the chain-constraints.[11] Some poses are more likely than others for reaching a target in front of the character e.g. the arms will most likely move with its whole body, instead of keeping the rest of the body fixed. Defining a path or trajectory which the animated character has to follow, can help to return a type of configuration. In general, inverse kinematics complexity increases with the number of serial joints. [22]

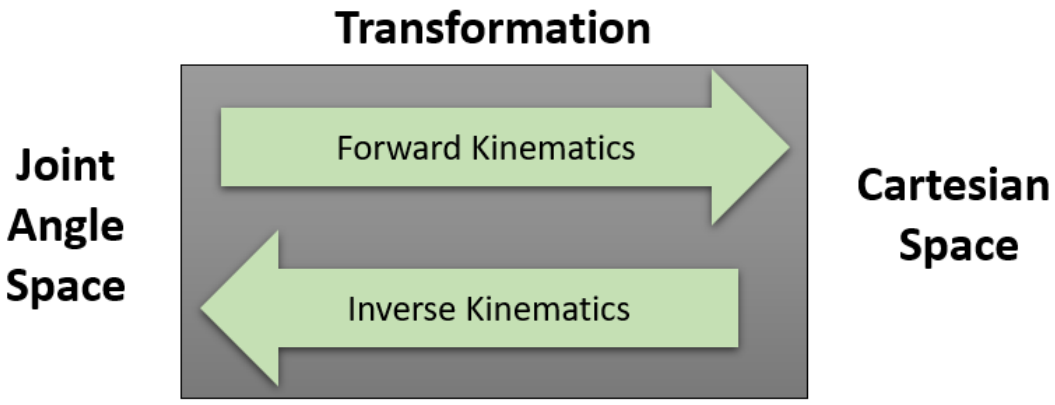


FIGURE 2.5: Mapping between joint angle and Cartesian space by forward and inverse kinematics.

2.3 Quadrupeds

Quadruped geometries as shown in Fig.2.6 are a challenging task for inverse kinematics in animation. Since they have four legs and multiple dependent joint chains, the root pose of the hip needs to be incorporated. To create animations and generate locomotion sequences, the joint couplings between fore- and hind legs need to be updated. Thus, a whole joint hierarchy must be solved simultaneously in real-time by using e.g. a full-body inverse kinematic algorithm.

FIGURE 2.6: A quadruped geometry using a wolf game character.
[30]

2.4 Character Control

Character control describes the problem of mapping from a set of control variables, such as the desired trajectory, to character motion which is both natural and responsive. This mapping can be done by data-driven methods such as a neural network, key-framing, motion matching, or any other method to generate low-level character motions out of high-level control signals.[13]

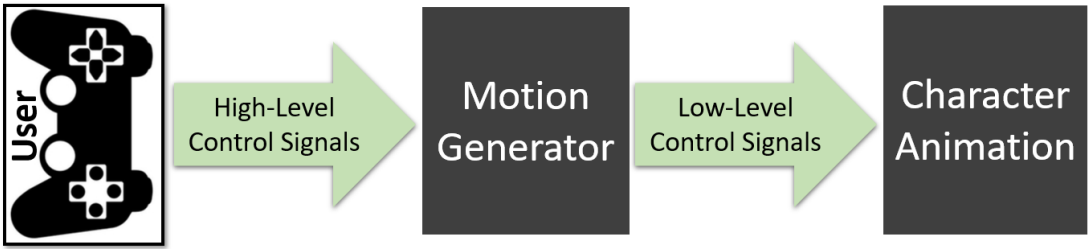


FIGURE 2.7: Character Control: The user gives simple high-level control signals e.g. a forward direction by a key on a gamepad controller, and a motion generator computes all required features (position, velocity, direction,...) for character movement.

2.5 Unity 3D

Unity is the creator of the world's most widely-used real-time 3D (RT3D) development platform, giving content creators around the world the tools to create rich, interactive 2D, 3D, virtual reality (VR) and augmented reality (AR) experiences. The platform was built in C++ and optimized since 2005 for performance. Therefore, many AAA game studios such as Ubisoft are working on this platform validating its reputation being a high-quality game engine. Building on the research from the authors of MANN [33] with their code-base published in Unity, the new authoring system is developed on the platform as a module component as well. Unity provides a workspace that combines artist-friendly tools with a component-driven design that makes development and game design more intuitive. Unity editor tools allow to simultaneously handle inputs for mice, keyboards, and game controllers which is especially desirable for the authoring systems purpose.

Chapter 3

Related Work

This chapter provides a literature overview of existing state-of-the-art methods for character animation. This specifically includes techniques both from kinematic and simulation-based methods in order to explain, discuss, and compare their strengths and limitations.

First, section 3.1 and 3.2 mention some basic data-driven animation methods to achieve interactive character control. Section 3.3 then discusses the counterpart of data-driven character animation techniques that make use of physics and simulate motions with respect to the character's environment. Afterwards, section 3.4 reviews the research results of a popular heuristic algorithm for character animation and a gradient-based method for numerical inverse kinematics. Lastly, section 3.5 presents two novel learning-based methods using deep-learning. The approaches are then further discussed in section 3.6.

3.1 Motion Graphs

The most classic way of achieving character control for games is to make use of motion graphs methods [16]. Their basic idea is to split motion data and cut them into clips. This is usually done by an animator setting keyframes (see section 3.1.1) to define where the motion transitions are between those clips. The decision on how to transition between the clips during run-time is either based on some automatic process or a state-machine. In practise, there is usually a huge state-machine and a big blend tree, which is blending between different sorts of motions. The state-machine is controlled by user inputs from a gamepad. A big problem with this method is that it requires a large number of animator inputs and manual processing by recording all the data, keyframing it, and labeling the sequences to define the states and store them into the state-machine.

3.1.1 Keyframe Animation

Some motions can usually be decomposed into small sequences by setting keyframes at their extrema. The positions of the character at the frames in between can then be interpolated. Keyframing is the original main method to produce computer animation and has been used for many years. However, not any motion can be done with this technique, due to non-smooth transitions

which make an animation look unnatural. The technique is straightforward and powerful, but also time-consuming and the results often depend on the skill level of the artist or animator. [3]

3.2 Motion Matching

Motion Matching is a simple and powerful method that fixed the problem of motion graphs, because it doesn't require any state-machine and much manual work once it's set-up. After recording motion due to motion capture the data is stored in a data-structure, such as a KD-tree. At runtime, this mocap-database is used to find the frame which matches the current pose and the desired future trajectory best, using a nearest neighbor search. This brute-force approach is implemented with a cost function to match the best candidate in the database. However, motion matching works best with lots of motion capture data, because it can only display motions that were captured. This comes with the cost of large memory usage, which only gets worse when the system and the environment conditions grow. This problem can be solved by training a neural network model, such used as in [13] and [33]. Motion Matching is the current standard in the game industry.[6]

3.3 Physics-based Animation

Static methods such as keyframing can't visualize complex interactions with the environment and struggle to accomplish lifelike character motion. Techniques in physics-based character animations allow dynamic animation with respect to user interaction, events, and the environment, by giving physically-based constraints. Simulated physics to create interactive character animation are studied in [9]. In [23], a reinforcement learning framework is presented which learns a policy that enables a physically simulated character to perform a task or action, while imitating a reference motion. Their technique is capable of a wide range of motion skills (from walks to highly dynamic kicks and flips), including locomotion, acrobatics, and martial arts. It is usable for a variety of models and allows motion re-targeting to different environments.

Also in [29] a technique is presented, where the gait of an animal is predicted of which no real-world data is available. This is especially useful to generate motions where no motion capture data can be obtained e.g. non-existing animals. Their method works for a variety of bipeds and quadrupeds and adapts the motion style to the size and shape of the animal. The trained model of motion styles is learned from a database of pre-captured animal gaits from online video sharing sites, such as YouTube and Flickr. The styles of the gaits they obtained are transferred on animals (dinosaurs and other extinct creatures) by using a joint inverse optimization algorithm with a regression function. This allows prediction of motions where the character appearance is known but the motion itself is unknown.

However, this approach only considers locomotion of the character and animations for actions e.g. turning the animals are not generalized. Also, they have struggles in producing realistic movements when responding to fast user input signals. Furthermore, the general downsides of physic-driven-kinematic methods are still that simulated characters can fall over easily and glitches may occur in video games when combined with reinforcement learning.

3.4 Inverse Kinematics

The inverse kinematic (IK) system in [11] can generate poses in real-time, even though it's not full natural which is impractical for the field of robotics, but especially desirable for character animation: Their human model can satisfy very specific constraints in video games (e.g., catching a ball or reaching a base) during run-time. In order to find the pose that is most similar to the space of poses in the training data, an algorithm called coordinate descent (CCD) is used. This algorithm solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes. [31] However, the algorithm is mainly optimising joint positions rather than joint values, which is often sufficient for applications such as motion capture and animation editing as well as post-processing. Also because the iterative steps of CCD-based IK solvers are heuristically driven, accuracy is normally the price paid for speed. A similar approach to the CCD techniques are differential- or gradient-based techniques [34] which also require multiple steps to find a solution, but are known for their ability to achieve high accuracy within a proper short amount of time. Many researchers have proposed hybrid techniques to allow efficient real-time IK solving, such as the in [18] presented Jacobian-based IK solver. The system of Jacobian relates small changes in joint configurations to positional offsets, which was also successfully used for real-time character animation tasks in [15]. They achieved good results on different articulated geometries.

3.5 Neural Networks

To overcome the limitation of scalability in the approaches of motion graphs and motion matching, neural networks are a novel used solution due to their virtual unlimited data capacity. Once they are trained they have a really fast run-time and don't use much memory, because only the network weights have to be stored. Also, a neural network can generalize and adapt to new run-time situations which can make a system flexible and improve smoothness. The downside of neural network systems is the long training time of the model which can go up to several days. This is especially problematic for industrial purposes because the whole network has to be trained from the start when adjustments in the data need to be done (e.g. stylization). Also, it is difficult for artists to edit and direct outcome of this kind of setup. Furthermore, it is hard to predict what the results will be like and why.

In order to generate motions, the network needs to learn the mapping from a user control signal to a motion. A desired user input or trajectory represents variables for multiple frames. For instance, a variable in the frame can be the speed or turning angle of the character. Learning character motion with RNNs that predict the next posture in the future from the previous frame are attractive frameworks for learning time series data such as human motion but are known to suffer from converging to an average pose (ambiguity). [8]

3.5.1 Phase-Functioned Neural Networks

In [13] an approach is presented where the weights of the neural network are generated as a function of a defined phase with multiple states. This phase function augments the learning with timing information of a given motion. The "phase" is a scalar variable in the range of 0 to 2π representing the timestamp in a locomotion cycle. Because the network learns when motions start and when they end it solves the problem of ambiguity which describes a mapping of the same input that maps to multiple different output motions. Ambiguity causes floating artifacts in the average pose during runtime. Their method leads to high-quality animation for biped locomotion with clear motion cycles.

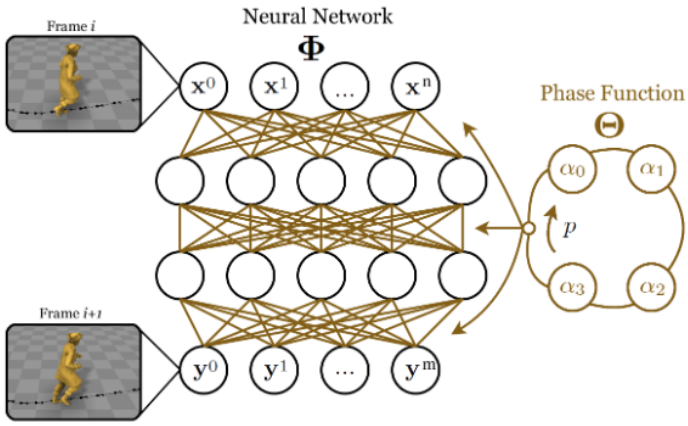


FIGURE 3.1: Architecture of the PFNN. The cyclic phase function with four states is shown on the right side. It generates the weights of the regression network.
[13]

3.5.2 Mode-Adaptive Neural Networks

When applying the PFNN framework in [13] to quadrupeds it fails because defining a phase for all four legs is undesirable within the transition between gaits of very different locomotion styles e.g. walk, pace, trot, and canter. Therefore, this thesis is based on the novel network architecture called Mode-Adaptive

Neural Network (MANN), cause it can learn from unstructured quadruped motion capture data. The authors in [33] present a time series model, that can predict the current state for characters who produce a wide range of periodic and non-periodic movements, such as quadrupeds. In general, the basic idea of their regression system is to automatically update weights of a neural network which are learned by neural networks as well. This differs from other neural networks for animation because the weights are interpolated instead of the output.

The MANN consists of two networks (see Figure 3.2) - the motion prediction network and the gating network. While the motion prediction network can compute the character's state in the current frame by using the state of the previous frame and a set of future control signals, the gating network is used to dynamically update the weights of the motion prediction network by interpolating consistent expert weights. Each expert weight is in a form of a neural network and trained to specialize a particular movement. This makes various types of locomotion possible, which are required to animate quadrupeds. For this, the gating network receives a motion feature $\hat{X}$ as input, which is a subset of X - the state in the previous frame and the user control signals. The architecture of the network is further described in section 4.3. [33]

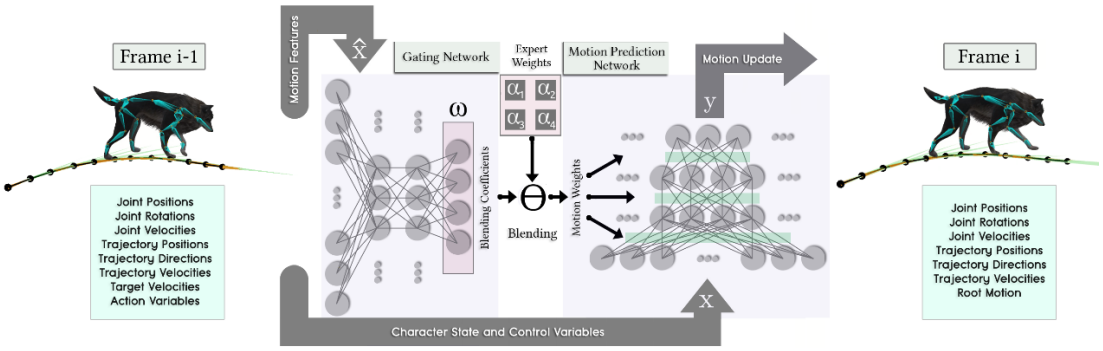


FIGURE 3.2: Architecture of the Mode-Adaptive Neural Network. The gating network takes the previous frame features of the character as input, such as foot velocities and computes the blending coefficient of four expert weights. The motion prediction network takes the characters posture and user's input control variables from the previous frame as input and predicts the updated posture and trajectory of the current frame. [33]

Note that the authors showed a locomotion authoring method in their results, but without any framework or user interaction.

3.6 Discussion

When a large amount of data is available, kinematic methods such as keyframe animation [3, 16], motion matching [6] and machine learning [13, 33, 26, 27] can be effective in character animation. Given a dataset of high-quality motion clips, data-driven kinematic methods will often produce higher quality motions than most simulation-based approaches such as physics-based methods [9, 23, 29]. However, for novel situations and complex environments, their ability is often limited. Collecting enough motion capture data to cover most situations can be very difficult and time-consuming. Another aspect is that many motions and actions, such as punches and kicks for box fights are often performed unrealistically in mo-cap-suits. The artificial motion capture environment makes it hard for actors to reproduce identical movements that would happen under real conditions e.g. playing soccer in a stadium. Also, there are no reliable motion capture systems for most animals: Yet it is not possible to track the motions of a spider with a mo-cap suit.

Chapter 4

Animation Authoring

This chapter presents the developed Authoring System. The concept of the system is mainly motivated by the literature review in previous chapter 3, and can be used to design paths and actions for AI-driven character animation. The following sections are based on the research that was published in [13, 33].

This chapter starts with the authoring system in section 4.1 which first presents an overview of the framework and then describes the implementations in Unity3D for the control signal as well as the corresponding user interface that is ready to be used in sections 4.1.1 and 4.1.2.

Followingly, data processing is presented in section 4.2. This particularly describes the editing of a starting pose to build new synthesized postures. Furthermore, it covers all steps of extracting the features from those motions to create a dataset for the neural network.

Section 4.3 then provides a detailed mathematical description of the input and output formats, the motion prediction network, as well as the gating network along with some explanations for the training phase of the network.

Finally, section 4.4 describes the motion synthesis, including motion and performance results during runtime in Unity.

4.1 Authoring System

Usually, animators must go through several steps when it comes to animating quadrupeds. This is because of their wide range of complex movements. Although the work of the MANN proposed a method to generate quadruped motion and construct controllers in a data-driven fashion, their framework is only capable of controlling the character during runtime. Therefore, the aim of this work is to further advance their scope of user applicability for authoring animations. The proposed animation authoring system seeks to extend the traditional understanding of character control by the ability to handle offline control of the character, but to also be responsive in real-time, and to carry out desired motion action types at a specified time or position. The system should allow people without programming knowledge easily to define paths for the

motions to be synthesized by the MANN model, as well as to transition between actions and styles, and to remain at states for a certain time. A further intention is to provide more detailed and realistic representations of control signals for achieving smoother transitions between motions.

The general scheme of the proposed authoring system using the MANN is visualized in Fig. 4.1, and represents an animation framework for offline character control. The user interface enables the user to create 3D control points in the world by using a mouse or key as an input. Based on these control points the authoring system computes a path for the character and predicts a future trajectory. In particular, this control signal consists of multiple features for the network's input such as positions, directions, velocities, styles, and speed. In addition, the previous pose of the actor which contains the bone positions, rotations, and velocities, is fitted into the network. The MANN then predicts the motions of the current frame, updating the root motion, the bone's positions, rotations, velocities, as well as the future trajectory. The output pose is post-processed to enable motions on terrain and reduce artifacts such as foot sliding by using inverse kinematics. Also, a trajectory correction is taking place in this step to improve motion quality and stability. Finally, the posture is assigned to the character which builds the updated pose of the current frame.

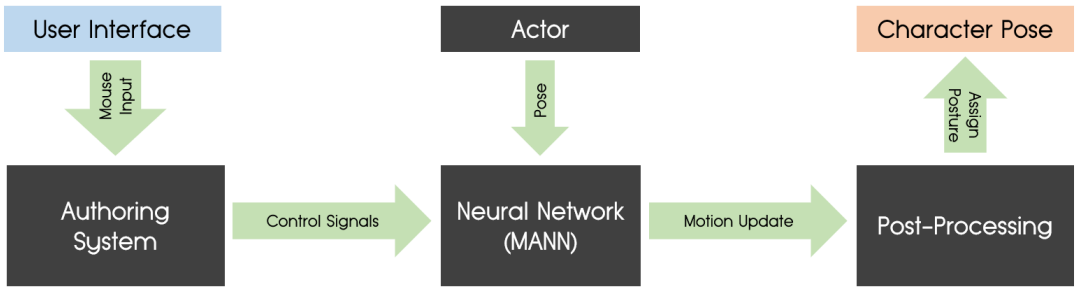


FIGURE 4.1: Pipeline of the Animation Authoring Framework.

4.1.1 Control Signal Generation

In order to generate a smooth path between a series of user-specified control points, the authoring system computes and stores points using the concept of centripetal Catmull-Rom spline interpolation. The name originates from Edwin Catmull, co-founder of Pixar, and Raphael Rom. This technique will not form loops or self-intersections within a curve segment and the spline follows its control points tight and smoothly. Furthermore, the points along the original set of points also make up the control points for the spline curve. Comparing this with the Bezier curve, which needs extra control points that are not a part of the spline itself. Here, the curve goes through its control points defined by four control points P_0, P_1, P_2, P_3, with the curve segment drawn only from P_1 to P_2. This can be done by solving a cubic polynomial as (4.1), where $p(t)$ is the path in space and a_0, a_1, a_2, a_3 the vector coefficients. To get the tangents of

p with respect to t, simply differentiate p as (4.1).

$$\begin{aligned} p(t) &= a_3 * t^3 + a_2 * t^2 + a_1 * t + a_0 \\ &= \sum_{k=0}^{3} a_k * t^k \\ p'(t) &= 3a_3 * t^2 + 2a_2 * t + a_1. \end{aligned} \tag{4.1}$$

The parameter t is a number between $[0, 1]$ for the distance between the curve moving from P_1 to P_2 as shown in Fig. 4.2. For instance, if t=0 the spline is exactly at the same coordinate as P_1 and at t=1 exactly at P_2. Followingly, equation 4.1 is examined to express some constraints:

$$p(0) = P_1 = a_0 \tag{4.2}$$

$$p(1) = P_2 = a_0 + a_1 + a_2 + a_3 \tag{4.3}$$

$$p'(0) = 0.5 * (P_2 - P_0) = a_1 \tag{4.4}$$

$$p'(1) = 0.5 * (P_3 - P_1) = a_1 + 2a_2 + 3a_3 \tag{4.5}$$

Then, subtracting (4.2) and (4.4) into (4.3) and (4.5) to get equations where a_2 and a_3 are the only variables:

$$\begin{aligned} a_2 + a_3 &= (P_2 - P_1) - 0.5 * (P_2 - P_0) \\ 2a_2 + 3a_3 &= 0.5 * (P_3 - P_1) - 0.5 * (P_2 - P_0) \end{aligned} \tag{4.6}$$

Finally, subtracting these two equations to solve for a_2 and a_3 to get all variables:

$$\begin{aligned} \mathbf{a_0} &= P_1 \\ \mathbf{a_1} &= 0.5 * (P_2 - P_0) \\ \mathbf{a_2} &= 3 * (P_2 - P_1) - 0.5 * (P_3 - P_1) - (P_2 - P_0) \\ \mathbf{a_3} &= -2 * (P_2 - P_1) + 0.5 * (P_3 - P_1) + 0.5 * (P_2 - P_0) \end{aligned} \tag{4.7}$$

To conclude, group all the $\{P_k\}$ terms together. In general, a point P located between P_1 and P_2 are calculated then as:

$$P = \alpha * \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} * \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix} * \begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix}^\top \tag{4.8}$$

Note that α is set to 0.5 to achieve centripetal Catmull–Rom spline. The parameter affects how sharply the curve bends at the interpolated points. When α=0, the resulting curve is the standard uniform Catmull–Rom spline and with α=1 the product is a chordal Catmull–Rom spline. [4]

This technique is also used to interpolate the **styles** of the control points, to provide smooth transitions in locomotion. Therefore, each control point stores

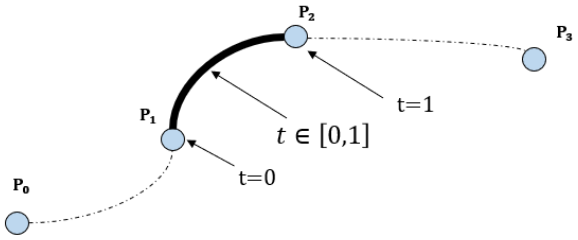


FIGURE 4.2: Catmull-Rom spline segment.

a number for each style in range $[0, 1]$ which are then interpolated by using these numbers instead of the position vectors. This number represents the motion action type for the character. During runtime, there is a high probability of launching the action if the value approaches 1. However, the speed, velocity, and direction feature for the trajectory are not computed as described above. Here, the **speed** is the length of the path starting at the point P and ended into one-second future. This describes the desired velocity at point P. Further, the **velocity** at each point is simply calculated as

$$P_{Velocity} = \frac{(P_t - P_{t-\delta})}{\delta} \tag{4.9}$$

in meters per second, where δ is the time interval between two points and t the current timestamp. Finally, the **direction** is defined as a normalized vector pointing forward to the next point. In particular, also note that all points with their features need to be computed once, and are then stored into a look-up-table, to allow interactions during runtime and to avoid recalculations of non-changing points. However, the system also can generate all features on-the-fly by giving a timestamp as input. This is especially required to create the authoring in the scene view such as shown in figure 4.4.

4.1.1.1 Trajectory Estimation

With the computed features described in the previous section, a control signal is built in form of a trajectory along the path. This trajectory consists of the ground point transformations, velocities in the past and future states, and an action type at those frames. More specifically the trajectory covers those features one second in the future and past by sampling them in each frame as shown in Figure 4.3. Later, these features are extracted for the network's input vector using 12 of those sampled frames of motion. In order to predict a future trajectory in every frame, the authoring system computes the current timestamp relative to the character's local transformation for sampling the spline-interpolated points one second into the future. For instance, this can be done by searching the nearest neighbor in a KD-tree or array.
During runtime, the future trajectory at the defined path is interpolated with that of the networks output, to produce character animation as realistic as possible. This is because there is a high probability that the user-specified path is

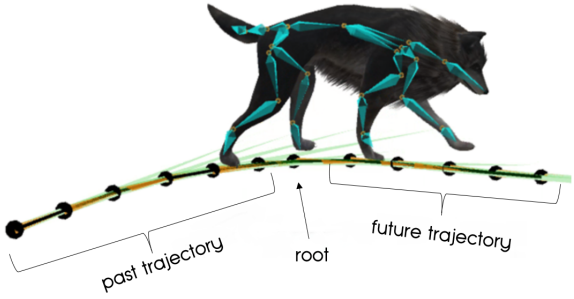


FIGURE 4.3: The visualized trajectory covering 6 frames in the past and 5 in the future.
[33]

unrealistic for the character motion in terms of position and high turning angles. Assume that F^* is a feature of the desired trajectory along the path and F^+ a feature from the updated corrected trajectory in the output of the network with $F^*, F^+ \in \{position, direction, velocity, styles, speed\}$. The feature F for the input trajectory is defined as $F = tF^* + (1-t)F^+$, where t is a blending parameter. Blending parameters are tuned and set sometimes depending on features, in order to achieve a better control signal. For example, $t_{direction}$ in $F_{direction}$ depends on F_{styles} with

$$t_{direction} = Min((1 - F_{style1})^{0.25}, (1 - (0.5 * F_{style2}))^1, \\ (1 - (F_{style3} + F_{style5} + F_{style6}))^{0.5}) \tag{4.10}$$

where F_{style3} to F_{style6} are actions with almost no velocity (sitting, lying, standing), while F_{style1}, F_{style2} are locomotion styles. The more the character falls into a state with no locomotion, the lower $t_{direction}$ gets and consequently results less blending in $F_{direction}$. This leads to less control from the authoring by using the last predicted output of the network. The result is a smooth trajectory, which is necessary for the next input of the neural network to achieve good looking transitions between motions.

4.1.2 User Interface

The Unity3D implementation consists of a single component that can be added to any game object. The component provides a user-friendly customized inspector for the authoring, which can be entirely used without any programming. The custom inspector for the script component is shown in Fig. 4.5. The first variable defines whether a new control point should be created or not. This is because it is not always desirable to have a raycast visualization for the new control point (see Fig. 4.4). The user can create a new control point at his cursor's position by pressing the key he defines with the second variable. Furthermore, the first variable in the "ControlPoints" section defines whether the authoring should cycle or not. The next parameters are used to let the user specify the desired time in seconds for each frame and a time interval between

two control points. Delta-time is set default to $\frac{1}{60}$ for $60Hz$ at runtime. The next box of variables specifies the color of each style, which the user can change by using a color picker. Next, every existing control point named with its index is presented as an extendable group. Each control-point is a game object with its transform, containing position, direction, and rotation. Also, the user can specify a layer-mask in the "Ground" variable for each control point. This allows placing the points on different layers in the scene. Each style value is in between $[0, 1]$ which represents the style or action at the given control point. Furthermore, it is desirable to have a motion time for some styles such as sitting, standing, lying, idling, eating, and hydrating to remain at states for a certain time. Therefore, the last parameter defines the time in seconds to perform the motion at the control point. Note that it is possible to apply changes to the parameters and variables of the authoring during runtime, as well as to enable, disable, add or remove them for dynamic interactive tasks. For instance, the user can change the position of the points via drag-and drop in the game- and scene-view.

Visualization of the authoring in editor mode is shown in Fig. 4.4. Each control point is represented as a grey sphere and labeled in order of the path. The interpolated points in between are connected by colored lines based on their style values and corresponding style color in the inspector. Note that there is an indicator at the cursor's position which shows how the path would look like if the user creates the new control point.

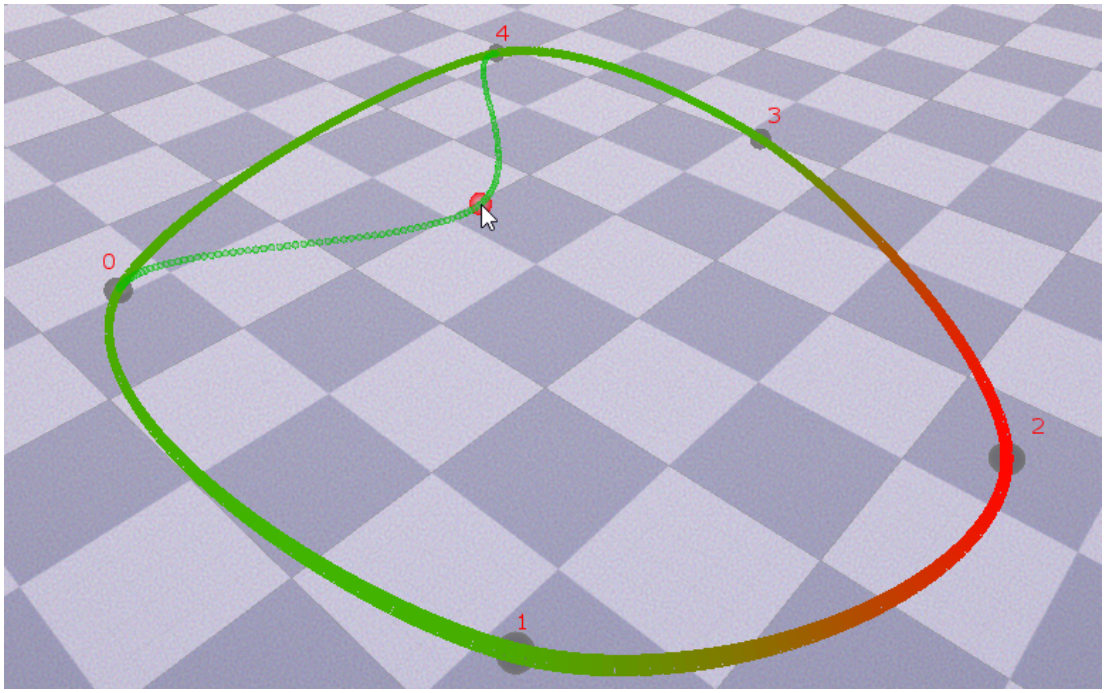


FIGURE 4.4: Result of the inspector settings of Fig. 4.5. Each control point is visualized as a grey sphere. The colored lines represent the spline-interpolated points.

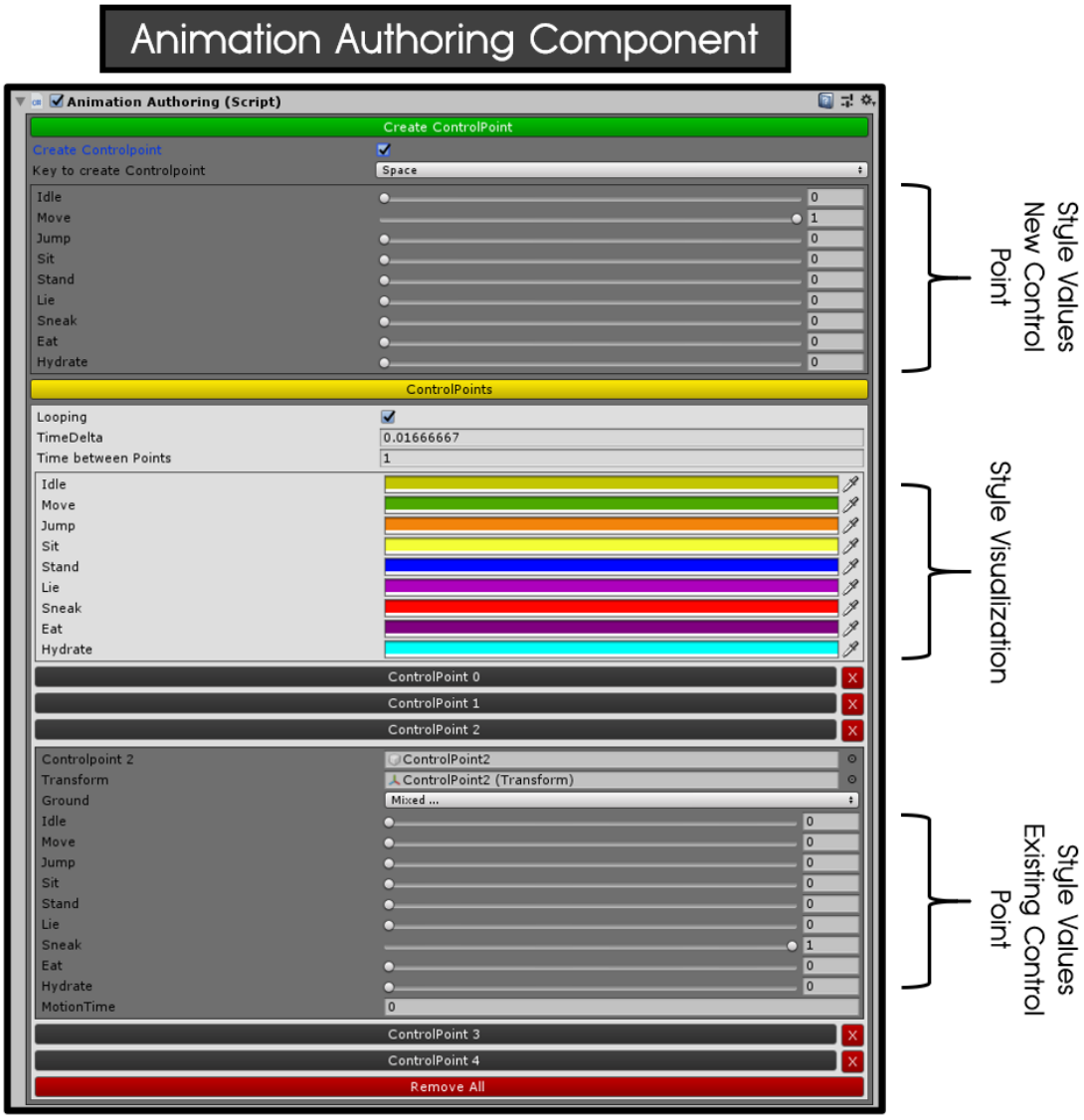


FIGURE 4.5: Inspector of the Animation Authoring System.

4.2 Data Processing

This section covers phases of data processing to provide data for the neural network. Therefore, the motion capture data [20, 21] from the authors is used which consists of 30 minutes of unstructured dog motion capture that includes various of locomotion styles such as walk, pace, trot, and canter, as well as other types of motions: sitting, lying, idling, standing and jumping. Here, this data is augmented with new synthesized data covering new styles that are generated by using full-body inverse kinematics. Therefore, section 4.2.1 describes the motion editing to create different types of character postures such as sneaking, eating, and hydrating, which are then post-processed within the original motion capture data. Followingly, the data preparation, motion classification, and feature extraction are described in section 4.2.2.

4.2.1 Motion-Editing

This section presents three new motion types that can be hard to obtain using motion capture but are easy to create via inverse kinematics. Intuitively, the basic concept of solving inverse kinematics on the full body is not used for character animation, because it is inappropriate for creating natural and smooth transitions between motions. However, it is a useful technique to result in an appropriate final posture for the character, by finding a suitable configuration of joint values and satisfying a given set of objectives. Building such posture, the network might be able to learn transition movements on its own by using the original data.

Since the skeleton structure of the wolf model is composed of 27 bones in total as shown in Figure 4.6, a whole joint hierarchy must be solved simultaneously. The joints are represented as raw transformations between bone segments and they can be controlled fully based on position and orientation information in joint space. In order to create natural motion, a joint is spheric and has a DoF (Degree of Freedom) of 3, which allows rotational motion about all three axes. The rigid bone segments have a DoF of 0 to not allow any motion but are used to define fixed connections between segments. The final DoF of the wolf model is 81 in total, which is calculated as the sum over all motion axes that are influencing the segment of the body. Note that, a higher DoF increases the amount of redundancy for available joint variable configurations to reach identical end effector poses.

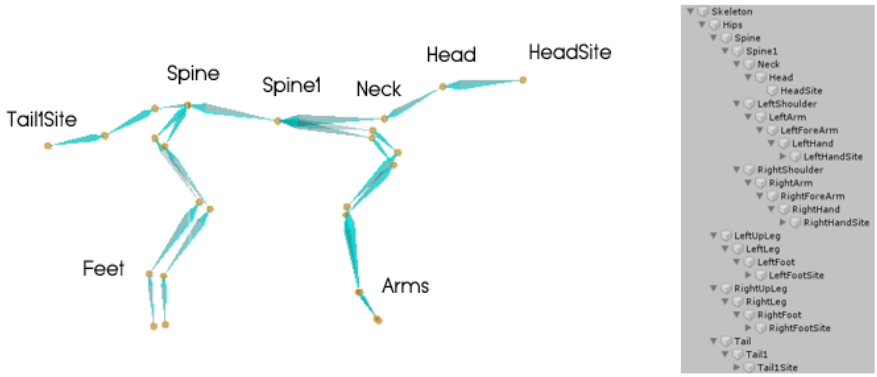


FIGURE 4.6: The labeled skeleton structure of the used wolf model along with its joint hierarchy.

In order to create poses, the "Ultimate Inverse Kinematic" algorithm [28] is used, which is a C# implemented full-body generic CCD solver for Unity. The algorithm requires a root transformation as input along with the end effectors to build a model. Each model can solve the kinematic chain from the root to its end effectors depending on the targets. Note that a target can be set for each objective and an end effector can also represent an intermediate segment within the kinematic tree. In order to obtain the pose of the single bones, the transformations along the hierarchy are calculated recursively starting from the root. The following postures are based on the starting posture in Figure 4.8 and are oriented towards the natural reference images showed in Figure 4.7.

FIGURE 4.7: Nature references of the sneak, eat and hydrate postures.
[25, 7, 14]

4.2.1.1 Sneak-Pose

The sneak posture is simply built by decreasing the height of the hip about $\frac{1}{8}m$ of the starting posture. Then, one model for each leg is created with the UpLegs and Shoulders (see Fig. 4.6) as root transformations. The arms and feet transformations are the end effectors with their position set as targets. All four models are solved which already leads to a sneak motion. Lastly, the neck and head are rotated 20 degrees, as well as the tail about 25 degrees around the z-axis. The result is shown in Fig. 4.9 where the top represents the starting posture and below the corresponding sneaking pose.

FIGURE 4.8: Starting Posture that is modified to create a sneak, eat and hydrate posture.

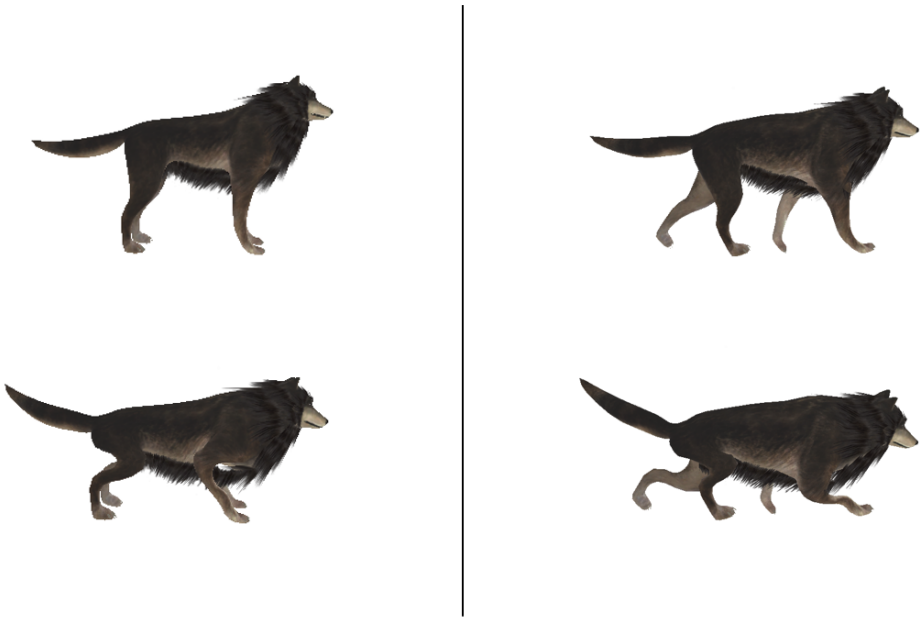


FIGURE 4.9: Sneaking posture synthesized by using full-body inverse kinematics. The top row shows the starting posture. The bottom row shows the corresponding sneaking pose.

4.2.1.2 Eat-Pose

Here, one model is solved with the Spine as the root and LeftHand, RightHand and HeadSite as the end effectors. The Head is targeted to a "food" object that is placed 0.5m in front of the character.This achieves that the character is looking at this particular point while other body components such as the tail can move. Note that the feet are clamped on the ground to avoid movements later during runtime. Figure 4.10 shows the final result of this posture.

FIGURE 4.10: Eating Pose

4.2.1.3 Hydrate-Pose

Finally, Figure 4.11 presents the hydrate posture which is constructed in three steps. At first, the feet and hands transformations of the starting posture are stored. They are set as targets for all four leg end effectors after rotating the hip -25 degrees around the x-axis and decreasing the height. Once the kinematic chains of all legs are solved, the right hind leg must be stretched out. This is done by creating a target objective at the desired position which transformation is set relative to the whole skeleton. The right foot's position and rotation are then targeted to this object. After solving this foot model the second time, the neck is rotated -60 degrees around the y-axis as well as the head with -30 degrees to look back left such as in the reference of Fig. 4.7. Finally, the tail is lifted by rotating 25 degrees around the z-axis.

FIGURE 4.11: Hydrate Pose

4.2.2 Feature Extraction

This section describes the pre-processing of new data to train the network. Therefore, the features for the new dataset are extracted and the motions are classified. In particular, the aim is to create a new dataset that contains all synthetic postures of the previous section.

4.2.2.1 Dog Motion Capture

The motion capture data is provided by the authors in a Biovision Hierarchy (BVH) format, which is a popular format for motion capture datasets that includes information about the skeleton hierarchy as well as motion data. Therefore, the authors provide a BVH importer that allows importing motion data directly into Unity. Once imported, the motions are edited via a motion editor script which applies the motions by bone index. Note that all the original motion data was captured on flat terrain.

4.2.2.2 Motion Classification

The motion editor as shown in Fig. 4.12 is a modular tool for processing and annotating the data. The script loads all frames of all files and allows the user to explore the captured animation. All data and motions are visualised while scrolling through the motion capture sequences. The tool allows not only to save motion labels, but also corresponding scenes for each file. Settings can then determine if a clip should be exported or not, amongst other things such as the export frame range and rotations of joints. There are 23 out of 52 clips used to create the new dataset: 11 for sneaking, 8 for eating, and 4 for hydrating. The style module enables us to annotate the animation, create transitions, and add as many styles as required: idle, move, jump, sit, lie, stand, sneak, eat, and hydrate. For instance, the style module allows creating keyframes where the user can specify a style label. The value is then linearly interpolated between those keypoints. It is very powerful and simple to use, but the process is done manually.

In order to apply a new pose to the character using the motion data, the style posture is solved to the character movements when the label value is set to 1. Each type of label can be added to the object as a module that derives from an abstract class. All three new action motions are post-processed when the current frame is loaded. Note that, only one-hot labels are used, because the network should learn the transition. Also, the size of the data is doubled by mirroring the motions. The Trajectory Module visualises the corresponding trajectory to the motion of the character. Before exporting the data, each file settings should be checked in terms of correct trajectory, styles, forward axis, mirror axis, and joint rotations.

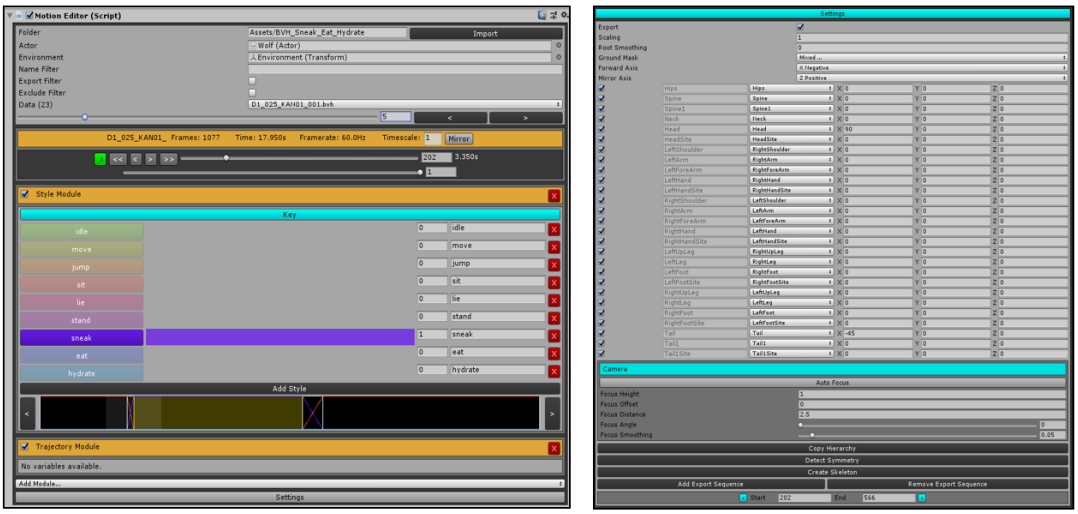


FIGURE 4.12: The Motion Editor's GUI consisting of the Style Module, Trajectory Module and setting options at the right.

4.2.2.3 Motion Exporting

The processed data can then be exported into an input and output file for training the neural network. Therefore, motions and annotations are exported to CSV format via a Motion Exporter script. The script loops through all frames in all files creating a state for the current frame and for the subsequent frame. Here, a state is a collection of information about an instant of time that is used to build the inputs and outputs to train the network. Note that the current frame is stored as the 6th(zero-indexed) point of the 12 points in the trajectory as in Fig. 4.3. The detailed input and output format for every frame is described in section 4.3.1. In this thesis, the motions are exported with a framerate of 60 that leads to 9.56min of synthesized data, which is mirrored to 19.13min.

4.2.2.4 Data Augmentation

After exporting the dataset with the new styles, it is augmented with the original data in a python script. Therefore, the script loads the original CSV and fills the $12 * 3$ missing style motion columns with zeros. This is done because the original dataset only has 6 motion action types. Filling the missing features allows to simply append the new data which builds the final dataset. Note that, having to many examples of one motion and too few of another can bias the training. The breakdown of the ratio of the motion data is shown in Table 4.1. In addition, the script computes contact labels for the feet in the output. This is desired later for runtime purposes to reduce foot sliding, where the label defines interpolation parameters for position and rotation. Here, the contact for each frame is defined as follows: Each foot connects to the ground if the height is below 2,5cm and the velocity is less than $1m/s$. Then the label is set to 1 at this instance, else 0.

Motion Type	time (sec)	frames	ratio (%)
idle	851.30	51078	15.77
move	2389.93	143396	44.27
jump	161.83	9710	3.00
sit	401.87	24112	7.44
lie	224.93	13496	4.17
stand	221.33	13280	4.10
sneak	586.30	35178	10.86
eat	288.97	17338	5.35
hydrate	272.50	16350	5.05

TABLE 4.1: The breakdown of the dataset for training. This dataset includes the original dog motion capture combined with the synthetically created sneak, eat and hydrate motion type.

4.3 Network Training

This section presents the neural network training process. More specifically, section 4.3.1 describes the format of the exported input and output CSV file via the motion exporter in section 4.2.2.3. Then, the architecture of the Mode-Adaptive Neural Network (MANN) in Fig. 3.2 is described, which is composed of the motion prediction network and the gating network. This includes a more detailed description than in section 3.5.2

4.3.1 Input and Output Formats

As mentioned in section 4.1.1.1, the features are extracted from 12 frames of motions covering the current frame and the frames one second in the future and past. The data is composed of the ground trajectory features at those frames and the body joint transformations and velocities of the current state. The input vector for the motion update at frame i is

$$x_i = \{t_i^p, t_i^d, t_i^v, t_i^{\hat{v}}, t_i^a, j_{i-1}^p, j_{i-1}^r, j_{i-1}^v\} \in \mathbb{R}^n \tag{4.11}$$

where $t_i^p \in \mathbb{R}^{2t}$ are the path trajectory positions of $t = 12$ samples in 2D-horizontal plane relative to the current state i, $t_i^d \in \mathbb{R}^{2t}$ are the forward facing directions relative to state i, $t_i^v \in \mathbb{R}^{2t}$ the trajectory velocities relative to state i, $t_i^{\hat{v}} \in \mathbb{R}^t$ are the desired trajectory velocity (speed) value of state i, and finally $t_i^a \in \mathbb{R}^{9t}$ are one-hot vectors for the character action (style) type. The relative joint positions, rotations and velocities of the previous state are $j_{i-1}^p \in \mathbb{R}^{3j}$, $j_{i-1}^r \in \mathbb{R}^{6j}$, $j_{i-1}^v \in \mathbb{R}^{3j}$ where $j = 27$ denotes the number of joints. Here, note that three more styles than the work in [33] are used. In total the input dimension of x_i is 516. Similarly, the output vector is

$$y_i = \{t_{i+1}^p, t_{i+1}^d, t_{i+1}^v, j_i^p, j_i^r, j_i^v, \dot{r}_i^x, \dot{r}_i^z, \dot{r}_i^a, f_i^c\} \in \mathbb{R}^m \tag{4.12}$$

where $t_{i+1}^p \in \mathbb{R}^{2t}$, $t_{i+1}^d \in \mathbb{R}^{2t}$ and $t_{i+1}^v \in \mathbb{R}^{2t}$ are the predicted relative trajectory positions, directions and velocities for the next frame of $t = 6$ samples in 2D-horizontal plane.$j_i^p \in \mathbb{R}^{3j}$, $j_i^r \in \mathbb{R}^{6j}$ and $j_i^v \in \mathbb{R}^{3j}$ are the relative joint positions, rotations and velocities for the current state. $\dot{r}_i^x \in \mathbb{R}$ and $\dot{r}_i^z \in \mathbb{R}$ are the root translational x and z velocities relative to the previous and $\dot{r}_i^a \in \mathbb{R}$ the corresponding root angular velocity in the 2D-horizontal plane. Lastly, $f_i^c \in \mathbb{R}^4$ are one-hot vectors for the foot contacts. This is similar to the data preparation published in [33], thought this thesis does output information about the foot contacts. In total the output dimension of y_i is 367. [33]

4.3.2 Motion Prediction Network

The motion prediction network has three layers - two hidden and one output layer. It receives the previous state x of the character, and outputs the data in the format of **y** as follows:

$$\Theta(x;\alpha) = W_2 ELU(W_1 ELU(W_0 x + b_0) + b_1) + b_2 \tag{4.13}$$

where α defines the whole network by

$$\alpha = \{W_0 \in \mathbb{R}^{h\times n}, W_1 \in \mathbb{R}^{h\times h}, W_2 \in \mathbb{R}^{m\times h}, b_0 \in \mathbb{R}^h, b_1 \in \mathbb{R}^h, b_2 \in \mathbb{R}^m | n, m, h \in \mathbb{N}\}.$$

Here, **h** is the number of units used in the hidden layers, **n** the dimension of the input vector and **m** the dimension of the output vector. In this thesis **h** is set to 512, **n** to 516 and **m** to 367. The activation function used is the exponential rectified linear function defined by

$$ELU(x) = max(x, 0) + exp(min(x, 0)) - 1. \tag{4.14}$$

The neural networks weights α are computed by blending K expert weights $\beta = \{\alpha_1, \ldots, \alpha_K\}$, where K is set to 8 for high-quality motions with sharp movements. Each expert is in a form of a neural network configuration: $\alpha = \sum_{i=1}^{K} \omega_i \alpha_i$ where $\omega = \{\omega_i, \ldots, \omega_K\}$ are the blending coefficients computed by the gating network described in Section 4.3.3. [33]

4.3.3 Gating Network

The gating network is, such as the motion prediction network, a three layer neural network and computes the blending coefficients ω given a subset $\hat{x}$ of the input data **x**:

$$\Omega(\hat{x};\mu) = \sigma(W_2' ELU(W_1' ELU(W_0' \hat{x} + b_0') + b_1') + b_2') \tag{4.15}$$

where $\hat{x} \in \mathbb{R}^{22}$ consists of the foot velocities $f \in \mathbb{R}^{12}$, the current action variables $t_7^a \in \mathbb{R}^9$ and the desired velocity $t_7^{\hat{v}} \in \mathbb{R}$ of the character. The parameters of the network μ are defined by

$$\mu = \{W_0' \in \mathbb{R}^{h'\times 22}, W_1' \in \mathbb{R}^{h'\times h'}, W_2' \in \mathbb{R}^{K\times h'}, b_0' \in \mathbb{R}^{h'}, b_1' \in \mathbb{R}^{h'}, b_2' \in \mathbb{R}^K\}$$

where h' is 32 - the number of hidden layer units. Note that, σ is a softmax operator, which normalizes the inputs such that they sum up to 1. [33]

4.3.4 Training

Finally, the entire network is trained using the processed data. The main goal of training the network is to produce a set of output variables $Y = [y_1 y_2 \dots]$ based on the input $X = [x_1 x_2 \dots]$ where y_i is the output of frame i and x_i the input. This is a regression task with the calculated cost function as the mean squared error between the predicted output and the desired output (ground truth):

$$Cost(X, Y; \beta, \mu) = ||Y - \Theta(X, \Omega(\hat{X}; \mu); \beta)||_2^2. \tag{4.16}$$

Also, the aim of training consists of learning a non-linear function for the motion action types that regress the missing motion frames to create more realistic transitions. The neural network is trained in TensorFlow which is the most popular learning library, published by Google. During training, the data is iterated in batches of size 32 where the training samples in each batch are selected randomly. This means the $\sim$324.000 samples of 883 scalars are divided into batches of 10.125 samples and fed to memory. At every layer (after the input) the layer's inputs and weights are multiplied and the neuron biases are added. Next, the activation function ELU is applied as shown above in equations 4.13 and 4.15. Then, a dropout technique is used to increase the network's ability for generalization by throwing away random samples. Note that, full training of 150 epochs takes around 20 hours on the NVIDIA GeForce GTX 1080 GPU. Also, notice that training this network with the proposed dataset using a CPU is undesirable in terms of training time.

4.4 Motion Synthesis

Once the network finished training, the trained weights are loaded into Unity. There, the network is running a port of the C++ Eigen library, which provides templates for most required mathematical operations to reconstruct the architecture of the network, since Unity requires C# script language. In order to predict the motion in the next frame during runtime, the trajectory is computed as described in section 4.1.1 based on the authoring system with the users input. By setting the action type value for the one-hot vector, the desired action, which is one of idle, move, jump, sit, lie, stand, sneak, eat and hydrate, is launched at the defined timestamp or position, along with a user defined motion time-interval that specifies a certain time to remain at this state. The character is fully controlled by the authoring system and all features of the control signal are included in the control parameters, which are smoothly interpolated and finally used to predict the future trajectory. The trajectory along with the characters pose of the previous frame is fed into the network as input. The network requires $\sim$ 2ms per frame to predict the motion single-threaded on an Intel-Core i-7 CPU. Further, the updated root motion is tuned by a control parameter, which prevents updating the root of non-locomotion styles to

avoid motion artifacts. Also, the updated trajectory is interpolated with the users desired trajectory in the previous frame based on a blending control parameter, which allows tuning a correction of the control signal. Finally, the whole posture of the character is computed and assigned.

Post-processing is applied on the updated posture to allow the character moving along uneven terrain, to reduce foot sliding, and to achieve good looking motion in general. Because the character is trained on flat terrain, the network has no data or information to generate motion for uneven terrain. For this reason, a CCD full body inverse kinematics is used and applied to the characters movements to walk along uneven terrain. The spine joints positions and rotations are updated with respect to the ground heights at the end of each frame. Also, the positions of the feet end effectors are post-processed using the contact labels from the networks predicted output such that they can incorporate the terrain height offset during locomotion. Here, the contact labels are values in range [0,1], which interpolate the position and rotation of each foot and the ground. It determines the foot update when connecting to the ground. A smoothstep activation function, similar to the sigmoid function operates on the networks predicted foot contacts to achieve even better results. Note that these inverse kinematic approaches are useful for simple motion synthesis, but cannot produce more dynamic movements such as jumping on to/off from high positions.

Although the authoring system needs to be defined before runtime, its implementation allows the designer to apply changes and adjustments to the authoring during runtime. For instance, the control signal can be changed by dragging and dropping the control points in the scene- and game view. Changing parameters of one control point requires to update the spline interpolation described in section 4.1.1. The system requires ∼140ms to compute the updated points in Unity editor mode on the fly. Note that, performance is increased outside of the editor, once the project is built.

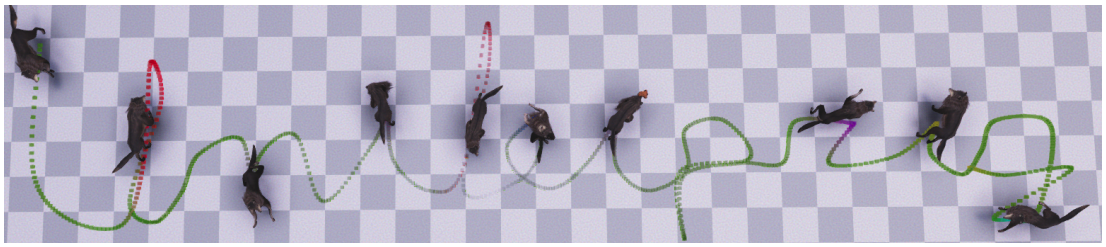


FIGURE 4.13: The character following the user defined path and control signal on the ground. The parameters for control and correction with the predicted trajectories of the network is set to 0.6 in order to perform realistic motions along the customized path. Further, the character is performing different action motion types at specified control points.

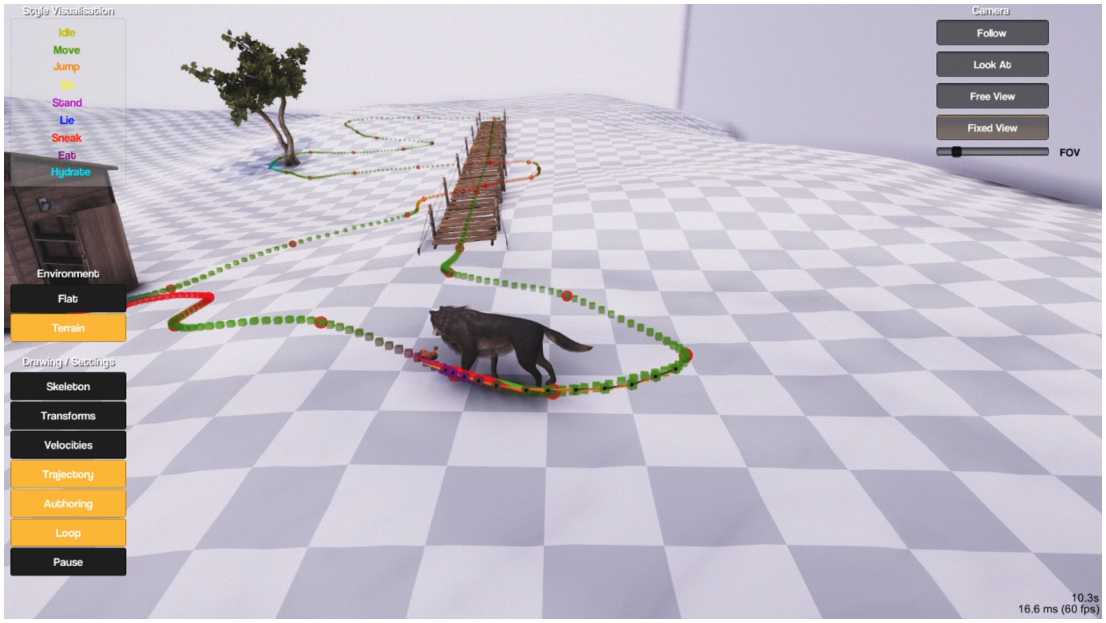


FIGURE 4.14: Runtime control in Unity with the quadruped character controlled via the authoring system over a smooth uneven terrain synthesized by full-body inverse kinematics.

Chapter 5

Evaluation

This chapter presents the experimental results of the proposed system, which aims to demonstrate its performance. Initially, section 5.1 gives a parameter evaluation in order to provide an intuition for a suitable choice in control and correction of the trajectory. Further, the system is evaluated through comparison with the results published from the authors in terms of motion quality and foot sliding artifacts in section 5.2. Finally, section 5.3 examines the activation of the new action motion styles with the corresponding learned transitions.

5.1 Control Signal Parameters

This selection of parameters has a major impact on the responsiveness and path-following accuracy of the character. However, choosing them is not straightforward, and is largely influenced by the users defined path and control signal. In order to make the character follow the trajectory, a control parameter τ defines all blending parameters t of the trajectory estimation, which blends all features F^* of the future desired trajectory T^*. Further, a correction parameter ρ interpolates between the future points of the corrected trajectory T^+ in the output of the network. To obtain the final input trajectory T for the motion update, both trajectories are blended as $T = \tau T^* + \rho T^+$. The position deviation between the actual trajectory of the character considering their root transformations and the points position of an artificial custom path is measured and visualized in Fig 5.1 and 5.3. It can be observed that the character is well following the path with a low average distance if τ is greater than 0.3. However, the distance rapidly increases when there is less control of the future trajectory. This is because the positions of the defined path are less valued. Note that, in Fig. 5.1 the average distance is measured to a maximum of 1 meter for visualisation purpose, while the real distance at zero control is approaching infinity. Here, the average position deviation of $\sim$ 9.0cm is similar to the result for the custom paths of the authors. Furthermore, note that this evaluation of the average distance does not provide any information about the motion quality.

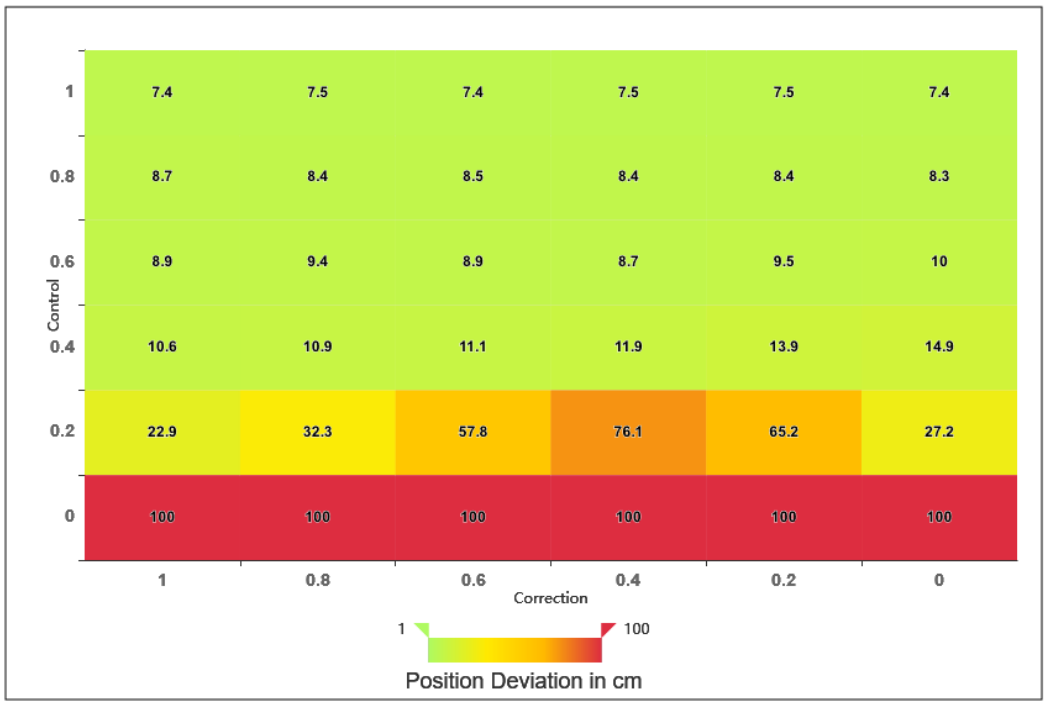


FIGURE 5.1: The average distance between the root transformations of the character and the path, in dependence of the trajectory control and correction parameter.

5.2 Foot Sliding Artifacts

The motion quality of the feet is evaluated in dependence on the angle turning speed of the character per second. Therefore, the foot sliding of all four legs is measured by adding the foot velocities if the position height h is within a maximum threshold of H=2.5cm. The sliding s during motion is then calculated as $s = v(2 - 2^{\frac{h}{H}})$ where v is the velocity magnitude in the horizontal plane. The exponent is clamped between 0 and 1. Further, the turning angle of the trajectory is sampled one second around the current timestamp during motion. The result is shown in Fig. 5.2 where the sum s_{sum} of all four legs are divided into containers. Note that, this is not the foot sliding in the ground truth data, but covers the data of a 5min runtime footage. It can be observed that the sliding on wide path curves is very low but can break out when the character has to turn relatively fast. Also, the peaks in Fig. 5.2 can show missing training data for some turning angles which are not covered in the current dataset. However, in comparison to the results of the authors, the foot sliding was almost reduced by half. This is because of the foot post-processing using the trained contact labels with the included contact information in the state vector, which is described in section 4.3.

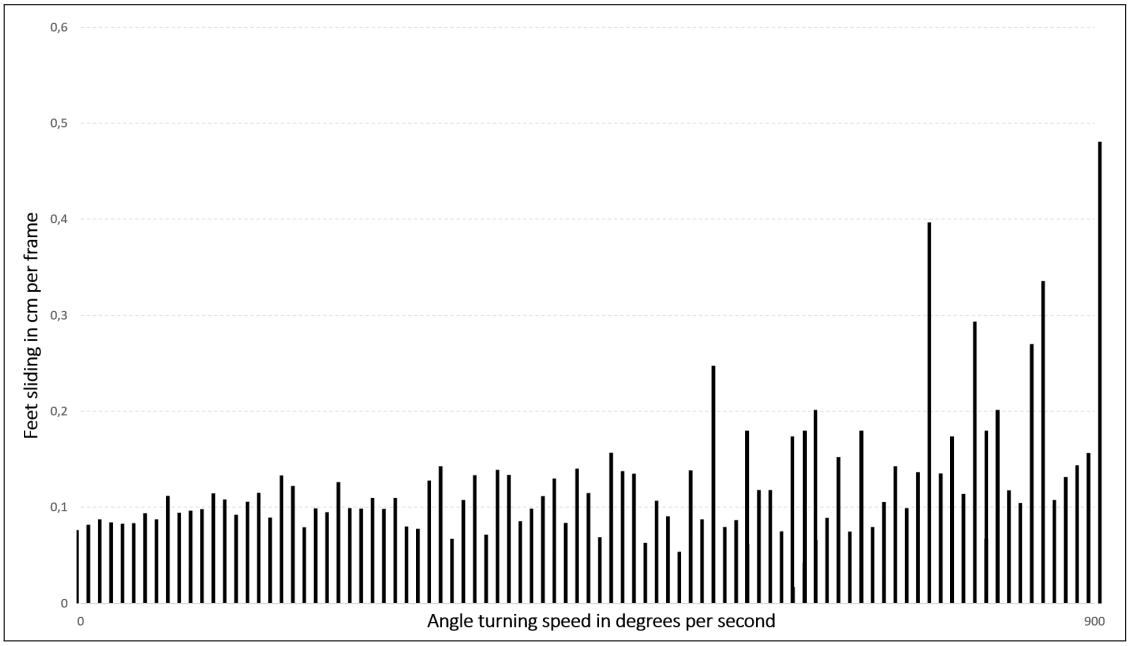


FIGURE 5.2: The average foot sliding for all four legs in an 5min runtime footage of walking along a custom path. Each bar represents the sampling of 10° angle turning speed.

5.3 Learned Latent Spaces

To examine the situation Figures 5.4, 5.5, and 5.6 show the transitions of the learned postures for the new different types of actions. All postures were obtained in real-time and are launched by a style value in range [0,1]. The transition of the learned postures is compared with a simple keyframe animation where the style value interpolates between the starting posture and the final pose. The style value for the networks input is pre-processed using a sigmoid activation function, which leads to starting the transition between a value of 0.6 and 0.7. This is done for visualization and control purposes because the raw network transition starts at $\sim$0.2 and tends already to end at $\sim$0.5. In addition, the figures show the corresponding latent space to the learned posture in each layer, which describes the multi-dimensional space of all feature values. It visualized the changing feature activation during motion and the neural network weights distribution between the first and the third layer. The value of each feature is the sum of the output weights of the neuron. Note that in this case, each latent space represents all 516 neurons, where the first layer values are visualized in range [-10,10], the second layers in range [-16,13], and finally the third layers in range [-2,2]. It can be observed that the network learns natural transitions for all three action styles using the data augmentation described in section 4.2. Note that, the learned postures of sneaking showed in Fig. 5.4 are deviating because the network learns the motion space and generates locomotion when the style values are approaching to 1. This is because sneaking was not trained without locomotion. Furthermore, Fig. 5.5 shows that the hind legs remain static in keyframe animation, while the network creates small movements.

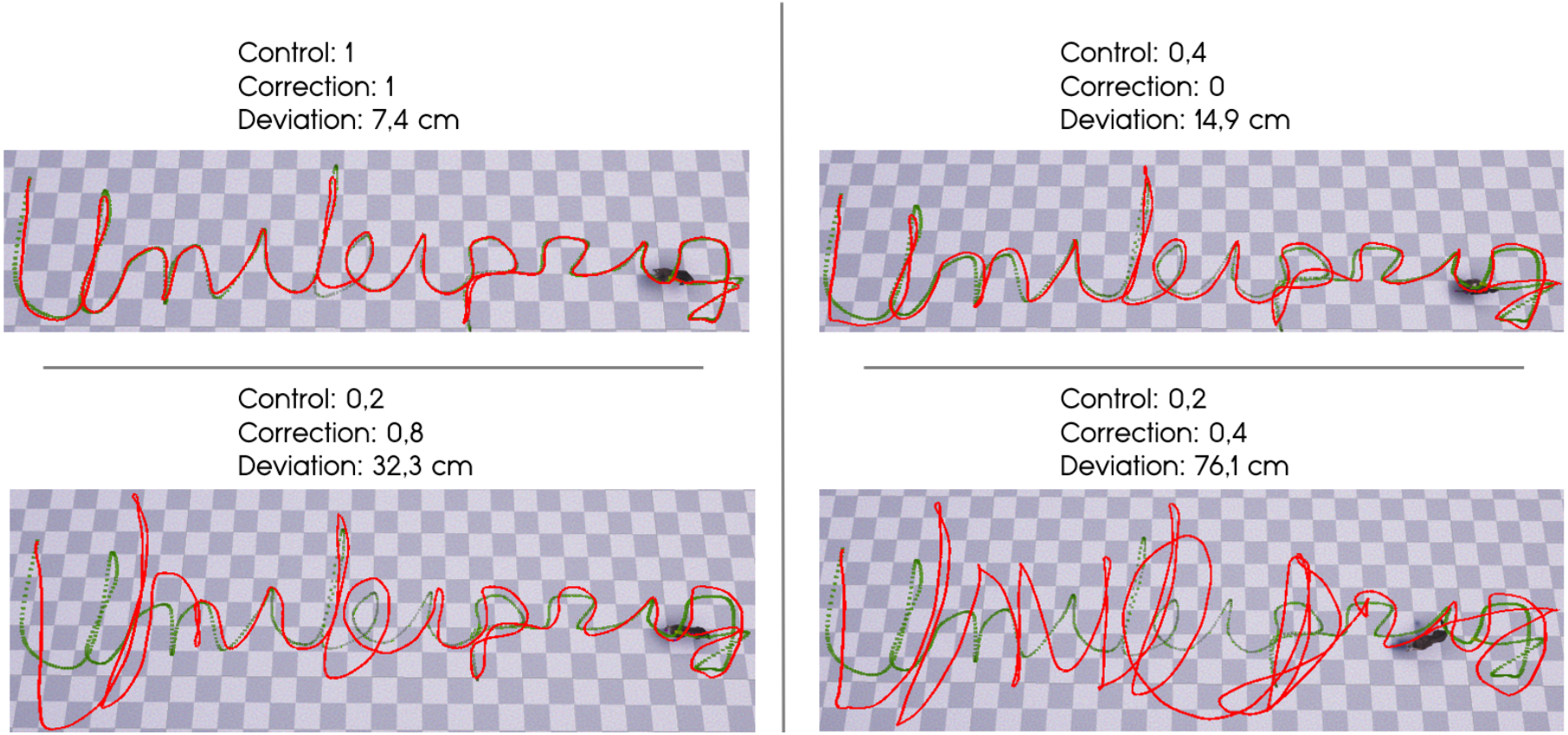


FIGURE 5.3: The average position deviation visualized in real-time in dependence on the trajectory control and correction parameter of Fig. 5.1. The actual trajectory colorized in red along with the desired path in green. In comparison each square on the ground with an area of $0.5m^2$.

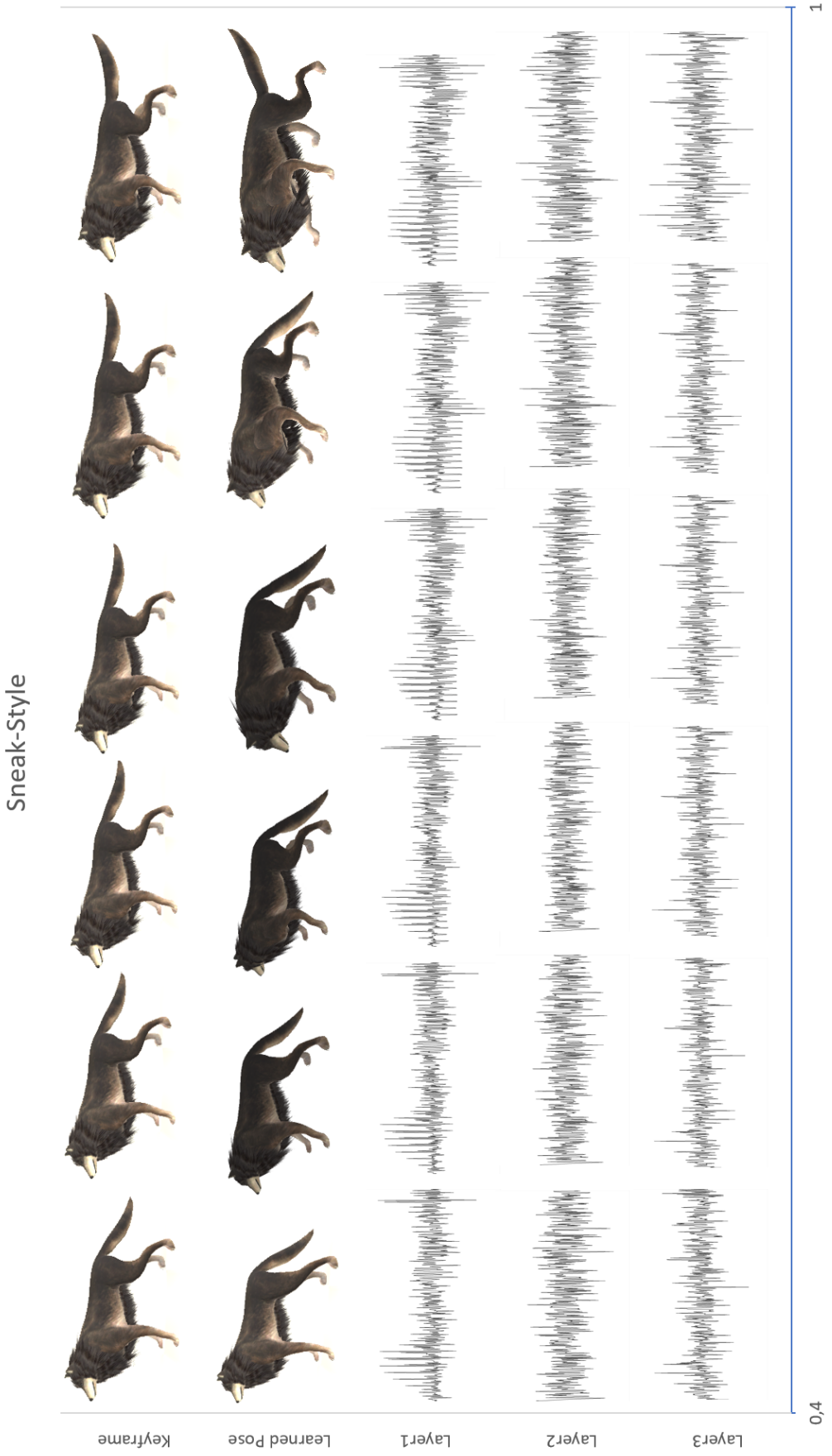


FIGURE 5.4: The transitions of the sneaking motion style obtained during runtime without using inverse kinematics on the wolf character. Each column represents the posture and latent space at the given style input.

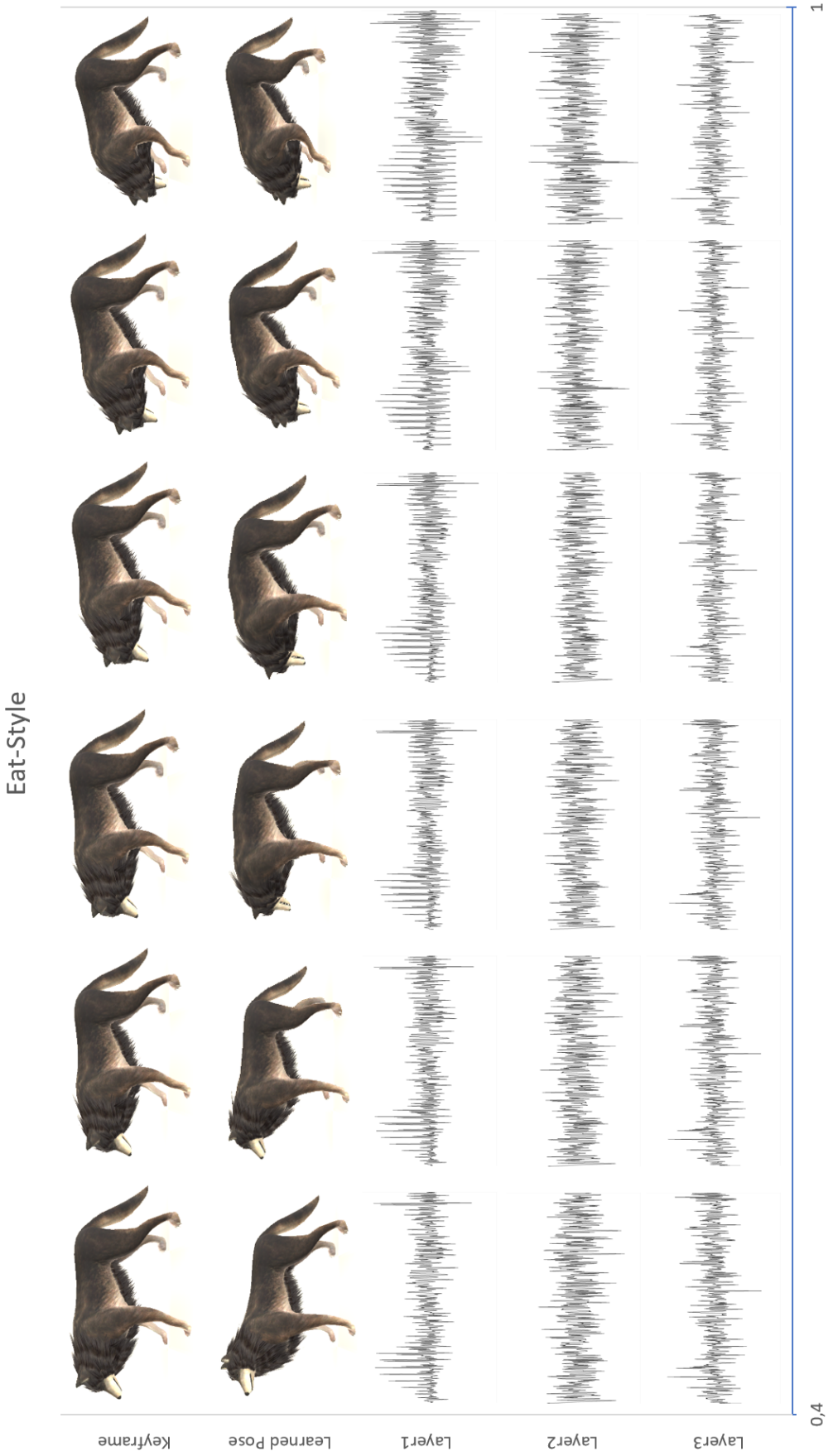


FIGURE 5.5: The result of the learned eating motion with the corresponding latent spaces at each style input as well as the comparison to linear interpolated keyframe animation.

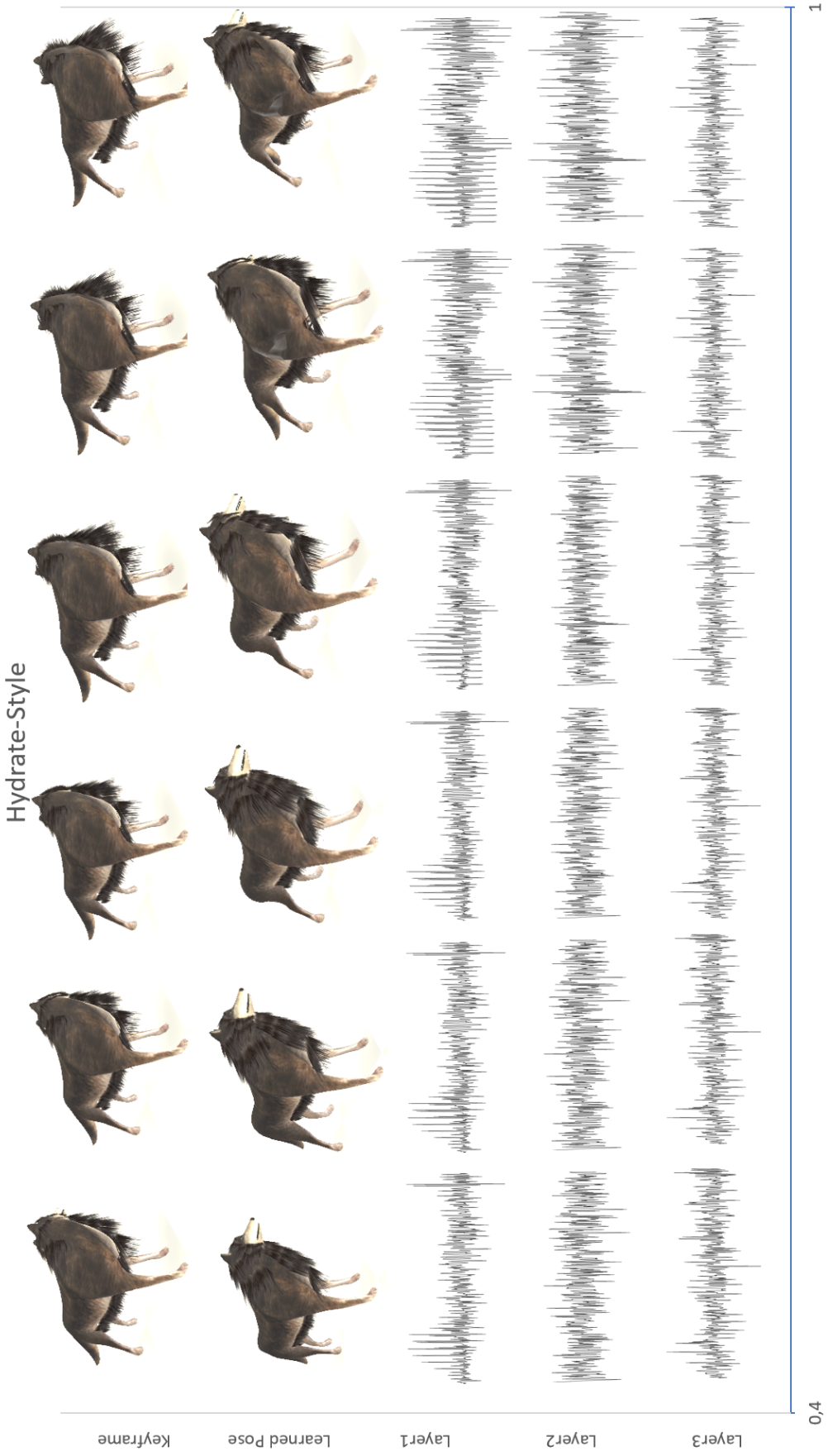


FIGURE 5.6: The transitions of the hydrate motion style.

Chapter 6

Conclusion

6.1 Summary

This thesis proposed an animation authoring system for quadruped animations learned by a mode-adaptive neural network. Given a set of user defined control points the system can generate control signals in form of trajectories that follow the path smoothly and tightly. It further extends the framework of the authors by the ability to specify desired actions at a timestamp or position with additional motion time. This can be used for designers to create natural-looking motion patterns in games as well as for the creation of film sequences. The system was made available as implementation for Unity3D(C#) and enables control of the character before as well as during runtime. Further, the research was motivated by the issue that many motions for quadrupeds did neither seem readily obtainable with motion capture, but are essentially required for various applications in animation. Therefore, this thesis provided a method to easily create a target posture by using full-body inverse kinematics with the aim to combine this synthesized data with the original motion capture data. This data augmentation enhanced the quality of the synthetic data when trained jointly with the original data, especially during transition movements that were not captured. The experiments on several new motion action types demonstrated that the network is capable of learning a non-linear function that can regress the missing motion frames, as well as to transform the synthetic movements into a more realistic motion. All experiments were done in real-time without requiring any offline precomputations. Further results show that, foot sliding artifacts were heavily reduced by providing the network with foot contact information. Furthermore the work shows that inverse kinematics is a simple, but powerful technique for full-body animation processing. Finally, this thesis demonstrated and aims to motivate neural networks as a competitive, efficient and flexible methodology for animating quadruped characters.

6.2 Limitations and Challenges

The proposed dataset presents one possibility to augment the data to achieve better motion quality for synthesized postures. However, the inverse kinematic data in the ground truth can cause worse quality on the motion capture data. This is because the model might overfit on the IK data and probably learns an easier function. Furthermore, the dataset is limited to those from flat

terrain and movements such as jumping from high positions cannot be synthesized. It was observed that the system is running at 60Hz which represents the usual real-time frame rate in games. This suggests that the framework is predominantly suited for control of one or some few main characters rather than for controlling multiple but less important characters at the same time, and for which computationally faster but less powerful methods are used in industry.

6.3 Future Work

So far, the authoring system has been applied to a quadruped wolf game character, but not yet considering the dynamics of the system. Extending the method to biped characters outlines a very promising goal for future work. Another goal would be to design a functionality for automatically annotating the motion data in dependence of velocity and height of the character bones. Further, the systems generated control signal could be extended to automatically adapt the characters actions to its complex environment, such as sneaking under a bridge without annotating motion actions of the control points by the user. Another possibility is to apply reinforcement learning to decide the control signal of the framework. Once the data is obtained, any terrain fitting and adaption can be easily done. Finally, applying physically based animation to synthesize the training data could achieve even more natural-looking motions where the character makes use of the preservation of momentum.

Bibliography

[1] *Activation Functions*. URL: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ (visited on 08/05/2020).

[2] *AI4Animation*. URL: https://github.com/sebastianstarke/AI4Animation (visited on 08/11/2020).

[3] Andy Beane. *3D Animation Essentials*. Sybex, 2012.

[4] Edwin Catmull and Raphael Rom. *A class of local interpolating splines*. 1974, pp. 317–326. DOI: https://doi.org/10.1016/B978-0-12-079050-0.50020-5.

[5] Dan C. Ciresan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber. "Flexible, High Performance ConvolutionalNeural Networks for Image Classification". In: *Addison-Wesley* 2 (Nov. 2011), pp. 1237–1242. URL: http://people.idsia.ch/~juergen/ijcai2011.pdf.

[6] Simon Clavet. "Motion Matching and The Road to Next-Gen Animation". In: *Proc. of GDC* (2016).

[7] *Eating Posture Reference*. URL: https://images.pond5.com/gray-wolf-eating-1-footage-000498632_iconl.jpeg (visited on 09/04/2020).

[8] Katerina Fragkiadaki, Sergey Levine, Panna Felsen, and Jitendra Malik. "Recurrent Network Models for Human Dynamics". In: (Dec. 2015). DOI: 10.1109/ICCV.2015.494.

[9] T. Geijtenbeek, N. Pronost, A. Egges, and M. H. Overmars. "Interactive Character Animation using Simulated Physics". In: *Eurographics* (2011). URL: http://graphics.cs.cmu.edu/nsp/course/15-869/2012/papers/PhysicsAnimation_EG11.pdf.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popovíc. "Style-Based Inverse Kinematics". In: *ACM Transactions on Graphics* (2004). URL: https://grail.cs.washington.edu/projects/styleik/styleik.pdf.

[12] Hansika Hewamalage, Christoph Bergmeir, and Kasun Bandara. "Recurrent Neural Networks for Time Series Forecasting: Current Status and Future Directions". In: 3 (Sept. 2019). URL: https://arxiv.org/abs/1909.00590.

[13] Daniel Holden, Taku Komura, and Jun Saito. "Phase-Functioned Neural Networks for Character Control". In: *ACM Transactions on Graphics* 36.4 (July 2017), pp. 1–13. URL: https://dl.acm.org/doi/10.1145/3072959.3073663.

[14] *Hydrating Posture Reference*. URL: https://cdn.akc.org/content/article-body-image/pug_peeing.jpg (visited on 09/04/2020).

[15] Ben Kenwright. "Real-Time Character Inverse Kinematics using the Gauss-Seidel Iterative Approximation Method". In: (2012). URL: https://www.thinkmind.org/index.php?view=article&articleid=content_2012_4_10_60013.

[16] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. "Motion Graphs". In: *ACM Trans. Graph.* 21.3 (July 2002). URL: https://doi.org/10.1145/566654.566605.

[17] *Max pooling*. URL: https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png (visited on 08/05/2020).

[18] Michael Meredith and Steve Maddock. "Real-Time Inverse Kinematics: The Return of the Jacobian". In: (2004). URL: https://www.researchgate.net/publication/228980657_Real-time_inverse_kinematics_The_return_of_the_Jacobian.

[19] *Neuron Activation*. URL: https://miro.medium.com/max/700/1*0lejoYyyQWjYzEP_BNW2nw.jpeg (visited on 08/27/2020).

[20] *Original Input Dataset*. URL: http://www.starke-consult.de/UoE/GitHub/SIGGRAPH_2018/Input.txt (visited on 08/05/2020).

[21] *Original Output Dataset*. URL: http://www.starke-consult.de/UoE/GitHub/SIGGRAPH_2018/Output.txt (visited on 08/05/2020).

[22] Rick Parent. *Computer Animation, Third Edition: Algorithms and Techniques.* Morgan Kaufmann, 2012.

[23] XUE BIN PENG, PIETER ABBEEL, SERGEY LEVINE, and MICHIEL VAN DE PANNE. "DeepMimic: Example-Guided Deep Reinforcement Learningof Physics-Based Character Skills". In: *ACM Transactions on Graphics* 37.4 (Aug. 2018). URL: https://arxiv.org/pdf/1804.02717.pdf.

[24] F. Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: (1958). URL: https://doi.apa.org/doiLanding?doi=10.1037%2Fh0042519.

[25] *Sneaking Posture Reference*. URL: https://live.staticflickr.com/563/21324327334_5ef5bcfff0_b.jpg (visited on 09/04/2020).

[26] Sebastian Starke, Yiwei Zhao, Taku Komura, and Kazi Zaman. "Local Motion Phases for Learning Multi-Contact Character Movements". In: *ACM Transactions on Graphics* (July 2020), pp. 1–14. URL: https://doi.org/10.1145/3386569.3392450.

[27] Sebastian Starke†, He Zhang†, Taku Komura, and Jun Saito. "Neural State Machine for Character-Scene Interactions". In: *ACM Transactions on Graphics* (Nov. 2019), pp. 1–14. URL: https://doi.org/10.1145/3355089.3356505.

[28] *Ultimate Inverse Kinematic Algorithm*. URL: https://github.com/sebastianstarke/AI4Animation/blob/master/AI4Animation/SIGGRAPH_Asia_2019/Unity/Assets/Scripts/UltimateIK (visited on 08/11/2020).

[29] Kevin Wampler, Zoran Popović, and Jovan Popović. "Generalizing locomotion style to new animals with inverse optimal regression". In: *ACM Transactions on Graphics* 49 (July 2014). URL: https://doi.org/10.1145/2601097.2601192.

[30] *Wolf Model*. URL: http://wp.me/P3dmoi-1uc (visited on 08/06/2020).

[31] Stephen J. Wright. "Coordinate descent algorithms". In: *Mathematical Programming* (2015). URL: https://doi.org/10.1007/s10107-015-0892-3.

[32] Andreas Zell. "Simulation neuronaler Netze". In: *Addison-Wesley* (2003). URL: https://www.worldcat.org/title/simulation-neuronaler-netze/oclc/249017987.

[33] He Zhang†, Sebastian Starke†, Taku Komura, and Jun Saito. "Mode-Adaptive Neural Networks for Quadruped Motion Control". In: *ACM Transactions on Graphics* 145 (Aug. 2018), pp. 1–11. URL: https://dl.acm.org/doi/10.1145/3197517.3201366.

[34] J. Zhao and N. I. Badler. "Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures". In: (1994). URL: http://graphics.cs.cmu.edu/nsp/course/15-464/Fall05/papers/zhaoBadlerIK.pdf.

Declaration of Authorship

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Ort, Datum

Unterschrift