Hierarchical development of physics-based animation controllers

M.J.P. Hagenaars
m.j.p.hagenaars@students.uu.nl

Version of January 17, 2014

MASTER'S THESIS

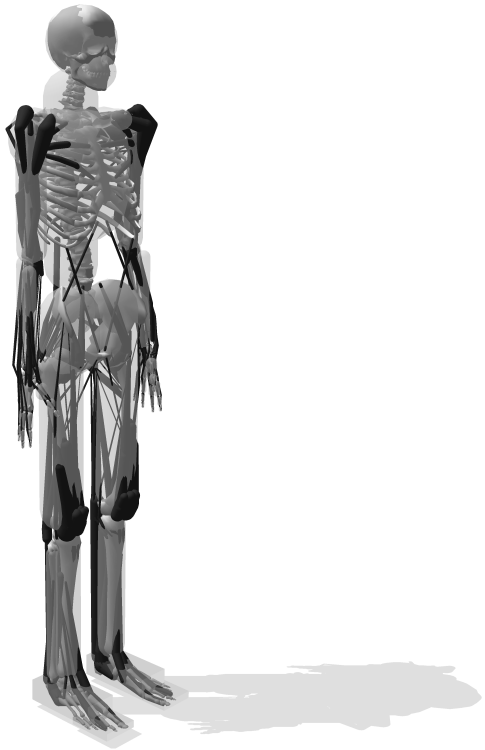
Game and Media Technology
Utrecht University

Supervised by dr. N.G. Pronost

ICA-3759482

Abstract

In this work, a *developmental hierarchy* is applied to the evolution of a relatively complex physics-based character animation controller. This means that the artificial neural network that makes up that controller is composed from a number of interdependent sub-networks; the *control modules*. It is hypothesized that evolving these modules one-by-one, with each of them dependent on its predecessors, will allow evolution to converge faster, and possibly to better results, than for a pair of baseline controllers. Both muscle-based actuation and joint torque-based actuation are tested, but only the latter succeeds. It is demonstrated that developmental hierarchies can lead to faster evolutionary convergence, while dealing with compound animations more adequately.



Acknowledgments

First and foremost, my very special thanks go out to dr. N.G. Pronost, whose tireless support and insight were indispensable to the realization of this project.

I would like to thank my colleagues and their supervisors from the animation group, especially dr. ir. J. Egges, being the second examiner for this thesis, for their critical thinking and feedback during our regular meetings. Also, thanks to my friends and family, for their patience, support, and fresh (interestingly different) ideas.

Special thanks go out to my girlfriend, who has been incredibly patient; always willing to listen to my random ramblings so very lovingly.

Contents

Chapter 1

Introduction

Before an infant is able to toddle about, it has usually mastered several other motor skills, such as rolling over, crawling, and sitting upright. Some of these abilities are necessary in learning to walk, as they lead to the development of both the required neural pathways and physical strength. In much the same fashion it can be argued that the ability to walk underlies the abilities of jumping and running, which, in turn, may underlie more complex activities, such as cycling and dancing. There seems to be a developmental hierarchy of motor skills in humans.

In the field of computer animation, and more specifically character animation, motion data is usually captured or generated in a fairly direct manner. If a virtual character needs to be walking, the required motion data is either hand-made, obtained through motion capture performance, or generated procedurally. Either way, this is a difficult and time-consuming problem, because these methods require a lot of man-hours per animation, while very little motion data can be used in more than one animation. This leads to the investigation of ways to automate large parts of the animation process.

The methods that are proposed in this thesis aim to re-use physics-based animation controllers by combining them in a hierarchy that is similar to the order in which humans develop motor skills. These modular animation controllers consist of artificial neural networks that are generated through a process of artificial evolution. They are applied to, and evaluated in, a physics-based simulation of a virtual human character. The idea behind this is that it is easier to train a neural network on a new task if it already has the solutions to any underlying problems at its disposal. For example, it may be easier to learn how to get from a supine or prone position to balanced stance if previous solutions for rolling over, crouching, and getting up are readily available. Using such lower level skills may lead to converging on a solution to a new task faster, because any commonalities between the new task and the previous tasks do not have to be trained for anymore. Also, choosing a particular structure and ordering for the developmental hierarchy that match the development of human motor skills, may provide a bias towards evolving more natural solutions.

This hierarchical organization of motor skills is supported by neuroscientific and psychological research. Berridge [6] provides an overview of the work of Teitelbaum [27], in which he describes one of his principles of

> "...hierarchy and levels of function. This hierarchical principle addresses how higher levels of brain systems control the activity of lower brain levels."

In his extensive theory of cognitive development, called "skill theory", Fisher [13] describes "control and construction of hierarchies of skills". The developmental hierarchies that are presented in this work are similar, albeit more simple, to Fisher's skill levels in that they deal with skills at different levels and of increasing complexity. However, the methods that are presented here are purely focused on (sensor-)motor skills, and not cognitive development in general.

1.1 Physics-based character animation

Recent approaches to computer animation are based on physical simulations of both characters and objects. Although the basic ideas have been around for a long time, interactive physics-based character animation seems to be still in its infancy, as most "commercial frameworks still resort to kinematics-based approaches when it comes to animating *active* virtual characters." [14]

In the past, artists could animate virtual characters quite convincingly by kinematics-based techniques, such as *keyframing*. Many modern approaches also rely on *motion capture* (mocap) data to compose animations that are inherently more realistic, because they are actually generated by a real human. With the constant demand for more and better animations, new projects focus on physics-based animation techniques. The nature of these approaches ensures that all motion data is physically realistic, while also eliminating the need for human actors. This is no guarantee, however, that the resulting animations will also look natural. These aspects, along with ever increasing cheap computing power, make physics-based character animation a very attractive state-of-the-art research topic.

The use of physics to simulate *passive* phenomena has seen a slow but steady increase over the last decades, including simulations of simple objects (e.g. boxes), cloth, fluids, and ragdolls. One of the main things that effects like these have in common is that they do not need a lot of control from the artist or end-user. Passive physics are commonly used in character animation by blending with traditional keyframe animation and motion capture data to create smooth transitions between actively controlled and reactive animations. An example of this is a game character being pushed or shot down while climbing or running, resulting in a rag doll-like animation. The climbing or running part is actively controlled, while the reaction to the punch or bullet—and the subsequent falling down—is the result of passive physics. Passive physics are, therefore, great for creating dynamic

environments and characters, but another animation system is usually required for the more active, purposeful animations.

Active physics-based character animation offers less direct control over a character's body pose over time than traditional methods would. This is because it works by applying torques directly at the joint, or by generating them with virtual muscles. These joint torques lead to a change in the character's body pose, which traditionally would have been set more directly by an artist. This means that creation of even basic functions like balancing or walking is non-trivial, especially when user interaction is allowed. An advantage of this kind of approach is that it results in more physically correct animations, particularly because active physics are inherently blending with the effects of passive physics. However, this does not mean that the resulting animations will be guaranteed to also look more natural. Simplicity and fine-grained control are traded for physical correctness, but not necessarily realism.

Directly applying joint torques is not realistic, conceptually, when compared to muscle actuation in biological organisms, thus giving rise to attempts at using virtual muscle models to match biology more closely. In the end, this should lead to more natural motion on top of physical correctness. However, the complexity of the optimization problems in such simulations is far greater, because each degree-of-freedom of the joints can only be fully controlled by at least two muscles, while biological organisms usually even have many more redundant muscles. Also, muscles can span multiple joints, which makes optimizing muscle group activation patterns a very hard problem to solve. This is further discussed in Chapter 3.

The methods of hierarchical modular animation controllers that are proposed in this thesis are designed to use active physics to control a virtual character—including some effects of passive physics, such as collisions between body parts and external forces.

1.2 Overview

The work that is presented in this thesis is an approach to physics-based and biologically inspired character animation, in which either joint torques or muscles of a physically simulated human body are used to generate motion. The scientific contribution of this approach lies in the method that is used to generate those torques or muscle activation patterns, which consists of an *artificial neural network* (ANN) that is created through a process of *artificial evolution*, or *evolutionary algorithm* (EA). The background of this thesis, therefore, lies on the borders of computer animation and artificial intelligence alike.

The main research goals of this project are to see if the proposed method is suitable to produce good animation controllers (1) in relatively short training sessions (2). Specifically, this is hypothesized to work by limiting the evolutionary search space of finding the parameters and topology of a complex animation controller. This is achieved by imposing a novel hierarchical and modular design on the final ANN, where the evolution of high-

level functionality is done step-by-step, as guided by many simple objectives, instead of by a single, complex fitness function.

All methods used in this work are implemented in the C++ programming language, using Microsoft Visual Studio 2010 as the IDE and Subversion for source control. Rigid body simulation and collision handling are done with the Bullet physics library [8], and visualizations are created through the OpenGL graphics API. The original C++ implementation of the NEAT method (NeuroEvolution of Augmenting Topologies) [23] is used to deal with neural networks and their evolution. Many assets and coding examples from the OpenSim project [10, 11, 31, 4, 5, 18] are used to generate the basic human body simulation. Training sessions are run on a mid-range to high-end (at the time of writing) desktop computer, containing an Intel Core i7 920 CPU (4 cores, 8 threads, clocked at about 2.67 GHz), 6 GB of memory, and a Samsung 830 Series SSD.

The remainder of this thesis is laid out as follows. A short literature study is presented in Chapter 2, relating this thesis to similar and otherwise relevant works. Chapter 3 describes in detail the methods of generating and evaluating animation controllers, the results of which are shown in Chapter 5, and discussed in Chapter 6. More detailed information about the software that is used in—or created for—this project is available in Appendix B, of which some parameter are listed in Appendix A.

Chapter 2

Related work

This chapter consists of a short literature study of works that are similar or otherwise relevant to the methods used in this thesis, particularly with relation to *stimulus-response network* control and neuroevolution. Many of the references are also found in a very useful review paper by Geijtenbeek and Pronost [14], who discuss the state-of-the-art in the field of interactive character animation by use of simulated physics in great detail.

Physics-based character animation controllers are usually developed in a framework that consists of multiple components, including a physics simulation, a physics-based character, and the animation controller itself [14]. For real-time applications, some of the most commonly used physics engines are Bullet, ODE, Havok, and PhysX. The Bullet physics library [8] is used in this work for its 3D collision detection and rigid body dynamics.

2.1 Stimulus-response networks

The artificial neural networks that form the animation controllers in this work are a type of *stimulus-response network*. There are a number of different kinds of stimulus-response networks, besides *artificial neural networks* (ANNs), including *genetic programs* (GPs), and *central pattern generators* (CPGs). What these types of networks have in common, is that they consist of interconnected *nodes*, or *neurons*. These nodes are processing elements that have a variable number of incoming and outgoing connections (*links*) with other nodes.

In ANNs, the output of each neuron (inspired by *axons* in biology) is determined by the weighted sum of all inputs (inspired by *dendrites*) and the *activation function* [3, 7], for which a sigmoid function is often used for computational reasons. The activation function $\sigma(x)$ is described by Equation 2.1 and Figure 2.1 [23], where y_i is the activation level of node i, and w_{ji} is the connection weight between nodes i and j.

$$y_i = \sigma\left(\sum_{j=1}^{N} w_{ji} y_j\right), \quad \sigma(x) = \frac{1}{1 + e^{-4.924273x}} \tag{2.1}$$

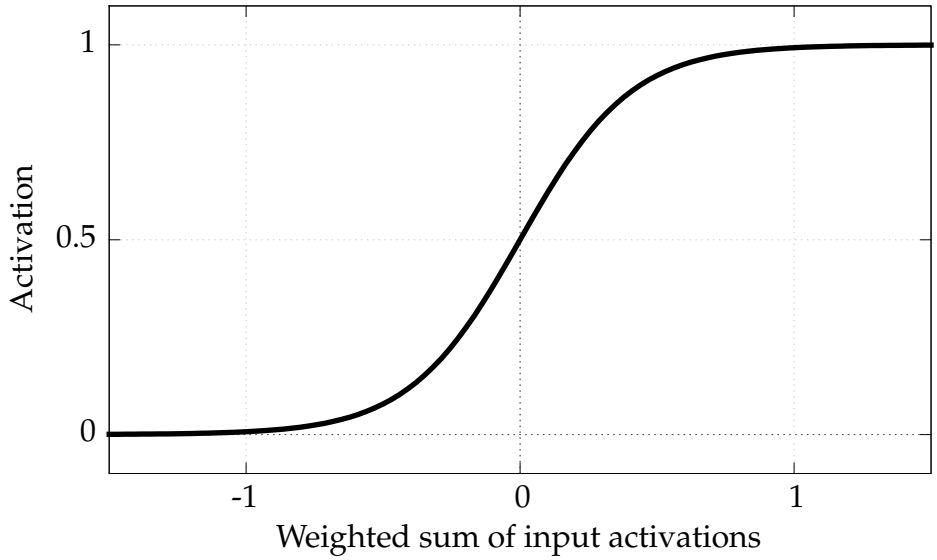


Figure 2.1: *Plot of the activation function as described in Equation 2.1.*

Many attempts at using artificial neural networks in physics-based character animation reduce the complexity of the problem by only looking at 2D characters, and by limiting the number of *degrees-of-freedom* (DoFs) in the joints [28]. Reil and Husbands [21] use an ANN with a fixed topology to animate a walking 3D character, by generating target joint angles that are fed into a *proportional-derivative* (PD) controller.

Allen and Faloutsos [3] use the same algorithm as is used in this work to optimize their neural networks, namely NEAT [23]. Therefore, their approach is probably the most similar to the methods in this work. Both feature the topology of the neural network growing gradually, allowing for increasingly complex behavior. And, in the same way, they do not assume any a priori knowledge of the appropriate actuation patterns, but require only the physical properties of the character model and a simple fitness function.

While also working with joint torques through PD control (see Subsection 3.3.3), rather than muscle actuation, they manage to produce somewhat unstable (but nonetheless amusing) locomotion controllers. In contrast, the controllers in this work are not dedicated to generating controllers for just a single task (e.g. locomotion), but rather on a set of different behaviors. Also, they use more sensor nodes than are found to be necessary in this work, when looking at the joint torque-based cases. Another important difference between the two approaches is that they generate two identical ANNs from each genome, reflecting bilateral symmetry in the human body. In the here presented methods, only in the muscle-based approach there are some parts of the ANN that are symmetrically duplicated, yet the way in which they are interconnected can still be influenced by evolution.

Genetic programs are a different kind of stimulus-response network, in the sense that they do not deal with input and output activations of nodes like ANNs do. Instead, each node in the network represents either a logical operation (*AND*, *OR*, *XOR*, ...), decision

operation (*IF*, *THEN*, *ELSE*), or memory reading or writing [9]. Sims [22] uses nodes that can generate periodical signals, like sine waves. Other possibilities include mathematical operators, or programming language syntax, but these are mostly outside the scope of stimulus-response networks.

Central pattern generators [25] contain relatively small clusters of interconnected neurons that exhibit rhythmic excitation behavior. The patterns that a CPG produces can be interpreted as either joint torques, muscle activations, or target joint angles [14]. These constructs are particularly useful in the development of cyclical motions, for example in walking [25, 26].

2.2 Optimization strategies

Physics-based character animation is particularly suited for off-line optimization techniques, because it is relatively cheap and fast to evaluate large numbers of candidate controllers. This in contrast to robotics research, where testing can generally not go faster than real-time, and on-line optimization techniques, such as reinforcement learning [19], are prevalent.

Evolutionary algorithms are the most commonly used type of off-line optimization strategy for stimulus-response network control. Although many papers suggest EAs could lead to more natural results [2], there is little evidence to support this, because no papers compare different optimization methods in this context [14].

The inspiration for evolutionary algorithms comes from the proven successes of biological evolution, being based on an elaborate method of trial-and-error [12]. Candidate solutions to the problem at hand are analogous to individual biological organisms. The parameters that should be optimized are encoded in every individual's genome (*genotype*), which consists of DNA in biology. The genotype determines an individual's observable properties and behavior (*phenotype*) during its life, or, its evaluation.

Individuals that are more successful as a candidate solution, which is quantified by the fitness function, have a higher chance of reproduction. This is similar to real life, where fitter individuals are often more likely to successfully mate and reproduce. Conversely, individuals that perform badly are killed off early, saving resources.

At the end of a generation (or epoch), after natural selection as determined by fitness and a survival rate, the surviving individuals reproduce, thus spawning the next generation's population. When two individuals mate, genetic recombination (usually *crossover*) can occur, effectively splicing different parts of the parent's genomes together to make up their child's new genome. *Mutations* are applied randomly (albeit within specified limits), slightly changing one or more genes. Both genetic recombination and mutation have their respective biological analogues as well.

Some evolutionary algorithms also employ *speciation*; grouping of individuals that have similar genomes. By limiting the chance of individuals from different species to

mate, innovations are preserved. Exploring new areas of the search space often leads to an initial decrease of fitness [23], even though further optimization may eventually lead to a much better solution than the current optimum. When species are not performing well enough, they can go extinct, making room for new innovations.

Killing off bad individuals and allowing good individuals to reproduce (*natural selection*), drives the search towards better solutions. Recombination and mutation keep the population diverse, as to explore the search space and find different kinds of solutions. Speciation protects innovations from being discarded too soon. The evolutionary process is terminated once the maximum allowed number of generations has passed, or as soon as one of the individuals reaches a fitness value that is higher than a predefined threshold.

Van de Panne et al. [28] propose Sensor-Actuator Networks (a kind of ANN), of which they optimize the parameters using a two-phased "dart throwing strategy" that can be seen as a simplified form of artificial evolution. In the inspiring work of Sims [22] virtual creatures are allowed to evolve both morphologically and neurally (through a form of genetic programming [20], not to be confused with [9]). Hase et al. [17] use a network of neural oscillators (a CPG) of fixed topology with a musculoskeletal model to generate human gait patterns. The parameters of their neuronal system are optimized through evolutionary computation.

More recent works on physics-based character animation are using the Covariance Matrix Adaptation (CMA) [16] evolution strategy. The work of Wang et al. [30, 29] results in some pretty robust locomotion controllers, and the work of Al Borno et al. [1] shows highly dynamic and athletic motion. Geijtenbeek et al. [15] create flexible muscle-based controllers for a variety of bipedal creatures, resulting in robust and natural looking animations. However, these approaches—like other physics-based character control frameworks that use CMA—are not based on stimulus-response network control.

The methods that are described in Chapter 3 aim to improve upon other approaches to artificial evolution as a form of off-line optimization. Most related works focus on creating a single controller for a single type of animation or behavior, without making many assumptions about how they may relate to each other, or how they might have developed *in vivo*. Because humans do not learn to walk "from scratch", without any previous motor skills, it makes sense to investigate methods that evolve controllers step by step. The approach in this work differs from other works in that one starts out evolving a controller for a simpler behavior, which is then kept static, so that another—more complex—controller can evolve on top of what has been learned so far. This process can then be repeated to build ever more complex controllers, consisting of multiple control modules in a *developmental hierarchy*.

This way, the modules are kept lean, only adding functionality that is not present in underlying modules already. Also, all low-level modules remain individually accessible when new modules are added, so the earlier behaviors can still be used by simply disabling any higher-level modules. Because controllers in this work are ANNs, this is

simply a matter of enabling or disabling all *links* (neural connections) that are part of those particular modules.

Chapter 3

Methods

The methods in this work can be described as a *neuroevolutionary* approach to physics-based character animation control. The desired controller, that generates deliberate actuation patterns, is created through an off-line optimization process, so that it may be used in real-time applications afterwards. In the following sections all components of the method are discussed individually and in conjunction, roughly in the order in which they are implemented over the course of the project.

3.1 Musculoskeletal simulation

For both the off-line optimization and real-time demonstrations, the same physical simulation is used, based on the Bullet physics library [8]. In the demonstration phase, a custom OpenGL-based renderer is used to visualize the simulation. The combination of this physics engine and rendering engine will from now on be referred to as "the simulation", as they are used together in simulating a basic virtual world with a virtual character in it.

The virtual character is formed from a combination of two datasets from the OpenSim project [10], namely the "Gait2329" model [11, 31, 4, 5] and the upper parts from the "ULB" model, that, in turn, are based on the "UpperExtremities" model [18]. These models consist of physical data of human body parts, joints, and muscles and how they fit together. The reason to combine these models is that no single OpenSim model covers the full musculoskeletal structure of a typical human body in high detail. The ULB model, for example, has fewer muscles in the lower regions, while Gait2329 has no upper body muscles at all.

In order to increase simulation performance and stability, a number of degrees-of-freedom are removed from the final model's joints, starting with any translational ones. The joints that govern radioulnar rotation are locked, as they caused numerical instability and are unlikely to be relevant to the desired animations in this work. One of the two joints that represent an ankle is similarly removed in both ankles, as to improve stability of

the leg. These changes are justified by assuming that the resulting animation controller is applicable only to this particular model, while the methods themselves should generalize to almost any physical body (bipeds, quadrupeds, or even non-existing morphologies).

The final human body model that is used here consists of 16 rigid bodies (loosely referred to as body parts), 15 joints, and 192 muscles (see Figure 3.1). The body parts, visualized by a collision shape and bone mesh and connected at the joints, form the skeleton "tree", originating at the *pelvis*.

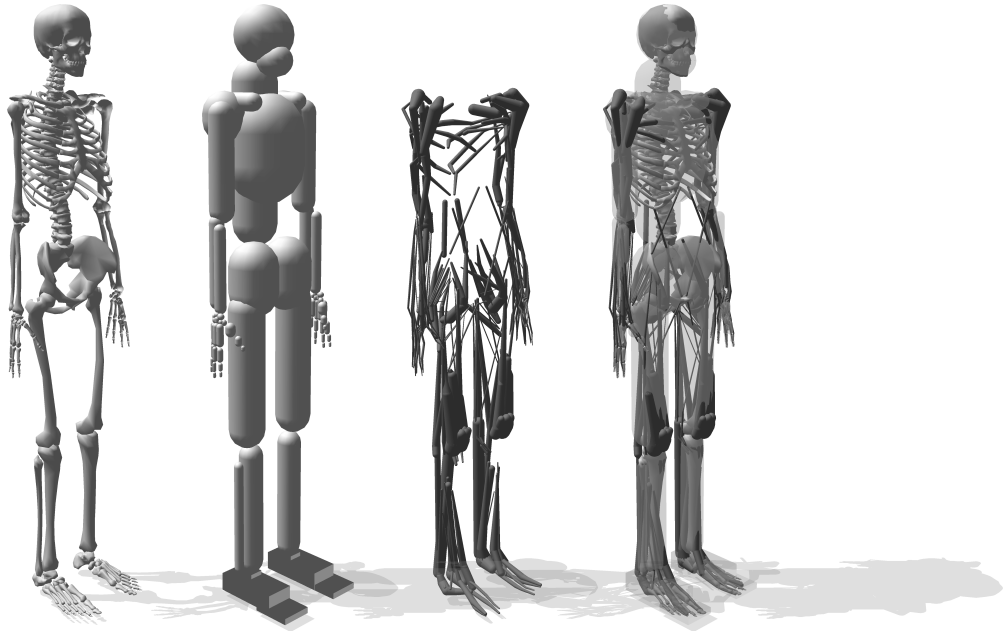


Figure 3.1: *From left to right: the* **skeleton** *is made up from OpenSim mesh data. From it, the* **rigid bodies** *are derived by using axis-aligned bounding boxes around the individual meshes. Virtual* **muscles** *are then parsed from the OpenSim data and attached to the collision shapes. These three components combined form the virtual physics-based human* **character** *that is used in all experiments.*

The *rigid bodies* are initially defined by their mass, moments of inertia, and center-of-mass. Polygon mesh data is also available to visualize the bones inside each body part, and in this project this is used in generating elementary collision shapes. This works by calculating the axis-aligned bounding box for each mesh, and fitting either a solid box (for the feet to be flat) or capsule shape in that volume. Although this "skin" is not very realistic in itself in terms of shape or plasticity, it does improve animation quality by not letting body parts pass through each other, or the world.

Joints impose constraints on the translation and rotation of the body parts they connect, and are defined by their location and orientation therein, as well as their *joint limits*. In practice, all joints allow their body parts to rotate around one point to some extent, while allowing no relative translations. A joint with, for example, three free axes can be seen as a ball-and-socket system (hip, shoulder), while a 1-DoF joint can be thought of as

a hinge (knee).

The *muscle* actuator data from OpenSim is quite comprehensive, and they use rather sophisticated models to simulate muscle activation and contraction dynamics. This includes the maximum isometric force of the muscle fibers, optimal fiber length, resting length of the tendon, and the angle between tendon and fibers at optimal fiber length. Also, the path of the muscle, along which the forces are exerted, is defined by path points that signify where the muscle is connected to a bone.

Integration or reimplementation of one of OpenSim's advanced muscle model is considered somewhat outside the scope of this project, so a simplified model is used instead, while holding on to the assumption that the key methods of this work should generalize to any other muscle models. The muscle model that is used here consists of an output muscle force (F_m) in Newton that is dependent on the input muscle activation ($a_m \in [0, 1]$) and the optimal muscle length ratio ($l_m \in [0, l_m^{opt}]$):

$$F_m = a_m \cdot \frac{l_m}{l_m^{opt}} \cdot F_m^{max} \tag{3.1}$$

So the force that a muscle can generate is in the range of $[0, F_m^{max}]$ Newton, where F_m^{max} is the maximum isometric force that this particular muscle can exert when fully activated. This maximum force is scaled by the optimal muscle length ratio, which reflects, in a simplified way, how muscle strength increases with muscle length.

3.2 Neuroevolution

As mentioned in Section 1.2, the animation controller is built from artificial neural networks, which traditionally consist of a collection of interconnected nodes, or *artificial neurons*. The way these nodes are connected (the *topology*), and the *connection weights* that are associated with their connections (or *links*), determine how information can be processed by the network. In this project the NEAT method (NeuroEvolution of Augmenting Topologies) [23] is used to optimize both the weights and the topology of the neural networks for the given fitness functions. NEAT is a *genetic algorithm* that works with genomes (*genotype*) that directly encode neural networks (the *phenotype* in this case), which means that information about the nodes and their connections is readily available from the genome itself. It is therefore possible to convert a genome to a neural network and vice versa.

At the beginning of the evolutionary process, an initial genome is loaded that represents a minimal network topology; the number of *input* and *output* nodes is predetermined and will remain fixed during evolution. Traditionally, the input nodes are fully connected to the output nodes, but no *hidden* nodes are present at first. However, for some of the problems faced in this work other initial topologies are more suitable, including ones that start out with no links at all, or ones that only partially connect inputs and

outputs. From the initial genome, a *population* is spawned of N *individuals* with randomized connection weights. Each individual's *fitness* value is determined by a *fitness function*, indicating how well an individual performs in solving the problem at hand. Then, the individuals are allowed to *reproduce*, which involves *crossing over* of genes (recombination), and genetic *mutation*. Individuals with a higher fitness value are awarded a higher chance to reproduce, while low-ranking individuals are marked for death. When *mutations* are applied to the genomes, this can involve a change in connection weight, the addition of a link, or the addition of a node (by splitting one link into two, with a new node in between), thus introducing new variations in the population to maintain genetic *diversity*.

The way in which the fittest individuals are favored over inadequate individuals drives the exploration of improvements upon promising genomes through mating and mutations, while "bad" solutions are killed off early on.

After these steps, the next *generation* in the evolutionary process is started, which involves the same steps of evaluating the (newly formed) population (consisting of survivors and newborns from the previous generation), applying reproduction, recombination, and mutations. When an individual is found that represents a perfect solution—or has a fitness value that is higher than a predefined threshold—the evolutionary process is terminated. Otherwise, evolution will continue, until a predefined number of generations has passed, and the individual that performed the best so far is regarded as the (possibly local) optimum.

Apart from addressing the problem of search space dimensionality by starting with minimal topologies, other characteristic features of the NEAT method include tracking of genes with historical markers (this allows *crossing over* between topologies), and protecting innovation through *speciation*. For more detailed information on these topics, and the NEAT method in general, see [23].

3.3 Developmental hierarchy

The desired animation controller in its simplest form is an artificial neural network that takes control and sensory information as input and produces a vector of actuation values as output (see Figure 3.2). The goal of this work is to create a method of generating such a controller that is both good (performing well and looking natural) and fast to generate (having a relatively short off-line optimization phase). The proposed method is inspired by biology, and more specifically by some of the phases in child development that are relevant to motor skills.

3.3.1 Intuition

In the first couple of months after being born, human infants develop only basic motor skills that allow them to feed, grasp objects (without being able to hold them), and turn

their head when in a supine position. Around the age of six months, skills like reaching with arms, sitting, and rolling over are learned. By the age of one year, infants can pull themselves up to a standing position, and sometimes even start to walk with the support of an adult. During the next year the infant, now becoming a toddler, can get up and stand without support, and is able to walk, although still falling quite often. In the following years, the child learns to walk more easily and upright, while avoiding obstacles and learning other balancing skills, such as squatting, jumping, and climbing.

Different regions of the human brain are responsible for different human functions and skills [27]. Sensory cortices receive and process signals of visual, auditory, haptic, and many other kinds of sensory information. Similarly, the primary motor cortex coordinates muscle actuation, while other parts of the brain are involved with thought and decision making, and so on. Most of these regions can be divided into pieces that correspond to more specific functions.

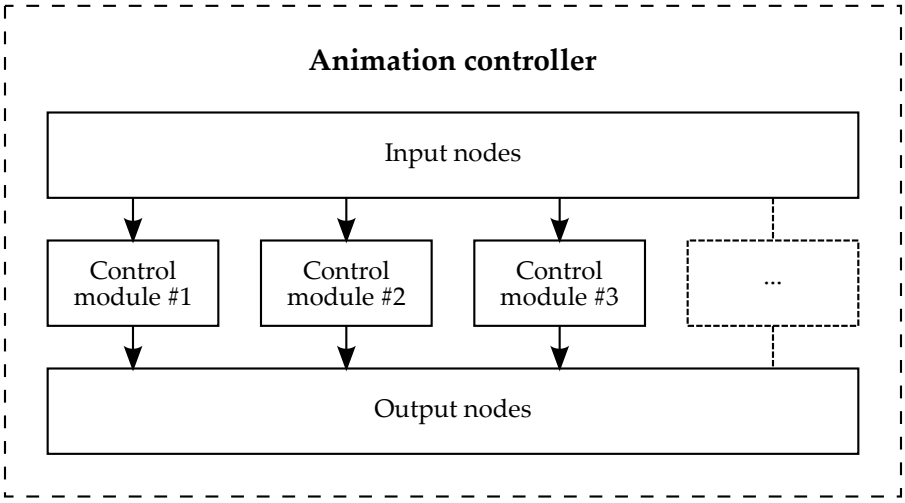


Figure 3.2: *A schematic overview of the animation controller in its simplest form. The input nodes feed sensor and control values into the control modules, which, in turn, produce an actuation pattern that is transferred to the character simulation through the output nodes. The arrows represent full or partial connectivity between in- and output nodes and hidden nodes, which are inside the modules. It is possible for a module to evolve direct connections between input and output nodes.*

The stages of motor development in humans are quite different from the training processes that are used in most physics-based simulations, where higher-level skills, such as walking, are being trained for rather directly. In reality, a toddler has first spent many months learning *other* motor skills, before learning to start walking from an upright body pose. This is the main motivation for the method that is proposed here, which involves a sequence or hierarchy of motor skills that lead up to a desired high-level motor skill. This hierarchy, consisting of one control module per motor skill, forms the final animation controller that will be used to animate a physics-based virtual human.

Analogous to the way the human brain is divided into regions that each have a particular purpose, each control module is an artificial neural network that controls one specific—in this case motor—behavior of the character. And just like an infant starts out by learning a set of rather simple behaviors, the first control module should represent a relatively low-level motor skill. Once the first module is fully trained for, the second module is added to the controller, representing a somewhat higher-level motor skill. During the training phase of the second module, the first module is also present and active, but it is not changed in any way. If the motor skill of the first module is similar to, or part of, the motor skill of the second module, the second training phase may be shorter than it would be if the first module were not present. Moreover, if the first motor skill is indeed—more or less—part of the second motor skill, it may act as bias towards desirable behavior for the rest of the second module, so that the second training phase may actually produce more natural-looking results. In the same way, a third module may benefit from its underlying motor skills, and so on, until the top-level module that represents the final desired behavior is completed.

This method of building control modules on top of one another may not directly match the structure or development of the human brain, but it can resemble the order in which humans develop their various (motor) skills [13]. One of the goals of the experiments in Chapter 4 is to see if this developmental hierarchy shows any benefits over training a single desired motor skill directly.

3.3.2 Specification

In order to animate a physics-based character, an animation controller must be built. In the proposed method, this controller consists of a hierarchy of control modules, that each have their specific target behavior. The animation controller is optimized by an offline optimization process called artificial neuroevolution, which is applied to the control modules—one by one, breadth first, in the order given by their developmental hierarchy.

Each control module is an artificial neural network that has a number of input nodes (for control and sensors) and a number of output nodes (for actuation). The control input nodes can be used for user interaction, while the sensor nodes provide feedback from the simulation to the controller. The output nodes are connected to physical actuators in the virtual human character. Conceptually it makes little difference whether these output values are interpreted as joint torques, desired joint angles, or muscle activations, because the control modules are trained accordingly during evolution.

Because many control modules use the same sensory information, while producing output for largely the same actuators, it makes sense to have a single set of input and output nodes that are shared by all modules across the animation controller. This is achieved by keeping the control modules separate during the evolutionary steps, while merging them together during evaluation (when determining their fitness). The input and output nodes are immutable by evolution, so they can be considered part of the animation

controller, but not part of the control modules per se. Conversely, all hidden nodes and connections that are grown in evolution, can only be part of one of the control modules.

This is why low-level behaviors always remain accessible, even after evolution of higher-level modules. Any low-level behavior can be expressed by simply disabling all neural connections that are part of any higher-level control modules, rendering them completely disabled. Only the control modules that were present during evolution of that particular low-level module remain active, thus the low-level behavior emerges. The other way around, higher-level modules may no longer work correctly if one or more of the underlying modules are disabled. This is because, in the current design, all modules are activated during evaluation, so high-level modules are evolved taking the lower levels into account.

It is important to distinguish the concept of a developmental hierarchy from *bootstrapping*. Although both techniques are used to smoothen the fitness landscape by gradually increasing complexity, bootstrapping does so *during* evolution, while developmental hierarchies act *between* evolutions. Also, in developmental hierarchies the intermediate behaviors remain accessible (as mentioned above), whereas they are generally lost when purely bootstrapping.

In summary, the resulting animation controller can be regarded as a black box that takes a vector of input values, and produces a vector of output values. The controller is connected to a physical simulation of a virtual human character, from which it receives feedback, and to which it provides actuation control information. The novelty of this controller lies in the way the structure of the internal artificial neural network is generated in terms of macro-topology using control modules. A developmental hierarchy is applied to these modules, evolving them in a particular order, and making them interdependent. Every higher-level control module is optimized while all lower-level modules are active, thus only adding to the pre-existing behavior.

3.3.3 Application

As a proof of concept of this method, Chapter 4 and 5, discuss the details and results, respectively, of two experimental setups where an example of a developmental hierarchy is tested. The following target behaviors are chosen because they are intuitively ordered in terms of difficulty and complexity:

Posing One of the most basic actions that is generally performed with virtual characters, is keeping them in a desired body pose. In a physics-based environment, this means that a character should actively compensate for any perturbations from external forces, to keep a predefined rigid body pose.

Standing Balanced stance is similar to posing in that the character should try to keep a (balanced) pose. The difference is that the character should also remain upright, with both feet on the ground, i.e. it should not fall down.

Reaching For the purposes of this work, the reaching behavior is defined as moving the right hand to a target position, while keeping a balanced stance. After reaching the target, the hand should return to its original position. This behavioral pattern is then repeated, until the (virtual) end of time.

From these descriptions it is intuitively clear that these behaviors are not equally complex. Furthermore, they each seem to add complexity to the previous behavior. In the experiments it is hypothesized that when the posing control module is already present, the standing control module will evolve faster (and possibly better). Similarly, the reaching module could then also evolve faster, being based on the combination of the posing and standing modules.

Before evolving the controller, the way in which the output of the artificial neural networks is interpreted needs to be decided. Possible interpretations—with corresponding actuators—include joint torques, target joint angles, muscle activations, and so on.

Because humans generate motion using muscles, virtual muscles may be the most biologically plausible device for animating a physics-based character. The first experiments in this work use musculoskeletal data measured from actual human bodies, as available through the OpenSim project [10, 11, 31, 4, 5, 18]. This includes the muscle path, attachment points to the bones, and other physical properties that describe the muscle's capabilities.

As muscles can only influence torques in the joints that they span, it makes sense to group muscles accordingly. Since muscles can span multiple joints, it is possible for a muscle to be in more than one muscle group. One way that muscles may be used to control a virtual character is by training a control module for each muscle group, so that it can actuate all the degrees-of-freedom of the corresponding joint. This way, higher-level control modules may abstract from having to actuate each muscle individually, by only controlling the muscle groups through their modules.

Having high-level control modules influence lower ones reinforces the idea of the developmental hierarchy, but it also complexifies the final animation controller greatly. When letting control modules evolve sequentially, but having them work in parallel, only the input and output nodes of the animation controller play a role in the complexity of the flow of activation, apart from the module's internal structures. Allowing connections between modules makes the search space much larger, slowing down evolution.

It turns out that using virtual muscles are a little bit too complex for now (more on this in Chapter 5 and 6). Instead of using muscles to generate joint torques, it is also possible to set joint torques directly in the physics engine—and thus interpreting controller output values accordingly. While this approach may be less biologically valid, the results can still look quite natural, which is why joint torques are still very commonly used in physics-based character animation [28, 21, 3, 1].

A second set of experiments involves the controller outputs being interpreted as target joint angles, which are then converted to joint torques using proportional-derivative

(PD) controllers. Each degree-of-freedom of every joint has a PD controller (see Equation 3.2), in which the same k_p gains and torque limits are used as Al Borno et al. [1] find to be appropriate by manual tuning. The k_v values are set to $2\sqrt{k_p}$, which maximizes convergence without overshoot (the PD controller is said to be *critically damped* [14]). The resulting torque τ is then clamped to be within $\pm 200Nm$ and applied to the corresponding joint DoF.

$$\tau = k_p(\theta_d - \theta) + k_v(\dot{\theta}_d - \dot{\theta}) \tag{3.2}$$

Chapter 4

Experiments

In order to get an indication of how good the proposed method works, it is important to have something to compare it to. Thus, a set of animation controllers, that are generated using the proposed method, is compared to a set of baseline controllers. Since the point of the method is to have a developmental hierarchy—consisting of control modules—that guides the evolution of its internal artificial neural network, the baseline controllers are using only a single control module. This is equivalent to evolving the structure and connection weights of a single neural network to fit the desired behavior.

In the following sections, the conditions and setup of the experiments are discussed, along with the fitness function and how it coaxes evolution towards producing better individuals. Section 4.2 and Section 4.3 discuss experiments that each interpret actuation patterns from the controllers in a different way; this influences the complexity of the search space, and may lead to different results.

4.1 General setup

The ultimate behavior of the desired animation controller is being capable of **reaching** a target position with the right hand, while standing upright (as described in Subsection 3.3.3). Intermediate behaviors are **posing** and **standing**. In order to evaluate the proposed method, controllers that are using the developmental hierarchy are compared to ones that are not.

Since posing is, in this project, an elementary behavior (meaning that it has no underlying behaviors), the corresponding experimental and baseline control modules are exactly the same. Therefore, no comparison is needed in this case.

At the level of standing, the experimental case is different from the baseline case in that it includes both the posing and the standing control modules. The baseline standing controller contains only the standing control module.

Similarly, the baseline reaching controller contains only the reaching control module, while the experimental reaching controller contains the posing, standing, and reaching

control modules. This is the most complex controller that is treated in this work, but many more target behaviors or further expansions are conceivable.

Firstly, the posing module is evolved. Secondly, the experimental modules of standing and reaching are evolved, both using their respective predecessor modules. Thirdly, the baseline control modules for standing and reaching are evolved individually. The resulting animations are then inspected visually, to compare their 'naturalness'. During evolution, metrics are stored so that, afterwards, timings and fitness progress can be compared.

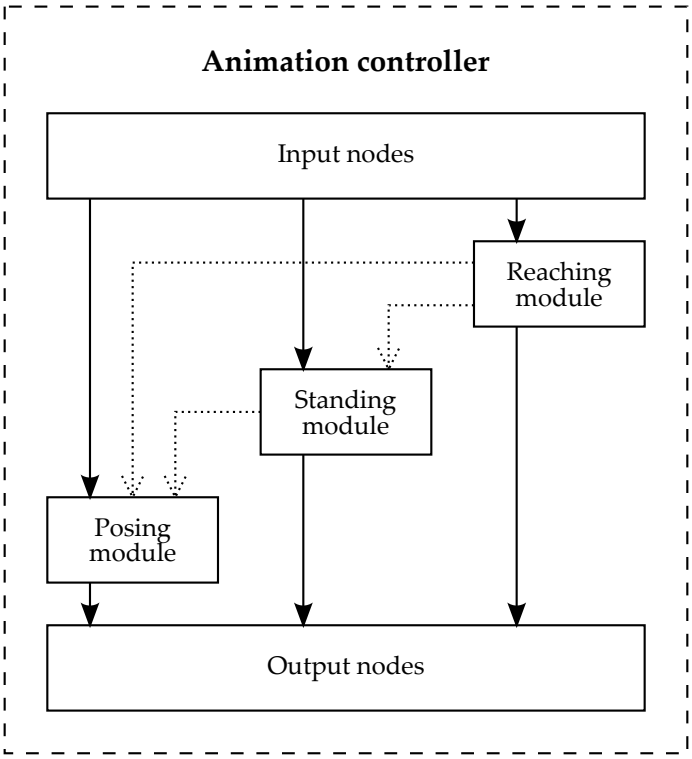


Figure 4.1: *A schematic overview of the final animation controller, as generated using a developmental hierarchy of posing, standing, and reaching. Similarly to Figure 3.2, sensor data flows to the individual control modules, that, in turn, produce an actuation pattern in the output nodes. The dotted arrows represent which lower-level modules are active during evolution of higher-level modules. Section 6.2 discusses how these arrows could, in future projects, represent actual connections, so that modules can influence each other's behavior.*

As discussed above, and shown in Figure 4.1, the control modules are evolved in a particular sequence. In Chapter 3 the method is stated to involve a developmental *hierarchy*, this sequence being a simple form of one. Because of limited time and resources, implementing a developmental hierarchy that includes multiple *branches* is left to future works. One might imagine expanding the controller with a **walking** control module, that derives from the posing and standing modules, and is at the same level as the reaching

module. To achieve these layers, however, it may be required to let high-level modules influence how and when low-level modules are activated. It is possible that not all low-level modules are necessary in establishing all high-level behaviors. This is an example of how employing a developmental hierarchy can lead to a rather complex controller, that can be used for a range of different animations.

4.1.1 Fitness functions

For every evolutionary generation, all individuals in the population have to be evaluated to determine their *fitness*. These fitness values are the driving force behind the evolutionary process (see Section 3.2). The evaluation of an individual consists of running the physical simulation of the virtual human character with the animation controller (derived from the individual's genome) attached to sensors and actuators. The motion of the character is then measured and compared to a number of *objectives*, determining how well the controller is performing. Each control module in the animation controllers is evaluated using a different fitness function, which can contain some of the same objectives.

The fitness values are calculated using the following formula, where f_W denotes the fitness value, given a set W of error-weight value pairs (x, w).

$$f_W = 100 \cdot \left(\frac{\delta t}{T} \prod_{(x,w) \in W} 1 + \frac{w}{1+x} \right) \tag{4.1}$$

The number of simulation frames that have passed without the individual *failing* is denoted δt, with relation to the target number of frames T. A factor 100 is used to boost fitness values, because the NEAT algorithm cannot deal with (initially) very small numbers. The specific error weights for each module are shown in Table 4.1. An individual fails, as a form of early termination, if one of its metrics exceeds a particular threshold, which depends on the control module and is fine-tuned by hand.

An *objective* is a way to express a desired property of the produced animation. For example, when standing, the character should not fall down. This can be quantified by measuring the distance between the character's center-of-mass and a target position. If this distance becomes too large, the character is said to have fallen; the objective is to minimize that distance. Most objectives are implemented using error metrics, meaning that the corresponding values should be optimized or minimized over the course of evolution.

Many error metrics that appear in this work are also used by Al Borno et al. [1], who demonstrate balanced stance, walking, and various other complex motions, using such simple specifications. These error metrics, among a couple of new ones, are used in the control modules as follows.

Posing

The fitness function of the posing control module consists of just two objectives. The first one has a corresponding error measure $E_{restPose}$.

$$E_{restPose} = \sum_{j \in \mathcal{J}} \sum_{t} (\theta_{j,t} - \overline{\theta}_{j,t})^2 \tag{4.2}$$

This term is used to have the character converge towards a particular body pose, as defined by target joint angles $\overline{\theta}_j$. This can be done for either all simulation frames t, or for just the end state, which is the second posing objective. The joint degrees-of-freedom j are in a set $\mathcal{J}$ that comprises a specific number of joints, that depends on which part of the body needs to be constrained.

In the second posing objective, where the end state is evaluated (in $E_{restPoseEnd}$), the sum over all simulation frames is removed from the equation.

$$E_{restPoseEnd} = \sum_{j \in \mathcal{J}} (\theta_j - \overline{\theta}_j)^2 \tag{4.3}$$

A posing individual fails as soon as its $E_{restPose}$ becomes larger than 200π.

Standing

For the standing module, the same objectives and early termination conditions are used as for the posing module, with the addition of two more. These are the ones that make sure that the character not only has an upright pose, but also keeps its balance, by shifting its center-of-mass, and by keeping its feet on the ground.

$$E_{COM} = (\mathbf{c} - \overline{\mathbf{c}})^2 \tag{4.4}$$

where E_{COM} is the squared distance of the character's center-of-mass $\mathbf{c}$ to a target position $\overline{\mathbf{c}}$ at the desired height. This leads to the behavior of the character trying to keep its balance.

$$E_{feet} = y^2_{leftFoot} + y^2_{rightFoot} \tag{4.5}$$

where E_{feet} is the sum of the squared altitudes of both feet, where minimizing leads to the feet staying on the ground, which helps in finding a balanced pose. Both E_{COM} and E_{feet} are calculated at the end of an evaluation, so it is the final body pose that counts.

A standing individual fails as soon as the altitude of the center-of-mass of the character is too low (20 cm sub-target), or when the feet are either too far off the ground (higher than 20 cm), or too far apart (more than 50 cm).

Reaching

Like, the standing module, the reaching module uses all of its predecessors' objectives, and adds two of its own.

The main reaching error metric is the sum of two components, representing moving the right hand $\mathbf{h}$ towards the target position $\overline{\mathbf{h}}_{target}$, and moving back to the rest pose $\overline{\mathbf{h}}_{rest}$. The notation of $t(f)$ is used for those simulation frames that are spent moving the hand towards the target, and $t(b)$ for the ones moving back.

$$E_{reaching} = \sum_{t(f)} (\mathbf{h}_t - \overline{\mathbf{h}}_{target}) + \sum_{t(b)} (\mathbf{h}_t - \overline{\mathbf{h}}_{rest}) \tag{4.6}$$

Finally, an error metric for control torque is added to prevent the reaching arm from showing jittery motions. Such chaotic forces need to be minimized because they can, along with the body's shifted center-of-mass, easily bring the character out of balance. Interestingly, this error metric does not seem to be required for the other two control modules.

$$E_{torque} = \sum_{j,t} \tau_{j,t}^2 \tag{4.7}$$

is the sum of the control torques τ over every degree-of-freedom j of the joints, for all simulation frames (time steps) t.

It should be noted that the reaching controller is the only one that is rewarded an additional fitness bonus for every time the hand successfully reaches the target, and every time it is successfully returned to the rest pose. This is necessary to stimulate evolution to select individuals who reach for the target more than once. Without a bonus for target reaching, additional reaching iterations would result in higher error rates, for which the reward for additional successful simulation frames cannot compensate.

The reaching module has similar early termination thresholds as the standing module, but in calculating $E_{restPose}$, the joints of the right arm are not included when it should be reaching for the target pose. One additional termination condition is a measure of how far the right upper arm is intersecting the thorax. This is undesirable behavior, yet can occur because no collision detection is done between rigid bodies that are directly connected to each other by a joint, in this case the right shoulder.

The behavior of alternating between reaching for the target and the rest pose is governed by a simple *finite state machine*, that switches back and forth between the two states each time one of the goals is reached and a predefined number of simulation frames i has passed. This is what makes the difference between "standing" and "not-reaching". Also, to award a bonus for reaching either goal, the fitness function for reaching individuals is augmented with the awarded bonus and the theoretical maximum bonus (set to $100\frac{T}{i}$).

Module	$E_{restPose}$	$E_{restPoseEnd}$	E_{COM}	E_{feet}	$E_{reaching}$	E_{torque}	Threshold
Posing	3	1	-	-	-	-	95%
Standing	1	1	1	1	-	-	94%
Reaching	1	1	1	1	2	1	55%
Posing (equal)	3	1	-	-	-	-	95%
Standing (b.l.)	2	1	1	1	-	-	93%
Reaching (b.l.)	2	1	1	1	2	1	55%

Table 4.1: *The error weights for each control module's fitness function are tweaked manually. The lower half of the table shows the baseline (b.l.) controllers, with the posing module shown twice for completeness. Threshold fitness percentages (of the theoretical maximum fitness value—which also depends on the error weights) for each module indicate when their performance is deemed "good enough", so that the evolutionary process may be terminated. These thresholds are hand-tuned to fitness levels that are not likely to be exceeded soon.*

4.2 Experiment 1: Muscle-based controller

The first experiment is based on animating the physics-based character using a virtual muscle model, as described in Section 3.1. Because each muscle in the character needs to be actuated individually, each muscle has its own output node in the controller. Because there are so many muscles (192 in total), evolving a neural network that can generate appropriate actuation patterns is a very complex task. To make things worse, in order to provide feedback to the neural network, sensor nodes for all joint DoFs are required. In an attempt to overcome this problem, an extra layer of control modules is added to the hierarchy, dedicated to producing actuation patterns for muscle *groups* instead.

This "lower motor layer" is evolved as an abstraction layer, so that higher-level control modules have a way of generating joint torques without having to address each muscle by themselves. A visualization of just this layer, as combined into a single artificial neural network, is shown in Figure 4.2. The lower motor layer consists of control modules (not to be confused with the way nodes can be clustered in the network) that each control one degree-of-freedom of a joint. Each control module addresses only the output nodes that correspond to muscles in that muscle group, while only one joint DoF sensor node is required per module. Quite similar to the $E_{restPose}$ from Subsection 4.1.1, the fitness function of these modules is based on converging to target joint angles.

The goal of the muscle-based experiment is to see if the proposed method of developmental hierarchies is useful in constructing animation controllers that can generate muscle activation patterns. This can be tested by generating the layers as described above, evolving them, and then comparing them to the baseline controllers.

4.3 Experiment 2: Joint torque-based controller

With muscle-based controllers being a rather ambitious goal, the second experimental setup focuses on a much more commonly used joint torque-based controller. In this setup, there are 27 output nodes to the neural network, each corresponding to a single degree-of-freedom of the character's 15 joints—not all joints having all three rotational DoFs unlocked. This is a number that is quite a lot smaller than the number of muscles in the body. Therefore, no extra abstraction layers are needed to support the poser and higher control modules.

The output actuation values are interpreted as target joint angles, in the same way as Allen et al. [3] do for their controllers. This involves the use of proportional-derivative (PD) controllers to convert target joint angles to joint torques (as discussed in Subsection 3.3.3). The initial structures of the control modules are similar to the initial baseline modules, as shown in Figure 4.3.

The goal of the joint torque-based experiment is to see if the proposed method of developmental hierarchies is useful in constructing animation controllers that can generate desired joint angles under PD control that lead to the desired motions. This is tested by evolving the posing, standing, and reaching modules sequentially, and then comparing the resulting controllers to the baseline controllers.

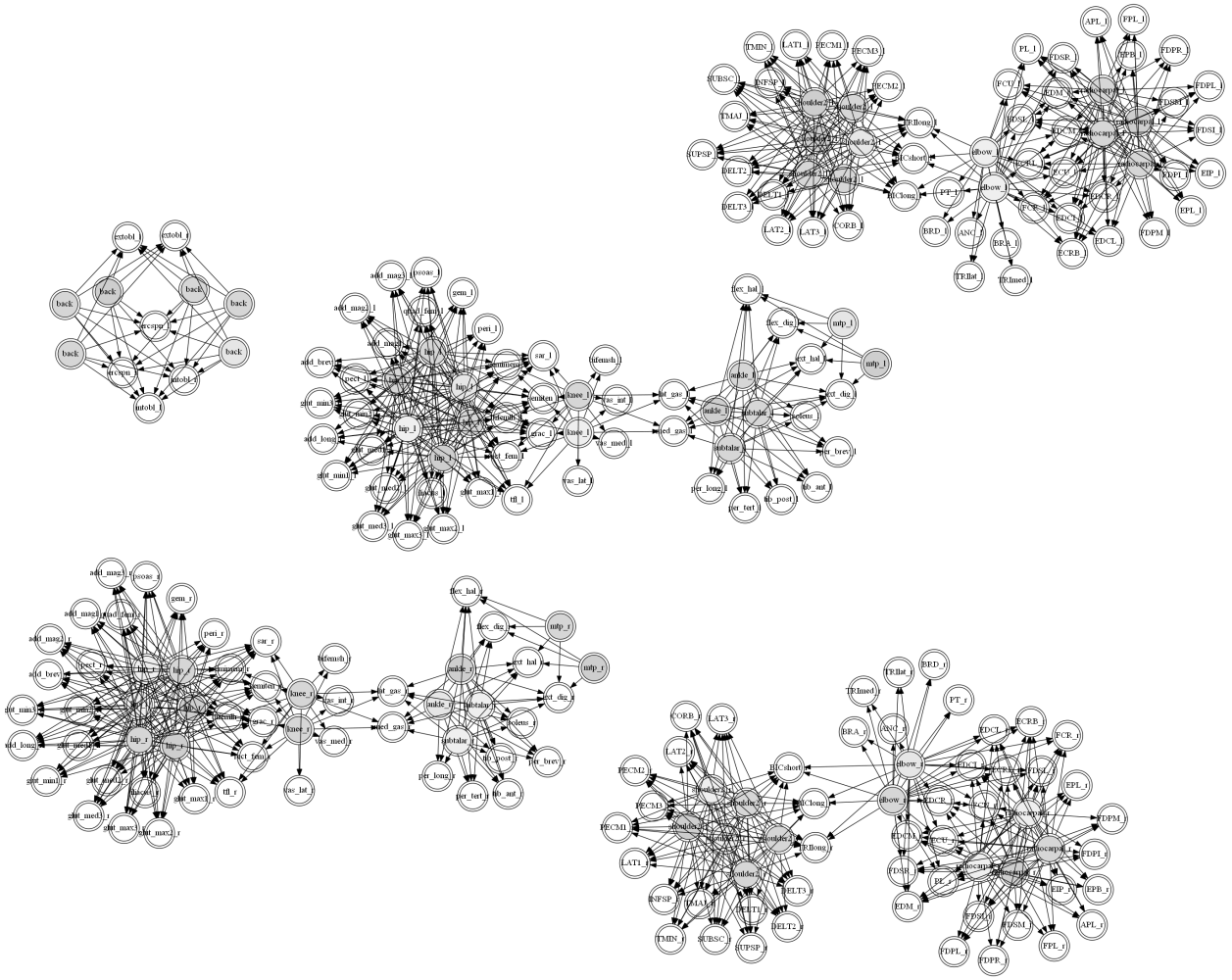


Figure 4.2: *A visual representation of the artificial neural network that forms the "lower motor layer". Note how the nodes are clustered in a way that corresponds to the human character's physiology. The smallest cluster—in the top left—corresponds to muscles that connect the pelvis and thorax. The other two pairs of clusters correspond to the pairs of legs and arms respectively. The white nodes are output nodes, that are each connected to a single muscle. They are fully connected by the colored nodes within the same cluster, which are input nodes, used to control joint actuation. The actual muscle-based animation controller also includes sensor nodes for each joint DoF, which are hidden in this overview. Also, this particular controller network is generated from a starting genome, so it has not been evolved yet. The evolved controller would include many more connections and hidden nodes.*

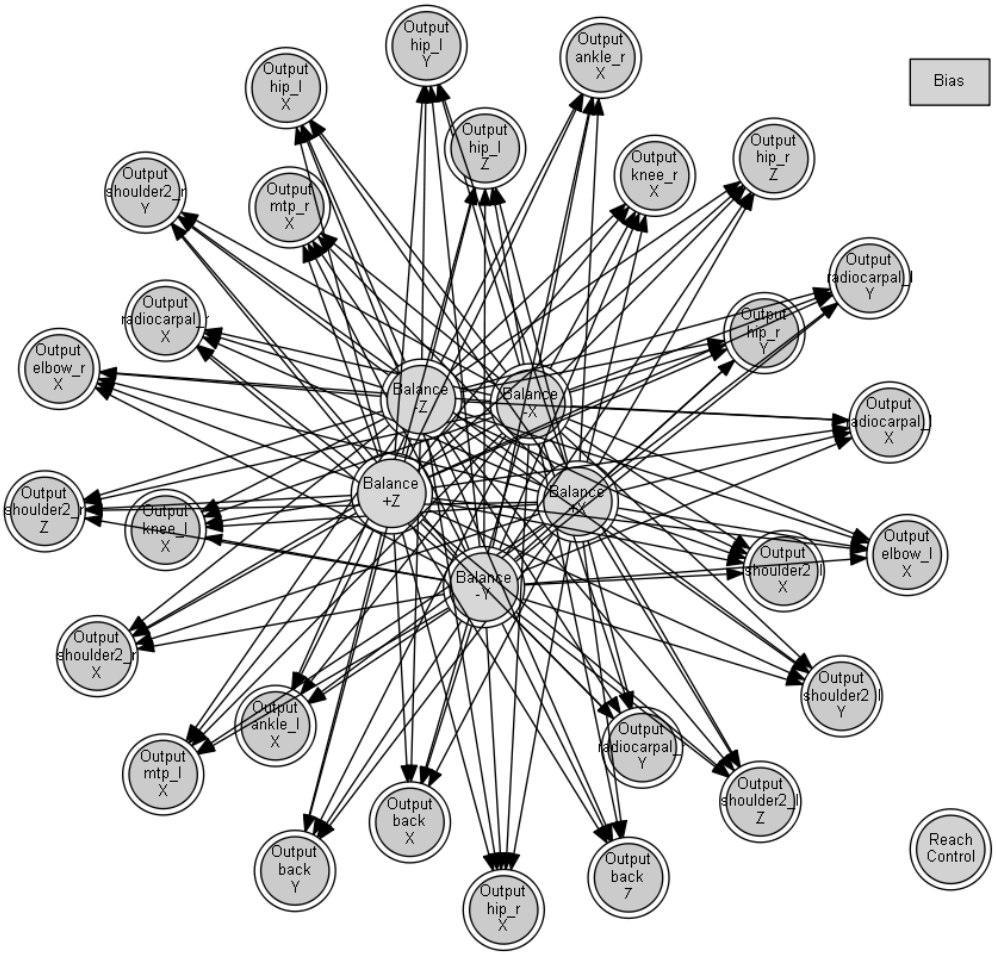


Figure 4.3: *Visualization of the baseline reaching control module's initial artificial neural network. Note how this initial network is much smaller than the muscle-based variant in Figure 4.2. The inner nodes (labeled "Balance") provide sensory data on where the character's center-of-mass is, with relation to a target position, in which the character would be balanced. The surrounding nodes ("Output") each correspond to a degree-of-freedom of one of the joints, of which the names and axes are also in the labels. The sensor nodes and output nodes are fully connected at the start of evolution, because much interaction is expected in the balance (sub-)task: the center-of-mass is easily shifted in all directions. The lower-right "Reach Control" input node is used to advance the reaching finite state machine (Subsection 4.1.1). The top-right "Bias" input node always has an activation of 1.0, allowing the network to spread activation, even though none may be available through other inputs. Any necessary connections from these last two nodes need to emerge over the course of evolution.*

Chapter 5

Results

In the next sections, the results of the various evolutionary processes and the final performance of the animation controllers are shown and compared. Any further interpretation of these analyses is left to the discussion in Chapter 6.

5.1 Results 1: Muscle-based controller

Due to the high complexity of the "lower motor layer", no complete animation controller is produced that is based on muscle actuation. Each individual control module (responsible for a single joint DoF) has to have its own evolutionary process, so it takes a lot of time to process the full body. This allows for but a few opportunities to see the results and to further tweak the evolution parameters.

A side effect of using muscles in the physics engine, is that forces are exerted on the rigid bodies, and not at the joints directly. This leads to instabilities at times when many large opposing muscles, that span a single joint, are activated all at once, causing rigid bodies to intersect and drift into other unnatural positions.

Even with most of the lower level working to some extent, it proves difficult to actually make use of it in terms of higher level control modules. Although each joint DoF can be controlled separately, with the DoFs of all other joints temporarily locked, controlling a complete joint in a meaningful way is quite another problem. For example, the evolution of the posing control module, on top of the lower motor layer, does not converge. Upon closer inspection, the muscle groups seem to produce rather twitchy forces that lead to a chaotic and unstable character.

With the muscle-based controllers being incomplete, it is quite hard to quantify these results. Also, it makes little sense to compare them to any baseline controllers, other than to say they are not working yet. Because of a lack of time to solve the encountered problems, the second experimental setup is given priority. Section 6.2 provides a some pointers for future work to continue with a muscle-based application of developmental

hierarchies, which, as far as the method itself goes, still seems quite promising. This is discussed in Chapter 6.

5.2 Results 2: Joint torque-based controller

The second set of experiments is completed successfully, so that its results can be compared to the baseline controllers. These results are twofold: the animation controller that is generated using a developmental hierarchy is compared to both the "standing", and the "reaching" baseline controllers, as "posing" is equivalent in both the experimental and baseline case.

As opposed to the first experimental setup, the joint torque-based approach actually produces viable animation controllers. Over the course of evolution, the developmental hierarchy of posing, standing, and reaching is playing out nicely, leading to a combined controller that can do all three of those things. The user can toggle individual modules on or off, thus changing the behavior of the controller. Details on the evolutionary progress of the different controllers are shown in Figure 5.1, and summarized in Table 5.1. Without exception, the baseline controllers took longer to evolve (if at all), or otherwise achieved lower levels of fitness.

Module	Generations	Processing time
Posing	42	8 min 25 sec
Standing	36	11 min 55 sec
Reaching	122	35 min 16 sec
Combined	200	55 min 36 sec
Posing (equal)	42	8 min 25 sec
Standing (baseline)	803	240 min 1 sec
Reaching (baseline)	> 1600	> 270 min
Combined (baseline)	> 2445	> 296

Table 5.1: *Shown here, are the number of generations and the amount of time taken for each of the control modules to fully evolve. Posing is shown twice, for completeness.*

5.2.1 Experimental control modules

When visually inspecting the results, the posing behavior results in the character becoming rigid, while trying to keep the target pose (which, in this case, is a rest pose). Applying external forces or collisions to the body makes the body parts diverge from their resting positions slightly, only to quickly return to them immediately.

The standing behavior works as expected, and in this regard it is a bit more interesting, because the character actually has to *do something*. An upright pose is kept, with the

character rocking back and forth a bit at the start. This is due to an initial imbalance, as the character is spawned with its feet a couple of centimeters above the ground. Without applying external forces, the standing behavior lasts indefinitely, but the character is slowly sliding away from its initial position in a sideways spiraling motion. This is caused by the character slightly swaying sideways in trying to balance itself. When recovering balance from leaning backwards, the toes are vibrating up and down a bit. This is easily remedied by imposing passive spring-dampers at the joints, with a spring constant of 30 Nm / rad [15, 30].

Reaching is the most complex task in this experiment, and it therefore takes the longest time to evolve a solution. One of the control nodes of the neural network is used to toggle between the two states of either reaching or not-reaching. This switch is automated in the implementation, so that it toggles once every one or two seconds (simulation time), with a pause of one second in between (stay at target), on the condition that the character has actually reached its target by then. This works: the right hand is raised towards the target, stays there (albeit somewhat shaky), and then moves back whence it came.

5.2.2 Baseline control modules

The baseline standing controller takes a lot (about twelve times) longer to evolve than its hierarchical counterpart, yet results in a balanced stance that is of similar quality. In terms of body pose it is slightly less symmetrical, hanging to its left side, and the upper body is a bit more shaky (originating in the pelvis) while keeping balance. Vibrations in the character's feet are about equally visible, but sliding over the ground plane is more slowly and in a mostly forward direction. The character adopts a wider stance, which may provide better balance under lateral external forces.

Evolution of the baseline reaching controller starts out with a long period in which the controller learns how to keep a balanced pose. The right arm is, indeed, not kept at its resting position to the right of the thorax. Instead, in later generations, it is pulled upward in a jerky motion, independent of whether the reaching state is activated or not. Most of the time this causes the character to lose balance, and fall to the ground. Unfortunately, after passing a time limit of 4.5 hours, evolution does not converge to an acceptable solution.

5.2.3 Comparison

When comparing the results of evolution, as presented above, the experimental modules are coming out ahead in terms of both the time and number of generations that are needed to converge to a solution. The experimental controller is generated in under an hour of evolution, while the baseline controllers take several hours apiece. Also, generations for the baseline controllers seem to take less time (19.9 versus 17.9 sec / generation on

Control module	Input	Hidden	Output	Links	Performance
Standing	6	0	27	135	913.9 act/sec
Standing (baseline)	6	9	27	171	473.0 act/sec

Table 5.2: *A comparison of the artificial neural networks that make up both the experimental and the baseline standing controller. Shown are the number of input, hidden, and output nodes, and the number of connections between nodes. Also, the performance of each network is given as the average number of network activations calculated per second (from 30 samples, each calculating 1 million network activations).*

average for the standing controllers, not tested for statistical significance), but they do require many more of them.

Another interesting observation is that the baseline modules tend to evolve larger genomes (having more connections and hidden nodes), while not necessarily showing better performance (in terms of fitness). This is probably due to the properties of complexification over time that are inherent to the NEAT algorithm. Computations on network activation do take more time on larger networks, but this is hardly noticeable in these applications. For example, when comparing the experimental standing controller network with its baseline (see Table 5.2), the number of network activations per second is much higher than the 60 frames per second that are commonly used in real-time rendering. Neural network visualizations are shown in Figure 5.2.

Judging the naturalness of the motion that is produced is quite hard, and may, in the light of the foundational nature of this project as it developed over time, not be appropriate for now. The goal of achieving natural-looking motion could be better suitable for future projects, where most of the elementary issues have been straightened out. For example, the reaching controller is solving a problem that is almost purely inverse-kinematics. That is, for now it is more important that the targets can be reached, and not so much that it also looks as natural as possible. That being said, because of the basis in biological data, and the possibility of future support for muscle-actuation, developmental hierarchies still show a lot of potential for naturalness.

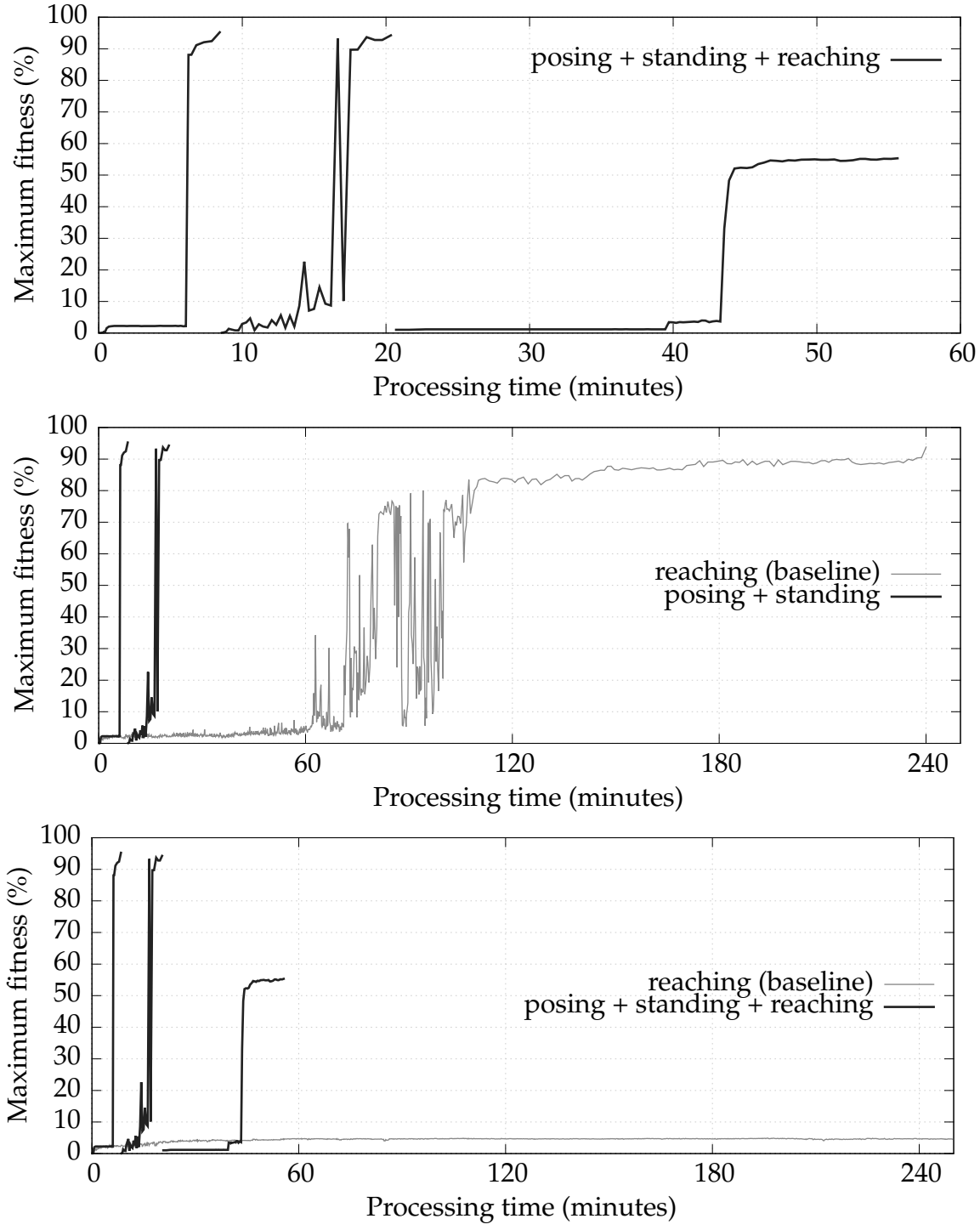


Figure 5.1: *The first graph shows the evolutionary progress of the experimental animation controller with developmental hierarchy. In the other two graphs, the evolution of the experimental controller is compared to that of the two respective baseline controllers.*

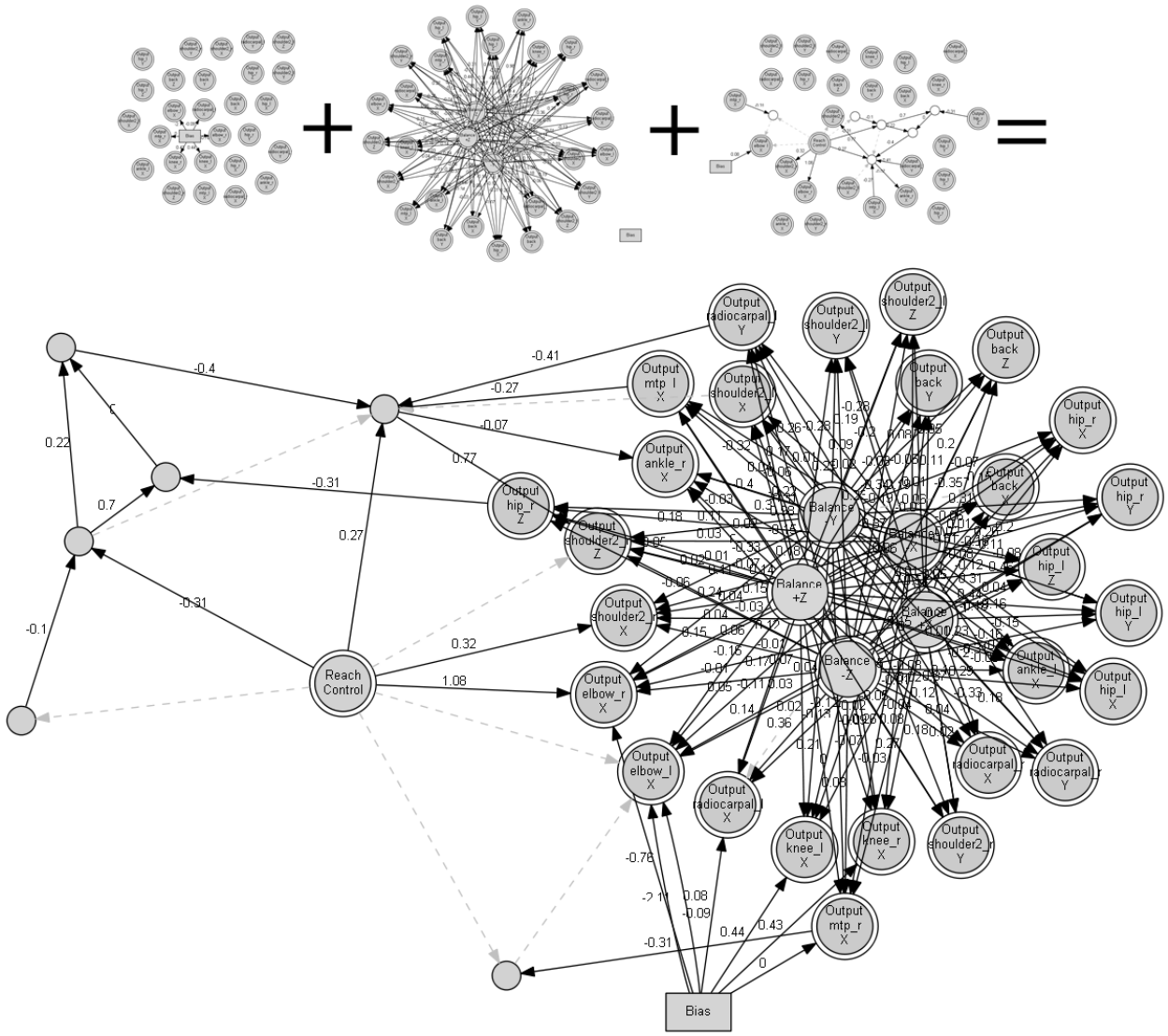


Figure 5.2: *An overview of the different controller module networks that are combined to form the artificial neural network for the "reaching" controller. Note how input and output nodes (doubly encircled) are merged across modules. The dashed links between nodes have become disabled during evolution. Numbers are shown next to all other links to indicate connection weights.*

Chapter 6

Discussion

In this work, a developmental hierarchy is applied to the evolution of a relatively complex physics-based character animation controller. This means that the artificial neural network that makes up that controller is composed from a number of interdependent sub-networks; the control modules. It is hypothesized that evolving these modules one-by-one, with each of them dependent on its predecessors, will allow evolution to converge faster—and possibly with better or more natural-looking results—than for a pair of baseline controllers. Both muscle-based actuation and joint torque-based actuation are tested.

The modern movie and gaming industries show an ever-increasing demand for high-quality computer animations. With manual keyframing out of the question, kinematics not looking natural enough, and motion capturing being very time- and resource-consuming, this encourages a lot of research towards finding a method of automatically generating flexible, physically accurate, and natural-looking animation controllers. Self-organizing modular neural networks, like the ones shown in this work, can be a step toward reaching that goal.

Evolving both the experimental and baseline control modules, based on joint torque actuation, resulted in the hierarchical ones converging to a solution much faster than the baseline. Generating the baseline standing module took about twelve times as long as its hierarchical counterpart, for example, but the resulting controllers performed just about equally well. The baseline reaching controller does not seem to converge to a good solution (at least not within the imposed time limit of 4.5 hours). However, the experimental reaching controller did, to all satisfaction, showing one of the benefits of this method. All in all, the concept of using a developmental hierarchy in neuroevolution works well in the context of this work.

Unfortunately, the experiments that are based on muscle-actuation did not succeed. However, in the light of the results of the joint torque-based controllers, and how hard it is to find the right evolution parameters for them, it is possible that evolving muscle-based control modules simply requires a different fitness function to converge to a solution. An

intermediate step may be to take a set of desired joint torques, and then use a muscle-based variation of Jacobian transpose control [24] to calculate the muscle activations that are necessary to achieve those torques, as done by Geijtenbeek et al. [15]. Investing more time and resources in finding a setup that is more suitable to muscle-actuation than the one presented in this work, may still prove to be worth the effort.

According to these results, the benefits of using developmental hierarchies in character animation controllers still hold. The modularity of the controller allows for all intermediate behaviors to be used, where the baseline controllers are like a black box in that regard. The hierarchical approach evolved faster, and also more reliably: tuning parameter settings results in different evolution times and solutions, where the baseline controllers would oftentimes fail to converge. It is also possible to take out and re-evolve just one control module, but any higher-level modules may malfunction: the separation of responsibilities between modules should be investigated further.

However, evaluating the naturalness of the results is rather difficult. Both baseline and experimental animation controllers generate motion that is, in terms of human behavior, somewhat awkward, to say the least. Of course, the target behaviors are of a fairly mechanical nature, so one may argue that naturalness is of little relevance for now. This may change as soon as larger hierarchies, generating more diverse motion, are tested in a context where behaviors are chained to show more complex actions. Think of navigating an obstacle course, or walking over to an object and picking it up, or interactions between characters.

It is important to mention that it is very hard to find a set of parameters that reliably leads to good controllers. The number of possible combinations of configuring the physical simulation, the evolution parameters, and the fitness functions combined, is absolutely huge. This makes it quite difficult to evaluate the robustness of the method's ability of generating useful animation controllers in general. Nonetheless, in these particular experiments, developmental hierarchies are easier to configure than the baseline cases. If only because their evolution converged faster, thus revealing the effects of the current parameterization much faster.

6.1 Conclusion

The proof of concept for developmental hierarchies in neuroevolution of physics-based animation controllers has succeeded. This work shows that, for at least the joint torque-based approach, the resulting animation controllers evolve at a faster rate of convergence than the baseline, while performing comparably well at generating the desired motions. For complex (compounded) animations, these methods may work even better, because in such cases the baseline method is unlikely to converge within a reasonable amount of time, if at all.

However, the large number of parameters that have to be manually tuned, and the un-

predictable nature of neuroevolution in general, form a complex problem on their own. Finding a set of parameters that will reliably lead to the best possible results is a non-trivial task. That being said, developmental hierarchies in neuroevolution are a promising, and—to the author's knowledge—novel approach to creating complex animation controllers from a set of simple objectives.

6.2 Future work

Generating modular animation controllers with relatively simple developmental hierarchies, like the ones used in this work, are but the beginning of exploring the new possibilities of this method. Many improvements, tweaks, and expansions are possible, both in the method itself and in its scientific context. The following listing discusses some of the ideas that came to mind over the course of this project's development.

Complexification: It may be interesting, and, indeed, necessary, to build and test more complex and diverse developmental hierarchies. The relatively simple posing - standing - reaching sequence that is demonstrated here clearly does not stretch the limits of what is possible using these methods.

Muscle actuation: Even though the initial experiment on muscle-based actuation did not succeed, it is conceivable that exchanging the current fitness functions for more suitable ones may eventually lead to better results.

Interconnectivity: In the current setup, higher-level control modules are *forced* to deal with all their predecessor's behaviors. One of the original ideas, which has not been tested due to time constraints, includes top-down interaction between modules, so that high-level modules can actively control the activation of all lower-level modules. This may enable even more complex behaviors, because unnecessary and unwanted modules can be switched off at any time, and vice versa.

Hot-swapping: It may be worth investigating to what extent behaviors between layers are overlapping. If all modules are strictly adding actuation patterns that are needed for their own target behavior, it may be possible to perform "brain surgery". This could enable the re-training of modules that are underperforming, while all other modules remain intact, or trying out different solutions to the same problem by swapping modules at will.

Off-line optimization: In this work, the original (yet slightly modified) implementation of the NEAT algorithm is used. In theory, however, the concept of developmental hierarchies can also be applied to other off-line optimization techniques. Examples of particularly interesting alternatives, as used in many related works, include HyperNEAT and Covariance Matrix Adaptation (CMA) [16].

Parameter reduction: The huge number of parameters that need to be tweaked in order to reach optimal evolutionary performance is undesirable, as it is a tedious and time-consuming task of trial-and-error. Finding a way to reduce this number would be of great benefit to many research projects. It may be possible to find a set of parameters that can be kept constant across projects, thus limiting the search space.

Reflexes: Fisher [13] mentions his suspicions that there may be a "reflex tier" in the hierarchical development of skills, of which reflexes in infants may be the initial units from which skills are constructed. It may be interesting to build a library of (biologically) known and verified human reflexes, and building abstractions in the form of developmental hierarchies from that.

Standard model: One of the problems on the physical simulation side of this work lies in the fact that there does not seem to be a standardized way of dealing with physics-based character models. Most of the works in the field use either ODE or Bullet, but there are no neuromusculoskeletal character models that are used by a wide range of researchers. This may be due to the pioneering nature of current active-physics-based research. Of course, the OpenSim project is of great value because of its tools and datasets, but building custom experiments from their data is still a lot of work. Having a common collection of such models may be beneficial to the field, as it can save people a lot of time and resources. But, most importantly, it allows for easy comparison of different methods.

Appendix A

Parameters

When working with neuroevolution there are so many parameters that need to be taken into account, that it is easy to lose track of them all. To address this, the following tables give an overview of the (most important) parameter settings that are used in this project. Apart from the NEAT parameters, also the fitness function parameters are included (Subsection 4.1.1), as well as some of the physical properties of the physical human body (Section 3.1).

Mass	Body parts	Joints	Muscles
75.16 kg	16	15	192

Table A.1: *A short summary of the body's physical properties.*

Body part	Mass	Friction coefficient
pelvis	15.6%	0.5
thorax	35.7%	0.5
femur (s)	12.4%	0.5
tibia (s)	4.9%	0.5
talus (s)	1.8%	0.8
toes (s)	0.3%	0.8
humerus (s)	2.7%	0.5
forearm (s)	1.6%	0.5
hand (s)	0.6%	0.5

Table A.2: *Mass distribution over the body parts, which are often named by bones they contain. The body parts marked with (s) should be counted twice, due to the body's bilateral symmetry. Friction coefficients for the feet are higher to reflect the effects of rubber shoe soles.*

NEAT parameter	Experimental	Baseline
trait param mut prob	0.0	
trait mutation power	0.0	
linktrait mut sig	0.0	
nodetrait mut sig	0.0	
weigh mut power	0.5	
recur prob	0.0	
disjoint coeff	1.0	
excess coeff	1.0	
mutdiff coeff	0.4	
compat thresh	3.0	
age significance	1.0	
survival thresh	0.20	
mutate only prob	0.25	
mutate random trait prob	0.0	
mutate link trait prob	0.0	
mutate node trait prob	0.0	
mutate link weights prob	0.9	
mutate toggle enable prob	0.001	
mutate gene reenable prob	0.0001	
mutate add node prob	0.03	0.003
mutate add link prob	0.05	0.005
interspecies mate rate	0.005	
mate multipoint prob	0.6	
mate multipoint avg prob	0.4	
mate singlepoint prob	0.0	
mate only prob	0.2	
recur only prob	0.0	
pop size	512	
dropoff age	32	64
newlink tries	20	
print every	never	
babies stolen	0	
num runs	1	
normalize	1	0

Table A.3: *NEAT parameters for experimental and baseline cases. Only different values are shown in the baseline column: the mutation parameters are lowered to prevent the genome size from exploding. The dropoff age is increased, to give younger species a better chance of developing good performance. The normalization parameter toggles normalizing for genome size in the genome compatibility calculations.*

Appendix B

Software

Over the course of this master's thesis project, a variety of software packages has been used. The following sections give a brief overview of the way in which these tools and libraries are used. All software is run on Microsoft Windows (7/8.1), with the exception of an SVN repository, which is being hosted on a Linux machine.

Microsoft Visual Studio: For the main implementation of the experimental setups Microsoft Visual Studio 2010 is used, with the programming language of choice being C++. All neuroevolutionary code is based on NEAT (NeuroEvolution of Augmenting Topologies) [23], by adapting the original implementation.

The Bullet physics library [8] is used for rigid body simulation and collision detection. A custom renderer is written using the OpenGL graphics API and The OpenGL Extension Wrangler Library (GLEW). Image manipulation is done through DevIL.

Revision control for this project is managed with Subversion (SVN), both through the TortoiseSVN shell extension and the AnkhSVN plugin for Visual Studio.

OpenSim: Most of the physical data used to construct the virtual human character comes from, or is derived from, data sets provided through the OpenSim project [10, 11, 31, 4, 5, 18]. In particular, the "Gait2329" model and parts from the "ULB" model, based on the "UpperExtremities" model, were used.

Eclipse Kepler: This thesis is itself written in LaTeX, through MiKTeX, using the TeXlipse plugin for Eclipse. The bibliography is managed with BibTeX. Revision control is managed with SVN, using the Subclipse plugin for Eclipse.

Other tools that are used include GraphViz (digraph, sfdp), to create all visualizations of artificial neural networks. The relevant scripts are generated during evolution by the experimentation programs. Schematic overviews of the animation controllers are made using InkScape.

Bibliography

[1] Mazen Al Borno, Martin de Lasa, and Aaron Hertzmann. Trajectory optimization for full-body movements with complex contacts. *Visualization and Computer Graphics, IEEE Transactions on*, 2012.

[2] R McNeill Alexander. Design by numbers. *Nature*, 412(6847):591–591, 2001.

[3] B Allen and P Faloutsos. Evolved controllers for simulated locomotion. *Motion in Games*, pages 219–230, 2009.

[4] Frank C Anderson and Marcus G Pandy. A dynamic optimization solution for vertical jumping in three dimensions. *Computer Methods in Biomechanics and Biomedical Engineering*, 2(3):201–231, 1999.

[5] Frank C Anderson, Marcus G Pandy, et al. Dynamic optimization of human walking. *Journal of Biomechanical Engineering*, 123(5):381–390, 2001.

[6] Kent C Berridge. Contributions of philip teitelbaum to affective neuroscience. *Behavioural Brain Research*, 231(2):396–403, 2012.

[7] CM Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

[8] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics.org*, 2006.

[9] H De Garis. Genetic programming - building artificial nervous systems with genetically programmed neural network modules, 1990.

[10] Scott L Delp, Frank C Anderson, Allison S Arnold, Peter Loan, Ayman Habib, Chand T John, Eran Guendelman, and Darryl G Thelen. Opensim: open-source software to create and analyze dynamic simulations of movement. *Biomedical Engineering, IEEE Transactions on*, 54(11):1940–1950, 2007.

[11] Scott L Delp, J Peter Loan, Melissa G Hoy, Felix E Zajac, Eric L Topp, and Joseph M Rosen. An interactive graphics-based model of the lower extremity to study orthopaedic surgical procedures. *Biomedical Engineering, IEEE Transactions on*, 37(8):757–767, 1990.

[12] Agosten E Eiben and James E Smith. *Introduction to evolutionary computing*, volume 2. Springer Berlin, 2010.

[13] Kurt W Fischer. A theory of cognitive development: The control and construction of hierarchies of skills. *Psychological review*, 87(6):477, 1980.

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

[15] Thomas Geijtenbeek, Michiel van de Panne, and A Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013.

[16] Nikolaus Hansen. The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer, 2006.

[17] Kazunori Hase, Kazuo Miyashita, Sooyol Ok, and Yoshiki Arakawa. Human gait simulation with a neuromusculoskeletal model and evolutionary computation. *The Journal of Visualization and Computer Animation*, 14(2):73–92, 2003.

[18] Katherine RS Holzbaur, Wendy M Murray, and Scott L Delp. A model of the upper extremity for simulating musculoskeletal surgery and analyzing neuromuscular control. *Annals of biomedical engineering*, 33(6):829–840, 2005.

[19] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *arXiv preprint cs/9605103*, 1996.

[20] John R Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[21] T Reil and P Husbands. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *Evolutionary Computation, IEEE Transactions on*, 6(2):159–168, 2002.

[22] K Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 1994.

[23] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[24] Craig Sunada, Dalila Argaez, Steven Dubowsky, and Constantinos Mavroidis. A coordinated jacobian transpose control for mobile multi-limbed robotic systems. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 1910–1915. IEEE, 1994.

[25] G Taga. A model of the neuro-musculo-skeletal system for human locomotion. *Biological Cybernetics*, 73(2):97–111, 1995.

[26] G Taga. A model of the neuro-musculo-skeletal system for anticipatory adjustment of human locomotion during obstacle avoidance. *Biological Cybernetics*, 78(1):9–17, 1998.

[27] Philip Teitelbaum. Levels of integration of the operant. *Handbook of operant behavior*, pages 7–27, 1977.

[28] M Van de Panne and E Fiume. Sensor-actuator networks. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 335–342. ACM, 1993.

[29] Jack M Wang, David J Fleet, and Aaron Hertzmann. Optimizing walking controllers for uncertain inputs and environments. *ACM Transactions on Graphics (TOG)*, 29(4):73, 2010.

[30] JM Wang, SR Hamner, SL Delp, and V Koltun. Optimizing locomotion controllers using biologically-based actuators and objectives. *ACM Transactions on Graphics (TOG)*, 31(4):25, 2012.

[31] Gary T Yamaguchi and Felix E Zajac. A planar model of the knee joint to characterize the knee extensor mechanism. *Journal of Biomechanics*, 22(1):1–10, 1989.