FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Using Neuroevolution for Physics-Based Character Animation in Two Dimensions

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Mag.rer.soc.oec. Andreas Gegendorfer, Bakk.techn.
Registration Number 0301584

to the Faculty of Informatics
at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

Vienna, 3rd July, 2017

Andreas Gegendorfer	Thomas Eiter

Erklärung zur Verfassung der Arbeit

Mag.rer.soc.oec. Andreas Gegendorfer, Bakk.techn.
Seidlgasse 8/1/2
A-1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Juli 2017

Andreas Gegendorfer

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Diplomarbeit unterstützt und mir dabei geholfen haben.

Insbesondere danke ich Herrn Prof. Thomas Eiter dafür, dass er mir ermöglicht hat mein Wunschthema zu bearbeiten und für die freundliche und konstruktive Begleitung während der gesamten Umsetzungsphase.
Besonderer Dank gilt auch meiner Freundin, Ruth, die mir zur Seite gestanden ist und natürlich meinen Eltern, die mich nicht nur während dieser Zeit sondern das ganze Studium über unterstützt haben.

Vielen Dank!

Kurzfassung

Die prozedurale Generierung von Inhalten zur Verwendung in digitalen Produktionen erfreut sich seit einigen Jahren großer Beliebtheit. Im Hinblick auf die Erstellung von Computerspielen profitieren sowohl große Firmen als auch kleine Studios davon, Arbeitsschritte und -abläufe an Maschinen zu delegieren. Im Gegensatz zur Verwendung relativ einfacher und statischer Inhalte wie Texturen oder Terrain, ist die Nutzung prozedural generierter Animationen im kommerziellen Bereich eher ungewöhnlich. Diese Arbeit untersucht einen Teil der Frage, inwieweit die aus dem Gebiet der künstlichen Intelligenz stammende Methode der Neuroevolution bei der automatisierten Erstellung unterschiedlicher Animationen einsetzbar ist. Konkret wird betrachtet, inwieweit sich Neuroevolution zur Realisierung physikbasierter Animationen von zweidimensionalen Charakteren eignet.

Die Analyse des momentanen Standes der Forschung und Entwicklung im Bereich der Neuroevolution, aber auch der physikbasierten Charakteranimation, bildet die Grundlage für das Design eines Konzeptes, das in weiterer Folge zur Umsetzung gelangt. Im Zentrum dieses Konzepts steht die Realisierung einer prototypischen Implementierung, die es erlaubt, ansprechende Animationen für unterschiedlichste Charaktermodelle und Kreaturen zu erzeugen. Neben der Fähigkeit, die gestellte Aufgabe gut zu bewältigen, sind große Flexibilität, möglichst einfache Verwendbarkeit und hohe Leistung Anforderungen, die mit Blick auf einen praktikablen Einsatz in der Spiele-Branche an das Programm gestellt werden. Als technischer Unterbau fungieren die Spiele-Engine Unity, die ihrerseits Anforderungen wie hohe Flexibilität und einfache Verwendbarkeit erfüllt, und die bereits darin enthaltene Physik-Engine Box2D. Die Umsetzungsphase wird durch umfangreiche Tests und Experimente unterstützt, die aufgrund der bis dato nur spärlich verfügbaren Erfahrungswerte zu diesem Thema von großer Wichtigkeit sind.

Am Ende dieser Arbeit steht ein vielseitiger Softwareprototyp, der bereits in den wesentlichen Punkten überzeugen kann und die ansprechende Umsetzung verschiedener Bewegungsarten für virtuelle 2D-Charaktere erlaubt. Untermauert wird diese Erkenntnis durch die Ergebnisse einer ausgedehnten Testphase, in der sowohl technische Messergebnisse als auch ästhetische Aspekte, deren Bewertung mittels einer Umfrage vorgenommen wird, von Relevanz sind. Das Fazit fällt großteils positiv aus und die Arbeit stellt einen soliden Startpunkt für weitere Unternehmungen im Bereich der Nutzung von Neuroevolution für die physikbasierte Charakteranimation dar. Die Untersuchung möglicher Anpassungen des Prototypen an die erhöhten Anforderungen von Echtzeitanwendungen wird als ein denkbarer nächster Schritt präsentiert.

Abstract

The procedural generation of content for usage in digital productions has become rather popular in recent years. With regard to the production of computer games, both large companies and small studios can benefit from delegating certain tasks and processes to machines. In contrast to the utilization of relatively simple and static content, such as textures or terrain, the usage of procedurally generated animations is only rarely seen in the commercial game development sector. This work examines part of the question as to how far the method of neuroevolution, a technique originating from the field of artificial intelligence, can be used for the automated generation of various animations. Specifically, the extent to which neuroevolution is suitable for the realization of physics-based animations for two-dimensional characters is considered.
The analysis of the current state of the art in research and development in the areas of neuroevolution and physics-based character animation forms the basis for the design of a concept that is implemented subsequently. At the heart of this concept stands the realization of a prototypical implementation, that allows the creation of appealing animations for different character models and creatures. In addition to being able to handle the given task well and in view of a practical application in the game industry, the program is desired to provide great flexibility, ease of use, and high performance. The game engine Unity, which in turn meets requirements such as high flexibility and ease of use, and the already integrated physics engine Box2D are the technological foundation for the prototype. The implementation phase is supported by extensive tests and experiments, which are of great importance due to the limited availability of intelligence on this topic up to the present time.
The result of this work is a versatile software prototype, that can already fulfill the most important tasks and handle the appealing animation of different types of movements for virtual 2D-characters. This property is underlined by the results of an extended testing phase in which both results of technical measurements and aesthetic aspects (which are assessed by means of a survey) are taken into account. Therefore the overall conclusion concerning the outcome of this work remains largely positive and its results provide a solid starting point for further research on using neuroevolution for physics-based character animation. The investigation of possible adaptations of the prototype to the special requirements of real-time applications is presented as a possible next step.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem

Using artificial intelligence (AI) for content creation and various other tasks in digital productions looks increasingly appealing with the advent of more and more powerful computers, that allow the sensible utilization of sophisticated AI techniques like neuroevolution. Looking at different digital products, an area that profits quite a lot from this development and where AI methods are already widely adopted is computer games (cf. [1]). Video games have become a big part of the modern media landscape with production values of hundreds of millions of euros not being unthinkable anymore. The costs of and the time needed for creating high quality content are ever increasing and techniques that can help in reducing these expenditures are more relevant than ever.
Although neuroevolution could be of value in regards to this dilemma and although it already has been sensibly and beneficially utilized in the development of computer games (cf. Chapter 2.2.2), it is rarely actually used for commercial productions. The reasons for that might be manifold. First of all the technique might not be known to many developers in the industry and specific use cases might be missing. But more importantly the results produced by neuroevolution are somewhat unpredictable[1] and its usage in a production environment could therefore be perceived as problematic. While this concern might be valid for a possible usage in a live gaming environment, this does not have to be an issue for lots of other tasks that can be performed before deploying a game.
One such task, that is often also resource intensive, is the animation of characters in a game. Physics-based character animation is an approach to creating naturally looking animation without having to model every tiny movement by hand. Although there has been quite some research concerning the movement of three-dimensional characters (cf. [2]), only little research has been done on the topic of moving two-dimensional creatures after some early work by Van de Panne and Fiume [3], presented almost 25 years ago.

[1] This is an issue for many other AI techniques as well.

While modeling locomotion of 3D-characters using a realistic layout of simulated joints and muscles is interesting from an academic point of view it is not easy and also not always necessary to use such a complex model in a gaming environment. The animation of simplified 2D models could help in attenuating problems with long animation times, performance, and animation quality. It is therefore potentially well suited to being used in the development of computer games and other digital media products.
Finding good and flexible solutions that allow the automated development of such animations in a gaming environment remains an open problem. In fact Togelius et al. [4] mention overcoming the *animation bottleneck* as one of eight major challenges in the field of procedural content generation. The usage of neuroevolution to approach the task of creating physics-based character animations seems promising and researching this possibility therefore a worthy endeavor.

1.2 Aim of the Work

The main aim of this work is the realization of a prototypical software solution that allows the automated simulation of appealing movements for various different 2D characters, which can be built from simple building blocks of a common physics engine. Such a character could be, for example, a four-legged creature where the legs and the torso are connected through joints and muscles or a worm without any legs at all. The software should be able to deliver movement strategies depending on the physical parameters that define a given character and the inputs and outputs connecting it to the surrounding world. These strategies will be developed using neuroevolution.
Next to delivering a practically usable solution for simple physics-based animation tasks, it shall be explored if such a solution would be interesting for an application in a real-time environment also. This means determining if movement strategies can be evolved in a relatively short period of time so that they could be used to animate characters that are created during a live gaming session[2].
Last but not least this work shall point out a specific area of application for neuroevolution in game development and potentially add an interesting approach to the wealth of procedural content generation techniques that are commonly used in this field.

1.3 Method and Structure

Corresponding to the main aim of the work stated in the last section, the strategy presented here is chosen first and foremost to enable a successful implementation of a fitting software solution for the given problem of 2D character animation. Subsequently this solution can serve as foundation for answering all further questions. The approach comprises the following parts:

[2]Such characters could be created by the players of the game or by using some suitable procedural content generation methods.

- *Conceptual research.* A theoretical basis for the work has to be established before actually starting with designing a specific software solution. Different types of neuroevolution have to be considered and ways of utilizing them for physics-based character animation are to be explored. Reviewing the literature and current state of the art is essential during this phase.
- *Designing a solution.* A solution will be designed according to the previously gained insights. Although many specifics are only determinable after the research phase (e.g. the best type of neuroevolution strategy to use or what types of movement can be implemented), there are some desirable properties that the software produced during this work should definitely possess in order to be usable in practice.
  Flexibility is probably the most important such property. In the end it shall be possible to simply feed a number of input parameters, such as the position of a character's legs and the rotation of its torso, to the program and get some output values that determine the actions this character will set in order to perform a movement. Ease of use would be another such property.
- *Implementing the solution.* A big part of this work will be the actual implementation of the software solution that has been developed before. Next to using neuroevolution as AI technique for evolving motion controllers for the various characters, the game engine Unity [5] is chosen as technical framework for this work. The reasons for using this technology are that it is widely used by game developers around the globe, that it is available for many different gaming platforms, and personal experience.
- *Evaluating the solution.* The implemented solution will be evaluated with respect to the achieved results. Some exemplary questions that shall be answered are:
  - Has it been possible to evolve efficient movement mechanics for different characters (e.g. how far can a creature run in a given amount of time?)?
  - Are there limits to the complexity of characters that can be animated (e.g. the number of legs)?
  - How long does it take the program to find an acceptable solution – is it fast enough to be used in a real-time environment?

The structure of this thesis reflects these parts to a great extent. Chapter 2 presents the concepts used here in greater depth and discusses some deliberations that lead to the choice of certain techniques. Chapter 3 is concerned with the explanation of the solution devised to tackle the given task and Chapter 4 shows how it is implemented. Afterwards a detailed account of the testing and evaluation phase is given in Chapter 5. Ultimately Chapter 6 completes this work by giving a final discussion and conclusion.

CHAPTER 2

State of the Art

In order to set the stage for the upcoming chapters and to lay the groundwork for more advanced topics relevant to this work, the essentials of some elementary concepts are presented in the first section of this chapter. The two remaining sections are concerned with the two concepts that are central to this work: neuroevolution and physics-based animation.

2.1 Background

Machine learning, artificial neural networks, and evolutionary algorithms represent building blocks for the technique of neuroevolution. The remaining subsections provide some additional context relevant to this work in general.

2.1.1 Machine Learning

Machine learning (ML) is an important part of the vast research field of AI that deals with the automatic improvement of already achieved results once new information is available. In respect to ML, Tom M. Mitchell states more formally that: *"[...] a machine learns with respect to a particular task T, performance metric P, and type of experience E, if the system reliably improves its performance P at task T, following experience E."* ([6, p.1]). Key to ML is the focus on automation, meaning that there might be human-computer interaction on the training part but that the learning is done autonomously by the machine.

While there are some overlapping areas, it is common to distinguish three different categories of ML:

1. **Supervised learning.** Under the supervised learning category a machine is provided with a set of labeled training examples that it can learn from. These examples consist of some inputs and the correctly corresponding outputs (*labels*)

and are often provided by human experts. The objective here is to learn a function that allows the correct labeling of new, initially unlabeled, inputs. One field of application for supervised ML that became very popular in recent years is image recognition (mainly due to the fact that the processing power of computers reached sufficiently high levels and that large image databases, providing a substantial amount of training examples, are available as of today).
A simple set of training examples in this area could be pictures denoted with the labels: "human", "horse", and "other". Following the notation of Mitchell mentioned above, the task T could be to learn a function that determines if a given picture depicts a horse or not. It hereby maps an input to an output. Based on all the training experience E a machine gathered by analyzing thousands of illustrations of horses, humans, and other motives it can make informed decisions on pictures it never encountered before and the quality of those decision can be measured as performance P (e.g. the number of correctly categorized images).

2. **Unsupervised learning.** In contrast to supervised learning, unsupervised learning works on unlabeled training data. Instead of input-output combinations it only requires that some input is provided and learning starts from there. The goal of unsupervised learning is often to find previously unrecognized patterns and structures within big amounts of data.
Although this technique might not be the first choice for image recognition, it can still be applied to the example from before to illustrate the differences a little better. In this case T would be to learn the underlying structure of the images in the database and to assign newly encountered pictures to one of the discovered groups that share certain features (*clustering*). As for supervised learning, if E increases the quality of the results should increase as well because the differences and similarities between different groups of images become more and more obvious. In this scenario P indicates the machine's ability to assign new pictures to the correct cluster (e.g. a horse to the group of horses).

3. **Reinforcement learning.** Of the three ML categories presented here reinforcement learning is the most general one. In contrast to supervised learning a machine utilizing this paradigm does not have to be trained using a set of labeled examples but at the same time it still has to receive some kind of feedback about the quality of the decisions it is making and therefore this technique also differs from unsupervised learning. As for the other categories there has to be some kind of sensible input that stimulates the machine to produce a corresponding output. This result is then either rewarded or punished/not rewarded to steer the learning process in the wanted direction.
Reinforcement learning is frequently used by agents in dynamic environments to learn certain abilities like pole balancing or navigating a maze. Still it might be applied to our example. Again T designates the task of identifying pictures of horses and P tells how well the program is able to tell if there is in fact a horse on the image. But this time there is another usage for the metric P because it might also serve as a measurement on how much reinforcement the machine should receive.

Correctly classified pictures result in a reward and subsequently the computer is more likely to classify another similar image the same way. Incorrectly classified images will not result in receiving a reward and therefore the machine might try classifying other similar pictures differently. Again the quality of the results usually improves if E increases (not indefinitely).

ML application

ML is widely used nowadays and its fields of application range from natural language processing over search engine optimization to robotics and various other areas. While in some cases it might be possible to use different learning types for a given problem (e.g. supervised or reinforcement learning for the task of identifying horses in the example above), techniques associated with different categories of ML usually work best on different types of problems. Which type of learning to use in a certain situation depends on the available information and the nature of the task at hand.
For the kind of learning that has to be performed by the agents developed during this work, reinforcement learning is the best and really the only reasonable choice. Some reasons for that are, among others, that there is no easy access to labeled data in order to allow the sensible application of supervised learning, that there are no big amounts of previously prepared input data available, and that it is useful to get constant direct feedback for taken actions in order to improve.

2.1.2 Artificial Neural Networks

The concept behind artificial neural networks (ANNs) is based on observations made in the areas of biology and medicine that reach back as far as the 19^{th} century. Since then researchers in the emerged field of neuroscience[1] have learned that the human brain consists of roughly 100 billion neurons (cf. [7]). While the exact way of functioning of the brain is still not completely explored, neurons and the communication between them are understood relatively well and the findings are used in computer science to create artificial networks of virtual neurons. ANNs as well as biological neural networks (BNNs) are structures built from their basic components, the neurons.
Figure 2.1 illustrates the artificial version of such a neuron. While there are some differences in the way biological and artificial neurons fulfill their tasks, their purpose is similar. Moreover there are quite a few properties of BNNs that come in handy during software development. Kriesel [8] identifies three main characteristics of BNNs that are of interest for developing an ANN:

1. *Learning and self-organization abilities.* It is not necessary to give detailed instructions to a neural network, rather it is able to learn from examples or from external feedback. An ANN can be trained using one of the learning methods known from ML – namely supervised, reinforcement or even unsupervised learning.

[1] Neuroscientists study the brain and the nervous system in general.

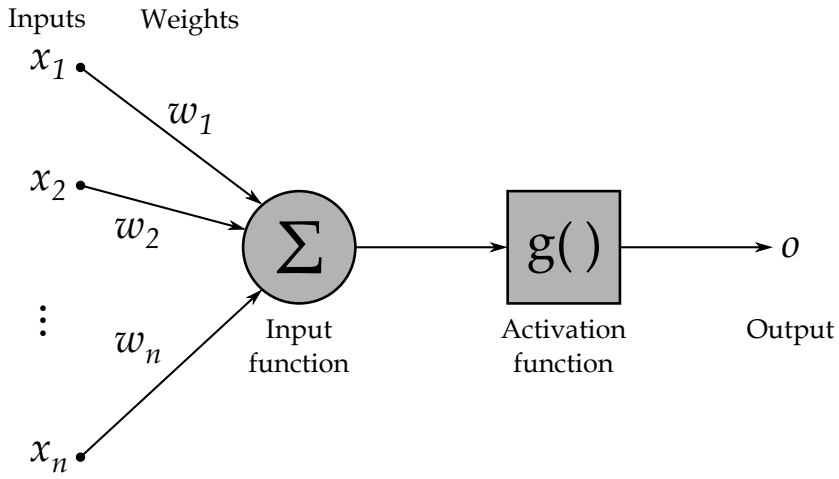

Figure 2.1: Artificial neuron

2. *Generalization capabilities.* Once trained, a neural network is capable not only of solving problems it is used to, but often also of solving tasks that are somewhat similar. Like a biological brain uses learned information for dealing with new situations so does an ANN, to some extent.
3. *Fault tolerance.* BNNs are quite resistant to both internal and external errors. If a part of a human brain is damaged by a trauma induced by heavy drinking or even a seizure, it is often able to react to it and keep its functionality mostly intact. The brain is also capable of dealing with erroneous inputs such as missing letters in a written sentence. Especially the resistance to external errors is something that is a nice property to have in an ANN, if dealing with noisy input data.

Mechanics and structure

All a single neuron as part of an ANN does, is: receive some input signals, produce a transformed signal, and communicate this new signal as output. This output again can be input for other neurons, since ANNs are organized as a sequence of layers[2] (cf. Figure 2.2). The first layer of this sequence is called input layer and the last one is the output layer. All layers in between are named hidden layers. Whereas the number of input and output neurons is usually fixed, the number of hidden neurons and also the number of hidden layers can be changed to increase or decrease the complexity and computational capacity of the network. Of course an increase of these numbers comes at the prize of higher computational costs and potentially – depending on the application – even worse results. The reason for naming the mid-ranged layers "hidden" is that only the first and the last layer interact with the environment of the net and that its inner workings can be regarded as hidden or as a black box.

[2]In a BNN it gets more complicated already since the neurons are often organized in various very complex structures.

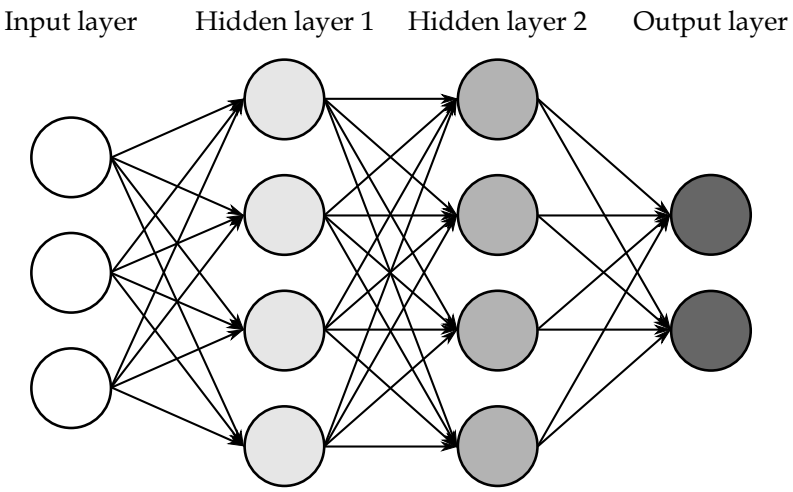

Figure 2.2: Topology of an artificial neural network

In an ANN, information is provided to the input layer and is propagated to and altered within the following layers until it is finally available as transformed output data (again Figure 2.2). How the information is altered depends on the parameters of the net and the implementation of the neurons. Figure 2.1 shows a characteristic blueprint for an artificial neuron. First the inputs $x_1 - x_n$ and the associated weights $w_1 - w_n$ are fed into a transfer function. This transfer function defines how this information shall be combined. Usually this is done by simply summing up the weighted inputs (as indicated by the Greek letter Σ). The calculated net input is further processed via the activation function $g()$ and results in the output o. The activation function determines not only how the output value is calculated but also in which range of numbers it will be and if it is a discrete or continuous output.
Two popular activation functions are plotted in Figures 2.3 and 2.4. Figure 2.3 shows the discrete Heaviside step function, where o is either 0 or 1, depending on the input being smaller or greater than the threshold of 0. Figure 2.4 depicts the sigmoidal logistic function, where the output is a floating-point value smaller than 0.5 if the input is a negative number and greater if the input is larger than 0.

ANN application

The three characteristics mentioned by Kriesel and other properties make ANNs interesting for many different applications. They are especially well suited to handle situations where there are big amounts of data available but it is very difficult to develop programs that can act on it. Referencing to the image processing example from Section 2.1.1, it would be almost impossible to explain all existing features of a horse to a static algorithm in order to enable it to recognize the animal in any given picture. Using labeled examples, an ANN could learn similarities of images of horses and use this knowledge on new data in the future.

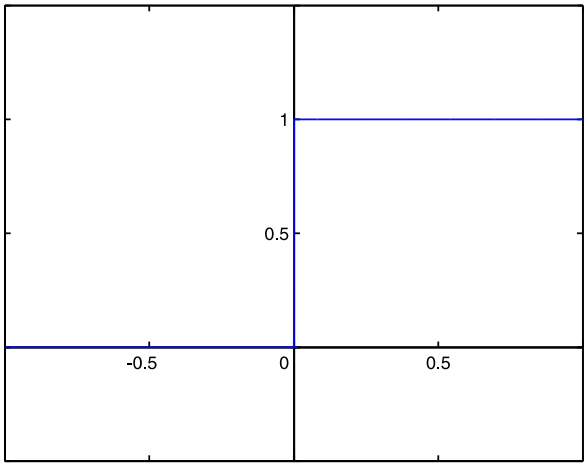

Figure 2.3: Heaviside step function (taken from [9])

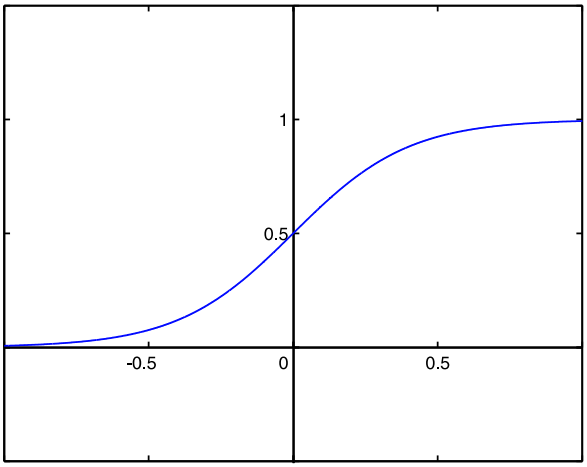

Figure 2.4: Logistic function (taken from [9])

In respect to this work, the ability of ANNs to deal with new situations and to learn by experience is of great interest. While it is not completely obvious initially how exactly an ANN has to be designed for the task at hand, it is safe to say, even at this point, that a continuous activation function is better suited than a step function since the agents are dealing with floating-point numbers only, as inputs but also as outputs.

2.1.3 Evolutionary Algorithms

Evolutionary algorithms (EAs), as metaheuristics and subset of evolutionary computation, deal with various problems in the domain of optimization. Their development was inspired by a principle that often leads to the development of better and better solutions for specific problems that can be found in nature, biological evolution. The idea of evolving a population of initially sub-optimal solutions towards optimal ones stands at the center of the concept of EAs. Reproduction, selection, recombination, and mutation of individuals within a population are means to that end. As for biological evolution, it can take several or even hundreds and thousands of generations until desired properties in an individual are evolved and the result is therefore satisfactory. The basic measurement for the quality of the results produced is defined by a fitness function that has to be designed appropriately for the problem at hand.

Types, components, and mechanics

Without going into much detail on the single types of EAs, it shall be mentioned that there are several different approaches to the topic. According to Eiben and Smith [10], the most widely known variants of EAs are: genetic algorithms, evolution strategies, evolutionary programming, genetic programming, learning classifier systems, differential evolution, estimation of distribution algorithms, and also particle swarm optimization.

While the concrete implementation usually differs for different EA variants, typically there are a few aspects that have to be considered:

- *Representation of individuals and solutions.* The first aspect is the problem of transferring the task at hand to the somewhat abstract environment of an EA and to enable a machine to process it.
  In analogy to biological evolution, individual solutions to a problem are named phenotypes and the blueprints that define how these solutions can be created are called genotypes. Illustrating this concept using a simplified example from nature, a genotype can, encoded in individual genes, describe certain traits like brown hair and blue eyes. A phenotype created from this genotype would therefore be a brown-haired and blue-eyed person. It is important to notice that the whole process of searching for an optimal solution is happening in the domain of genotypes and that it is therefore also important that all possible solutions to the initial problem (phenotypes) can be represented as such.
- *Population.* Genotypes are not only blueprints that can be used to produce individual solutions – the phenotypes – but are individuals themselves. They are individuals in a so-called population, which is a multiset[3] of the representations of possible solutions. A population has a certain, usually fixed, size and its individuals compete against each other for a spot in the next generation of it. The limited size of the population is what puts evolutionary pressure on the candidates and it plays a vital role in creating better solutions.
- *Fitness evaluation.* To determine which individual of a population is of highest quality it is necessary to find a suitable evaluation criterion. The fitness value of a genotype is a number that indicates how well the phenotype that can be produced by it fulfills the task given to the EA. It is therefore the basis for the upcoming selection process that determines the members of a future generation of the current population.
- *Parent selection.* The prerequisite for creating a new generation of individuals is parent selection. Which candidates of the current generation become parents for the next one is generally decided by ranking them according to their fitness. An important aspect of this selection is that not only the best individuals become parents but that also weaker ones have a small chance to reproduce. This helps in avoiding premature convergence of a population and therefore prevents getting stuck in local optima.
- *Recombination.* Recombination denotes the process of exchanging parts of the chromosome (the sequence of genes) of two or more parents two create at least one child. It is one of the basic insights gathered from natural evolution that sexual trumps asexual reproduction in most cases.
  How exactly the parents are combined to create new individuals depends on the concrete implementation and also on chance. Usually the chromosome is severed in one or several places and the fragments are exchanged. The points of separation

[3] A multiset can contain multiple instances of its elements.

Algorithm 2.1: Basic scheme for EAs (Pseudocode)

```
Create population of genotypes using random candidate solutions (phenotypes);
Evaluate individuals of the population;
while not terminated do
    Select fitting parents;
    Recombine them to create offspring;
    Mutate offspring;
    Evaluate new individuals;
    Select candidates for next generation;
end
```

are often determined randomly. That leads to many different combinations of traits and is expected to produce offspring that sometimes has lower, sometimes equal, and sometimes higher fitness than its predecessors.
In contrast to biological evolution, it is feasible in artificial evolution to allow selected parents to skip their turn and move over to the next generation without any alterations at all.

- *Mutation.* Mutation is a stochastic operation performed on single genotypes. Each gene is chosen for mutation with a certain probability and if selected its value is changed by a small randomly determined amount. This allows the introduction of new traits to the population. Ones that cannot be developed by mere recombination of properties.
- *Replacement strategy.* The last aspect that has to be examined here is how to select the individuals that will form a new generation. Since the size of the population is usually fixed, it is necessary to remove some individuals from it in order to make some space for newly created ones.
  As mentioned before, it is feasible to see not only the offspring created during the recombination phase but also the parents as candidates for the next generation. One way of determining the new population is by simply using fitness selection on the multiset of combined parents and children. Another possibility is an age-based selection but also a combination of both or even random selection are possible.

Algorithm 2.1 describes schematically (in pseudocode form) how the individual components discussed here come together and form an EA.

EA application

EAs, as heuristic optimization techniques, are valuable in situations where the exact solution to a problem is very hard to find[4]. They are able to provide solutions of sufficient quality in many cases and proved themselves useful in areas as diverse as engineering, where they have been used successfully to design sophisticated antennas, or economics,

[4]Many of these tasks fall into the category of NP-hard problems.

where they find usage in stock market analysis, and many others. It is relatively easy to adopt EAs to new kinds of problems as long as the quality of a solution can be described by a fitness value.
The flexibility of EAs and their properties as optimization algorithms, that allow the development of unexpected and yet practicable solutions, qualify them for a sensible utilization during this work. The specific type of EA that will be used in the following is the genetic algorithm (GA).

2.1.4 Animation

The beginnings of animation reach back to the era of black and white silent films that have been produced in the first half of the 20$^{\text{th}}$ century or, depending on what types of animation are considered, even further (cf. [11]). The artificial animation of initially inanimate objects and characters became very popular over the following decades and led to the development of sophisticated techniques that found its way into many of today's works in the film and the computer games industry also.
Meanwhile a wealth of different animation techniques has been developed and while some of them are used more frequently than others, most of them have interesting application scenarios. Following the research focus of this work, they are divided into two broad categories here, the second being of more relevance:

1. **Analog animation**[5]**.** Since computers have not been around when the first animators started out their work, the first known animations, but also newer ones, fall into this category and many of them are still used today. Some interesting types are:
   - *Mechanical animation.* Mechanical devices are used to either achieve a sense of motion or create some actual one. Flip books and puppetry belong to this subcategory but also animated robots that can be found in theme and amusement parks.
   - *Cel animation.* The single frames are drawn by hand and are subsequently concatenated to give the illusion of movement. Cel animation became widely known through the many animated cartoon films produced in the last century.
   - *Stop motion.* Real-world objects are moved, transformed, and photographed frame by frame to create a continuous animation. The objects can be modeled using materials like clay, paper, toy blocks, and others.

2. **Computer animation.** Digital animations play a big role in several areas of content production today. Hardly any blockbuster action movies are produced without computer generated effects and naturally even less video games. Important techniques for generating such computer animations are:

[5]The term "analog" refers to the circumstance that big parts of the animation process are done, or have been done traditionally, unassisted by computers. Computers can still play a part in developing these animations but are no prerequisite.

- *Sprite animation.* A sprite is a single picture, often containing a single character or object, that can be incorporated into a bigger image to appear as part of it. This avoids the need for the computer to redraw the whole scene if only a little part of it changed. Similar to Cel animation, in sprite animation several slightly different pictures are shown in sequence to create a motion effect. It is common to wrap all the pictures needed for an animation into a single one and feed this so called sprite sheet to the computer. Sprite sheets are mainly used for 2D animations.

- *Key-frame animation.* Key-frames define the starting and end points of a continuous motion. They describe how a subject or an object should look like at a certain point in time. All the frames that have to be created between two key-frames to create a smooth movement have to be interpolated. While this process can be conducted manually, in computer animation it is performed automatically by the machine. As in sprite animation, it is also possible to morph the target into different shapes and to change its color, next to simply moving it around.
  The technique of key-framing is equally suited for 2D and 3D animation but while the interpolation between mere images may work smoothly in 2D, it is a little more complicated to achieve convincing motion in 3D.
  This is where *rigging* comes into play. Rigging is the process of fitting a virtual skeleton with joints and bones to a character or even an object mesh. This so-called rig allows animators to easily manipulate the 3D mesh surrounding the skeleton by moving the single parts around and thereby bending the target into the desired position for the next key-frame.

- *Motion capture.* In motion capturing it is not the creation of artificial movement that stands in the focus but rather the recording of real one and the transformation into digital imagery. The motions of characters or objects are tracked, either with or without the help of special markers, and can be used to achieve realistic behavior in computer generated content. Because of its connection to actual movement that can be performed in the real world, motion capturing is not as flexible as other computer animation techniques. However, the quality of the outcome is usually very high and it is possible to combine this technique with others to mitigate the restrictions.

- *Physics-based animation.* With the advent of more and more powerful computers, the simulation of moving entities based on the laws of physics becomes increasingly interesting.
  Since the utilization of this type of animation plays a key role in this work it is described separately and in greater detail in Section 2.3.

2.1.5 Procedural Content Generation

The term of procedural content generation (PCG) refers to the automatic generation of a predefined type of content for a certain purpose. In the context of gaming, Togelius et al. define: *"PCG as the algorithmical creation of game content with limited or indirect user input"* [12]. PCG is interesting for creating big amounts of contents in a relatively short amount of time and is therefore of great interest in areas where lots and lots of pieces of content are needed. Nevertheless, not only the sheer amount of data that can be created is of interest but also the capabilities of PCG techniques to create many different versions of a single object and to modify already existing ones, as well as the focus on automation. Particularly this last point, the automation aspect, is important here.

Methods used in PCG

Figure 2.5 shows a taxonomy of established techniques that are used for automated content generation, adopted from work of Hendrikx et al.[13].

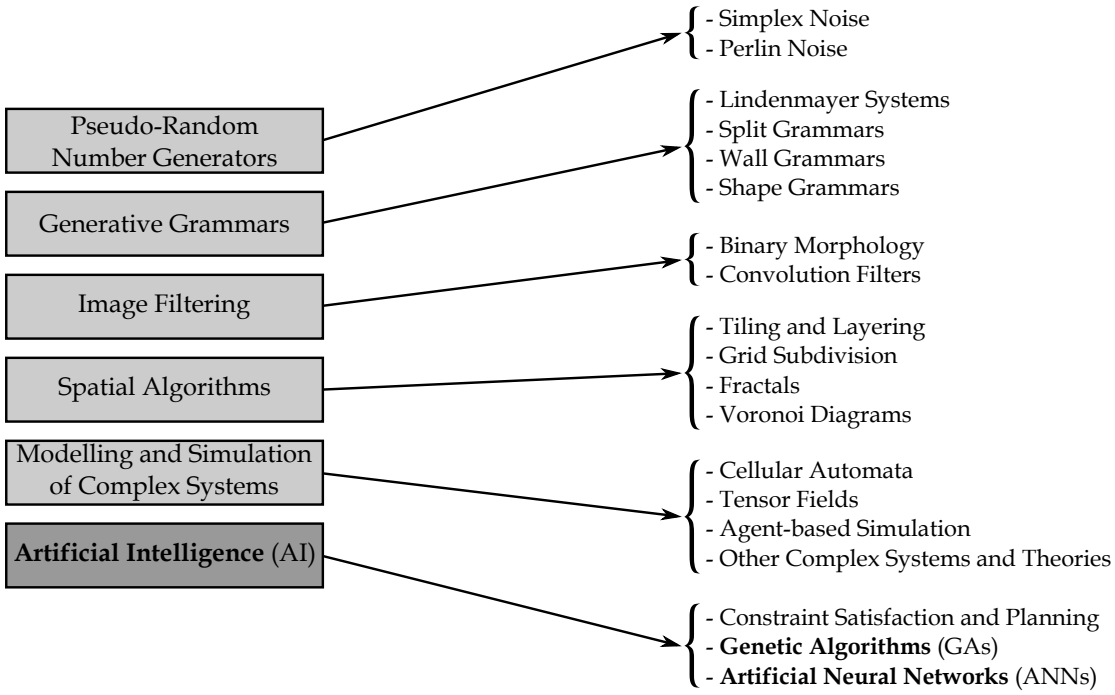

Figure 2.5: Taxonomy of procedural content generation techniques

It is out of the scope of this work to discuss all the different methods in detail but the last category in Figure 2.5 (AI) contains two already introduced concepts of relevance: GAs, belonging to the family of EAs, and ANNs. Although AI methods might not be the ones that are used most often in PCG, they have some interesting areas of application and can be used successfully for several tasks. They can be utilized on their own for creating new content, they can help to increase the quality of already existing content,

or they can be combined with other types of PCG techniques to create interesting new results (cf. the work of Ochoa [14], combining GAs and Lindenmayer systems).

Potential of PCG for animation

Though techniques like ANNs and GAs can be used for directly creating content, like textures for characters or surroundings, the possibility for improving on already existing content is of greater interest for this thesis. Especially the automatic generation of character animations.
The animation of characters in movies or games is a very time intensive procedure. The more prominent a certain character is set into scene and the more time it is visible on a screen, the higher the quality of its animations have to be. While fully automated generation of motion for animation movies might not be realized sensibly in the near future it can be valuable for many other purposes in the area of digital productions.
Imagine a strategy gaming environment with dozens or hundreds of different units that have to be animated. It is still possible to handle all these units by hand, nonetheless a solution helping in procedurally generating all the animations can be a big help. But lets go one step further. In a game that allows its players to freely create their own characters it is virtually impossible to create perfectly fitting animations for all the different kinds of creations before shipping the game. Another use case scenario is the animation of procedurally generated characters, or other objects, that are created automatically and have to be actuated accordingly.

2.2 Neuroevolution

Neuroevolution (NE) can be considered a special variant of reinforcement learning that displays several advantageous qualities and introduces some new interesting ideas to the field of ML. Next to using the fundamental principle of reinforcement learning, it draws on the concepts of EAs[6] and ANNs.
The basic idea behind the technique is to evolve a population of genetic encodings, that represent different building plans for individual neural networks, towards a solution that can solve a given task optimally or at least satisfyingly. Following the terminology used for GAs, the encodings of the building plans are the genotypes and the corresponding ANNs the phenotypes. Figure 2.6 illustrates this idea and shows how the evolution takes place. The procedure is the following:

1. An initial population of individuals is created and encoded as genotypes.
2. Each genotype is decoded separately and used to generate an ANN, a phenotype.
3. In an evaluation phase the different ANNs interact with the environment and try to solve the given problem. At the end, each of them gets a fitness value assigned, according to how well it performed on its task[7].

[6] As mentioned in Section 2.1.3, the specific type of EA used in this work is the GA.
[7] It is important to notice that no learning takes place at this point.

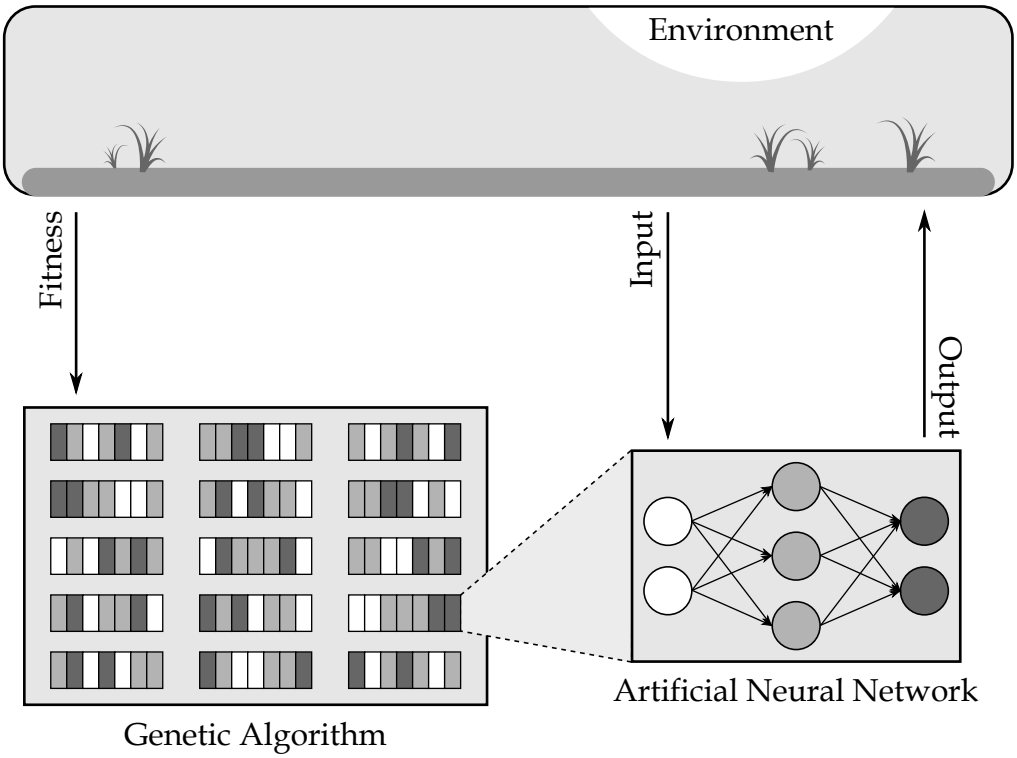

Figure 2.6: Schematics of neuroevolution

4. A new generation of genotypes is created/encoded. How exactly the new population will look like and which individuals will still be in it, or have an influence on its generation, is determined by the fitness values and the concrete implementation details of the GA that is used.

After creating a new generation the process jumps back to Step 2 and continues with the decoding of genotypes. The process of NE usually stops if the fitness level is high enough and the problem has been solved successfully or if a certain amount of time/computing cycles has passed.

2.2.1 Types of Neuroevolution

Though partially overlapping, there are several different forms of NE, developed over the last decades (cf. [15]), that can be distinguished. One way to do that is by looking at the way the ANNs are encoded into genotypes for further usage by the EA, but there can be other distinctive features as well.
Some important concepts, ideas, and approaches in the field of NE, all of them being well suited to certain kinds of tasks, are:

- *Conventional NE.* In conventional NE a genotype is represented as string of concatenated ANN weights, that are assigned to the edges between the nodes of the nets. These weights are typically binary or floating point values, depending on the use case. As a consequence only the weights of an ANN get changed during the recombination and mutation phases of the corresponding EA and the underlying structure of the net stays fixed.
  This approach works well and leads to good results in a relatively short amount of time in many cases, especially if the task at hand is not too complicated and a fitting network structure can be developed by the designer. However it has some drawbacks as well. First, in more complex settings it can be challenging to optimize all parameters at the same time and a divide and conquer strategy might lead to better results. Another issue is the circumstance that similar ANNs with almost the same functionality but different arrangement of weights can be evolved in parallel (*competing conventions*), which creates unnecessary computational overhead. Finally the conventional NE, without alterations, is susceptible to getting stuck in local optima.
  Since the disadvantages of this technique can be circumvented relatively well for the software solution that is developed during this work and the advantages of its usability and good performance, if a suiting network structure is predetermined, outweigh these drawbacks, it is of great interest here.

- *Topology and weight evolving ANNs.* As the label topology and weight evolving ANNs (TWEANNs) suggests, this technique does not only evolve the weights of a neural network but also its structure. The idea behind TWEANNs is that not only the weights but also the topology of a net itself has a big impact on its performance. Although it should be possible for a sufficiently large and fully connected ANN to solve the same problems, it is often more efficient to choose different network structures for specific tasks.
  While TWEANNs can be used successfully to tackle more complicated problems by developing well fitted solutions, there are also a few issues that have to be addressed during their implementation. In contrast to the comparatively straightforward alteration of weights that has to be performed in conventional NE, it is a lot more complicated to find a good way of evolving the network topology. Two ways of starting this process are to either randomly initialize the structure of the ANNs or to start from a very simple structure and increase the complexity subsequently. After starting the evolution it has to be ensured that branches of individuals within the population showing new and promising innovations in their network topologies have sufficient time to improve in order to live up to their potential. This protection for newly evolved structures is called *niching* or *speciation* (the evolution of species). Other topics that have to be addressed are, as for conventional NE, the issue of how to deal with competing conventions and how to properly encode the network in order to conveniently store the information of both the network's weights and its topology.

One of the most popular implementations of NE uses the TWEANN approach and is called NEAT, which is an acronym for "NeuroEvolution of Augmenting Topologies" (cf. [16]). It meets the points mentioned above in the following way:

- Topology evolution is started with a population of ANNs that make use of a network structure without any hidden neuron layers. New hidden nodes within newly created layers and also new connections between the nodes are added gradually via mutation and only successful creations are propagated through the different generations.
- Speciation is achieved by the introduction of historical markings that track different genes in order to make it possible to tell which individuals share those genes and belong to a particular species. By identifying different species, whose members share similar topologies, it becomes feasible to protect the initially weaker ones and to give them a chance of evolving.
- Competing conventions are avoided by using the tool of historical markings once more. Since each gene is tracked by a marking over its entire lifespan it is easy to tell which genes of two individuals that have been selected as parents for recombination are present in both of them and which are not. During the recombination phase overlapping genes are selected randomly from the parents and the genes that are only present in one of the parents are chosen from the fitter one. In that way accidentally choosing the same genes from both parents can be prevented.
- Encoding of network topology as well as network weights is achieved by using two different types of genes that are arranged sequentially to form a genotype. Node genes hold information about the nodes (mainly their positioning in the input, output, or a certain hidden layer) that belong to the corresponding phenotype, the ANN. Connection genes tell how the nodes are interconnected, which weights are assigned to these connections, and they also store lots of other info.

Because NEAT has been used successfully in many different situations since its initial presentation and it is still widely applied in the field or NE and because it is potentially well suited to the task at hand, it is also considered during the phase of designing a solution in Chapter 3.

- *Solution components.* As mentioned before, it can be sensible to apply a divide and conquer strategy to some of the problems that can be handled by NE. Instead of evolving networks on a very complex task it is sometimes possible to divide the desired solutions into individual components and to develop smaller ANNs, or parts of ANNs, for these components separately. These partial solutions can be combined to a full solution afterwards, according to some predetermined layout or even an evolved structure.
  Solution components constitute a higher level concept that provides an instrument for dealing with more sophisticated problems. The specifics of how to implement the evolution of the partial solutions still have to be addressed from case to case.

- *NE ensembles.* The idea behind NE ensembles is similar to the concept of ensemble learning known from standard ML. Instead of using multiple learning algorithms, different ANNs are applied to a certain problem. Rather than taking only the best individual from a population, several, preferably fairly diverse, nets are selected from it.
  Because the advantages gained by the diversity of the ANNs that are used comes at the cost of increased computation time, it is reasonable to use this technique mainly where there is some variance in how to find good solutions as well. For tasks that can be solved in almost the same way each time they are started it might be better to simply use a single ANN that proved capable before.
- *Indirect encoding.* Most NE methods utilized nowadays (including conventional NE and NEAT) use a way of representing the ANNs within the genotypes that is called *direct encoding*. In direct encoding each part of an ANN is represented explicitly as data in the genotype. To illustrate this further, for conventional NE the single values can be the weights of a network and for a TWEANN they would represent both weights and information about the network topology. In contrast to this explicit storage of information about the neural nets, indirect encoding only provides a kind of building plan for how to construct the ANN. Various methods like cellular encoding, an approach that is inspired by cell splitting in organic lifeforms, and also certain PCG techniques, like the generative grammars listed in Figure 2.5, can be used to devise such a plan (cf. [17]).
  Some advantages of indirect encoding are its compact representation, which maps a bigger phenotype to a smaller genotype and therefore generates a smaller search space, that the search space ideally corresponds well to the problem space, and that it is possible to make use of recurring structures. However, it is quite challenging to decide on how exactly the encoding should look like and further on how to optimize the solutions produced during the evolution.

2.2.2 Performance and Applications of Neuroevolution

Since NE can be seen as alternative to classic reinforcement learning methods, it is of interest that it is quite competitive in continuous and only partially observable domains that are well suited for the latter. A popular example showing the good performance of NE is the benchmark task of balancing an inverted pendulum without velocity data provided. It has been shown that NE is both faster and more capable of solving harder versions of the problem [18]. This example is not only a mere artificial benchmarking task, it rather points to one of the main areas where NE performs well in many cases: the control of physical devices like robots or rockets (e.g. [19], [20]). Because it is impracticable to develop the controller for such a physical device in the real world, it is necessary to use simulations during the training phase of the NE algorithms. A high quality and accuracy of these simulations is therefore key to ensuring the results are usable in practice. Though it can be a useful substitute for conventional reinforcement learning, it brings good results in other domains as well. Some examples for a successful

utilization and competitive implementation of NE in a supervised learning setting are given by Schmidhuber et al. [21].
As NE is not limited to reinforcement learning problems but can be used for supervised and furthermore unsupervised learning tasks too, it is highly flexible and therefore widely applicable. Hence there are lots of other potential areas of application besides the ones mentioned above. The two most interesting for this work naturally being physics-based animation, considered in Section 2.3, and computer games.
Risi and Togelius [22] point out a number of examples where the technique has successfully been used for various tasks in digital games. These tasks range from controlling non player characters all the way to designing parts of the gaming environment using PCG. Table 2.1 shows the roles NE takes in some selected games and lists the types of ANNs and NE methods that have been chosen to fulfill them[8]. Aside from applications in PCG, where compositional pattern producing networks (CPPNs) [23] are especially attractive, the predominant type of neural network in use is the multi-layer perceptron (MLP) already presented in Figure 2.2. Next to popular EAs, like the multi-objective EA NSGA-II [24], GAs, or evolution strategies, and fixed user-defined topologies for the networks, specialized NE algorithms such as NEAT, respectively its extension HyperNEAT, are also used in this field.
Something that maybe does not attract attention at a first glance is the fact that although the underlying survey has been published not too long ago, the games listed in Table 2.1 are either quite old (some released before the turn of the millennium) or of relatively small production value and often both. Additionally the possibility of using NE for animation purposes is not explored by the games presented here. These observations underline the impression formulated in the introduction that although it has been shown that NE can be of great value to game development it is still hardly used in commercial game productions and that taking a look at the combination of NE and physics-based animation might turn out fruitful.

[8] Table 2.1 intends to give a first impression of how NE can be utilized in games. For a more detailed description of the techniques and methods used and for references to the mentioned games consult [22].

[9] MLP = multi-layer perceptron, CPPN = compositional pattern producing network, LSTM long-short term memory

[10] UD = user defined, ES = evolution strategies, NSGA = non-dominated sorting genetic algorithm

NE Role	Game	ANN Type[9]	NE Methods[10]
State/action evaluation	Checkers	MLP	UD, GA
	Chess	MLP	UD, GA
	Othello	MLP	Marker-based
	Go (7x7)	CPPN (MLP)	HyperNEAT
	Ms. Pac-Man	MLP	UD, ES
	Simulated Car Racing 1	MLP	UD, ES
Direct action selection	Quake II	MLP	UD, GA
	Unreal Tournament	Recurrent, LSTM	UD, GA, NSGA-II
	Go (7x7)	MLP	NEAT
	Simulated Car Racing 2	MLP	UD, ES
	Keepaway Soccer	MLP	NEAT
	Battle Domain	MLP	NEAT, NSGA-II
	NERO	MLP	NEAT
	Ms. Pac-Man	Modular MLP	NEAT, NSGA-II
	Simulated Car Racing 3	MLP	UD, GA
	Atari	CPPN (MLP)	HyperNEAT
	Creatures	Modular MLP	GA
Selection between strategies	Keepaway Soccer	MLP	NEAT
	EvoCommander	MLP	NEAT
Modeling opponent strategy	Texas Hold'em Poker	MLP	NEAT
Content generation	Galactic Arms Race	CPPN (MLP)	NEAT
	Petalz	CPPN (MLP)	NEAT
Modeling player experience	Super Mario Bros.	MLP, Perceptron	UD, GA

Table 2.1: The role of neuroevolution in selected games (adopted from [22])

2.3 Physics-Based Animation

Physics-based animation displays some fundamental differences to the various other approaches presented in Section 2.1.4. The most important being that it does not depend on previously prepared motion data but moves objects and characters based on the laws of physics and according to given physical capabilities and constraints. Instead of telling an entity exactly how to move, it is subject to different forces and therefore a certain motion is executed as a result. Often such a physics body is affected by two or more forces, like the passive force of gravity and some active forces created by its artificial muscles, at once and therefore it can be difficult to achieve the intended motion under all circumstances.

At this point two important distinctions have to be made. First, there are different challenges for active and passive physics-based animation. While passive animation is

already widely used for achieving naturally looking effects on bodies of water, rocks, cloth, and many other objects, the active animation of physical entities is still not very widespread and the kinematics-based approaches, already presented earlier, are most prevalent. Second, it is necessary to distinguish between animating arbitrary objects and doing character animation. While all types of physics-based animation might be worth exploring, only the active animation of characters is of real interest for this work and will be discussed from here on under the term of physics-based character animation (PBCA).

2.3.1 Mechanics and Types of Physics-Based Character Animation

A character in PBCA usually consists of several components. The most obvious one is the physical body itself, including its torso, legs, wings, fins, or other parts. Sensors and actuators act as input and output devices for the character, where the sensors perceive information about the surroundings, but also internal body signals, and the actuators are the muscles or motors that allow it to move actively. The processing of all input and output signals is performed by a so-called *motion controller*, the brain of a character. Motion controllers are the most interesting part from the perspective of PBCA, since they decide how the body should be moved by its actuators in respect to the inputs received from the sensors. It is important to notice that the actions performed by a character influence future inputs by changing the environment or its internal state. In contrast to *open-loop control systems*, a system where the outputs impact the coming inputs is referred to as *closed-loop control system*. How this closed-loop control works for PBCA is illustrated in Figure 2.7.

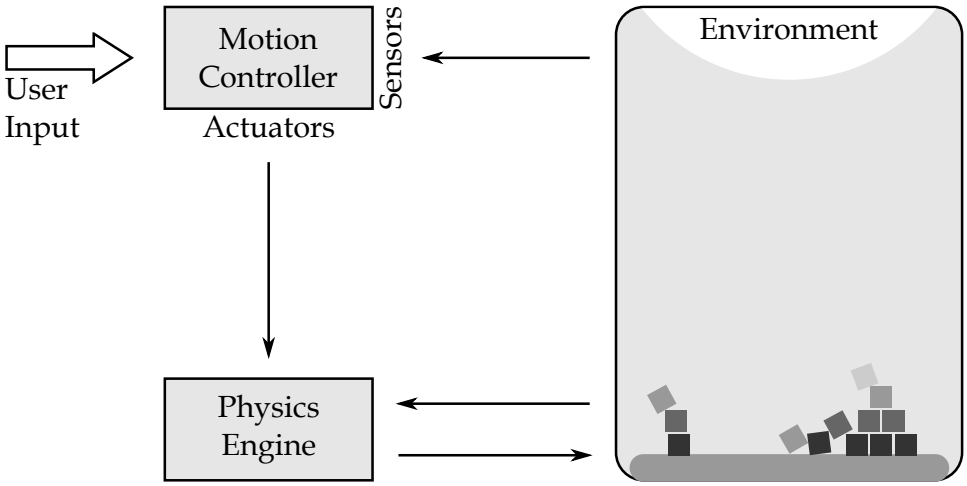

Figure 2.7: Closed-loop motion control in a world with simulated physics

Figure 2.7 indicates that the inputs given to a motion controller by a potential user are rather high-level commands, that tell the character what to do but not how to do it. Such commands would be: walk left, jump right, duck, follow something, and many others. How exactly these instructions are followed depends on the type of the PBCA

used, more precisely the type of motion controller. Geijtenbeek and Pronost [2] define three different PBCA control approaches, based on the research area they originate from:

1. **Joint-space motion control.** In joint-space motion control the characters are provided with target trajectories that describe the movement that should be performed. By getting feedback from the environment, via the sensors attached to the body, the associated movement controller is able to detect if the character leaves the predestined path and act accordingly by activating the right actuators to take steps against it. Similar to key-frame animation it is also possible to define certain key poses that a character should adopt within certain time intervals. However, as it is to be expected from PBCA due to the many factors that influence movements, it is not guaranteed that the desired pose is successfully adopted at the right time or that the given trajectory is followed perfectly.
   Although joint-space motion control is relatively easy to handle and implement and can bring good results, relying on predefined target trajectories or poses can be a big drawback if the goal is to automatically build flexible motion controllers that do not only work in predefined settings but can be easily adopted to new characters and environments.

2. **Constrained dynamics optimization control.** The idea behind constrained dynamics optimization control is to find optimal values for the actuators attached to a character's body, using constraints on the character and its environment and objectives that represent the intended behavior which, in turn, is defined by high-level goals. As the name of this method suggests, the mathematical technique of *constraint optimization* is key to finding these optimal values.
   Next to being less intuitively applicable by animators, the circumstance that the actuator values are optimized constantly during execution makes this technique computationally expensive. Nonetheless if these drawbacks are of no relevance, or are mitigated using suitable approaches, constrained dynamics optimization control can produce high quality animations.

3. **Stimulus-response network control.** The technique of stimulus-response network control takes inspiration from motion control as it can be found in nature. Sensors and actuators are connected by a network that somehow transforms the inputs received by the sensors to sensible outputs for the actuators. The quality of this transformation can be measured according to how well a given high level task is fulfilled. Finding the right setup for the stimulus-response network is the optimization problem that has to be solved here. In the light of the topics that have been discussed in the previous sections, a specific technique seems especially well suited for developing this type of motion control: NE. Using NE, the role of the stimulus-response network is taken by an ANN, how well the task of executing a certain movement is performed serves as fitness criterion, and the optimization of the network is accomplished by using an EA.

Concerning stimulus-response networks Geijtenbeek and Pronost concede that: *"The main appeal of this approach is that it allows motion controllers to be constructed automatically using only an optimization metric [...]"* ([2, p.11f.]), but they are skeptical when it comes to developing complicated high quality animations using this technique and state that: *"The most distinctive quality of stimulus–response network control may very well be its capability to produce unexpected and entertaining animations [...]"* ([2, p.13]). They also mention the lack of examples where this type of motion control has been successfully used for full-body humanoid bipedal locomotion.
Apart from the fact that future research in this area might bring results that can be used for such cases, this observation raises a more general question that has to be answered before deciding which type of PBCA should be applied to an animation task: What type of character should be animated and which additional factors have to be taken into account?

2.3.2 Two vs. Three Dimensions and Other Factors

The expectations on the outcome of PBCA and its area of application influence the way it should be implemented greatly. Shall it be used in the rather static production environment of a digital short film or for flexible, maybe even real-time, animations in a digital game? Is it to be performed in a two- or three-dimensional world? What about the degree of physical accuracy and how complex is the model of the character that has to be animated?
The ultimate desire for PBCA in the context of this work is to develop flexible high quality animations for characters that can be used sensibly in computer games. In this light, and to increase the chances of meeting this desire, the remaining questions raised above can be answered as follows: The animation will be performed in a two-dimensional environment, using a physics engine that favors performance over physical accuracy and rather simplified character models.
The reasons for that are easily explained. Although the high accuracy of some physics simulation frameworks might be important for scientific use cases, like calculating accurate models for avalanches and mudslides, it is not to be confused with being necessary for displaying good animations. Modern physics engines used for computer games and other digital productions are capable of presenting convincing animations, that are based on simplified physical models to increase performance. The decisions of choosing not too complex character models and the restriction to two dimensions go hand in hand. The higher the complexity of a character – the more body parts, sensors, and actuators, it consists of – the more parameter have to be optimized. The more parameters have to be optimized the more difficult the task of doing so gets and the longer it usually takes. Restraining the dimensionality of the environment reduces the number of parameters quite naturally and therefore boosts the chances of developing fitting and flexible motion controllers in a relatively short amount of time.

2.3.3 Utilization of Physics-Based Character Animation

Like NE, PBCA is not used very widely in the sector of commercial computer game development, or commercial digital applications in general, yet. However, as the amounts of content for productions in this area are ever on the rise, the desire for finding new ways of animating it realistically, and preferably even automatically, is so too.
Addressing this interest, several promising different approaches to PBCA have been, or are still being, researched up to this day. Out of these, a couple of works deal with animating bipedal or humanoid 3D characters (e.g. [25] and [26]), but some other creatures are considerd as well. Hecker et al. [27] present a system for creating animations for 3D fantasy creatures with morphologies unknown at animation time. Coros et al. [28] show different gaits and locomotion skills for four-legged creatures (quadrupeds) and Tan et al. [29] discuss the animation of swimming creatures. Aside from dealing only with the locomotion of characters, some research is concerned with various other topics, as diverse as learning bicycle stunts ([30]) and the parallel optimization of control and morphology ([31]), as well.
Interestingly a lot of research work over the last years has been focused on the development of realistic and accurate movement of 3D characters, often bipedal, and there has been less interest in the promising field of creating animations for different 2D creatures and characters, which might be developed to a level where they are usable in practice (e.g. in the genre of platform games) more easily. This work addresses that issue and, because high flexibility, fast execution, and the automatic generation of animations have been identified as desirable factors for the solution, it does so using a stimulus-response network and NE rather than one of the other types of PBCA.

CHAPTER 3

Solution Design

Several aspects influence the design of the solution that is presented here. The first section in this chapter describes the fundamental concept behind it and the second one provides information on the technological framework to go along with it. Afterwards the importance of repeated experimentation for the development of a solution of high quality, that is in fact usable in practice, comes up for discussion.

3.1 Concept

In order to develop a concept able to deal with the challenges outlined in the introduction, the first step is to determine what exactly is expected from a good software solution. All design decisions elaborated subsequently are targeted at fulfilling these expectations.

3.1.1 Desired Features and Restrictions

As already indicated in Chapter 1, while the combination of NE and PBCA is an approach that can be useful not only for games but also for other digital media productions, like movies and short films, this work is mainly focused on game development. This choice leads to a certain weighting of aspects like performance and flexibility. For a utilization in movies both aspects might be secondary to animation quality but in game production they can be of high importance.
Against this background, the most wanted abilities, properties, and features for the software solution that is to be implemented are:

- *Automated PBCA.* The most important ability is the automated simulation of appealing movements. Although the desire for this functionality might seem obvious, there are a few points and restrictions that have to be considered.
  First, defining what makes a movement appealing is definitely not completely obvious and this is an issue that has to be considered during later evaluations.

Second, while the umbrella term of PBCA comprises all kinds of movements, it is out of the question to implement all of them for testing during this work. Rather than trying to do so, the only movement types that are examined here are running and jumping. Running (resp. locomotion in general) is a natural choice for demonstrating PBCA, since almost any land-based character is able to perform it. Therefore it is also in the focus of research in this field (cf. [2] and [32]). Similarly, the motion of jumping can be performed by many terrestrial animals and it is therefore well suited to serve as second movement type here.
As a result of this choice of motion types, the characters that shall be moved will be placed on solid ground and are required to interact with it. That means that there are no floating, flying, or swimming creatures involved in the experiments and tests to come.
Another restriction is imposed on the shape of the ground. Because the development of quality animations for different characters is of more interest here than the capability to deal with different terrain types the underground remains flat.

- *Flexibility.* The restrictions for automated PBCA mentioned above are mainly relevant for the specific implementation produced during this work. The theoretical solution as such is a lot more flexible. Basically all kinds of different movement types can be implemented in arbitrary environments, given that it is possible to reward the desired behavior of a creature using an appropriate fitness function and that the character in question embodies all the required physical parts.
  Apart from this theoretical flexibility of the solution design, a different kind of flexibility is desired for the realization of it here. It shall be possible to easily swap the type of character that is to be trained, modify its body, and also to add new functionality and creatures.
- *Ease of use.* This property goes hand in hand with flexibility. In the end it shall be possible for graphic designers and artists to simply create new characters in an editor environment and these characters should be able to learn the defined kinds of movement without the necessity for adapting the underlying program code.
  The motions that are learned by the program have to be easily accessible for further usage. That means that they can be implemented separately into any game without the need for incorporating the whole logic used for training purposes.
- *Performance.* In comparison to the production of animations before shipping a product, the requirements for situations where animations have to be produced in a live gaming environment are completely different. Waiting hours during the processing of movements on high end computers might lead to high quality animations but these are completely useless if content is procedurally generated during a gaming session. Since one of the goals of this work is to explore the possibilities of using NE for real-time PBCA usable in games, a high performance of the implementation and short animation times are of interest.
  Even if real-time animation cannot be achieved it makes a difference for the solution's usability in practice if animations can be developed in minutes, hours, or even days.

Summing up, the solution designed here is supposed to enable users (and not only programmers) to create different animations for different types of characters quickly and efficiently. Although the theoretical capabilities of the presented concept shall be wide-ranging it has to be restricted in order to allow for a sensible later implementation that stays within reasonable bounds. Nonetheless, the desired high flexibility of the solution design ensures expandability.

3.1.2 Design Decisions

In order to guarantee that the desired objectives for the solution are met as good as possible, the design choices presented here incorporate the insights gained during researching the current state of the art in this field. Still, there is only little information on the task of using NE for PBCA in two dimensions, and even more specific in the context of digital gaming, to be found in the literature. For that reason the whole design process is an iterative one, accompanied by extensive experimentation and testing phases and partially intertwined with the implementation itself. This is of interest here because the first and most important design choice is based partly on observations made while experimenting with different NE types.
Two approaches to NE that have been identified for further consideration in Section 2.2.1 are conventional NE and the TWEANN-method NEAT. First basic experiments with both, a conventional NE implementation built from scratch and an already existing NEAT variant (cf. Section 3.2), suggested that the self-made program has the potential for providing the desired quality for the targeted types of animation. Although the NEAT variant also showed promising results, the higher level of control over the self-made implementation and the potential overhead of having to spend time on evolving the network topology for NEAT tipped the scales in favor of the conventional NE solution. Additionally it might be advantageous to be able to fit a newly implemented piece of software to the highly flexible program that is to stand at the end of the development process instead of having to integrate an already existing framework.
Now that this fundamental decision has been made it is time to deal with the specifics of the concept. To give a clearer picture of why certain choices are made and how the single parts fit together they are set into the context of the process of developing a character animation as it is envisioned for the final solution.

The process of creating an animation

Seen from the high-level perspective of a designer or artist the process looks rather intuitive, as illustrated by Figure 3.1. He or she will have to perform the following steps:

1. Create a character in an editor environment using simple physics components and/or select it to be used for learning the animation.
2. Select a predefined type of animation to learn.
3. Start the program and wait until a satisfactory level of animation has been reached or continue with the next step.

4. (Optional) Stop the program, adjust some parameters if necessary, and continue at Step 3 if the result has not been satisfactory.
5. Save the animation for later usage.

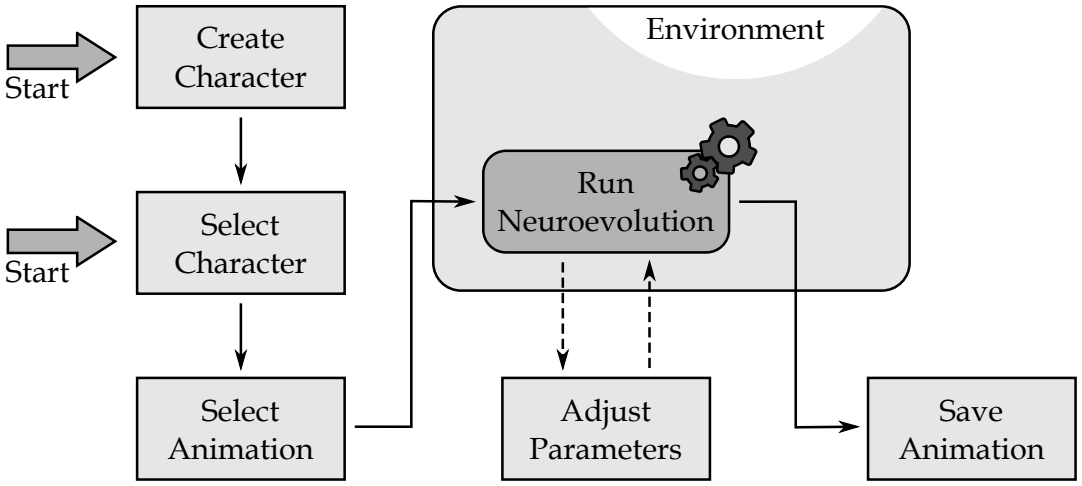

Figure 3.1: High-level perspective on the process of creating an animation

What happens behind the scenes is more interesting of course. How to handle the first step, the character creation and selection, depends mainly on the chosen technology and will be explained in the next section.

However, already the second step is of more relevance at this moment. Selecting a different type of animation essentially corresponds to switching the fitness function used for training the neural networks. Finding the perfect fitness function for a concrete task is not a trivial endeavor. Is it sufficient to reward a character for rapidly moving in a certain direction if the goal is to teach it running? Are there other important factors that ensure that naturally looking movements are developed?
In nature the movements of all animals are restricted by external forces, the strength of their muscles, and many other things. Additionally a trade-off between effective movement and the consumption of energy has to be found[1]. An artificial creature does not have to deal with those issues but the person designing a solution for PBCA definitely has to. At this point the problem is addressed by simply using the distance covered during a certain amount of computing cycles for running and the maximal height the torso of a creature reached for jumping as benchmark. If the achievable quality for animations is considered as insufficient the fitness function can still be enhanced during the implementation phase.

The body of the work that is to be done has to be handled in Step 3. This is where the NE of animations takes place and decisions about the structure of the ANNs and GAs required for it have to be made.

[1] This trade-off is one of the reasons for animals and also humans to develop different gaits.

Because conventional NE is chosen as ML technique the topology of the ANNs has to be predetermined. As for the fitness function, the approach here is to choose a supposedly fitting approach initially and to improve it later on if new insights suggest so. The initial layout consists of the following: a number of input nodes corresponding to the number of sensors a character utilizes, one hidden layer with eight neurons in it, and an output layer with as many output neurons as the character has actuators. While the number of input and output neurons has to change for every new character, it is easy to decide on their numbers since they are more or less given by the physical components a creature possesses. The best number of neurons for a hidden layer and the best number of hidden layers in an ANN have to be determined experimentally but the stated amount should be a good starting point for the creation of appealing animations. Another thing that has to be determined is the activation function that the ANN uses. A natural choice at this moment is the continuous logistic function presented in Figure 2.4, that is able to handle the floating point numbers that will be delivered by the input sensors and are desired by the actuators.
When it comes to the design of the GA a few more initial decisions have to be made. Again because of the choice of using conventional NE, the single genes within the genotypes of the GA are all representing the weights of the connections in the associated ANNs and nothing else. Inspired by the preceding literature research, the values for the crossover and mutation rates are set to reasonable initial values and a *fitness proportionate selection* (also referred to as *roulette wheel selection*) method is chosen for determining the single crossover point that will be used here. In addition to the basic GA model, where a new generation consists of individuals that are created by recombination and mutation or unchangedly taken from the previous one, the GA developed here shall support elite selection. Elite selection is the process of deliberately taking the best, or some of the best, individuals from the old generation and let them lead by example in the next one. This feature does not have to be used during the training process but it is nice to have in order to protect good solutions that have already been discovered.
A fact that becomes obvious at this point is that many design decisions made here have to be revisited and reevaluated during the process of implementing this solution design. While literature research can suggest a good starting point, it is necessary to check if the produced piece of prototypical software lives up to the expectations repeatedly. Three already mentioned parts of the solution are expected to influence the outcome of the program particularly: the network structure of the ANNs, the activation function used by the single neurons, and the fitness function chosen for determining the quality of the solutions. In order to substantiate this assumption, those three components are tracked during the whole implementation process and their influence on the development of the program is analyzed and documented in detail in Section 4.2.2.

Apart from the choice and adaption of core elements of the solution, different parameter settings for some of the parts presented above can change the animation results in one way or the other. Although sensible values for these settings shall be predetermined to ensure a good outcome without big adjustments, it can be a good idea for the user of the software solution to play around a little bit in order to fine tune the developed

animations. All of this happens in the optional Step 4. The desire to enable the end-users to change certain values requires the final solution to provide means for them to do so in a convenient way, without having to alter the program code. Fortunately this is a feature that can be easily achieved via the Unity framework used here.

If a satisfactory animation has been produced it finally has to be saved to a reusable file in the fifth and last step of the animation process. This can be achieved by saving all the necessary information needed to reconstruct the result in a simple text file. The information that has to be stored includes the chosen character and motion type, the structure of the ANN, and of course the weights evolved for this neural network.

Enclosing the process and using the animations

To enable a sensible utilization of the software solution it is not enough to merely implement the functionality needed for creating animations but it is also necessary to make it easily usable. For that reason the implementation developed here features a graphical interface that on the one hand allows software users to interact with an ongoing simulation and on the other hand provides the possibility to watch and supervise the animation process in a real-time and also in a time lapse mode. Furthermore the settings mentioned before, other additional preferences, like the number of individuals and the duration of an animation run, and means for saving and loading good solutions shall be readily accessible.
One last point that is of great importance for the implementation phase is the issue of how to actually use the saved animations in other applications and games. The motion controllers that make use of the evolved animations have to exist as separate entities and independently from the program used for training them. They shall be encapsulated as lightweight components that can be used for character animation in various pieces of software later on.

3.2 Technology

In addition to the advantages already identified in the introduction, the chosen game engine Unity provides a graphical editor that is well suited to being integrated into the solution developed here. It allows accessing public code variables via simple text and numerical input fields that can be used for allowing quick changes of simulation settings. Though the most practical feature of the engine usable for this work is the possibility for creating so-called *prefabs*. Prefabs can be seen as blueprints for various objects usable in a scene that can be instantiated at almost any point during the execution of a program. Therefore they can be used for conveniently storing previously created character models that shall take part in an animation process. These characters have to interact with a world that follows the laws of physics and for that reason they have to be designed using physics components.
The basis for using such components is the utilization of a physics engine. There are quite some different physics engines that would be up for the task but it is reasonable to resort to one that is already integrated into Unity – and actually there are two of them.

Next to the PhysX [33] engine that is designed mainly for physics applications in 3D, the second engine Box2D [34] is, as the name already suggests, targeted specifically on 2D usage and is hence the one chosen for this project. However, not all of the concepts and component types provided by Box2D will be used here. The most interesting ones are *rigid bodies*, *box* and *circle colliders*, *hinge joints*, and *motors*. Rigid bodies and the colliders define the consistency and shape of a character's body parts whereas the joints and motors determine how they are fitted together and how they can be moved.

One more piece of already existing software plays its part during the realization of the devised solution design, even if it cannot be found in the final implementation anymore. It is the software used for experimenting with the NEAT approach when trying to decide which type of NE to choose (cf. Section 3.1.2).
The chosen implementation is a variant of NEAT introduced by Jallov [35] under the name of UnityNEAT. As the name suggests, it is directly integrated into Unity. The circumstance that this port of NEAT is easily usable in connection with a game engine makes it interesting for this work but also for game development in general. In fact, Jallov et al. [36] demonstrated its usability for game productions using a strategy game called EvoCommander.

3.3 Accompanying Experimentation

Many details of the design decisions made before have to be evaluated using an integrated experimentation and testing approach. This is not to be confused with the testing and evaluation phase following the implementation of a final solution that will be discussed in Chapter 5. Rather it is necessary to experiment with different parameters and settings for certain parts of the solution, many of them already mentioned above, in order to even find a usable final software solution. Consequently the questions that have to be answered here differ from the ones that will be asked during the final evaluation of the finished program. Instead of trying to find the limits of complexity for characters and testing other boundaries, questions that interplay with taken design decisions, but also the ones that arise later on, shall be answered using accompanying experiments.
Apart from helping to make big design decisions, like the type of NE and fitness function to use mentioned before, experiments can deliver insights on the smaller, often more practical and technology related, issues as well. Some interesting exemplary questions from this area are:

- How to simulate artificial joints and muscles?
  Unity and Box2D provide different kinds of joints that could be used to connect the single body parts of a character. They allow and restrict movements in several ways. A distance joint, for instance, ensures that two body parts are kept at a fixed distance from each other. The probably most suitable type of joint for the solution developed here is the hinge joint. This joint is defined by a hinge around which two connected parts rotate. While unlimited hinge joints can be useful for animating the rotating wheels of a car, the joints of most characters will have to be restricted

to allow the development of fitting motions. How exactly they should be restrained can be determined most efficiently by experimenting with different settings.
In nature the usage of flexor and extensor muscles goes hand in hand with motion along the axes determined by joints. Much PBCA research follows this idea and tries to simulate the movement of these muscles to achieve appealing motion (cf. [32]). However, since the simulation of realistic muscle movement is costly and the aim of this work is not to model natural movement as accurately as possible, it is of interest to experiment with functionality already provided by Unity, the chosen game engine. Motors are physics components that allow the movement of joints in all allowed directions and can possibly replace the functionality of real muscles here – best if their parameters are carefully fine tuned during experiments.

- What are the restrictions on the numbers of individuals that can be simulated simultaneously?
  A higher number of individuals in a population could lead to faster evolution of satisfactory animations, measured by the number of iterations, or even better solutions. Unfortunately this advantage comes at the cost of higher computation times. A good balancing between the number of individuals per generation and the duration of the calculations can be achieved experimentally.
- Which kinds of information shall be gathered by the sensors of a character?
  Varying types of sensors deliver different information about the state of a character and its surrounding environment. Visual sensors inform a creature about the shape and consistency of the landscape. Contact sensors are activated once the body parts they are attached to collide with another object (e.g. if a leg touches the ground). Other sensors could deliver knowledge about the location and rotation of certain body parts to the characters motion controller.
  Using many sensors helps with the acquisition of detailed knowledge about the world but again increases the computational complexity. Additionally it is more difficult for a NE controller to optimize a solution if it is receiving a great number of input signals. The goal here is to use as little sensors as possible but as many as necessary to allow for quality animations to emerge. Experimentation can assist in reaching it.

CHAPTER 4

Implementation

Some general decisions concerning the implementation of the previously designed solution have already been made. Next to identifying Unity and Box2D as basis for the software that has to be developed, there are some other initial remarks and considerations to be made and all the details of the implementation are presented hereafter. The final section of this chapter shows the user interface of the implemented solution and discusses how the system can be used successfully.

4.1 Preliminary Considerations and Remarks

An important point to consider is the practical realization of functionality that allows content creation. Unity already provides an editor that is well suited for designing digital characters and whole game worlds. Since it would be unreasonable and completely out of the scope of this work to try and develop a new one from scratch, it is only sensible to make use of the possibilities provided by the game engine.
For this reason the software is not implemented primarily as standalone application but rather it has to be used in combination with the engine to utilize its full potential. To clarify this point: the resulting piece of software can in fact still be used without Unity but the users are restricted to producing animations for predefined characters, using predefined settings and predefined types of movement only. This standalone version has its own right of existence for showcasing animations and the process of animating characters to interested parties (e.g. via mobile devices) but does not provide all the desired functionalities.

While it has been decided before that the landscape surrounding the created characters remains flat and that only land-based creatures will be examined, the concrete form of the characters in use has yet to be determined. In order to observe the quality of the resulting animations under diverse preconditions, several different fantasy creatures, showing varying amounts of legs and joints, are modeled and animated in the following.

Out of the pool of designed character models the seven creature types chosen for demonstrating different properties (and for later testing also) are:

1. *Apod (0 legs, 6 joints).*
2. *Biped 1 (2 legs, 2 joints).*
3. *Biped 2 (2 legs, 4 joints).*
4. *Tetrapod 1 (4 legs, 4 joints).*
5. *Tetrapod 2 (4 legs, 8 joints).*
6. *Decapod 1 (10 legs, 10 joints).*
7. *Decapod 2 (10 legs, 20 joints).*

How the structures of the physical bodies of these creatures look like is illustrated in Figure 4.1 and Figure 4.2. The thin green lines depict the physical bounding boxes of the bodies, the white areas surrounding it are merely semi-transparent textures.

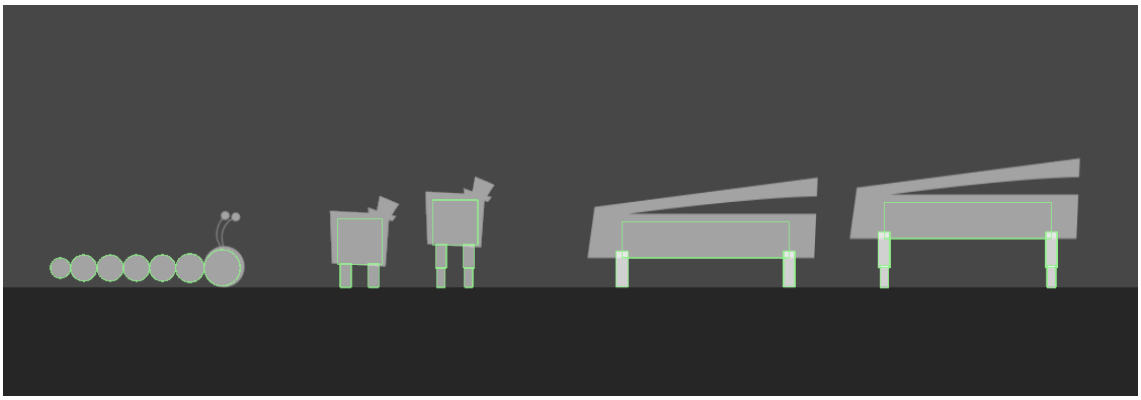

Figure 4.1: Bodies of selected character models – 1 (Screenshot)

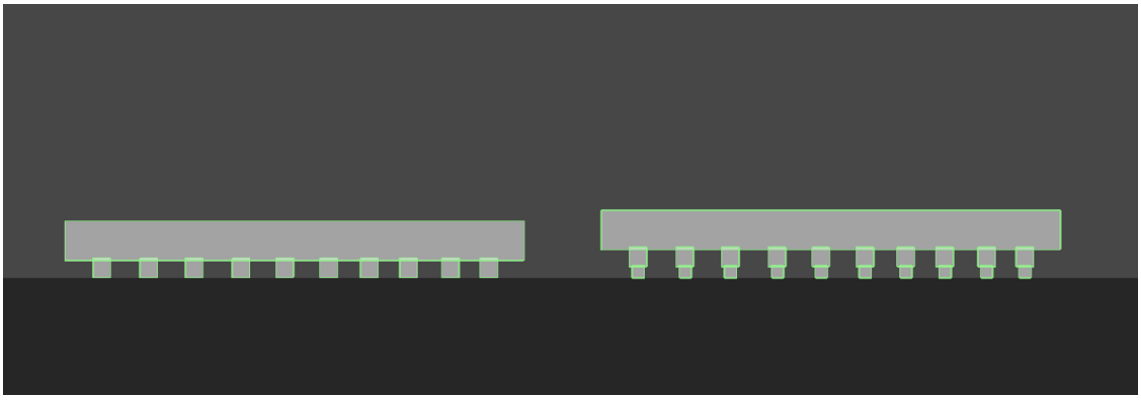

Figure 4.2: Bodies of selected character models – 2 (Screenshot)

The reason for the bounding boxes of Figure 4.1 not matching the white shapes exactly is rooted in design choices for the characters' appearances. Because the five characters in

this first screenshot are used during later quality assessments (cf. Chapter 5) they are outfitted with more elaborate textures. In contrast, the two decapods depicted in Figure 4.2 only received placeholder textures that fit the contained bounding boxes exactly.

One last thing to consider before moving on to the actual process of implementing the solution has to do with the dimensionality of the game world that is used. If the list of creature types given above is compared with the provided screenshots it becomes obvious that something does not add up. Instead of two bipeds and two tetrapods it seems like four bipeds are depicted in Figure 4.1. The reason for that is that two of the legs of the two rightmost creatures are behind the others, as suggested by the slightly more opaque color of their legs.
This points in the direction of possible complications for the process of animation. In two dimensions it is not necessary to use paired sets of legs to ensure the balance of a character. It is possible for creatures with all their legs lined up, like the bipeds and decapods here, to move successfully even if they might not be able to do so in a 3D world. This has to be kept in mind, as a possible source of error if problems with the emergence of natural looking movements occur later on during the implementation phase.

4.2 Implementation Details

The task of creating a solution that allows its users a comfortable and largely automatic creation of animations for 2D characters can be divided into a few separate topics. Put together they constitute the working prototype that is the centerpiece of this work.

4.2.1 Character Creation and Mapping

The basic process of character creation has been presented before: Some physics components are combined in the unity editor to form the body parts of a creature. An aspect that has not been discussed in detail yet is the fact that this character creation has to abide to certain rules. Because an automatic linkage of a character's anatomy to its brain is one of the goals here, it is required to impose some restrictions at this point. To understand why this is the case and which restrictions are needed it is important to take a look at how this mapping is achieved.
The aim is to connect the outputs of the ANN used as motion controller for the character to the motors that move the single parts of its body. In order to do that, it is necessary to know how many movable parts a creature has – here all the movable parts are legs. This information is not only required to map the output neurons to the legs but also to decide on how many of these neurons are actually needed. The approach chosen here for enabling the developed software to deal with this issue, is to assume a certain structure for the character models. The program analyzes this structure sequentially and handles it as follows: The first encountered body part is always taken as the torso of a creature. Which part of the creature this is can be chosen freely by the designer but it has to be ensured that this component is encountered first in the Unity object hierarchy for the character in question (by simply dragging its identifier to the uppermost position in the

graphical editor). All other parts are considered to be legs and have to be linked via a hinge joint, containing a motor. These motors are taken as actuators for the model and their total number corresponds exactly to the amount of output neurons needed.
While the above layout for the character design has to be followed strictly in principle, it is still quite easy to adapt it to the potential need for a different one. Using a tag system that allows single parts to be assigned to a class of components ensures this quality. As an example, while searching through all the parts of a character's body only those tagged as "leg" could be considered and counted for further usage. Parts tagged as "hand" or "head" would be ignored for animation purposes.

With respect to the devised structure, several characters were created for usage during further implementation steps and testing phases. They have already been briefly introduced with help of the Figures 4.1 and 4.2. These characters lay the ground for starting to work on the actual ML parts of this implementation, since they are needed for experimentation purposes.

4.2.2 Neuroevolution Implementation and Improvement

With the support of the newly designed characters, the most demanding part of this development process can be tackled. As discussed during the solution design, the process of finding good ways of creating animations and hence a good approach to implementing NE is an iterative one, accompanied partly by experiments. For that reason an initial basic solution is implemented at first, using approaches and settings suggested in the literature. This serves as starting point for further improvements. Only the three components that are expected to have the biggest influence on the overall outcome (cf. Section 3.1.2) are presented and tracked subsequently, for the most important milestone versions produced during this process.

Version 1.0

Apart from some minor decisions on all the single parameters, the main parts of the first solution are chosen as follows:

- *Network structure:* As conventional NE is used, the ANN in use is fully connected. The initial numbers of neurons and neuron layers have been determined in Chapter 3 and are eight, resp. one. The number of output neurons corresponds to the number of joints a character has and is determined automatically. The only thing missing here is the number of input neurons, which depends on the number of sensors used for a certain character.
  This number is in fact also determined automatically, by adding up the following sensors that are assigned to every creature: one rotation sensor for the torso, one sensor per joint, measuring its angle, and one contact sensor per leg sending a signal if a foot[1] touches the ground. For the Tetrapod 2 in Figure 4.1 this would result in a count of 13 (one torso sensor plus eight joint sensors plus four foot sensors).

[1]Here a foot is defined as the last segment of a leg (the one that is supposed to have contact with the ground). If a leg consists of one segment only, it is also considered the foot.

Therefore this exemplary character's ANN would have 13 input neurons, 8 hidden neurons, and 8 output neurons. Also there is a bias input to be considered for every layer but the last one.

- *Activation function:* Also determined before, the sigmoidal logistic function is chosen as activation function initially. The corresponding formula is:

$$f(a) = \frac{1}{1 + e^{-a/p}}$$

For this formula the letter a denotes the activation that goes into a neuron and p is used to alter the shape of the curve produced by the function – it is usually set to 1. The logistic function maps all inputs for every neuron to a floating point number in the range of 0.0 to 1.0 (cf. Figure 2.4). The final values received from the output neurons are remapped a bit to span the range of -0.5 to 0.5. This little adjustment makes the assignment to the two possible movement directions of a joint motor more intuitive.

- *Fitness function:* The fitness function used here simply rewards the distance the torso of a character has moved to the right ($endPosition.x - startingPosition.x$) during a certain amount of computation cycles. Of course this function can only be used to train running animations and is not suited for learning to jump. Running is chosen as the default type of movement used during all the software iterations at this point, because it is easier to observe here if a satisfying motion is evolving and the program is behaving as expected.

Some observations have been made after realizing this first version. On the bright side, it has been possible to create reasonably swift motion in the right direction after a few generations have been simulated. On the other side, the movement looked rather jittery and therefore quite unrealistic. Looking at the Tetrapod 2 again – the creature that has been mainly used for experimentation purposes – all that happened was that it lifted two of its four legs as high as possible and used the other ones to sort of vibrate over the floor. Vibrating here means very tiny but also very quick movements where the feet touch the ground only briefly but do so extremely often.
After trying to get rid of this behavior and achieve a more realistic type of movement by altering different parameters, it has been decided that a new fitness function might lead to better results. The thought behind that was that the characters might in fact have learned the fastest type of movement but that this motion comes at the cost of very high energy consumption – something that would not be retained in nature.

Version 1.1

To deal with the issue of high energy consumption, the energy costs have to be integrated into the function that assigns fitness values to the characters. The other main parts did not change substantially and this results in the following updates for the new version:

- *Network structure:* No major changes.
- *Activation function:* No major changes.

- *Fitness function:* Several approaches can be taken to reducing the energy consumption of a creature's movement. Attempts tried here are the penalizing of very rapid movements, of the overall distance a leg moved, and of a high count of directional changes.

Although it could be achieved with help of this new type of fitness evaluation that the characters reduced the vibrating movements, the tetrapods still lifted two legs up and used the two remaining ones in order to perform unsatisfying movements. This lead to considering the question implied in Section 4.1: "Will a 2D character make use of all its legs if they are arranged in parallel and not really needed to keep its balance, as it would be the case in a 3D world?" Whilst the answer to this question might be "No." in some cases, it was not the core of the problem here.
After careful investigation of the issue and a thorough revision of the software produced so far, some conclusions could be made. The problem's source has been located in the area of mapping the in- and output values of the physics components to the motion controllers, resp. the ANNs.

Version 1.2

The different input sensors deliver different types of inputs. The rotation sensors of the torso and the sensors measuring the angles of joints provide values in the range of 0° to 360°, whereas the contact sensors deliver the boolean values 0 or 1. Even though the weights within an ANN might automatically adjust to inputs of different magnitudes, the high numbers delivered by the rotation sensors do not play well with the chosen activation function.
A look at the logistic function in Figure 2.4 immediately shows why. Inputs below the value of -1 or above the value of 1 all result in an output of 0 resp. 1. To illustrate what exactly this means: If the rotation sensor of the torso communicates an input of 1° it results in the same activation as an input of 359°, given that this input is multiplied with a weight of 1.0. While a proper weighting can mitigate the issue (for example the weight for this sensor could be set to 0.0025), this scaling problem has to be handled differently, especially since the left half of the curve produced by the activation function is completely ignored using these settings. As a result the subsequent changes are applied:

- *Network structure:* No major changes.
- *Activation function:* The most important changes are not concerned with the activation function as such but with the discovered scaling issues. Nonetheless a new function has been identified to be better suited to this implementation's needs and is hence used hereafter. This new activation function is called hyperbolic tangent function ($tanh$) and corresponds to the formula:

$$tanh(a) = \frac{1 - e^{-2a}}{1 + e^{-2a}}$$

The letter a is, as before, denoting the activation provided to a neuron. Figure 4.3 depicts the resulting curve.
To adjust them to this new function, all input signals are scaled to span the range of -3 to 3 (e.g. $0° = -3$ and $180° = 0$). Another aspect that deserves attention is

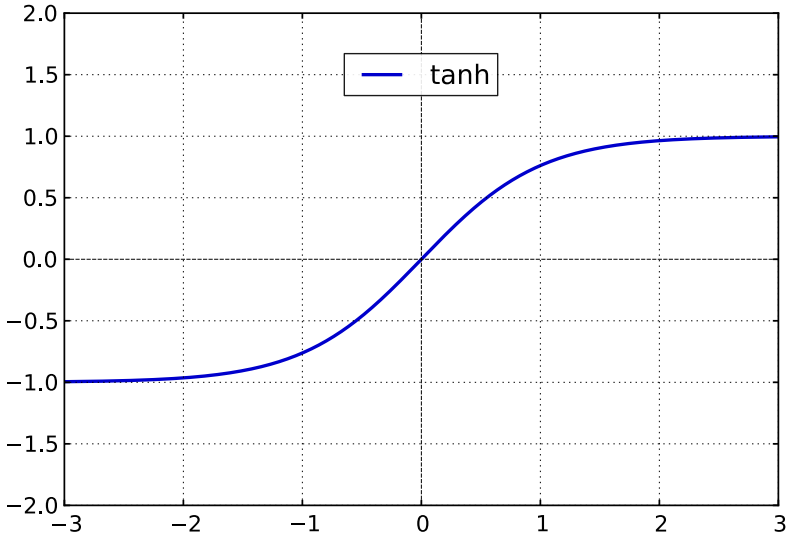

Figure 4.3: Hyperbolic tangent function (taken from [37])

the output produced by the function. In order to ensure an input of -3 to 3 also for the neurons in the layers behind the first one, the output has to be scaled to the desired range as well.
One last remark here concerns the jump between 359.$\dot{9}$° and 0°. Although the angle of a body part is nearly identical for the rotations of 1° and 359° this might lead to very different activation values. Unfortunately no easy solution to this problem could have been found, but it is better to use a bodies initial rotation as starting point and add resp. subtract the occurring changes from there. If mitigated in that way, this issue does not impact performance noticeably, especially since most rotations are restricted anyway.

- *Fitness function:* No major changes.

The modifications to the way sensory signals are handled and the adoption of a new activation function improved the achievable results greatly. Finally the characters started to evolve appealing and naturally looking movements and even the tetrapods started to use all of their four legs to gallop off. The time has come to optimize a few little things and produce a final version of the NE component for the prototype.

Version 2.0

After performing extensive experiments, using the latest and fully functional version, some last adjustments to improve on the developed solution can be made. The core parts are affected in the following way:

- *Network structure:* Interestingly, it proved to be enough to provide fewer inputs to the characters than expected. Instead of using the rotation sensors for the torso and the joints, as well as the contact sensors for the feet, it is sufficient to measure only the different angles. All contact sensors could be removed without decreasing

the animation quality. The usage of less input information results in fewer neurons in the input layer and therefore in a less complicated training process and better performance also.
Additionally the number of neurons in the hidden layer is increased to ten, in order to enable a higher flexibility for the creatures with more legs. Using the exemplary tetrapod introduced in Version 1.0, the neuron and neuron layer layout of the ANN shifts from 13|8|8 to 9|10|8.

- *Activation function:* No major changes.
- *Fitness function:* As it turned out in the light of the improved implementation, ignoring the energy consumption of the characters did not impact the resulting movements negatively – at least for the scenarios examined during this work. The changes made to the fitness function during the development of Version 1.1 are hence rolled back again. As in the beginning, only the covered distance is used as fitness criterion.
  Since this is the description of the final version of the ML part, the second type of movement shall not go unmentioned. For the motion type of jumping, the maximal height a character's torso reached during the training phase is taken as fitness measurement, but not before deducting its initial height.

This latest version of the central part of the software constitutes the final outcome of the NE logic development phase and represents the basis for the upcoming testing and evaluation procedures. It promises to enable the users to easily and effectively train various characters, without having to alter all the different parameters for each of them.

4.2.3 Training and Animation Visualization

Next to enabling users to create characters and providing the logic for animating them, a third important part of the implementation is the visual presentation of the inner workings of the NE process and the resulting animations.
The main differences between the visualizations of the training phase and the resulting animations can be found in the number of individuals on the screen and the way the camera captures what is going on. In training mode the camera stays in a fixed location, so that the evolution of the characters, which are all drawn simultaneously, can be observed properly. In animation mode only one character is instantiated and animated using the best (i.e. fittest) solution developed so far. The camera is no longer fixed but follows this character at all times.

Figure 4.4 exemplary illustrates the visualization of the training of an apod character[2]. The bodies of all the individuals that are created in every generation are drawn semi-transparent and it is therefore easily possible to track their development. The first third of the picture shows the population at the end of generation 1. Not much has happened yet and most individuals stay around their initial spawning point, just left of the continuous white line which marks the starting point. The second third depicts the situation at the

[2]The relaxed initial state of the physics body of the apod can be seen in Figure 4.1.

end of generation 3. Several individuals have successfully passed the starting line and it seems like a single one already developed a promising way of moving forward. The last third of the image corresponds to generation 7. Many creatures followed the example of the leading character of the third generation and its movement strategy has been refined over the last few iterations. As a result, the maximal covered distance, and hence the fitness, is more than six times as large as in the first generation.
While all the information concerned with the animation of the characters is theoretically available in text form, it is of great help to the designers to actually see and follow their development. Really, this is the only sensible way of determining if an evolved movement is not only efficient but also visually satisfying, even in an early stage.

After waiting a few dozen more generations and after a potentially usable animation evolved, it is time to look at the result of the NE process. Figure 4.5 shows how these results are presented to the user. The texture of the only protagonist in the scene can be changed from semi-transparent, as it is used in the training mode, to filled. This ensures a more final look of the character. If the evolved motion of the creature is appealing, it can be saved for later usage at this point.

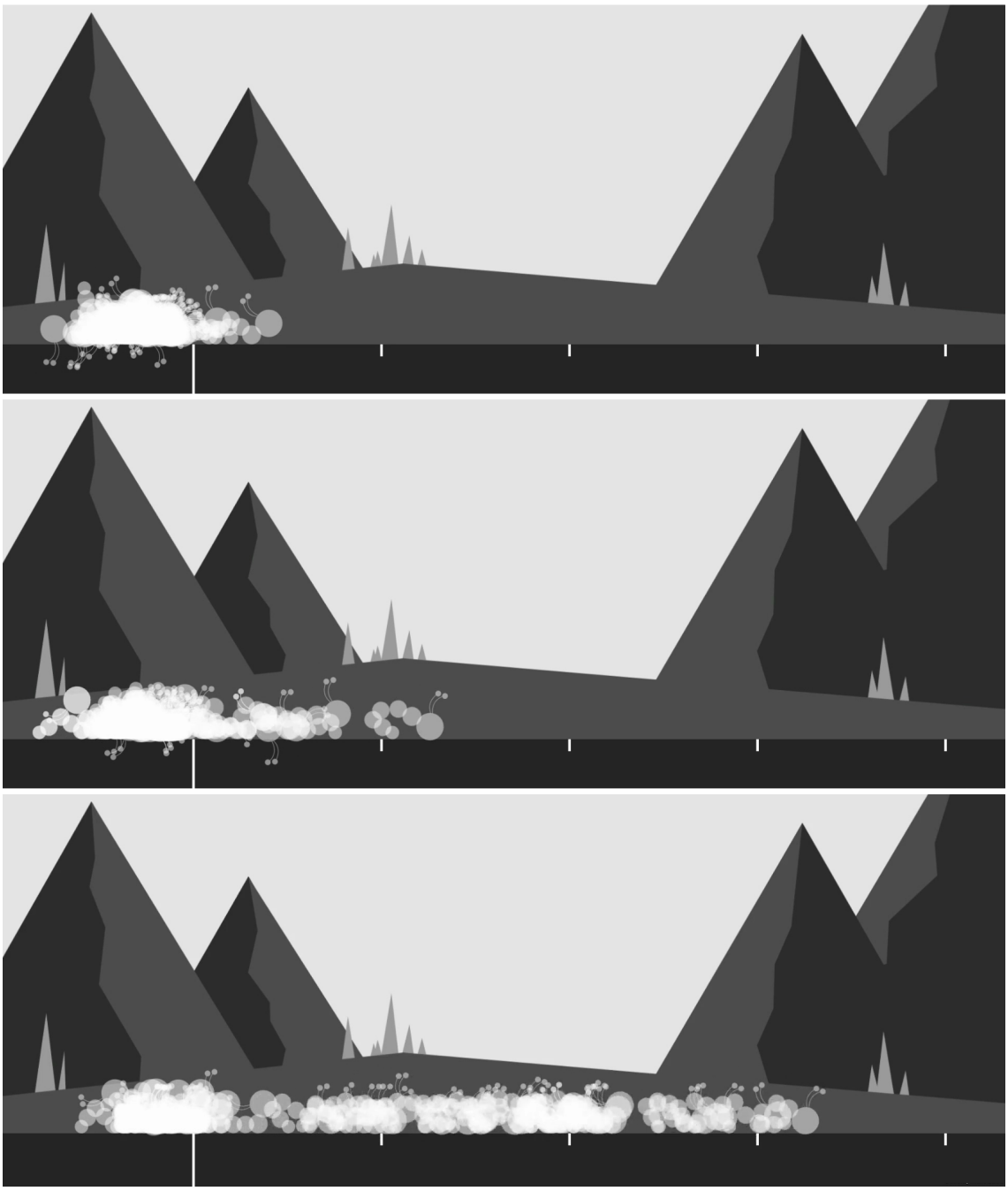

Figure 4.4: Visualization of different training stages of an apod character (Screenshots)

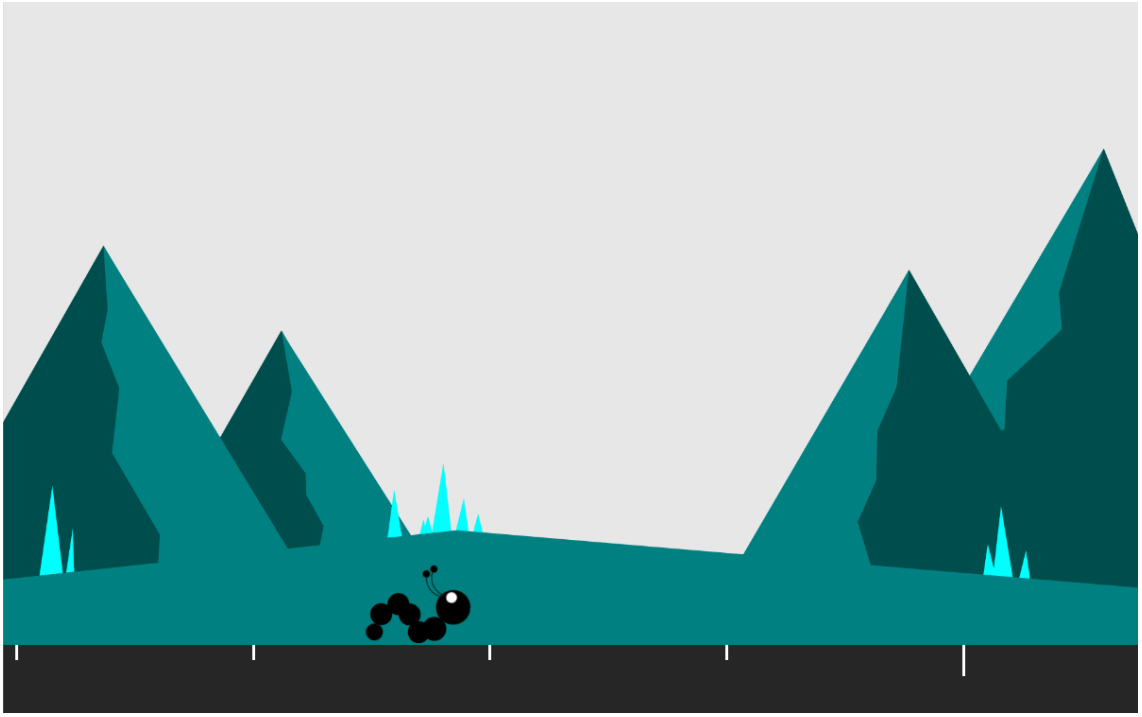

Figure 4.5: Showcase scene for character animations (Screenshot)

4.3 Usage of the System

In addition to the capability of showing the movement training phase and the resulting animations, the implemented software solution also provides a user interface that supports the process of creating an animation, as identified in Figure 3.1.
Figure 4.6 depicts this interface. It consists of two separate parts. The upper half of the picture represents the completely custom-made part of the solution, that allows the control of the actual training and animation procedures. The lower half shows Unity's inspector window, which makes it possible to alter certain values within a script called "Parameters" but without having to change the program code. Using both parts in combination enables users to carry out the full process of animating a character. The only task left out is the optional character creation, which of course has to be done in the Unity editor.

Before the process of creating an animation can be started some parameters have to be checked and set. While most of the fields visible in the inspector window can be ignored because the initial settings work just fine, at least two have to be filled out correctly. Namely they are: the type of movement to learn and the active character type. In the standard implementation provided here, the number 1 is assigned to the motion of "moving forward" and the number 2 to "jumping", for the type of movement to learn. The value of the active character type corresponds to the position of the creature in the list of character prefabs, listed right below the field.
Using the parameters[3] presented in Figure 4.6, hitting the start button results in the training start of the forward movement for an apod character. With assistance of some of the other buttons on the left hand side of the scene, the speed of the training process can be altered at any time and the process can be paused completely, in order to take a look at the currently best individual. The button "T/S Mode" allows the users to switch between the semi-transparent and the solid style for displaying the characters. A last element that is aligned on the left part of the screen is a text display that indicates the current generation and the best fitness that has been achieved by a character up until now. On the right hand side, a few interface elements that allow the saving and loading of previously evolved animations can be found. They aid in completing the process of creating an animation and hence in concluding this chapter.

[3] Precisely these parameters have been used for producing the Figures 4.4 and 4.5.

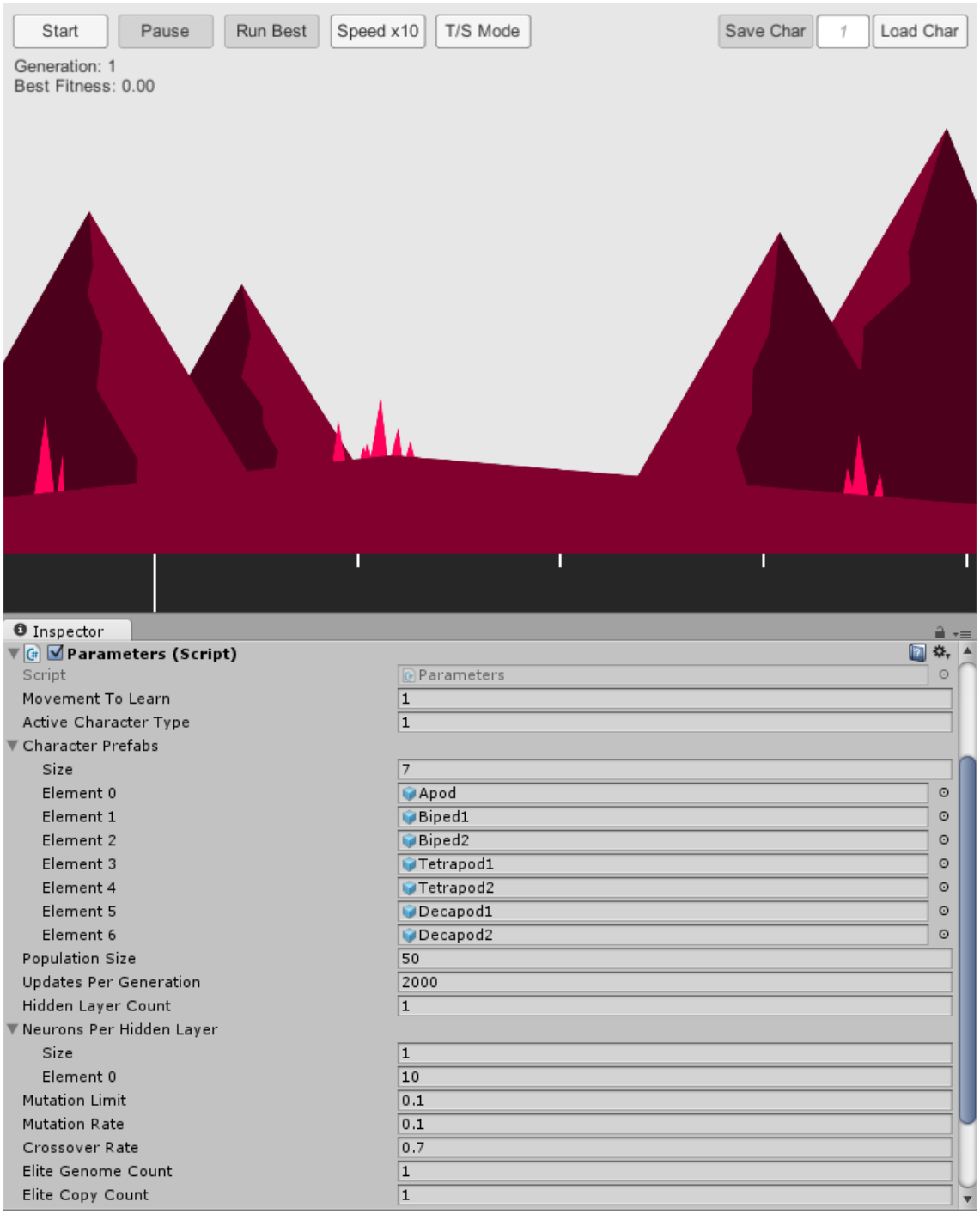

Figure 4.6: User interface (Screenshot)

CHAPTER 5

Testing and Evaluation

Evaluating the produced software solution is a rather complex matter. Some aspects of the results produced by the implementation can be simply measured, like the training duration for a certain character and a particular number of generations, while other parts are more difficult to handle. Especially the assessment of the quality of the resulting movements can be challenging.
How these different aspects are evaluated, using test cases that are identified as being suited for the task at hand, and all the detailed results of the performed testing runs are presented in the following. After the discussion of the results, the limitations of the software solution and the concepts behind it, as well as some insights gained during the implementation and testing phases, are mentioned to give a frame of reference.

5.1 Test Cases and Criteria

Both, the two different movement types used for testing purposes as well as the characters that will participate in the testing procedures have been introduced before (Sections 3.1.1 and 4.1). Now the time has come to go into more detail and look at the specifics of the test cases that lay the foundation for this evaluation phase.
A total of 14 distinct scenarios are considered: First, the 7 different characters (Apod, Biped 1, Biped 2, Tetrapod 1, Tetrapod 2, Decapod 1, and Decapod 2 – cf. Figures 4.1 and 4.2) are observed while trying to learn a suitable forward movement. After that, the motion of jumping will be trained. While creatures like the Apod and the Biped 1 only consist of a few components and should be relatively simple to animate, the two decapods are assembled out of a considerably larger amount of extremities and should hence be more challenging to train. All in all, the animation of the various characters selected here should, together with the choice of two very common types of motion, cover a wide area of use cases that can be found in practice.

In order to get a grasp on how the different creatures performed after running the experimental training phases, the above test cases are evaluated using the following four performance criteria:

1. *Movement efficiency.* This first criterion is relatively simple to analyze. The distance a character moved or jumped after a certain number of generations is measured. More specific, the fully covered distance or the maximal reached height is noted after 300 generations and evaluated subsequently.
   The primary evaluation goal here is not to determine which character ran the fastest or jumped the highest, but to be able to check if an efficient movement could be achieved and how it evolved. Some questions that should be answered are:
   - Does a creature even move in the desired direction or is it lingering around the starting coordinates?
   - Is the evolution continuous, with small improvements in many generations, or rather step-by-step, with big performance jumps in a small number of generations?
   - Is a good result obtainable using only a few runs?

   Because of the last question all experiments are conducted only three times initially. While getting an acceptable result from three consecutive runs is not enough to prove that an efficient movement can be produced at every try, it is sufficient to suggest that it might be possible for a designer to produce usable animations without having to run the program unreasonably many times.
2. *Robustness.* Although three distinct runs might give a first indication about a good result being achievable in a reasonable amount of time, a further question that is of interest here is if the results are reliable to some extent. For that reason, seven additional test runs are conducted – which leads to a grand total of ten runs per character and movement type (i.e. 140 runs altogether).
   Out of these runs, the first three for every test case will already be analyzed and described in detail for the criterion of movement efficiency in Section 5.2.1. These descriptions provide some insights into the way certain movements evolve for the different creatures and they can give a good impression of which difficulties and challenges are to be expected for the other runs as well. The most interesting aspect of the robustness evaluation is therefore the investigation of the variation within the test runs. Does the overall outcome change completely with every new run or does it stay almost the same for three and for ten runs?
3. *Training duration.* Because the training duration is of high importance for the technique's usability in real-world or even real-time applications it is also of interest here. Next to measuring the training duration for the full 300 generations that will be computed during all the single experiments, two other tests will be performed as well.
   Instead of taking the time for a fixed number of generations it is tested if satisfying movements can be evolved during a fixed amount of time. 5 minutes are chosen as first time interval. Animations achievable in this time span could be of use for a live utilization in slow-paced games. The second time interval is 1 minute. Evolving

movements in such a short time frame would allow the sensible usage of NE for developing character animations in a wide segment of game development.

4. *Quality of the results.* The quality of an animation is not easy to determine. A motion that looks appealing to one person might be dismissed as unfitting by the next. That said, it is also true that not every animation a designer creates has to be tested for its mass appeal. The creator of a character is usually able to develop a fitting motion for his or her creation without substantial external input.
   The approach for assessing the quality of the results chosen here is sort of a middle ground. Some animations are preselected from the wealth of received results and are presented via social media[1] for public evaluation afterwards. These movements are chosen from the results of the testing runs which are performed to determine the movement efficiency. This means that the motions are probably not the best that could be discovered if more runs would be conducted, but in this way they also do not represent artificially staged results that are only rarely achieved in practice. Apart from the subjective visual impression, the main criterion for preselecting an animation is the corresponding character's ability to use all limbs sensibly.
   This preselection process is the reason why the characters presented in Section 4.1 are displayed separately. While the animation results for the first five creatures (Figure 4.1) reached promising levels, the movements of the two decapods depicted (Figure 4.2) never got beyond an unsatisfying state – not even using more computing cycles and a higher number of training runs. The legs did not work well together and the resulting movements are not very naturally looking. For that reason it has been decided to omit those two characters and the associated test cases from the external evaluation.
   In order to allow an easy access to the developed animations during the public evaluation phase, a video[2] briefly illustrating the process of NE and showing the final animations is produced. A simple survey asking about how well these animations fit the characters is provided to the viewers to enable them to participate in the evaluation and to give feedback. Apart from some simple demographic questions, the two questions asked that are of most interest for this work are: "Does the evolved forward movement fit the character?" and "Does the evolved jumping motion fit the character?". One further question asked is concerned with the evolution of characters as game mechanic in its own right and reads as follows: "If you're interested in video games: Would you like to play a game that allows you to create and automatically evolve your own creatures?".

A last point to make in this section is that since one of the goals of this work is to create a flexible solution that allows users to utilize it without the need to optimize all parameters for every single animation procedure, the settings used during the experiments are fixed – to the values depicted in Figure 4.6. The only exception here is the number of updates per generation, which is reduced from 2000 to 500 cycles for the second motion type of

[1] The social media services used here are mainly Survey Tandem (`http://surveytandem.com`), Reddit (`https://www.reddit.com`), and Facebook (`https://www.facebook.com`).

[2] The video can be watched online at: `https://youtu.be/Y6OmG1ZZOz4`

jumping. The reason for that is that it does not take a lot of time for a character to perform a jump and that this saving of time leads to substantially shorter experimentation and training durations[3].

5.2 Results and Performance

The results corresponding to the test cases and evaluation criteria identified in the last section are presented and discussed here. For every part the procedure of doing so stays the same. First the the animation of moving forward is reviewed for all relevant characters and only after that the jumping motion is thematized[4].

5.2.1 Movement Efficiency

Moving forward – Apod (0 legs, 6 joints)

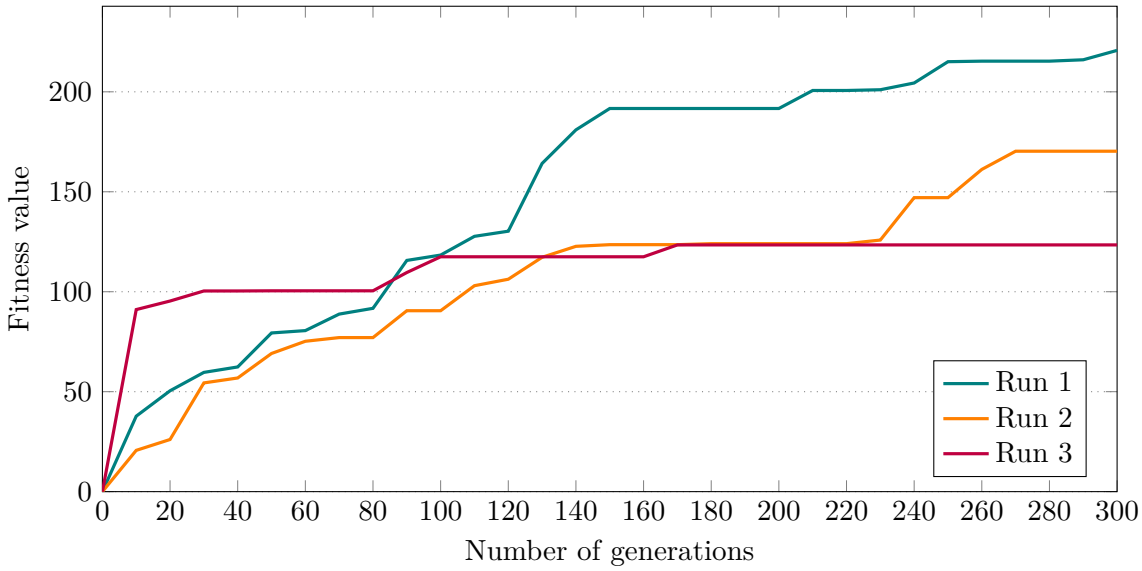

Figure 5.1: Fitness development for the Apod (moving forward)

The final fitness values for the three initial runs are:
Run 1: 220.73 Run 2: 170.29 Run 3: 123.42

The first examined character, the apod, started moving in the right direction already early on. Figure 5.1 depicts how the reached fitness values increase over time for the first

[3]Using 2000 cycles per generation for learning the jumping movement would not have decreased the quality of the results. Hence it would have been possible to use completely identical values for all characters and movement types.

[4]Many of the evolved movements can be witnessed by watching the video that has been presented before for usage during the quality assessment phase (`https://youtu.be/Y6OmG1ZZOz4`).

three runs. Interesting here is that although the third run started out quite promising, the associated curve flattens considerably after a few generations. What happened? While the creature adopted slow but easily improvable crawling motions during the first two runs, it abandoned this behavior for the third. Instead it favored a different approach, a sort of rolling movement, but had difficulties improving on the initial motion. It seems like it ran into a local optimum. A local optimum in this context is the situation where a character develops a movement that is obviously not the best one but it is so good that a better one cannot be found under the restrictions imposed by the mutation rate, the crossover rate, and other parameters.
Nonetheless, without looking at the aesthetic aspects yet, the animation development went just fine for this character. All three runs produced forward movements that allow the apod to move in the desired direction at a relatively high pace. Even the resulting motion of the worst run is fast enough to be usable in a real-world scenario – an outcome that stays in fact unchallenged by observations made for any of the remaining seven runs for this test case (cf. Figure A.1).

Moving forward – Biped 1 (2 legs, 2 joints)

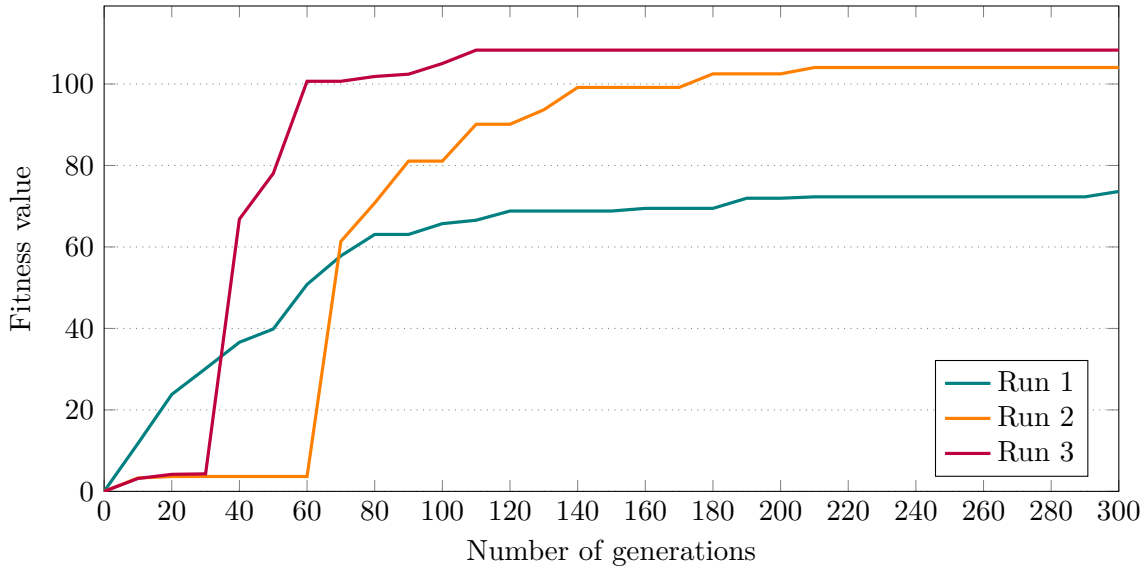

Figure 5.2: Fitness development for the Biped 1 (moving forward)

The final fitness values for the three initial runs are:
Run 1: 73.62 Run 2: 104.05 Run 3: 108.32

It already gets more challenging for the implemented software solution to find an efficient movement type for the second character in question, the first biped. It has been relatively easy to motivate an apod, a creature without legs but lots of joints, to move forward by randomly activating all its motors. Almost any motion moves its body in one way or another – if only seldom in the desired way. This is different here. The biped can

maneuver itself into a position where it is impossible to continue moving on. Quite often this character loses its balance and ends up lying on one of its sides or, even worse, on the top. Only if the balance is kept successfully, a promising movement for the creature can be evolved.
This circumstance explains what is going on in Figure 5.2. The curves of run two and three look very similar, with the only exception that it takes approximately twice as long to find a balanced way of moving during the second run. The flat parts at the beginning of those curves indicate that all individuals kept tumbling over and the steps around generation 30 resp. 60 mark the point where a single individual managed to stay upright and actually started to run. Although some improvements can be achieved over the next few dozen generations, a fitness plateau is reached eventually. Small changes might still occur but apparently a very good movement strategy has evolved. After all, the possibilities for creating movement variations with only two legs and joints are limited. The outcome of the first run looks relatively continuous in comparison with the others. What happened here is that the biped fell on its right side but managed to use its right leg, which was now behind the creature, to push itself forward. This movement strategy did not turn out to be as effective as the others but it is sufficient to create a reasonably fast forward movement. Interestingly, this shows some resemblance to the development process of babies. It is difficult to start off with the gait of walking and trying to crawl can lead to good results earlier. Unfortunately the biped has not been able to eventually evolve the crawling movement into an upright walking one during the first run.
As for the apod, the results for the first biped turned out to be satisfactory and also satisfyingly reproducible. Again, this does not mean that in fact every run produces an almost ideal result but that it is quite easy to get one that is well usable (cf. Figure A.2 for this test case). It is important to note at this point that although the charts in this section generally have the same dimensions and the number of generations on the abscissa stays the same, the scaling of the ordinate varies from test case to test case. The best fitness achieved during the first three runs for the biped is actually a little lower than the worst fitness for the apod. As indicated in the last section, this is not automatically a sign of worse performance but it is mostly explained by the characters' anatomy.

Moving forward – Biped 2 (2 legs, 4 joints)

The final fitness values for the three initial runs are:
Run 1: 50.38 Run 2: 41.05 Run 3: 179.69

With an additional set of joints, the task of balancing gets even more difficult for the second version of the biped. The result of the second run shown in Figure 5.3 shows similarities to the first run of the last character. It also looks more continuous than the other two and it again stands for a lateral movement using only one leg. The other two motions are actually rather similar, even if the curves do not suggest that. The character developed a type of locomotion that can be compared to a repeated flic-flac or somersault movement. It continuously rotated around its center of gravity and achieved an astonishing tempo doing so. The difference between run one and three is that during run one the creature always landed on its head and could not continue moving anymore

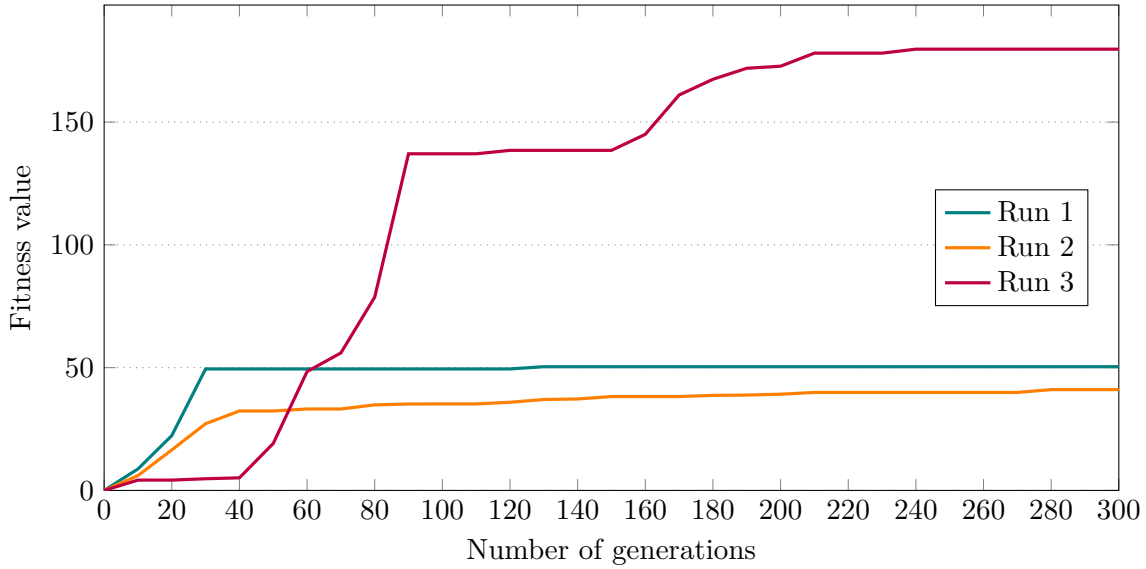

Figure 5.3: Fitness development for the Biped 2 (moving forward)

but only helplessly spread its legs into the air.
This is the first test case where not every of the exemplary runs led to a more or less usable outcome. Due to the increased difficulty of balancing the character, it cannot be guaranteed that a continuous forward movement will be evolved at all times. However, it is still possible to find such movements and doing so should be no problem in an offline environment (cf. also Figure A.3).

Moving forward – Tetrapod 1 (4 legs, 4 joints)

The final fitness values for the three initial runs are:
Run 1: 147.71 Run 2: 71.37 Run 3: 209.01

Because holding the balance and the risk of tipping over can be largely ignored here, the evolution of animations for the simpler tetrapod variant passed by mostly uneventful. All runs depicted in Figure 5.4 led to the development of a certain forward movement. Only in the second run, the character skipped a few beneficial development steps that it took during the first and third one.
As a consequence of the robust stature of the tetrapod it is safe to use the solution created here for animation purposes without having to fear that the creature gets stuck, as it happened for the second biped. Still, as the extended experiments showed (Figure A.4), if the movement efficiency has to be high it might be necessary to perform several animation runs before getting the desired outcome.

Moving forward – Tetrapod 2 (4 legs, 8 joints)

The final fitness values for the three initial runs are:
Run 1: 136.93 Run 2: 72.27 Run 3: 75.41

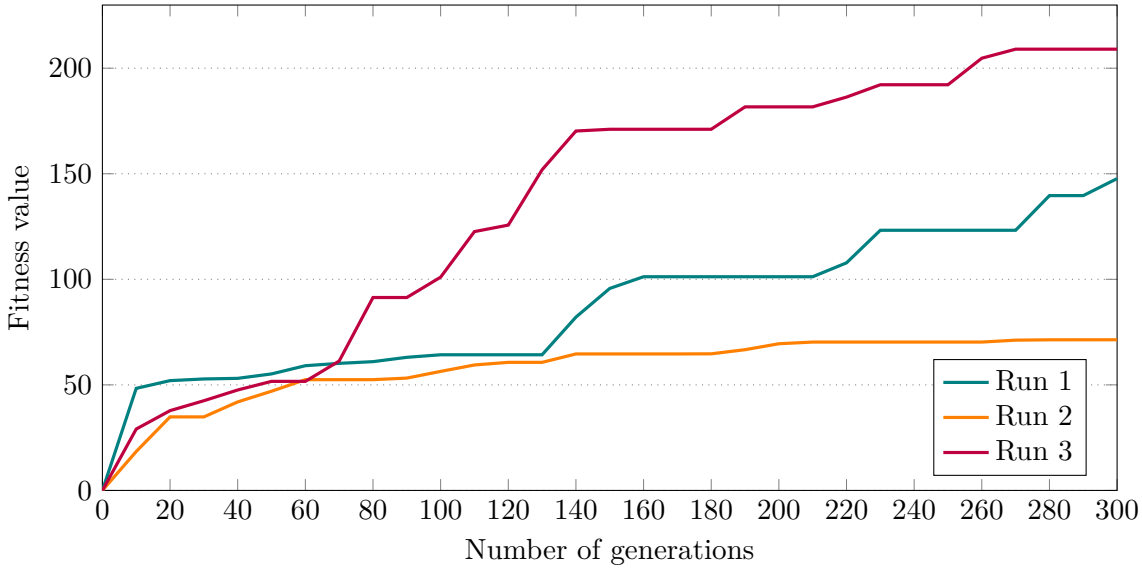

Figure 5.4: Fitness development for the Tetrapod 1 (moving forward)

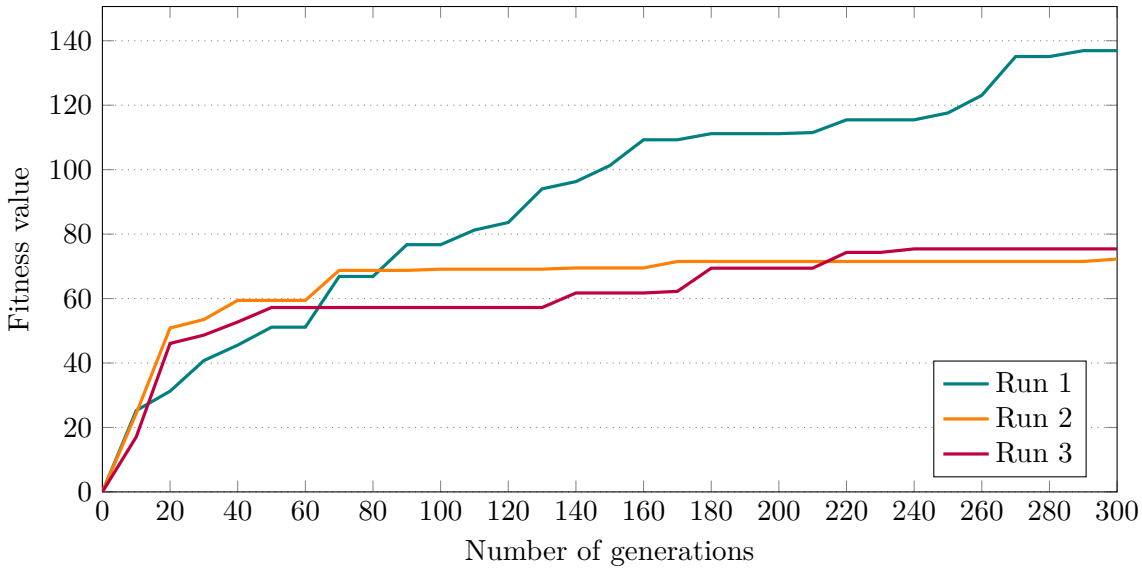

Figure 5.5: Fitness development for the Tetrapod 2 (moving forward)

The second tetrapod, consisting of 4 legs and 8 joints, is the character that has been used for most of the experiments that have been conducted during the implementation phase. The galloping movements that are mentioned in Section 4.2.2 could indeed be reproduced in the course of testing.
Because the desired movement sequence is already quite complex and all motors and legs have to be synchronized to achieve a suitable solution, the development of such a

movement can take some time. In fact, this type of character would also profit from a higher number of generations. In more lengthy experiments a fitness value of 200 and more has been reached. But time is not the only constraint that limits the quality of the movement. Once more the topic of running into local optima becomes an issue. The more complex a motion sequence gets, the harder it gets to discover it – especially if it is not easily producible by performing small incremental improvements on any arbitrary starting setup.
Figure 5.5 indicates what actually happened during the testing process. Although the movements in the two latter cases started out promising, they could not be improved beyond a certain point. It was simply too hard a task for the program to jump from the already cultivated sub-optimal solution to a completely different one. Only in the first run it has been possible to achieve the appealing and also efficient galloping motion.
As for some of the other characters, this means that not every test run brings results that are satisfactory but that suitable movements can definitely be evolved without too much effort on the designers' side (cf. also Figure A.5).

Moving forward – Decapod 1 (10 legs, 10 joints)

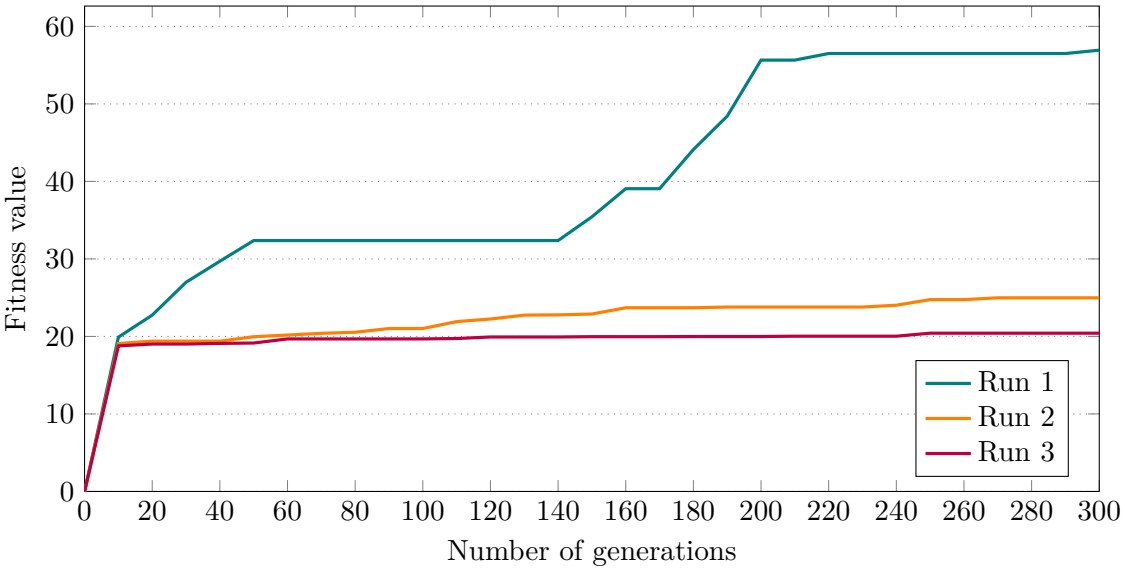

Figure 5.6: Fitness development for the Decapod 1 (moving forward)

The final fitness values for the three initial runs are:
Run 1: 56.93 Run 2: 24.98 Run 3: 20.42

The animation of the two decapod characters are the hardest test cases selected for the implementation. Handling the many legs correctly in order to produce efficient movements calls for a considerable amount of coordination capabilities. A problem that could not be solved completely satisfyingly by the prototypical software examined.

Figure 5.6 shows the outcome of the first runs after trying to do so. Although no really good results could be achieved, it was still possible to achieve a forward movement in every run. The problem was that the solution for a synchronized way of moving the ten legs has never been found and that this lead to the development of an unsatisfying and unsteady rocking motion. The most promising run has been the first one, where the development of an efficient movement slowly came into sight. Several of the legs had already been synchronized and reaching a promising solution might have been possible, with considerably more time.
Since also the additional runs illustrated in Figure A.6 confirmed that no efficient movement type could be evolved within the given frame, it has been decided that this type of character will not be considered during the later assessment of movement quality. While no satisfying animations were achieved during the test runs, this does not mean that the program is generally not able to produce them. Different settings of the parameters, a different fitness function, more time for the evolution process, and other changes might very well lead to the desired results.

Moving forward – Decapod 2 (10 legs, 20 joints)

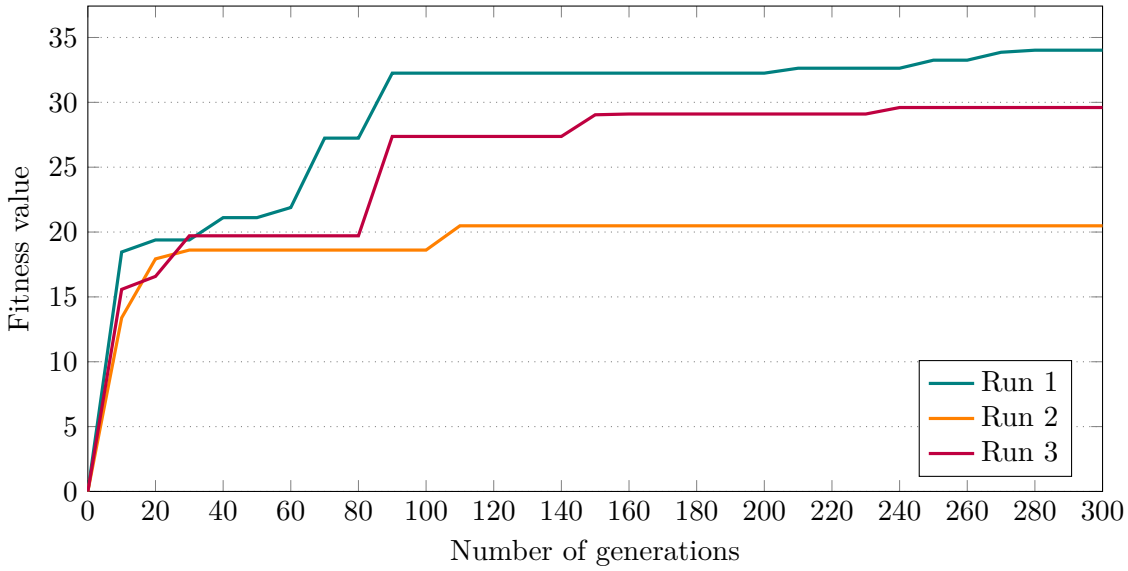

Figure 5.7: Fitness development for the Decapod 2 (moving forward)

The final fitness values for the three initial runs are:
Run 1: 34.02 Run 2: 20.48 Run 3: 29.60

Every problem identified for the first decapod character is even more true for the second one. As it is to be expected, doubling the number of joints and motors did not improve the situation. Instead it became even more difficult for the creature to find an efficient way of moving forward.

Consequently none of the runs depicted in Figure 5.7 and, in extension, Figure A.7 produced a satisfying motion sequence and hence also this character is excluded from the quality assessment phase. Again, there is still a good chance of improving the results by tweaking and improving the chosen settings.

Moving forward – Comparison

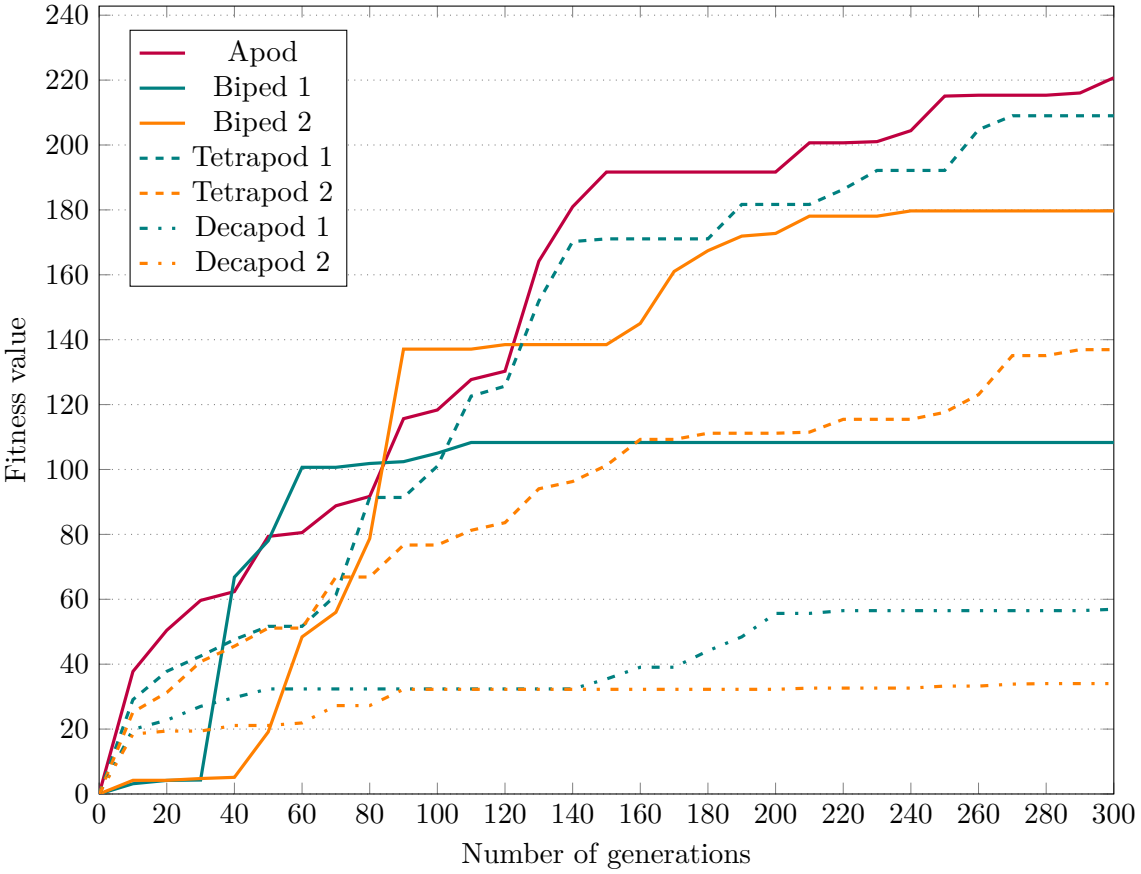

Figure 5.8: Best testing runs for all examined characters (moving forward)

Even though a direct comparison of the fitness of the various characters is only moderately meaningful, due to the different body shapes and abilities, it is interesting to at least take a look at it and see which creatures moved the farthest. Figure 5.8 gives an overview, depicting only the best from the three initial runs for all characters, that allows such a quick glance.
It becomes obvious immediately that the two decapods did not only perform badly in respect to developing a synchronized movement but that their uncoordinated motions also did not carry them very far, in comparison with the other characters. From the five remaining creatures, it is the first biped that achieved the lowest fitness score here. As mentioned, this is caused by its anatomy – the developed movements are actually looking

quite efficient[5]. The curves of the movement sequences of the other characters show that their motions are already on a comparatively high level, with the second tetrapod still having some room for improvement.

Jumping – Apod (0 legs, 6 joints)

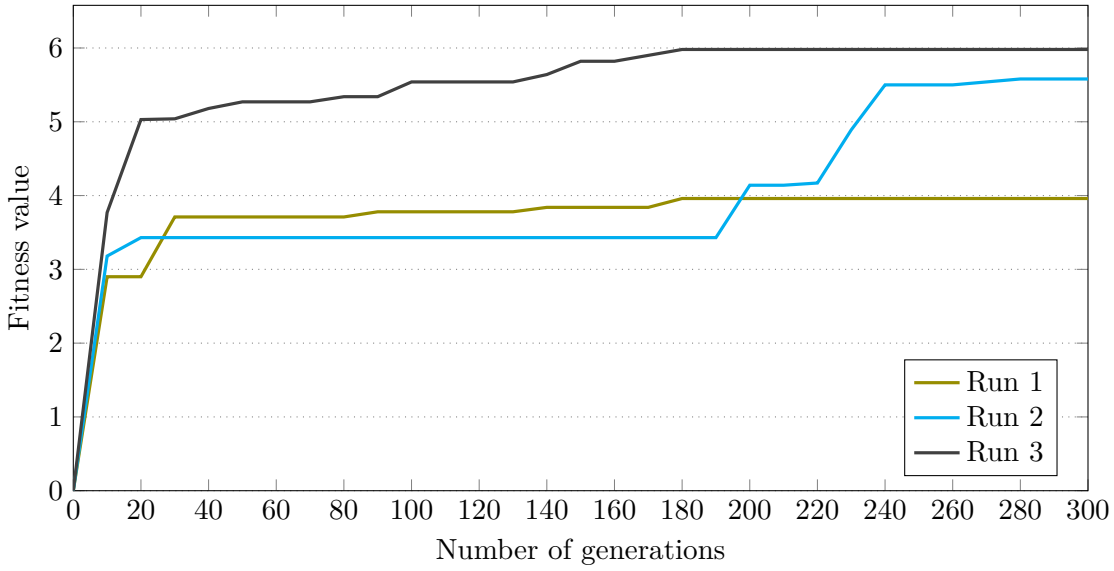

Figure 5.9: Fitness development for the Apod (jumping)

The final fitness values for the three initial runs are:
Run 1: 3.96 Run 2: 5.58 Run 3: 5.98

Compared to the forward movement, the jumping motion is relatively easy to learn. No complicated and lengthy movement sequence has to be discovered and evolved but instead only a few coordinated motor operations are often enough to reach a certain height. However, there is a difference between simply stretching as high as possible and actually performing a jump. The former being quite painlessly achievable often leads to large characteristic increases of fitness at the beginning of the training process.
As in many of the other figures concerned with the movement of jumping, this can be witnessed in Figure 5.9, which is associated with the animation of an apod, also. As the curves suggest, a stretching motion has been quickly achieved with a big performance step within the first 20 generations of evolution. In the runs two and three the character was able to improve this initial motion substantially and catapult itself into the air, by firstly rolling itself up and subsequently rapidly unrolling again. For the first run the creature chose another approach which also resulted in a jumping motion but did not enable it to reach the same height as with using the other technique.

[5]Some curves in Figure A.2 seem to contradict this statement but instead of a walking motion, which is illustrated here, they depict a continuous somersault movement, similar to the one observed for the second biped.

In conclusion, the testing runs showed that the task of finding a jumping animation can be accomplished reliably for the apod character (cf. also Figure A.8).

Jumping – Biped 1 (2 legs, 2 joints)

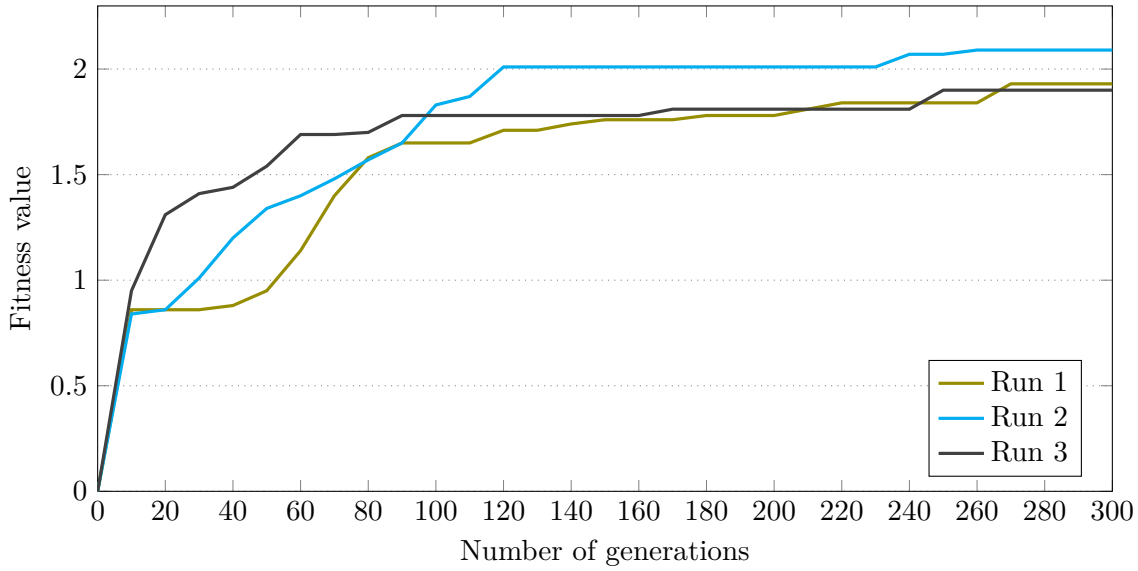

Figure 5.10: Fitness development for the Biped 1 (jumping)

The final fitness values for the three initial runs are:
Run 1: 1.93 Run 2: 2.09 Run 3: 1.90

No big problems occurred while training the first biped character. All three runs illustrated in Figure 5.10 reliably and continuously led to the development of satisfying jumping motions (this is also true for the remaining runs, cf. Figure A.9). The only interesting thing here is that the best evolved movements rotated the character around its own axis, instead of simply launching it upwards straightly.

Jumping – Biped 2 (2 legs, 4 joints)

The final fitness values for the three initial runs are:
Run 1: 2.34 Run 2: 3.12 Run 3: 2.87

Like the first biped, the second one developed a jumping strategy rotating around its center of gravity. Mainly due to its greater height and increased number of motors it has been able to reach a noticeably better fitness.
After the initial stretching phase, it also managed to improve the developed motion reliably in order to produce an efficient style of movement in every single one of the three runs shown in Figure 5.11 – and further the remaining ones in Figure A.10.

Jumping – Tetrapod 1 (4 legs, 4 joints)

The final fitness values for the three initial runs are:
Run 1: 2.12 Run 2: 2.69 Run 3: 2.51

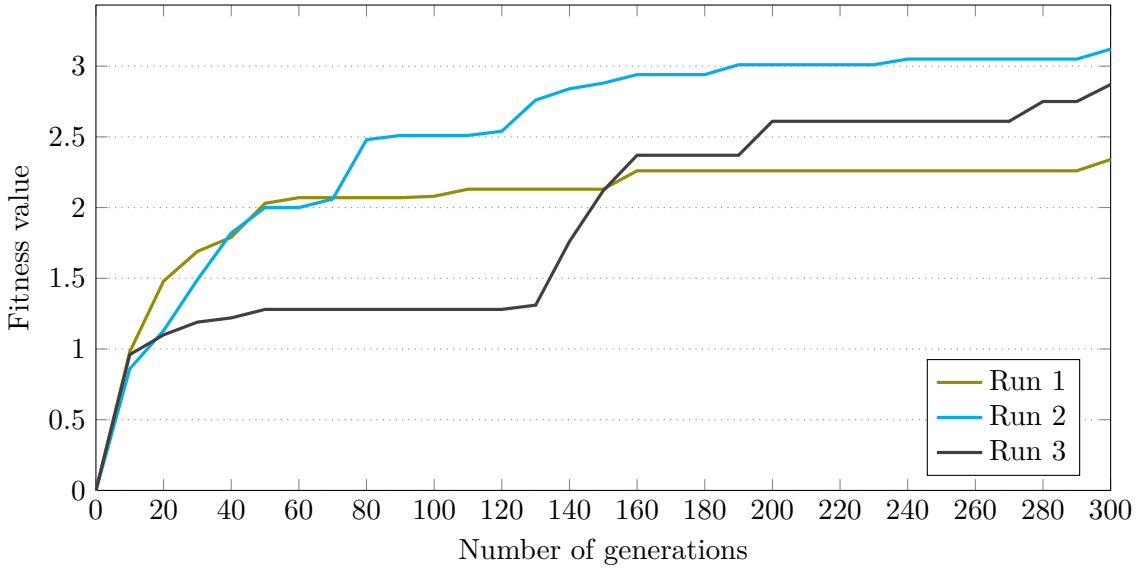

Figure 5.11: Fitness development for the Biped 2 (jumping)

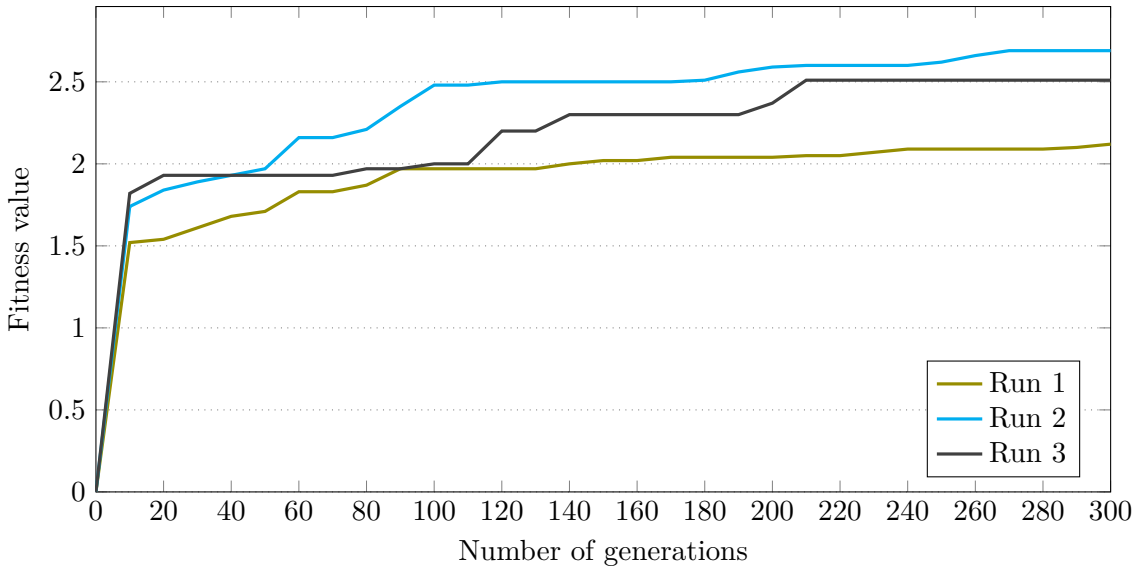

Figure 5.12: Fitness development for the Tetrapod 1 (jumping)

Figure 5.12 indicates that the improvement that can be achieved, after the familiar initial fitness step, is maybe not as high for the first tetrapod as it has been for other characters anymore. The reason behind it can be found justified in the anatomy of the creature once more. The short legs are not able to build up a big amount of action potential, which would be needed to lift the heavy torso high up into the air. Additionally this character lacks the possibility for catapulting itself upwards by using the rotation trick

invented by the bipeds.
The resulting jumps might not be as spectacular as the ones before but the task of lifting the creature into the air as high as possible is fulfilled anyway. The developed jumping motions are efficient within their limitations and they can be produced repeatably (cf. also Figure A.11).

Jumping – Tetrapod 2 (4 legs, 8 joints)

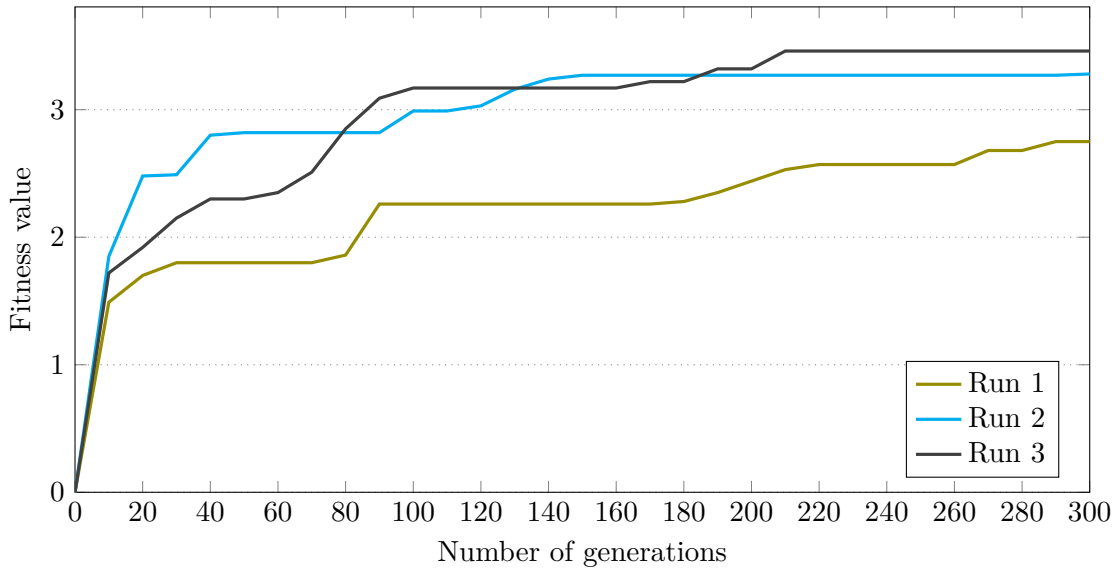

Figure 5.13: Fitness development for the Tetrapod 2 (jumping)

The final fitness values for the three initial runs are:
Run 1: 2.75 Run 2: 3.28 Run 3: 3.46

The second tetrapod successfully uses its extended limbs to develop an efficient jumping motion, which is both forceful and coordinated. At first the character adopts a ducking stance just to jolt up immediately afterwards. Not only is this type of movement effective, it – or at least a similarly efficient one – can also be developed reliably, as witnessed by the curves in Figure 5.13 (and Figure A.12).

Jumping – Decapod 1 (10 legs, 10 joints)

The final fitness values for the three initial runs are:
Run 1: 1.23 Run 2: 1.49 Run 3: 1.17

In contrast to the situation described for the forward movement before, where the two decapods could not even be trained to perform an efficient forward movement, the results look slightly different for the jumping motion. The first decapod was indeed able to lift itself up and even jump a little after the initial stretching phase.
However, although similar results could be produced several times (cf. Figures 5.14 and also A.13) the creature always only used a few of its ten legs for actually performing

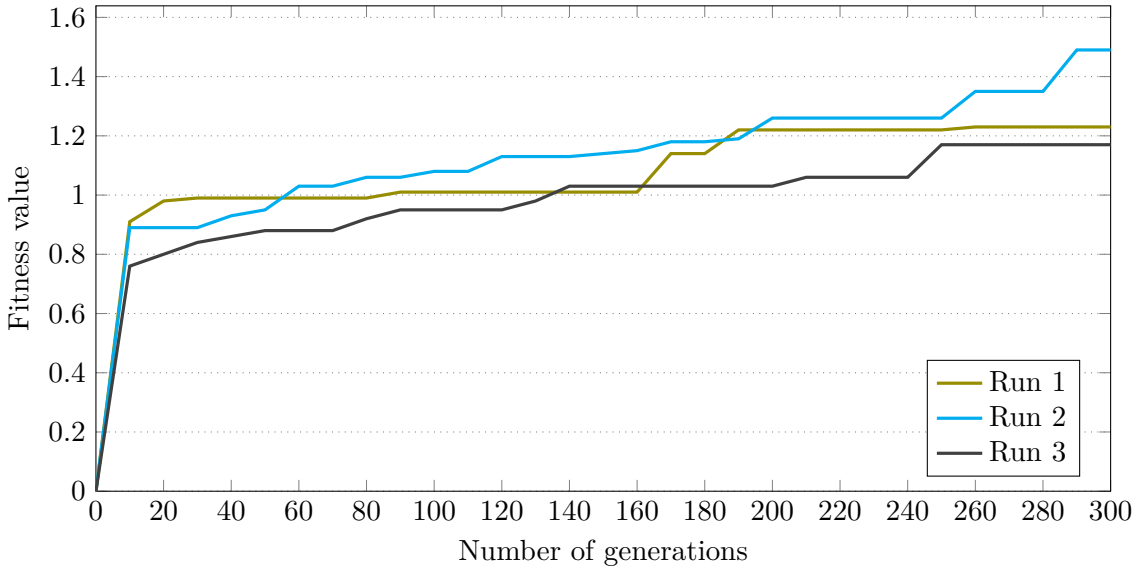

Figure 5.14: Fitness development for the Decapod 1 (jumping)

the movement and activated the remaining ones quite randomly. In all likelihood this behavior would not have led to a positive evaluation during the quality assessment phase. Therefore the results received here do not contradict the decision to remove this character from the list of selected participants for this phase.

Jumping – Decapod 2 (10 legs, 20 joints)

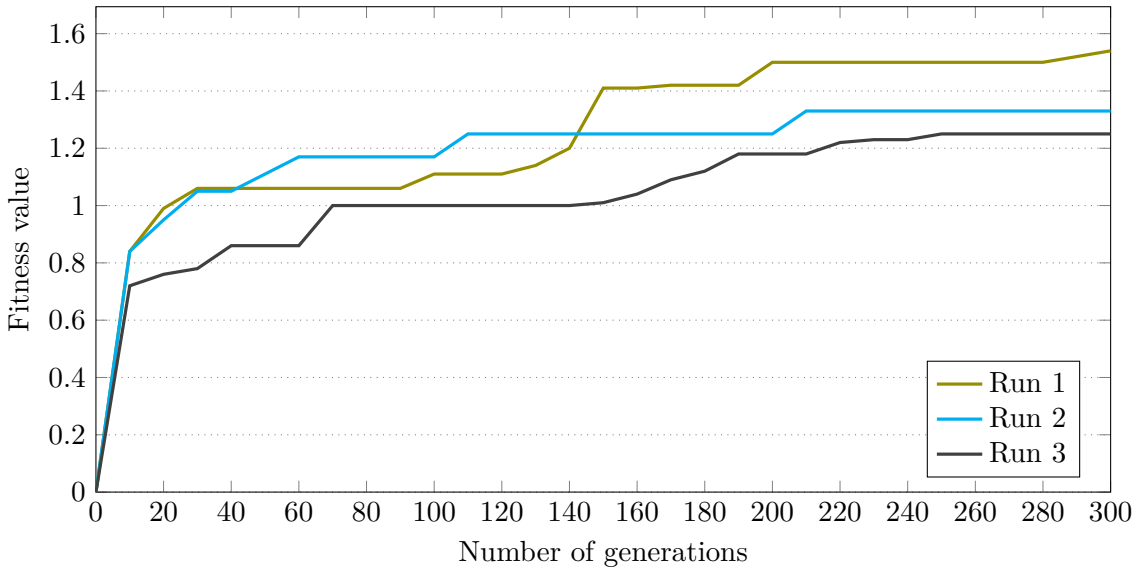

Figure 5.15: Fitness development for the Decapod 2 (jumping)

The final fitness values for the three initial runs are:
Run 1: 1.54 Run 2: 1.33 Run 3: 1.25

The results of the second decapod mirror the ones from the first. Figure 5.15 illustrates that a small jumping movement could be achieved consistently for all three initial runs. But again, the animation of the legs looked rather uncoordinated and unnatural. Such motions are not to be expected to be perceived as appealing by users and as the additional runs (Figure A.14) did not improve the situation, this character can still be ignored safely during the later evaluation phase.

Jumping – Comparison

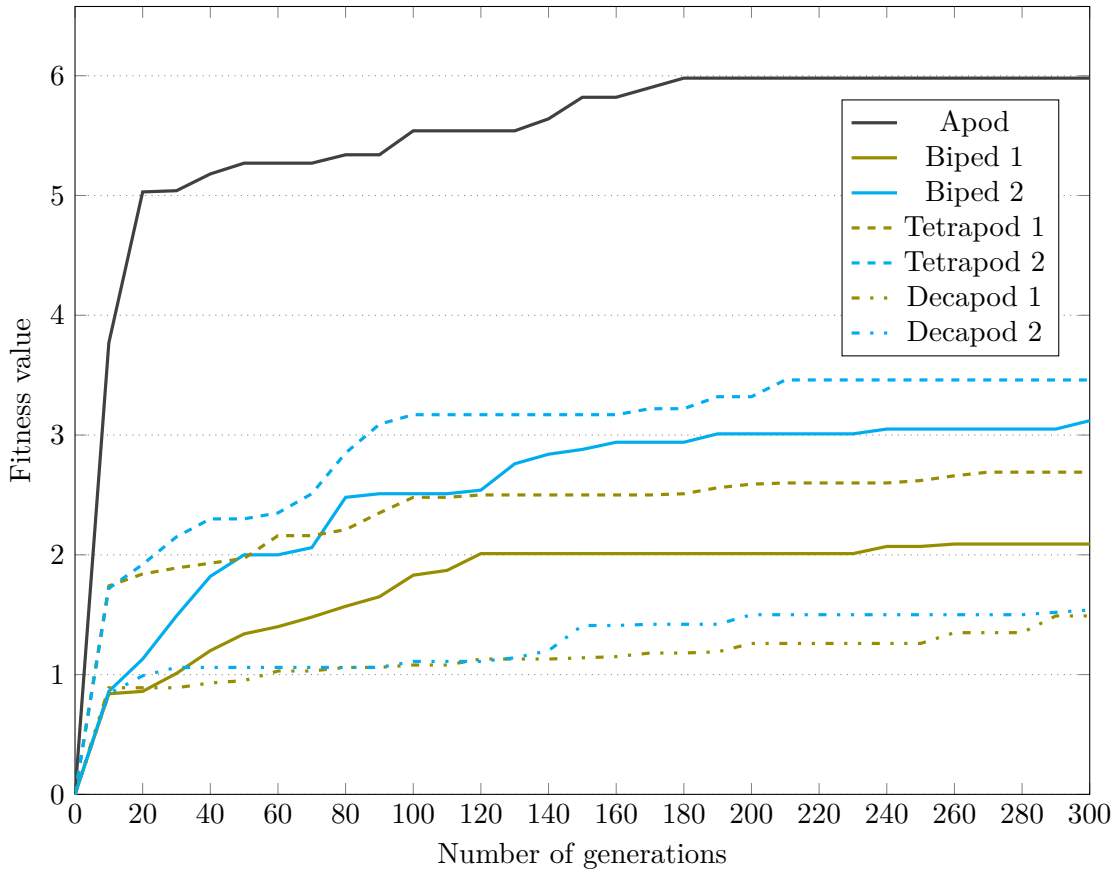

Figure 5.16: Best testing runs for all examined characters (jumping)

Figure 5.16 shows the comparison of the best initial training runs for all seven characters, absolved for developing the jumping motion. Even more so than for the training of the forward movement, the properties of the bodies of the characters are of importance here. Not only did the apod, which is able to twist its body in several places and use this advantage to catapult itself into the air, take the lead unchallenged, also other characters profited from their anatomical advantages.

In each and every case did the creature variant with more joints and motors perform better in this scenario – if only slightly for the decapods. At least the second biped and the second tetrapod managed to utilize their greater movement freedom to develop more sophisticated jumping techniques and a higher fitness on the side.

In general it can be said at this point, that it has been easier for the implemented program to develop efficient jumping motions than forward movements. There have been no total failures while developing these simpler animations and, mainly due to the reduced number of computing cycles necessary, they evolved in considerably less time.

5.2.2 Robustness

The brief comments, concerning the relation of the first three to the later seven runs, that were stated within the discussion of the movement efficiency criterion (Section 5.2.1), already point to a high repeatability of the results. And indeed, while there are certain test cases where single runs deviate noticeably from the others (e.g. Figures A.4 and A.7), often the results are quite similar, particularly for the jumping test cases (Figures A.8 – A.14). But what exactly does this mean?
First and foremost this means that the picture painted by the three initial runs is reflected by the additional seven runs in large part and that the implementation shows a certain robustness by producing usable solutions at a high probability. However, this does not imply that every run will yield the same outcome or that it is ensured that the result will be usable. This is a fact that has already been mentioned and should always be kept in mind. What the outcomes of all the testing runs actually do provide are certain ranges in which new results can reasonably be expected. The fitness scores of all the runs plotted in the Appendix (Figures A.1 – A.14) illustrate these ranges. Again, they are often quite similar for the sets of only three and all ten testing runs.
While it could be argued that ten runs are not enough to statistically secure that the result of a new run will come to lie within a certain range, it can be said that complete confidence in the outcome of a single run is not an important aspect here. For this work's needs the ranges spanned by ten runs per test case are completely sufficient – especially since even the poorer results are most often still good enough. A look at the plots shows that the prototype is able to quickly and reliably produce good animations and that it has been possible to achieve at least one above-average result within the first three runs for every test case. Therefore it should be possible for the users of the implementation to get good and usable movements within only a few runs.
Of course, if the requirements for an implementation are different it might be necessary to further ensure that new results do not deviate too much from the already existing ones. For example, in real time applications it might be important that a usable solution is reliably produced at first try. In this case it would be sensible to conduct a lot more testing runs in order to avoid unpleasant surprises. At the same time it would be a good idea to try to tweak the fitness function in such a way that the fitness range between the best and the worst runs becomes significantly smaller.

Character	Duration	Generations	Best Fitness	Usable
Apod	1′	15	92.37	**yes**
Apod	5′	75	203.51	**yes**
Apod	20′	300	220.73	**yes**
Biped 1	1′	15	19.38	**yes**
Biped 1	5′	75	59.31	**yes**
Biped 1	20′	300	108.32	**yes**
Biped 2	1′	15	3.68	no
Biped 2	5′	75	109.09	**yes**
Biped 2	20′	300	179.69	**yes**
Tetrapod 1	1′	15	26.47	no
Tetrapod 1	5′	75	36.43	**yes**
Tetrapod 1	20′	300	209.01	**yes**
Tetrapod 2	1′	11	19.90	no
Tetrapod 2	5′	55	30.26	no
Tetrapod 2	27′	300	136.93	**yes**
Decapod 1	1′	–	–	–
Decapod 1	5′	–	–	–
Decapod 1	27′	300	56.93	no
Decapod 2	1′	–	–	–
Decapod 2	5′	–	–	–
Decapod 2	58′	300	34.02	no

Table 5.1: Training results for different time frames (moving forward)

5.2.3 Training Duration

Table 5.1 lists the results for the three different test case scenarios concerned with computing time for developing the forward movement animations, which have been identified as interesting in Section 5.1. Table 5.2 does the same for the jumping motion. The tables are ordered by character and for each of them there are three rows that correspond to the test cases with fixed time frames (1 minute and 5 minutes) and the test cases that have been used for getting the movement efficiency before (using a flexible amount of time but the fixed number of 300 generations).

All test runs have been performed on a middle class home computer, running *Windows 10 Pro (64 Bit)* under the following technical specifications:

CPU: *Intel Core i5-4670K 3.4GHz*, GPU: *AMD Radeon HD 7950 3GB*, RAM: *8GB*

Time for moving forward

Next to the first four self-explanatory columns in Table 5.1, the last column indicates if a developed movement seems usable in practice or not. "Usable" here means that the

desired motion is actually executed successfully. While the fitness of the characters is naturally lower after only a few generations than at the end of the full 300-generation period, it can still correspond to satisfying movements.
This is already the case for the first character, the apod. The fitness of 92.37 that has been reached after only one minute of training (and 15 generations that have been computed during that time) stands for a forward movement that is maybe not as refined as the final one but still usable for character animation purposes. Since already the movement that could be evolved during a time frame of one minute is satisfying, it is to be expected that also the animations produced during the 5 minute and the 300 generation testing runs are usable as well; and indeed they are. The last row in the apod table section, like the last row in the section of every other character, contains data on the test runs conducted during the evaluation of the movement efficiency in Section 5.2.1. Because the best fitness values and the usability achieved during these runs have already been presented above[6], the only interesting value in these rows is the training duration. In the case of the apod, it takes 20 minutes to compute 300 generations of the associated ANN population.
The results for the first biped are similar. Again already the movement developed after one minute could fulfill the task of moving forward. As the fitness value suggests, the motion was rather a slow-paced stroll than a run, but a forward movement it was. Again, the training duration for computing 300 generations stayed at the 20 minutes mark established for the apod.
The next two characters, the second biped and the first tetrapod, also managed to complete 300 generations during 20 minutes but failed to develop usable animations during only one minute. Nonetheless, satisfying movements could be found for both creatures within five minutes.
This has not been possible for the second tetrapod anymore. A satisfying movement – the gallop mentioned in the last section – is only to be observed after the full 300 generations. Another premiere here is that the training duration for evolving those 300 generations is longer than 20 minutes.
Because it has not been possible to evolve efficient forward movements during the full length runs for both decapod characters, the shorter time frames were simply ignored for this evaluation. The training duration of 27 minutes for the first decapod is the same as for the second tetrapod and the calculations for the second decapod reached an all time high of 58 minutes.

Time for jumping

There are two big apparent differences between Table 5.1 and Table 5.2, the one that is of relevance here, that can be found. First, the training duration for the animations is much lower. This is easily explained by the already mentioned reduction of computing cycles from 2000, as for moving forward, to 500. Second, although the number of generations that can be calculated quadrupled, it has not been possible to develop usable motions within a single minute. It seems that while the motion sequence that has to be evolved

[6]The best fitness values for the 300 generation runs presented in the Tables 5.1 and 5.2 are based on the three initial runs for every movement efficiency test case.

Character	Duration	Generations	Best Fitness	Usable
Apod	1′	60	4.29	no
Apod	5′	300	5.98	**yes**
Apod	5′	300	5.98	**yes**
Biped 1	1′	60	0.86	no
Biped 1	5′	300	2.09	**yes**
Biped 1	5′	300	2.09	**yes**
Biped 2	1′	60	1.96	no
Biped 2	5′	300	3.12	**yes**
Biped 2	5′	300	3.12	**yes**
Tetrapod 1	1′	60	2.07	no
Tetrapod 1	5′	300	2.69	**yes**
Tetrapod 1	5′	300	2.69	**yes**
Tetrapod 2	1′	37	2.85	no
Tetrapod 2	5′	187	3.21	**yes**
Tetrapod 2	8′	300	3.46	**yes**
Decapod 1	1′	–	–	–
Decapod 1	5′	–	–	–
Decapod 1	8′	300	1.49	no
Decapod 2	1′	–	–	–
Decapod 2	5′	–	–	–
Decapod 2	22′	300	1.54	no

Table 5.2: Training results for different time frames (jumping)

here can be developed with a relatively small amount of computing cycles, it takes a little while to get a body off the ground and actually jumping instead of just stretching. Because the results for all characters are so similar, there are not many details of Table 5.2 that have to be pointed out here. Almost all relevant characters were able to finish their full 300 generation cycles within the five minute time frame. Therefore it has been possible to evolve satisfying movements for all of them. This is also true for the second tetrapod, which needed eight minutes to the finish line but managed to develop a good enough jumping motion in five.
Again the shorter time frames can be ignored for the two decapods. The training durations for completing the full length runs are 8 resp. 22 minutes.

5.2.4 Quality of the Results

The results presented here reflect the opinion on the achieved movement quality of 40 participants, who answered the simple survey questions presented in Section 5.1. There has been no selection process implemented that restricted the applications for the

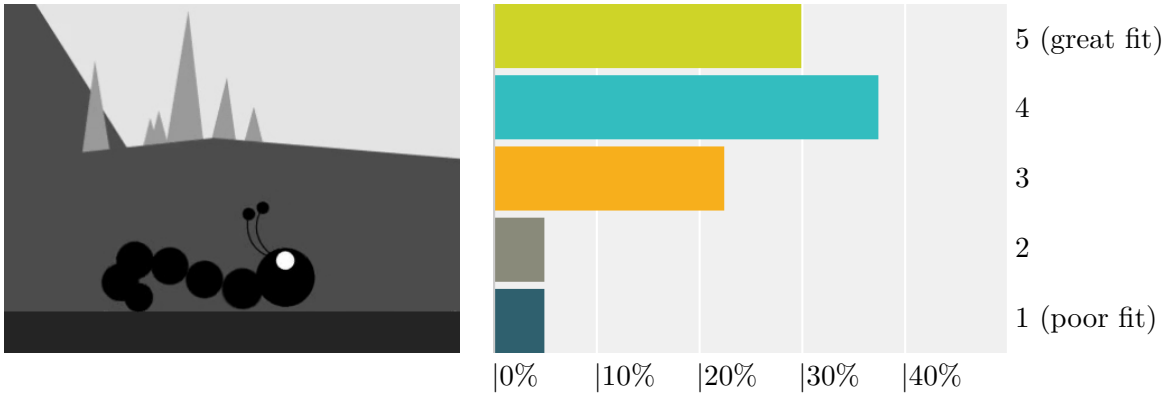

Figure 5.17: The Caterpillar's forward movement and its ratings (Screenshots)

questionnaire. Everybody could participate and indeed this resulted in a quite diverse population of respondents. The participants come from all continents, are between 15 and 59 years old, and both genders are well represented. Additionally their gaming backgrounds span a wide range as well, from persons that do not play at all to gamers that play more than 5 hours daily. Some more detailed statistical results[7] are:

Age
15-19: 7.69% 20-29: 56.41% 30-39: 28.21% 40-49: 5.13% 50-59: 2.56%

Gender
Female: 58.97% Male: 41.03%

Continent
Africa: 2.70% America: 27.02% Asia: 5.41% Australia: 5.41% Europe: 59.46%

Gaming frequency
Never (or almost never): 17.95% Several times a year: 20.51% Several times a month: 10.26% Several times a week: 23.08% Daily (or almost daily): 28.21%

Average gaming duration
<15 minutes: 16.22% <1 hour: 32.43% <2 hours: 32.43% <5 hours: 13.51% 5+ hours: 5.41%

While it would be interesting to analyze the data with a focus on how the different demographic groups answered the questions, this is not of primary interest for this work. The aim here is to get a good first impression of the evolved movements being appealing to many different kinds of persons, with and without gaming experience – a goal that has been reached successfully.

Moving forward – The Caterpillar (Apod)

The average rating assigned as movement quality: 3.825

As usual, the presentation of the test cases starts with the apod. A small detail that might attract attention at this point is that the names of the observed characters have changed.

[7] Only 37 out of 40 persons answered all the corresponding (optional) questions.

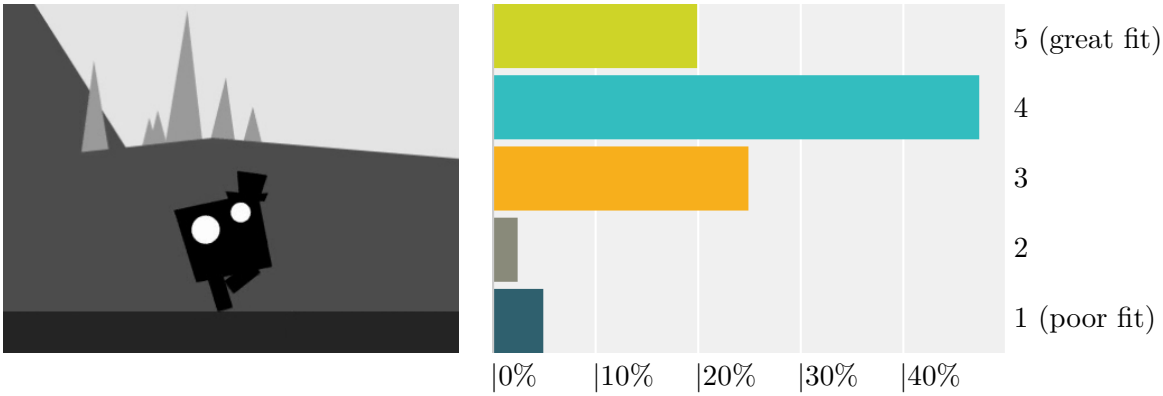

Figure 5.18: The first Gentleman's forward movement and its ratings (Screenshots)

This is not a big deal, the new names are simply the more colorful and descriptive titles that have been chosen for the video presentation and they can be used interchangeably with the old ones.
That said, it is time to look at the first results from the survey. Figure 5.17 shows the apod/caterpillar in a typical pose that occurs during its forward movement efforts and next to it, the ratings associated with the developed sequence. These ratings are generally favorable for this first creature. Approximately two thirds of the participants rewarded the developed forward crawling movement with 4 or 5 points, which corresponds to a good resp. great fit. Only 5%, or two persons, said that the evolved motion does not fit at all (1 point).

Moving forward – The Gentleman 1 (Biped 1)

The average rating assigned as movement quality: 3.75

Not as many people as before found the developed animation to be a great fit for the first biped and as a result the average rating is a little lower. Nonetheless, almost 50% awarded 4 out of 5 points and again about two thirds of the participants deemed the motion to be a good or great fit (cf. Figure 5.18).

Moving forward – The Gentleman 2 (Biped 2)

The average rating assigned as movement quality: 2.90

The innovative flic-flac motion developed by the second biped, which has been described in Section 5.2.1, does not seem to excite the public. In fact this animation achieved the lowest rating of all that are presented for evaluation and it is the only one that received a below-average assessment of its quality.
Figure 5.19 shows that while not even 10% found the results to be a great fit, 15% think that the evolved motion does not fit the character at all. Still, the ratings are not exclusively bad and almost 70% of the people think that the achieved results are okay (3 points) or better.

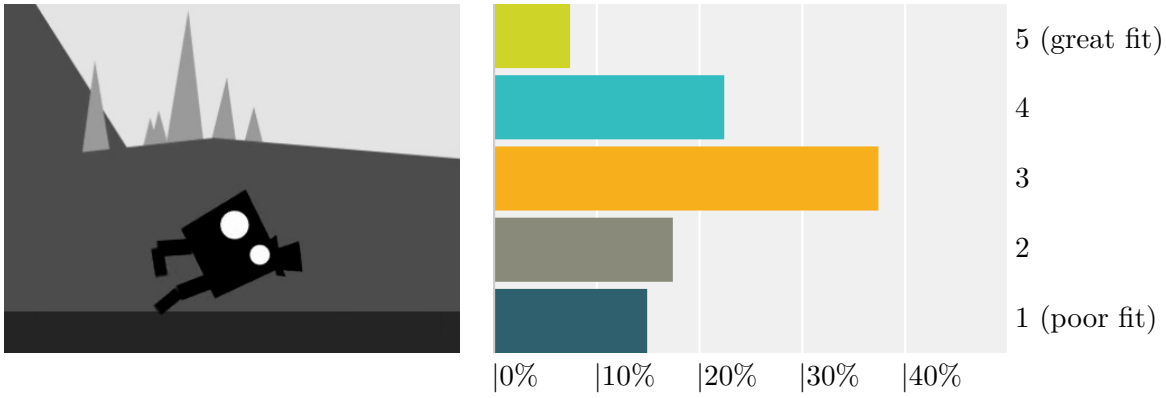

Figure 5.19: The second Gentleman's forward movement and its ratings (Screenshots)

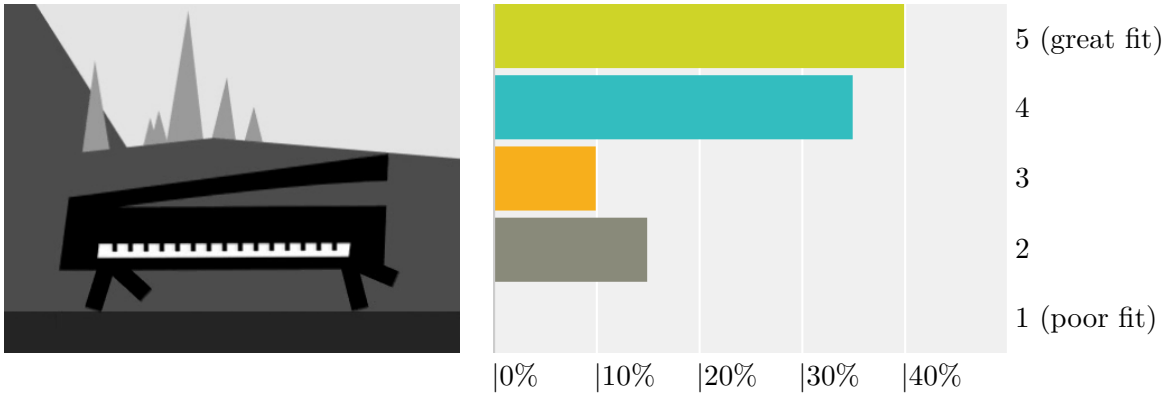

Figure 5.20: The first Piano's forward movement and its ratings (Screenshots)

Moving forward – The Piano 1 (Tetrapod 1)

The average rating assigned as movement quality: 4.00

The first tetrapod, as the last creature's counterpart, is the one with the best average quality assigned. It is also the only one that did not receive a single 1-point rating. The simple, yet effective, movement developed by the character in Figure 5.20 was appreciated by most survey participants and awarded with a high rating of 4 or even 5.

Moving forward – The Piano 2 (Tetrapod 2)

The average rating assigned as movement quality: 3.65

The last test case concerned with the motion type of moving to the right is starring the second piano character, depicted in Figure 5.21. The galloping movement developed by it has been rated as a good or great fit by more than two thirds of the participants. Approximately one sixth of the viewers have not been satisfied with it.

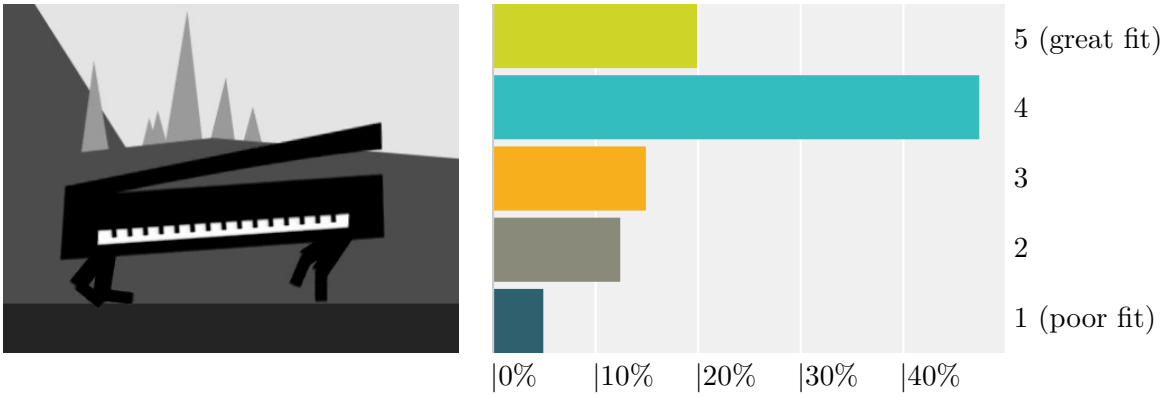

Figure 5.21: The second Piano's forward movement and its ratings (Screenshots)

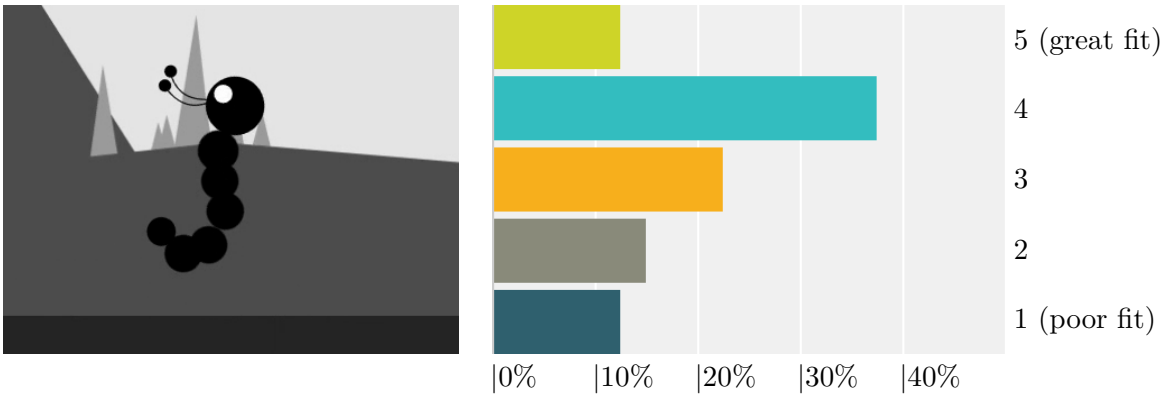

Figure 5.22: The Caterpillar's jumping movement and its ratings (Screenshots)

Jumping – The Caterpillar (Apod)

The average rating assigned as movement quality: 3.225

Out of all jumping test cases, the animation evolved for the apod has received the lowest rating – if not by far. In general, the average ratings are a little lower for the jumping motions than for the forward movements, with the exception of the flic-flac style of the second tetrapod discussed above.
The way the caterpillar illustrated in Figure 5.22 rolled itself up before a jump (again, the motion is described in Section 5.2.1) did not resonate well with everybody. Over 25% of respondents assigned only 1 or 2 points to this movement. However, as with all the other animations, there are also many persons that are quite satisfied with it.

Jumping – The Gentleman 1 (Biped 1)

The average rating assigned as movement quality: 3.475

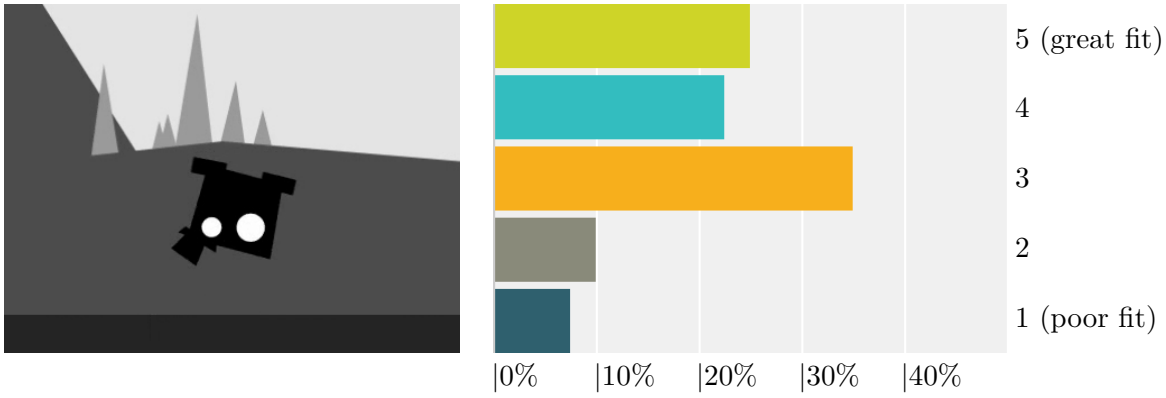

Figure 5.23: The first Gentleman's jumping movement and its ratings (Screenshots)

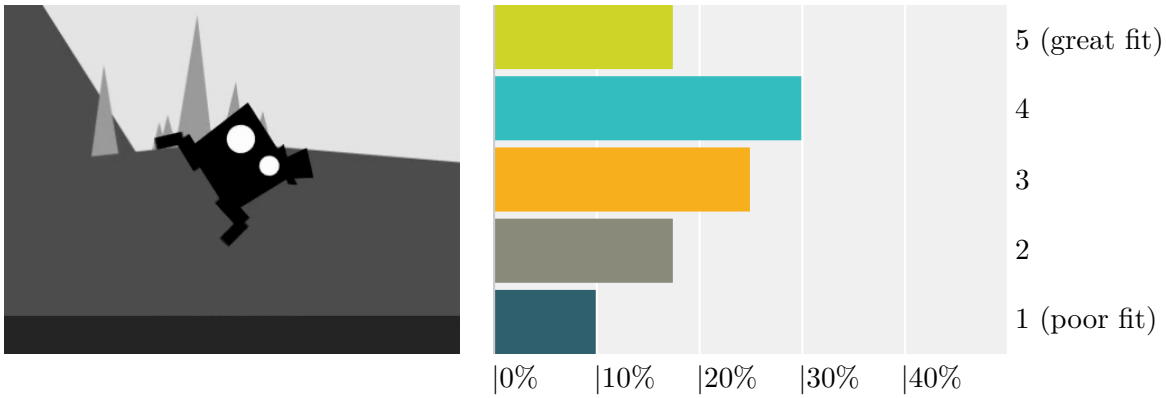

Figure 5.24: The second Gentleman's jumping movement and its ratings (Screenshots)

Although the rotating motions of the second biped have not been appreciated too much for the forward movement, rotations during jumps were received a little better here. Even so, Figure 5.23 shows that again most of the participants rated the resulting animation as average.

Jumping – The Gentleman 2 (Biped 2)

The average rating assigned as movement quality: 3.275

The second biped perfected the art of effectively rotating around ones own axis to perform efficient movements. As far as the jumping motion, depicted in Figure 5.24, is concerned this is rewarded by almost 50% 4 and 5 point ratings.

Jumping – The Piano 1 (Tetrapod 1)

The average rating assigned as movement quality: 3.275

The jumping animation evolved for the first tetrapod has been rated exactly the same as the motion shown by the second biped. Every single bar in the chart in Figure 5.25 mirrors the corresponding one in Figure 5.24.

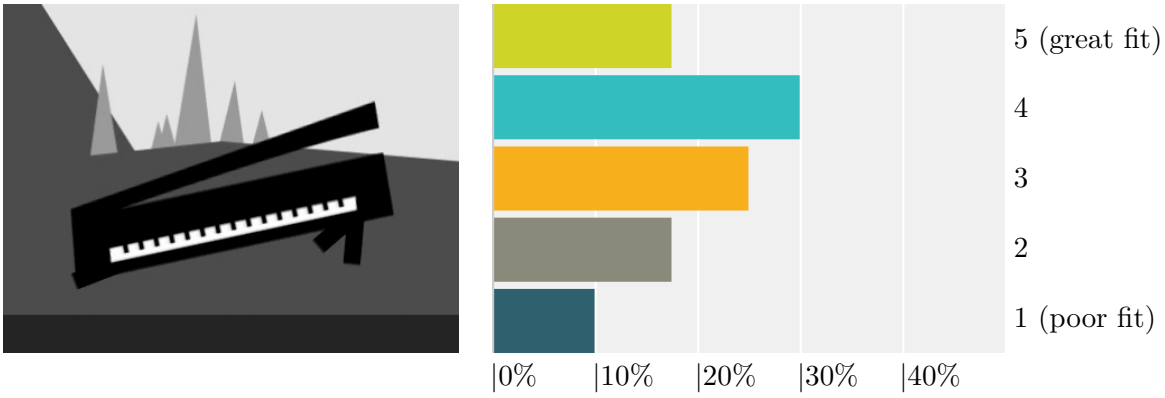

Figure 5.25: The first Piano's jumping movement and its ratings (Screenshots)

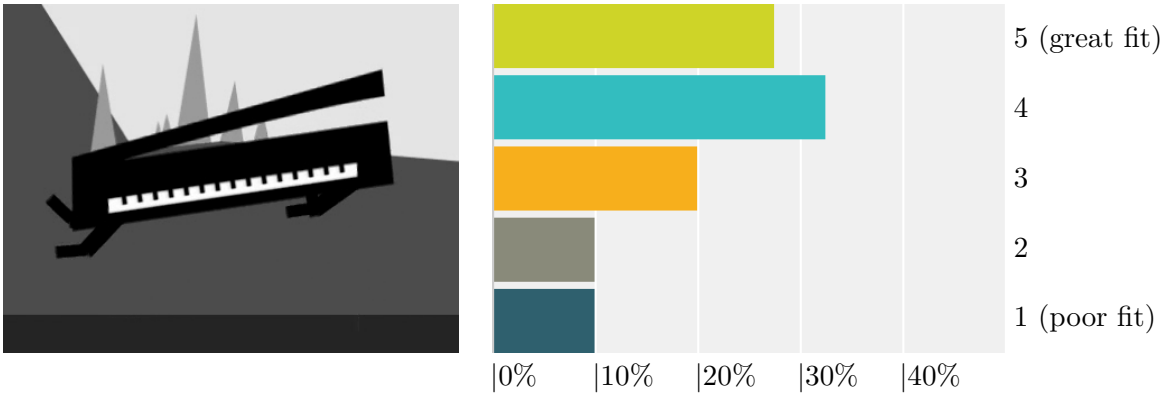

Figure 5.26: The second Piano's jumping movement and its ratings (Screenshots)

This might indicate that indeed the quality of the movements and not their efficiency (in this case the reached height) has been assessed by the participants, since the maximal height reached by the tetrapod is noticeably smaller than the one of the biped and the character still received the same ratings. Although it would be interesting to assess how much people's initial expectations and other factors influence the results, it is only of little relevance here and therefore not explored any further.

Jumping – The Piano 2 (Tetrapod 2)

The average rating assigned as movement quality: 3.575

The final results in the movement category of jumping are also the best ones. The powerful jumps performed by the second tetrapod, shown in Figure 5.26, lead to favorable feedback from the survey participators. More than 25% awarded a perfect 5 point score and another 30% assigned a value of 4 to this motion. As for most of the other animations concerned with jumping, the percentage of people that rejected the developed movement as unfitting stayed around 10%.

5.3 Assessment

Now that the different aspects of the implemented software solution have been analyzed and tested in detail, the time for summing up the findings has come. First the final testing results are shortly recapitulated and considered once more, after that the observations about the limitations of the program and some insights gained during the testing and also the implementation phase are discussed.

5.3.1 Testing Outcome

The overall testing outcome has been satisfactory in large part but some bounds, restrictions, and deficiencies have been identified as well.

It has been possible to develop efficient movements for most characters. Only the two creatures with ten legs and ten or more joints could not be animated satisfyingly in a reasonable amount of time and under the restrictions imposed by the desire for a uniform parameter setting. The terms "efficient" and "satisfyingly" here refer to a character being able to move all its limbs in a coordinated and at least moderately appealing way. In fact all characters, even the two decapods, were able to develop a movement in the desired directions but not all of these movements have been, as said, efficient or satisfying.

Relating to the implementations applicability in real-time scenarios there is a trend to be observed. As it is to be expected, longer animation times lead to better results and the movements of more complex creatures have been more difficult to develop.
While usable motions could be evolved within a 5 minute time frame for all relevant characters and motion types, with the exception of the forward movement of the second tetrapod, this could not be achieved for the 1 minute time span. Only the forward movements for the simplest creatures, the apod and the first biped, have been successfully evolved within that frame.
Also because of a certain unpredictability of the animation quality that will be produced, a completely unrestricted utilization of the implementation for real-time use cases is therefore not possible. This does not mean that the movements produced by it are not usable for real-time purposes at all – but more on that in Chapter 6.

As far as the perceived movement quality is concerned, the results gained during the testing phase are again mostly positive. While there are big differences on how single persons assess the quality of a movement, the average ratings for the characters and movements are largely satisfying. Only the forward movement of the second tetrapod has been rated slightly below 3 points (moderate fit) on average.
Although many people are pleased with the evolved animations anyway, another point helps putting the received quality assessments of those that are not into perspective: Movement quality is not something that is evaluated and criticized by players on a regular basis. As long as a motion is not looking completely off, there is a good chance that most players will not even give a second thought to an animation that is presented to them in a gaming context.

5.3.2 Lessons learned

Apart from the testing outcome and the animations that have been produced by now, some other useful results could be acquired during the course of implementing and testing NE for PBCA as well. Namely these are the insights, the knowledge about the limits of the technique and the implementation, and other observations that might lead to further improvements in the future.

Limits

A few limitations have already been identified during the testing phase but there are others too. Some of them are technological restrictions, some are conceptual, and some more limitations originate simply in design decisions.

The animation time, to give an important example, is mainly limited by technology, but not exclusively. While it is true that a faster computer would be able to produce shorter time frames, there are other factors that impact the performance as well. A closer look at the Tables 5.1 and 5.2 reveals that the training times stay the same for the first four characters. What does this mean?
Well, first of all it is important to know that all training procedures have not been performed in real-time but in a time lapse mode. All physical calculations were performed ten times faster. For relatively complex characters it is pointless to increase the speed much further because the computer is not able to keep up anyway, but the less sophisticated a creature gets the higher the time lapse multiplier can be set without running into performance issues. For the simplest character, the first biped that consists of only two legs and two joints, the value of the multiplier can be set as high as 30 in order to get the fastest results on the test machine. This leads to a considerable performance boost from 15 to 40 generations that can be simulated within the time frame of 1 minute and consequently considerably better animations.
Another way of decreasing the time required for evolving movements would be the parallelization of the evolution. The task at hand is well suited to being distributed among several processing units. Not only would the needed animation time drop significantly but potentially also the quality of the motions could be increased at the same time, because more individuals could be simulated per generation without drastically prolonging the process of doing so[8].

Other major limitations are concerned with the complexity of the characters that have to be animated and the types of animation that can be created. The first one can be mitigated by using different training parameters, more sophisticated ANNs, and longer training durations, as mentioned before. The second one is a more fundamental one.
Since not all details of an animation can be designed by hand or predicted before the animation has been actually produced, PBCA is not suited for handling all animation challenges.

[8]This would require a high number of processing units, as it can be found in modern graphics cards, that corresponds to the number of individuals per generation.

A last relevant restriction that shall be discussed here is the need for being able to describe the animation that should be produced via a fitness function. Even if a NE implementation would be able to produce the desired motion in theory, it is still possible that a developer is not able to find the right fitness function to describe to the program how it is to be searched for.
To give a simple example of when the change of the function could be necessary: on the one hand, the flic-flac forward movement developed during the test runs of the second biped is a very efficient way of moving to the right, on the other hand it has received not so favorable ratings by the participants of the conducted survey[9]. This leads to the desire for changing the fitness function that motivates the development of this motion originally. One way of forcing the character to avoid rotating while moving is to penalize each rollover heavily.

Insights

Next to finding possible solutions for some of the limitations that have been discussed above, there are also several other insights that could be gained. A few of the most interesting topics are:

- *Avoidance of local optima.* Running into a local optimum is one of the most common things that hinder the development of an appealing movement. Randomizing certain aspects of the search or allowing larger mutations, once the evolution of a population slowed down significantly, can help in mitigating this issue.
- *Overfitting a solution.* It might sound strange that an evolved movement can get too good but in fact the problem of overfitting, known from many other ML scenarios, is also an issue here. If the training goes on for too long, it can happen that the developed motions are perfectly suited to the training situation but do not work under different other settings anymore. In contrast to robust motions that withstand changes in the environment, even little alterations, sometimes even the change of the time scale, can break an overfitted animation completely.
- *Adjustments on the character level.* While overfitting a solution can be problematic, a good fit of the parameters and settings on a per-animation basis can be beneficial. Adjusting all relevant details (e.g. the angular restrictions for the joints or the mutation rate) can help in finding better solutions, or finding solutions of equal quality within a shorter amount of time. The drawback of using this approach is, of course, the reduction of flexibility.
- *Inputs and their representation.* The need for a careful selection of input parameters became obvious through the problems encountered around this subject during the implementation phase (cf. Section 4.2.2). That means that the number of inputs, the range of values for the weights, and the range of values allowed for the inputs themselves matter. They have to play well with the chosen activation function.

[9] Again, it would be interesting but is out of the scope of this work to analyze how people's expectations and preferences influence the ratings.

- *Choice of the activation function.* The choice of an activation function impacts, similar to the choice of the inputs representation, how animations are evolved. Different functions can be suited to different tasks and the clever choice of a simplified activation function might even lead to better performance, considering that it is executed once for every neuron of every character in every computing cycle. For instance, the hyperbolic tangent activation function (*tanh(a)*) chosen in Section 4.2.2 can be substituted by a computationally less expensive function (*f(a)*), that looks almost the same but gets along without using exponents:

$$tanh(a) = \frac{1 - e^{-2a}}{1 + e^{-2a}} \qquad \text{vs.} \qquad f(a) = \frac{a * (27 + a * a)}{27 + 9 * a * a}$$

- *Physics preferences.* The settings used for the physics engine, that works behind the scenes of the animation process, also play a big role in how a motion evolves. Many details, like the scale of the limbs, the forces used on the joints and motors, and also the gravity scale, impact the development greatly and it is of high importance that the according values are chosen reasonably and everything works together well.

CHAPTER 6

Discussion and Conclusion

Automated content creation for computer games is a research area that promises to provide benefits for many situations. Professional large-scale productions as well as smaller indie projects can profit from the utilization of procedurally generated landscapes, textures, buildings, and other types of assets. A type of content that has not been very much in the focus of procedural content generation (PCG) research, is the active animation of physics-based gaming characters. This work embarks upon the journey on changing that situation.
The machine learning technique of neuroevolution (NE), a method combining evolutionary algorithms and artificial neural networks (ANNs) that can be used for evolving motion controllers, has been found to be potentially well suited for approaching the tasks at hand.

The main goal identified for this thesis is the design and development of a prototypical software solution, that allows for the successful and automated simulation of appealing motions for various 2D characters. Some more goals are to reach a high flexibility and ease of use of the program, as well as to achieve low animation times. Additionally, the desires for suggesting NE as a viable approach for PCG and an evaluation of the implementation's suitability for real-time application have been formulated in the introduction.

6.1 Final Outcome

After designing a solution and implementing it subsequently, extensive testing and evaluation of the resulting software lead to the following conclusions – organized along the objectives mentioned above:

- *Automated physics-based character animation (PBCA).* It is safe to say that the main goal of the work has been met successfully to a great extent. It has been shown that different types of movements can be produced by the implemented software for several types of characters. The results presented in Section 5.2 illustrate that many

of the evolved animations are not only functional but that they are found to be appealing by many different people from different areas and gaming backgrounds. Although it is true that most characters could be animated satisfyingly, it is also the case that there have been problems with some. Especially the more complicated creatures proved difficult to animate, under the restraint of finding a single parameter setting that shall be used for all of them. However, this issue can be mitigated by dropping the premise of uniformity and fitting the settings more tightly to the single characters.

- *Flexibility and ease of use.* The implemented solution is easy to handle for designers and artists alike. All the basic functionality is accessible without the need for reprogramming parts of the software. The characters that shall be animated can be designed in the editor provided by Unity and all parameters are passed to the program via a simple input form. This allows users without programming skills to animate characters independently, meaning without support of a programmer.
  The high level of usability does not come at the cost of reduced flexibility. First, it is possible to switch the training process between already created characters and types of animations, add new creatures, and change existing ones at any time. Second, it is relatively easy to introduce new types of motions that shall be trained (e.g. swimming or climbing), different environments, and new ways for the characters to interact with the surroundings. While the first tasks, the ones that are directly related to animating various characters, can be managed without altering the program code, the second involve adapting the software to some extent. New types of movements that are to be trained require new fitness functions that describe what exactly should be rewarded. Introducing new environments would most probably result in different sensory and actuatory requirements, to allow the creatures to properly interact with it.
  Another property of the solution that ensures a flexible and easy usage is that all animations are being stored in simple data files, which can be accessed and loaded into a character's motion controller (its brain) without integrating the whole program developed here. Instead only a very lightweight, separable part of the implemented software can be utilized to transform the stored information into movement commands that are usable in a gaming environment.
- *Performance.* The production of animations for real-time usage looks most promising for rather simple characters. The more complex a creature's anatomy, the more sensors and actuators connecting it to the outside world it has, the more time is necessary for the development of usable movements. In addition, if only a single run is performed in real-time to produce a suitable motion, the chance that an unsatisfying animation evolves is relatively high. Under these aspects, a question that arises and that has already been indicated in Section 5.3 is whether the approach of using NE for PBCA in real-time is reasonable.
  The answer that can be given at this point is: "It depends.". It depends mainly on the complexity of the character that shall be animated, on the efficiency of the implementation used for training purposes, and on the specific requirements imposed

on the developed movements. The influence of the complexity of a creature on the animation time has been outlined above and the efficiency of the implementation developed here and ways to increase it are already examined in Section 5.3. This leaves the last point, the identification of requirements, to be discussed here.
In a scenario where all movements have to look a certain way and only small deviations can be tolerated, like the animation of a player's character in a realistic setting, the sometimes unpredictable motions that can be evolved via NE are not very well suited for the job. However, in such a situation it might not be the best idea to try to create animations on the fly anyway. On the other hand, if the important point is that a creature develops an animation that fits the character to some extent and that allows it to quickly start moving around, then the combination of PBCA and NE constitutes a promising approach.
Especially interesting here is that the process of evolving a motion itself could be of interest to the players as well and that it might be a concept worth exploring to let them watch and actively influence the evolution in real-time. This claim is supported by the answers to the last question related to NE asked in the survey (cf. Section 5.1). Almost 40% of the participants awarded 5 out of 5 points to the idea of creating and automatically evolving their own creatures in a game. Another 30% gave a rating of 4 points and only about 16% gave a rating lower than 3 points[1].

Real-time performance aside, it is to be stated that the overall animation times have been quite satisfactory and that almost all training results have been achieved in less than 30 minutes – many in as little as 5. Achieving high quality animations, without any specialized hardware equipment, in such a short time frame ensures the technique's and also the implementation's usability in practice.

- *NE for content generation.* Shortly summing up and elaborating on the conclusions from above, the combination of NE and PBCA has been identified as a promising approach to mitigating the problem of high costs and efforts for content generation in the area of animation. While there are still many challenges that have to be faced in order for the technique to be usable under certain circumstances, there are other situations where it could be used successfully even today.

Apart from these conclusions, regarding the goals for this thesis, and the results and insights gained in Chapter 5, the final outcome of this work in terms of software development for using NE for PBCA is the following: a highly customizable prototypical application that already allows the procedural generation of animations for various characters and constitutes a fitting starting point for further endeavors in this research area.

[1]The average rating is 3.84 points – a relatively high value, considering that more than 38% of the survey's participants said that they play computer games only several times per year or even never.

6.2 Future Work and Improvements

Next to the possibilities for approaching the limitations of the current prototypical implementation that have been identified and discussed before (cf. Section 5.3), there are a few other starting points for improvements and extensions that shall be addressed here.

The most promising and important point is, of course, the improvement of the NE part of the solution itself. To build upon the good results that have been achieved for relatively uncomplicated 2D characters in a simple environment during this work, it is a good idea to look at ways of allowing the software to deal with more complex situations.
A sensible first step is to revisit the different types of NE that have been presented in Section 2.2.1. Especially the NEAT[2] approach, that has already been considered for utilization here but has been excelled by the benefits provided by conventional NE for the needs of this work, is of great interest. The more complicated a problem gets and the more difficult the search for a good ANN structure for handling it becomes, the more promising it seems to use NEAT for handling the task at hand. It could also be of interest to develop a useful network topology for a certain class of characters with the help of NEAT, and later on train the creatures using conventional NE in combination with the previously optimized ANN topology to decrease the training duration.
Another strategy, that can also be combined with using a different type of NE, is to use a technique called *shaping* or *incremental evolution*, which has been introduced by Gomez and Miikkulainen [38] more than 20 years ago. The idea of incremental evolution is to start evolving a solution on a simple task and to increase the difficulty once this simple job has been completed successfully. For example, once the characters that have been examined here developed a satisfying forward movement, they could be equipped with some additional visual sensors in order to allow them to learn to jump over holes that start appearing in the ground. Alternatively, or additionally, they could be trained to use less and less energy once they learned how to run, by changing their fitness function. Many other incremental learning tasks are conceivable as well and, as it also became apparent during the testing phase, it is often easier to improve on an existing solution than to come up with a completely new one.
Apart from using specialized methods for increasing the performance of NE, it can also be fruitful to look for some further optimization techniques in the area of artificial intelligence and other fields as well, to find inspirations for possible future improvements.

Besides concentrating on the NE part of the PBCA process, the character models and the environment can also be the targeted for future improvements. If the motion controllers are sophisticated enough to cope with more complex physical settings, an increased level of realism in this areas can lead to better animation results as well. Much work that is concerned with PBCA already focuses on the lifelike animation of 3D character models (cf. [32]) and of course, once the basics of developing fast and appealing animations for arbitrary 2D characters in almost any situation have been mastered, increasing the degree of realism is a next logical development step.

[2] NEAT is an acronym for NeuroEvolution of Augmented Topologies (cf. [16]).

However, as animating more complex characters comes at the price of higher computational costs and longer animations times, the production of realistic animations for creatures, that can be carried out in real-time within gaming sessions, still appears a long way off. After all, this has been one of the main reasons for focusing on 2D animations for this work in the first place.

Finally it might be worth exploring if the software solution produced during this work can be extended and put to good use in other areas as well. The combination of a NE concept and its implementation using a popular game engine seems to be interesting for many different scenarios.
Naturally the program's usage in areas of game development besides PBCA comes into mind first, and it is easily conceivable to envision its application for other PCG tasks. For instance, it is indeed possible to not only evolve animations for characters but also the characters themselves. Even a co-evolution of both, a creature's morphology and its movements, can be imagined – although this can be a rather complicated process, as demonstrated by Cheney et al. [31]. In addition many other jobs, some of which are listed in Table 2.1, could be fulfilled potentially.
Other fields of application can be academia or certain industries. The limiting factors, really, are the complexity of the task that is to be accomplished and the ability of the software engineer in charge to properly fit the program to it. Among other things, this means: developing a good fitness function, finding suitable inputs and outputs that match the problem, choosing the right learning strategy, and maybe even finding a way of integrating the implementation into an already existing system. In respect to the last point it would be a big advantage if the surrounding environment is also implemented in Unity, but an integration as a plugin to an existing platform, like the DLVHEX framework [39] developed at the Knowledge-Based Systems Group[3] of the Vienna University of Technology (cf. [40]), is also imaginable.

[3]This is the group in the Institute of Information Systems where this thesis has been created and supervised.

References

[1] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, 2015.

[2] T. Geijtenbeek and N. Pronost, "Interactive character animation using simulated physics: A state-of-the-art review," in *Computer Graphics Forum*, vol. 31(8), pp. 2492–2515, Blackwell Publishing Ltd., 2012.

[3] M. Van de Panne and E. Fiume, "Sensor-actuator networks," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 335–342, ACM, 1993.

[4] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: Goals, challenges and actionable steps," in *Dagstuhl Follow-Ups*, vol. 6, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2013.

[5] Unity Technologies, "Unity." `http://unity3d.com`. Online Resource, last accessed: 20. June 2017.

[6] T. M. Mitchell, *The Discipline of Machine Learning*, vol. 9. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[7] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg, J. M. Farfel, R. E. Ferretti, R. E. Leite, W. J. Filho, R. Lent, and S. Herculano-Houzel, "Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain," *Journal of Comparative Neurology*, vol. 513, no. 5, pp. 532–541, 2009.

[8] D. Kriesel, *A Brief Introduction to Neural Networks*. Available at `http://www.dkriesel.com`, 2007. Last accessed: 20. June 2017.

[9] Chrislb (Wikimedia Commons), "Artificial neural networks/activation functions." `https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Activation_Functions`. Online Resource, last accessed: 20. June 2017.

[10] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2 ed., 2015.

[11] R. Parent, *Computer Animation: Algorithms and Techniques*. Newnes, 3 ed., 2012.

[12] J. Togelius, E. Kastbjerg, D. Schedl, and G. N. Yannakakis, "What is procedural content generation?: Mario on the borderline," in *Proceedings of the 2nd Workshop on Procedural Content Generation in Games*, 2011.

[13] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, p. 1, 2013.

[14] G. Ochoa, "On genetic algorithms and lindenmayer systems," in *Parallel Problem Solving from Nature - PPSN*, no. V, pp. 335–344, Springer, 1998.

[15] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: From architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, 2008.

[16] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

[17] J. Fekiač, I. Zelinka, and J. C. Burguillo, "A review of methods for encoding neural network topologies in evolutionary computation," in *Proceedings of the 25th European Conference on Modeling and Simulation ECMS 2011*, pp. 410–416, 2011.

[18] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Accelerated neural evolution through cooperatively coevolved synapses," *Journal of Machine Learning Research*, vol. 9, no. May, pp. 937–965, 2008.

[19] D. E. Moriarty and R. Miikkulainen, "Evolving obstacle avoidance behavior in a robot arm," in *Proceedings of the 4th International Conference on Simulation of Adaptive Behavior*, pp. 468–475, MIT Press Cambridge, MA, 1996.

[20] F. J. Gomez and R. Miikkulainen, "Active guidance for a finless rocket using neuroevolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, (San Francisco), pp. 2084–2095, Morgan Kaufmann, 2003.

[21] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez, "Training recurrent networks by Evolino," *Neural Computation*, vol. 19, no. 3, pp. 757–779, 2007.

[22] S. Risi and J. Togelius, "Neuroevolution in games: State of the art and open challenges," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, no. 99, 2015.

[23] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic Programming and Evolvable Machines*, vol. 8, no. 2, pp. 131–162, 2007.

[24] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[25] B. F. Allen and P. Faloutsos, "Evolved controllers for simulated locomotion," in *International Workshop on Motion in Games*, pp. 219–230, Springer, 2009.

[26] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen, "Flexible muscle-based locomotion for bipedal creatures," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, p. 206, 2013.

[27] C. Hecker, B. Raabe, R. W. Enslow, J. DeWeese, J. Maynard, and K. van Prooijen, "Real-time motion retargeting to highly varied user-created morphologies," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3, p. 27, 2008.

[28] S. Coros, A. Karpathy, B. Jones, L. Reveret, and M. Van De Panne, "Locomotion skills for simulated quadrupeds," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 4, p. 59, 2011.

[29] J. Tan, Y. Gu, G. Turk, and C. K. Liu, "Articulated swimming creatures," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 4, p. 58, 2011.

[30] J. Tan, Y. Gu, C. K. Liu, and G. Turk, "Learning bicycle stunts," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 50, 2014.

[31] N. Cheney, J. Bongard, V. SunSpiral, and H. Lipson, "On the difficulty of co-optimizing morphology and control in evolved virtual creatures," in *Proceedings of the Artificial Life Conference*, pp. 226–234, 2016.

[32] A. L. C. Ruiz, C. Pontonnier, N. Pronost, and G. Dumont, "Muscle-based control for character animation," in *Computer Graphics Forum*, 2016.

[33] NVIDIA Corporation, "PhysX." `https://developer.nvidia.com/physx-sdk`. Online Resource, last accessed: 20. June 2017.

[34] E. Catto, "Box2D." `http://box2d.org/`. Online Resource, last accessed: 20. June 2017.

[35] D. I. Jallov, "Evolve - introducing a novel game mechanic based on the indirect control of evolving neural networks," 2014.

[36] D. I. Jallov, S. Risi, and J. Togelius, "EvoCommander: A novel game based on evolving and switching between artificial brains," *IEEE Transactions on Computational Intelligence and AI in Games*, 2016.

[37] Geek3 (Wikimedia Commons), "Hyperbolic tangent function." `https://commons.wikimedia.org/wiki/File:Mplwp_tanh.svg`. Online Resource, last accessed: 20. June 2017.

[38] F. Gomez and R. Miikkulainen, "Incremental evolution of complex general behavior," *Adaptive Behavior*, vol. 5, no. 3-4, pp. 317–342, 1997.

[39] Vienna University of Technology, "DLVHEX." http://www.kr.tuwien.ac.at/research/systems/dlvhex. Online Resource, last accessed: 20. June 2017.

[40] T. Eiter, C. Redl, and P. Schüller, "Problem solving using the HEX family," in *Computational Models of Rationality*, pp. 150–174, 2016.

Appendix

This section contains 14 figures that are connected to the 14 different test cases presented in Section 5.1. They provide some additional information on how the evolution of fitness values progressed for the various characters and movement types. Often they are referenced relating to the test criteria of "movement efficiency" in Section 5.2.1 and "robustness" in Section 5.2.2.

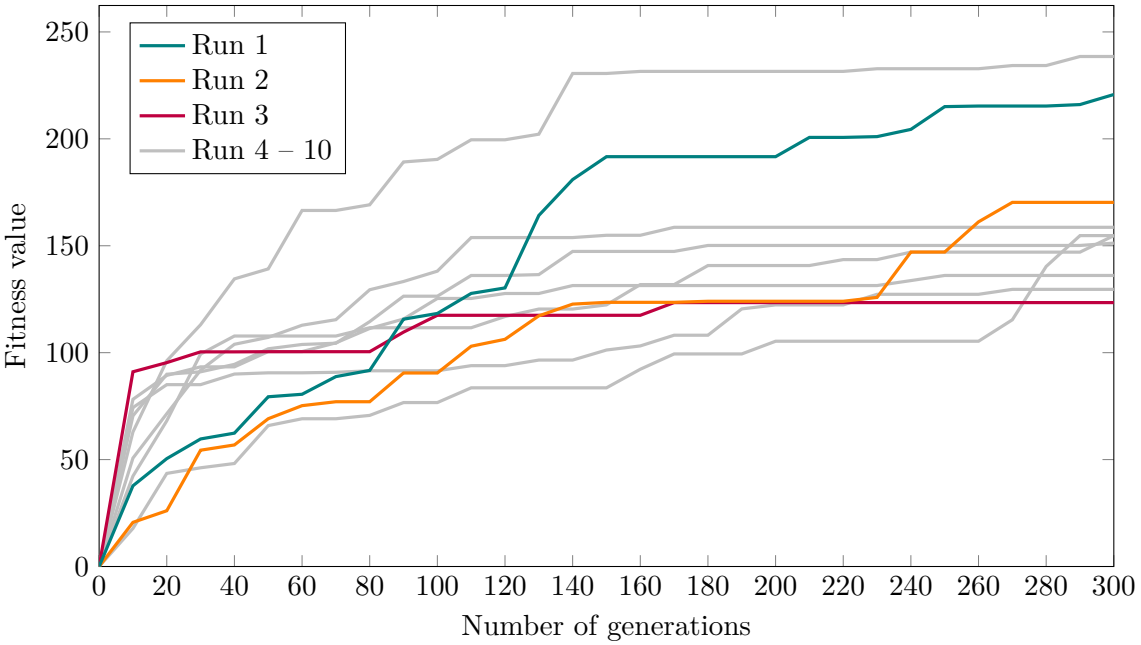

Figure A.1: Evolutionary comparison for the Apod (moving forward)

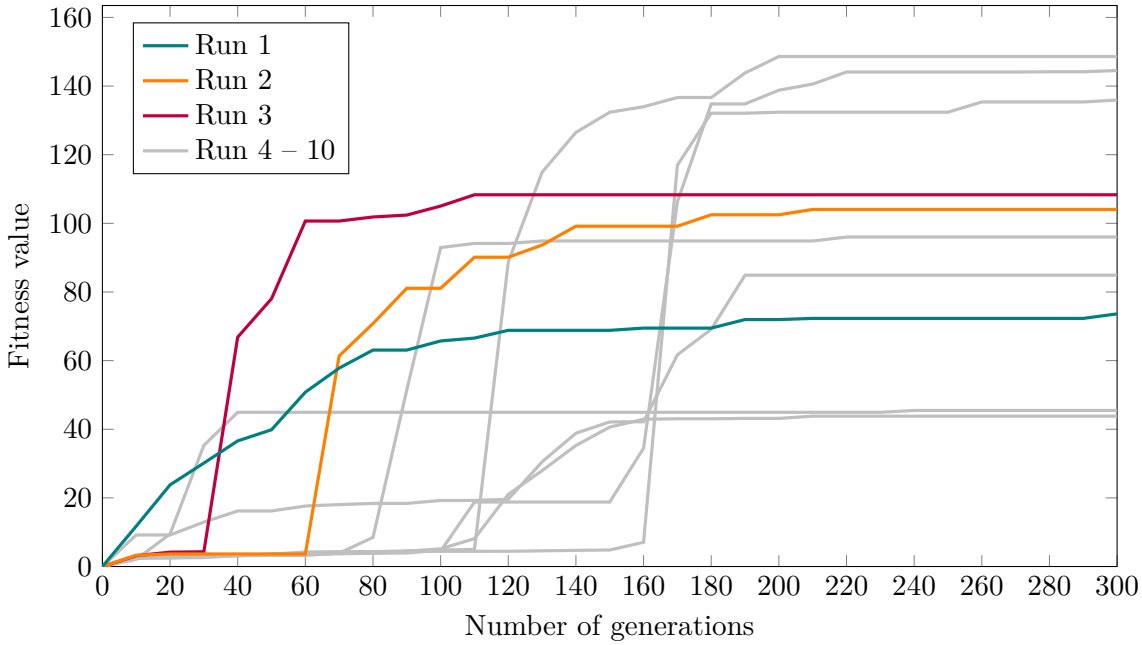

Figure A.2: Evolutionary comparison for the Biped 1 (moving forward)

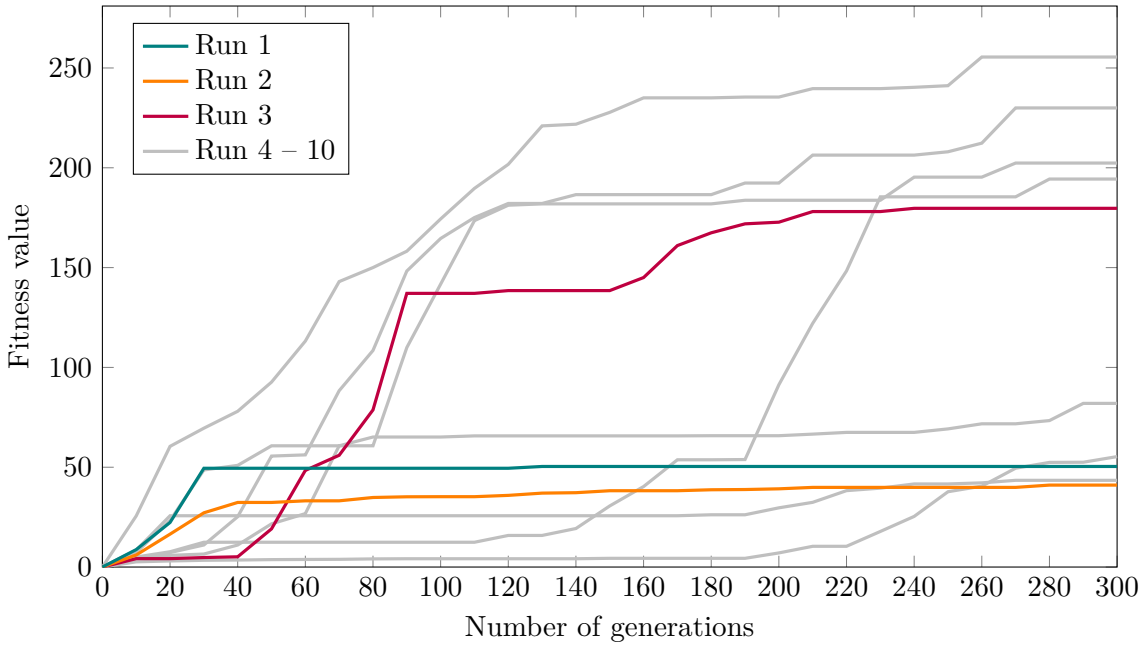

Figure A.3: Evolutionary comparison for the Biped 2 (moving forward)

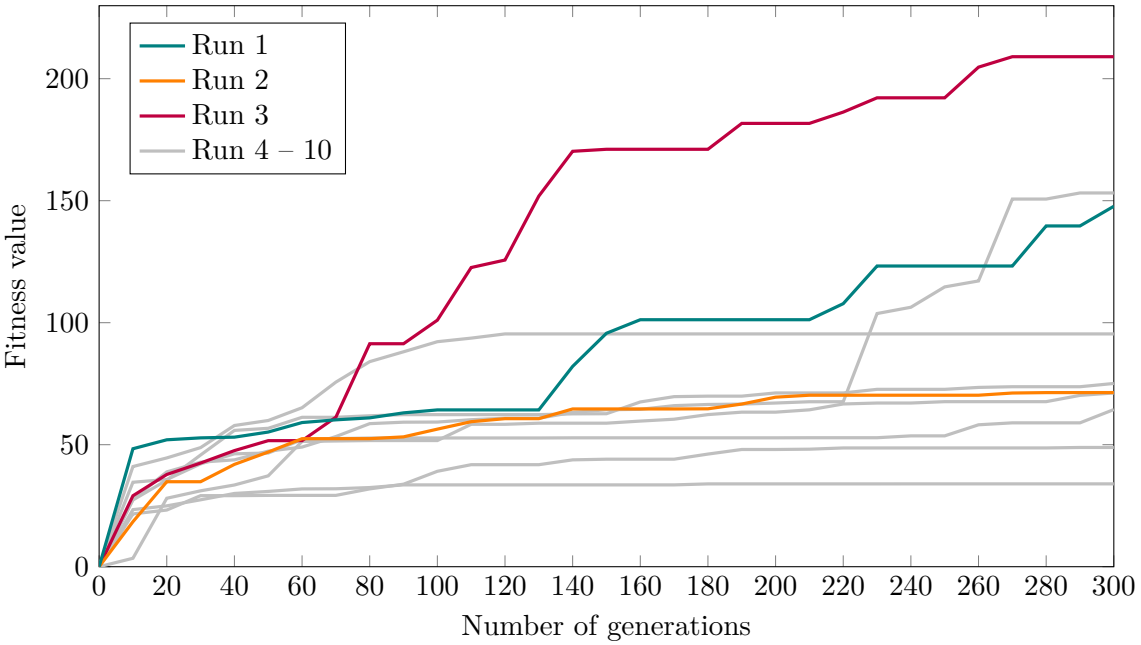

Figure A.4: Evolutionary comparison for the Tetrapod 1 (moving forward)

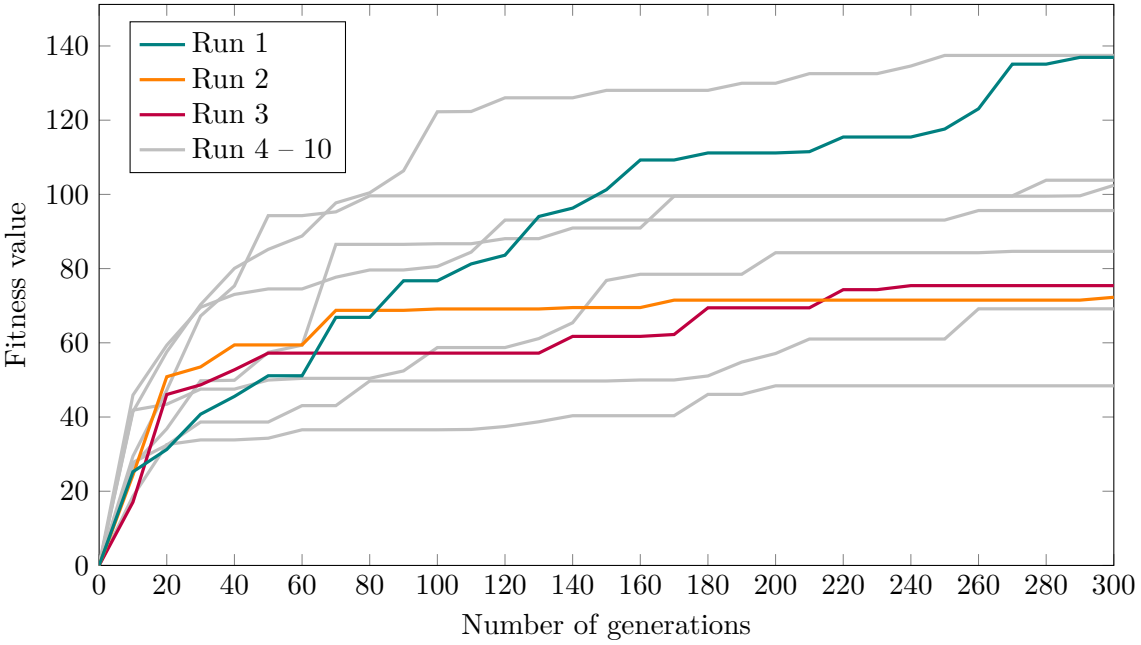

Figure A.5: Evolutionary comparison for the Tetrapod 2 (moving forward)

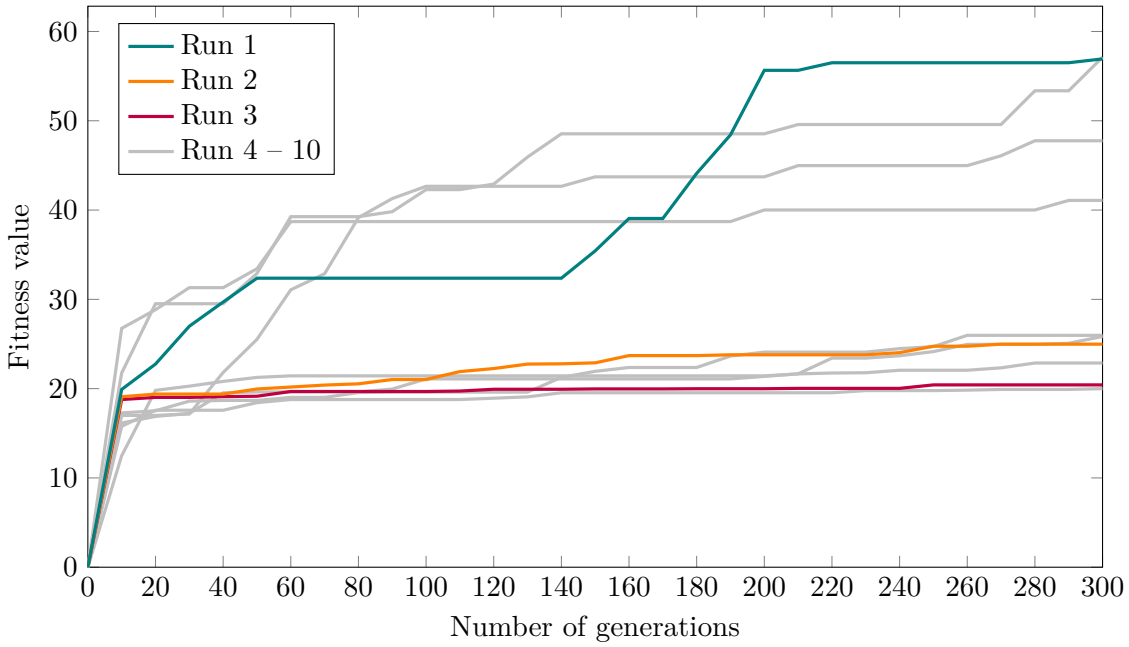

Figure A.6: Evolutionary comparison for the Decapod 1 (moving forward)

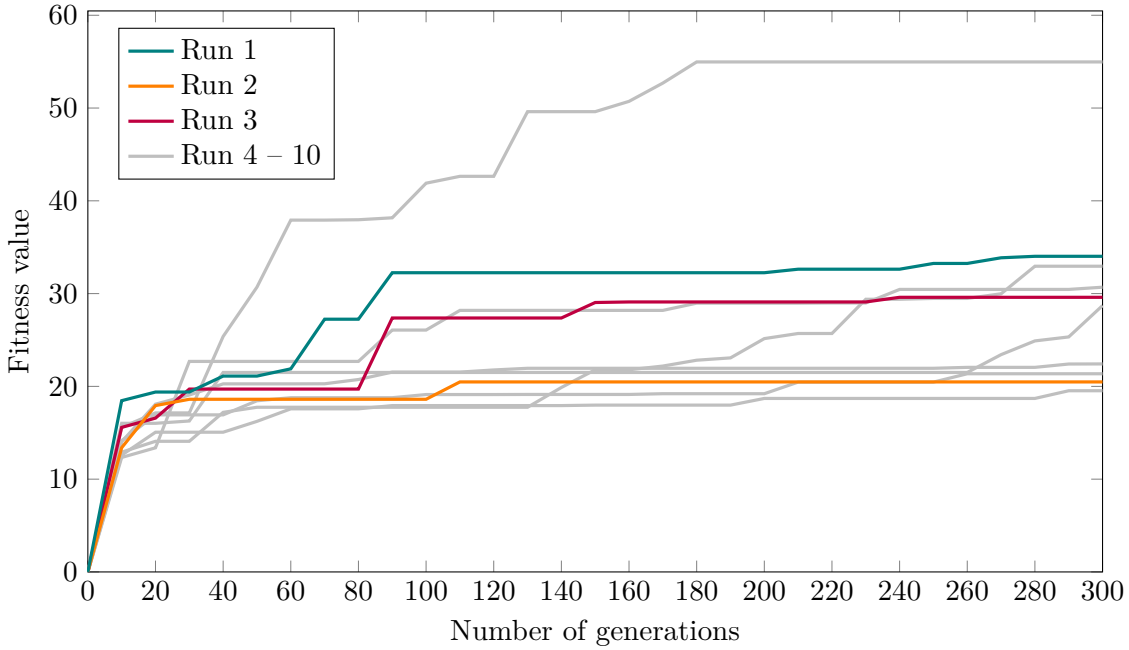

Figure A.7: Evolutionary comparison for the Decapod 2 (moving forward)

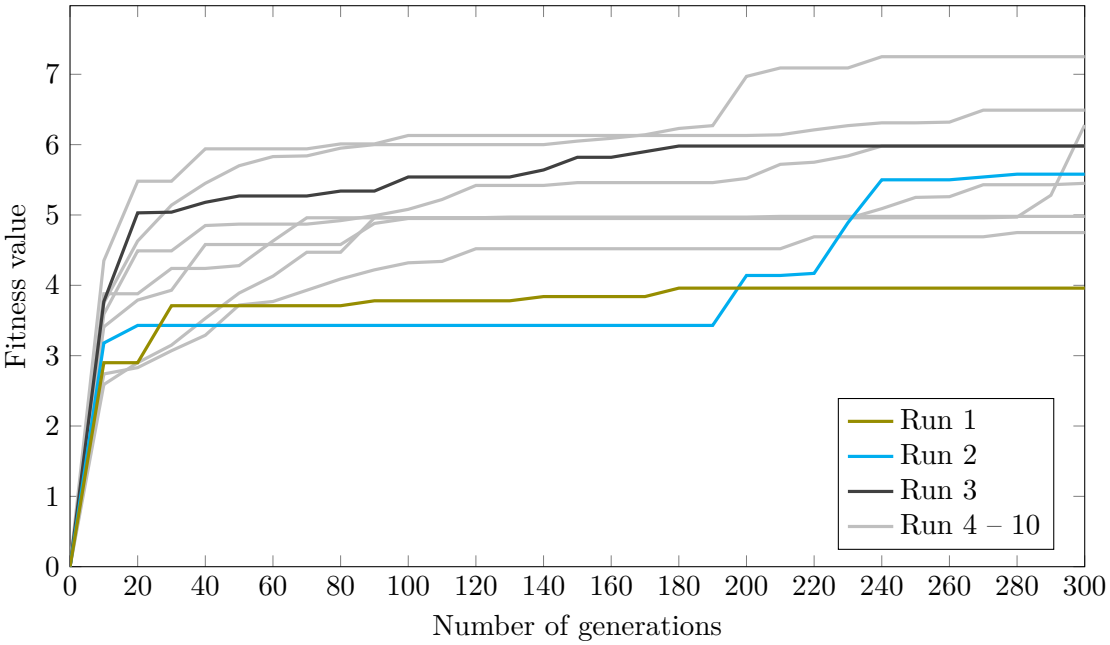

Figure A.8: Evolutionary comparison for the Apod (jumping)

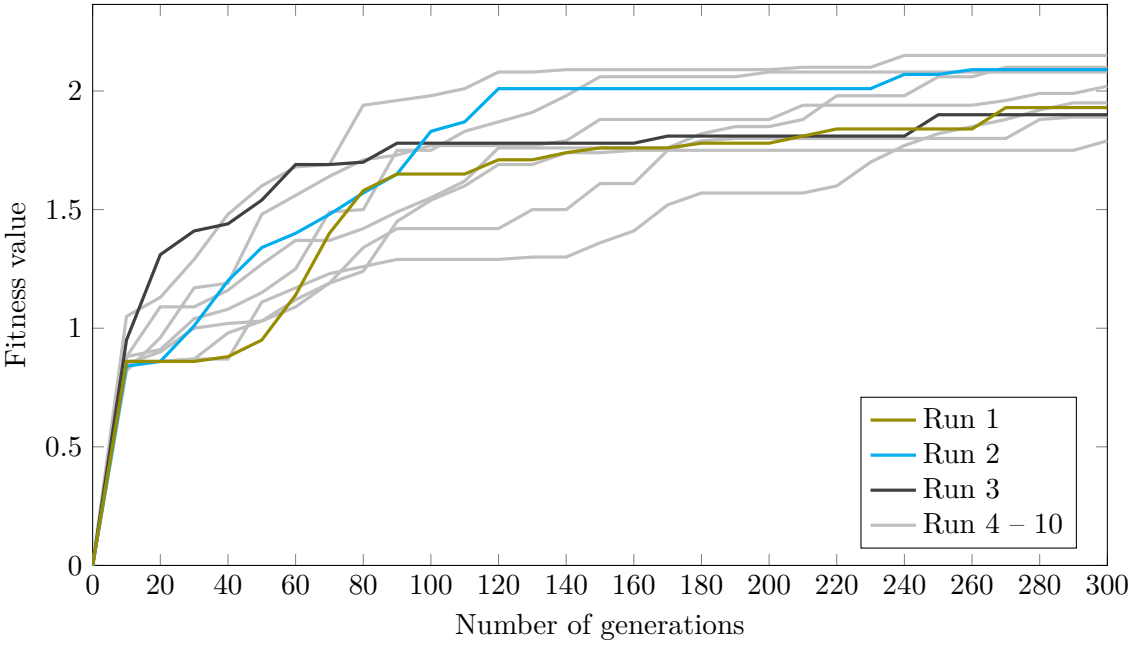

Figure A.9: Evolutionary comparison for the Biped 1 (jumping)

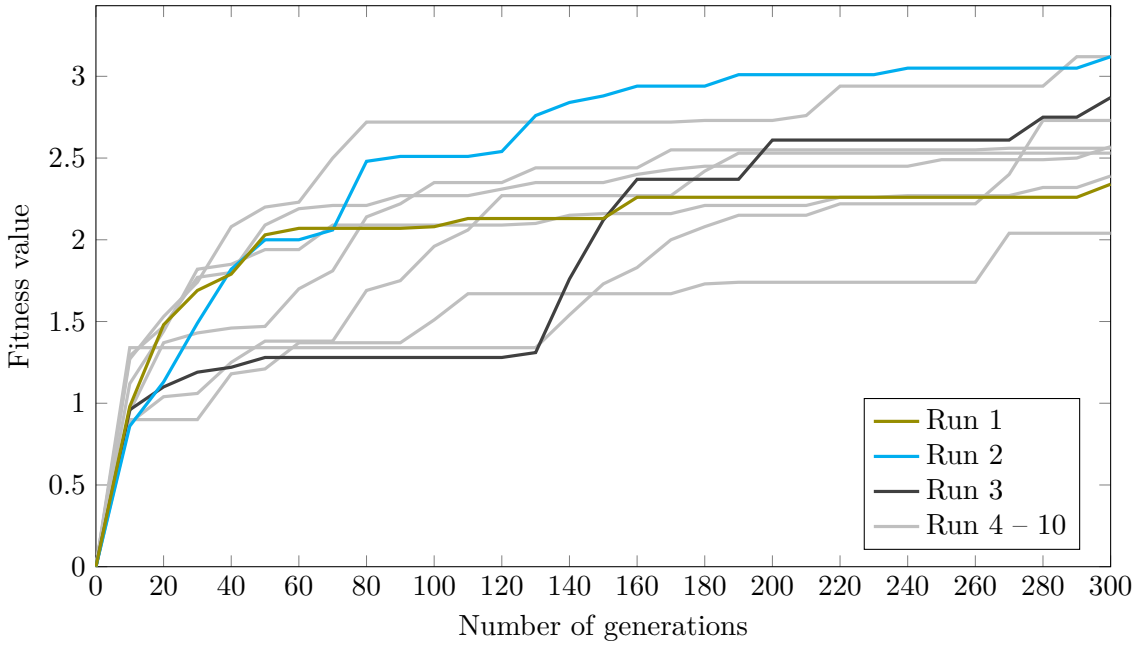

Figure A.10: Evolutionary comparison for the Biped 2 (jumping)

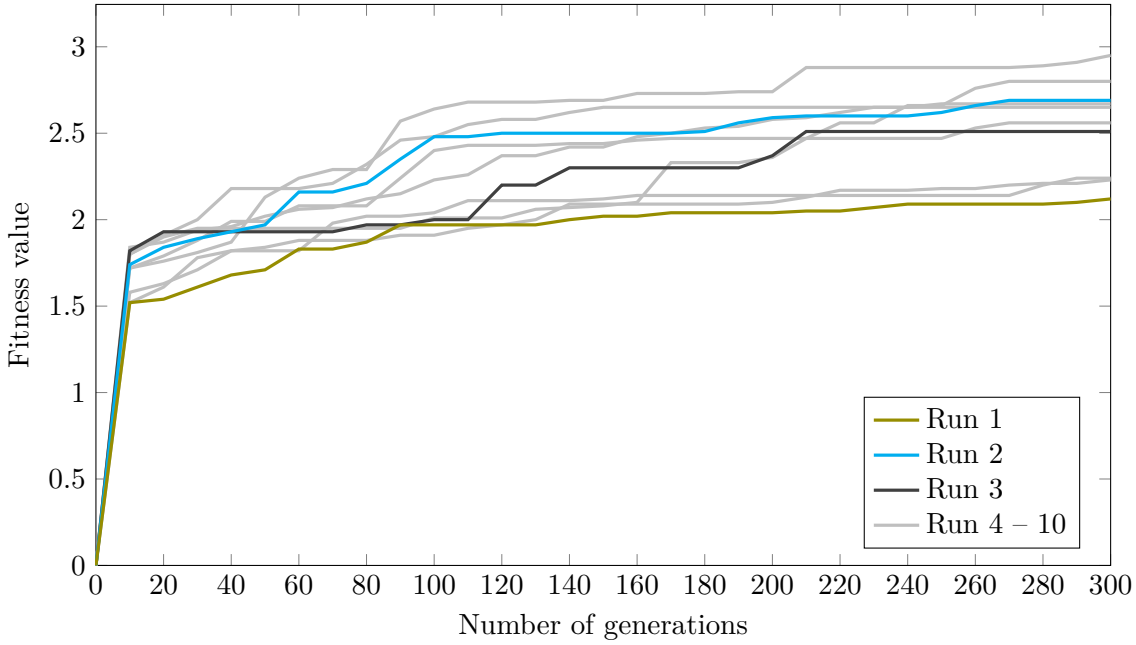

Figure A.11: Evolutionary comparison for the Tetrapod 1 (jumping)

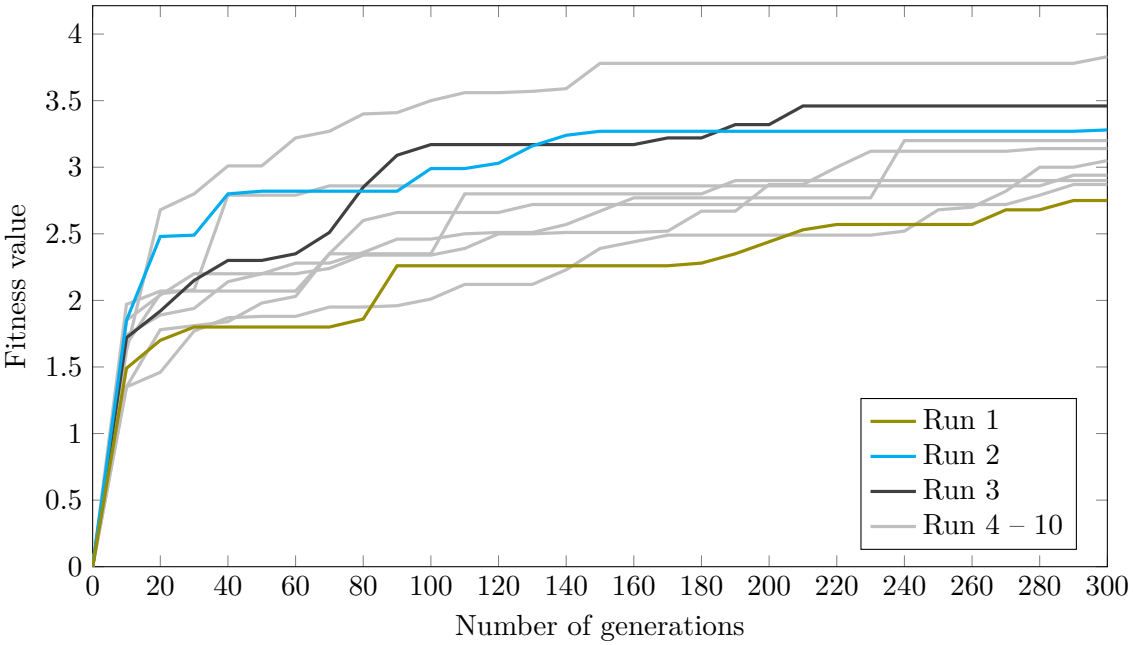

Figure A.12: Evolutionary comparison for the Tetrapod 2 (jumping)

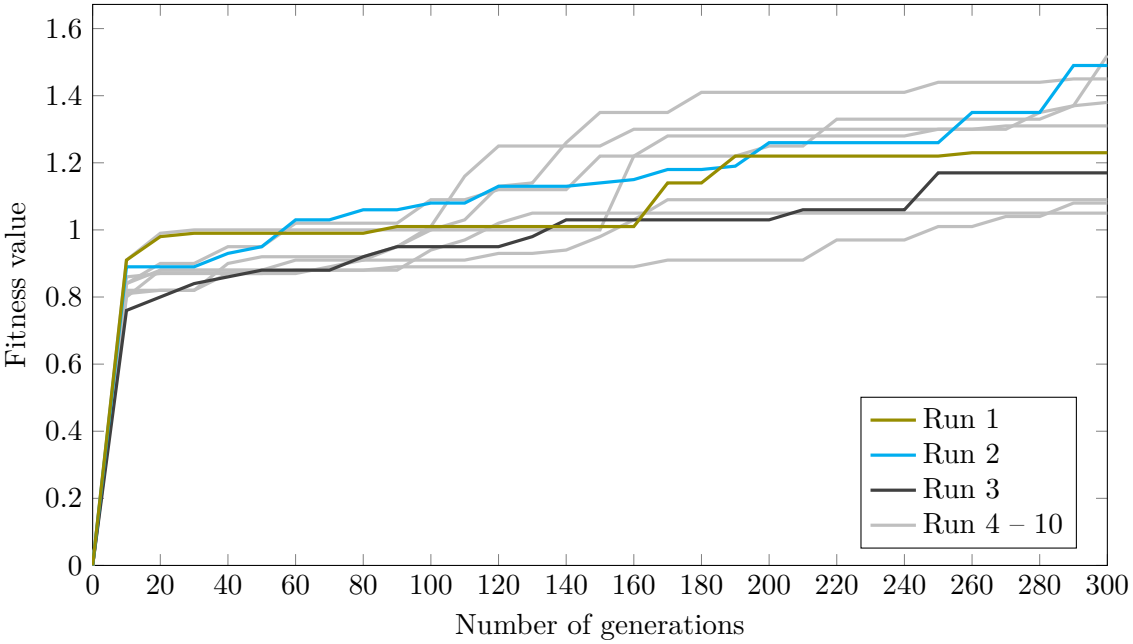

Figure A.13: Evolutionary comparison for the Decapod 1 (jumping)

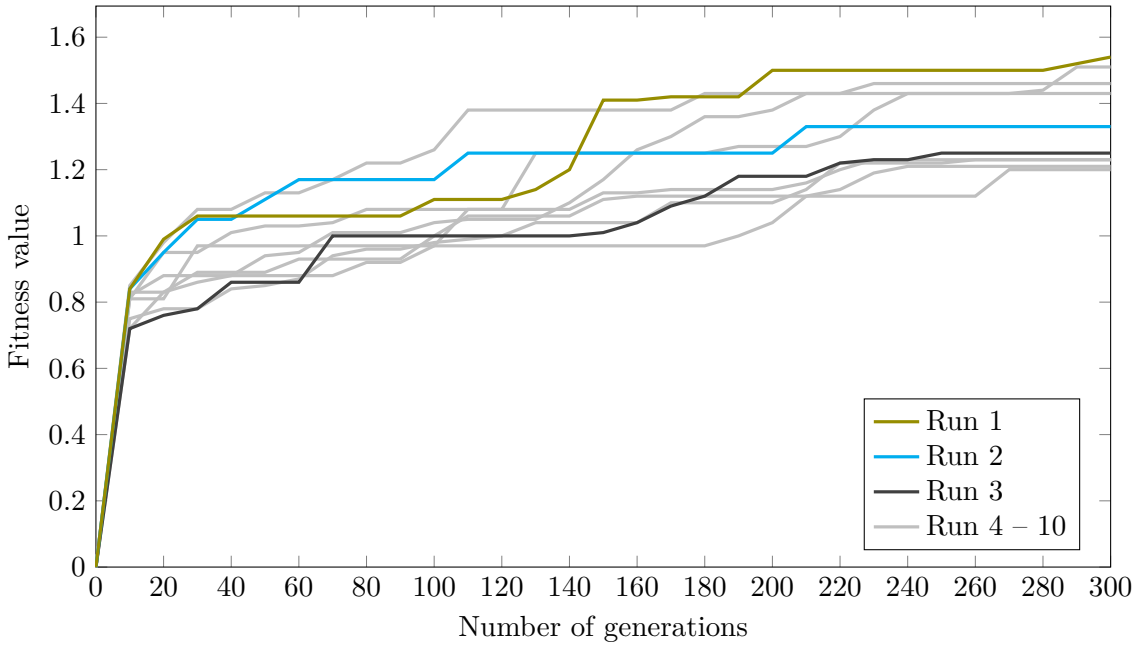

Figure A.14: Evolutionary comparison for the Decapod 2 (jumping)