Evolution of a Neural Network for Gait Animation

Anonymous

Anonymous

Abstract

In this paper we describe efforts to create a physically based system that automatically produces realistic real-time animations of walking figures, controlled by a neural network, the weights and functionality of which is evolved by genetic algorithm techniques.

In traditional computer graphics, the animator is forced to use intuition about the physical world in specifying the motions of objects in a scene. The scenes must then be manually inspected for collisions. Since humans are sensitive to detecting anomalies in everyday physics, and real motions tend to be complex, manual control techniques have generally proven to be unsatisfactory.

The use of dynamics greatly improves motion realism, and shifts control of the animation from specifying absolute positions of objects to applying forces and torques to the objects in the scene. In addition, it is possible for a dynamics system to automatically detect and respond to object collisions. This is a much more difficult task for a human to control.

A neural network makes an ideal controller for the figures to be animated. Manually programming a neural network is yet more difficult for a human, and setting up training examples to make the neural network learn by back propagation is not straightforward, since there is no functional solution to be solved.

Through the use of a genetic algorithm, one can determine the performance of a neural network as a whole, and select for whatever behavior is desired. Animator control is then directed through model design and behavior choices. In our system the genetic algorithm maximises the distance that the walking figure covers over a randomly generated terrain.

Key words: gait control, genetic algorithm, neural network controller, physically based animation

1 Introduction: Animation of Jointed Figures

Creating a realistic animation of jointed figures can be a difficult task [**?**]. Today, there are two major methods in use by animators. The first method uses *inverse kinematics*, where an animator moves some limb of an object and all attached skeletal elements are adjusted to compensate, using reactive forces [**?**]. The second uses *motion capture*, where an actor or prop rigged up with sensors performs in front of a camera, and the motions are digitized as a near-complete animation. An animator can then edit the animation so produced. Basically, all methods try to overcome the problem of making a figure move realistically, such that they appear to conform to the laws of our physical world, and also look fluid and efficient in their motion.

1.1 Animation Dynamics

Another method that is gaining popularity in the graphics world for creating realistic animations is the use of dynamics [**?**]. Since the 1980's [**?**] dynamics has been used for computer animation to produce highly realistic results, where the animator has control over the forces and torques applied to the model, rather than its absolute position.

David Baraff states in his paper [**?**], "A realistic physical-based simulation of rigid bodies demands that no two bodies inter-penetrate. In order to enforce this constraint, a simulator must first detect potential inter-penetration between two bodies, and then act to prevent the two bodies from penetrating." The former is fundamentally a kinematic problem, involving the positional relationship of objects in the world. The latter is a dynamic problem, in that it involves predicting behavior according to physical laws [**?**].

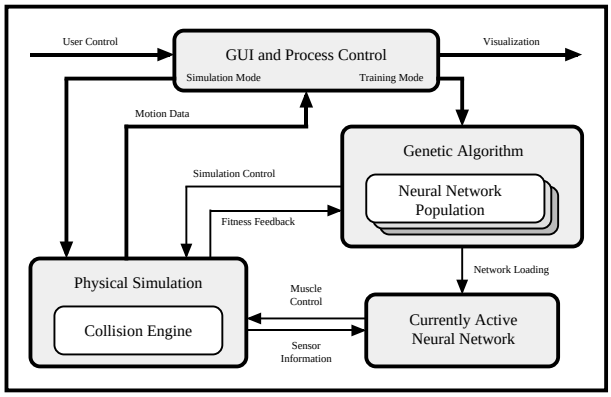

Figure 1: An overview of Creeper

1.2 *Creeper*: Dynamics Through Neural Networks and Evolution

The system described in this paper is called *Creeper*. The system makes use of a neural network to control a physically based walking model. The model uses a genetic algorithm to train the inputs to a neural network in order to learn a gait which will achieve a greater distance over uneven terrain. An overview of the system is shown in Figure 1.

This paper is organized as follows: Section 1 serves as an introduction to the computer animation and artificial intelligence methods used in this work. Section 2 introduces our system and describes the physically based animation methods used. Section 3 deals with the collision detection methods. Section 4 introduces the artificial intelligence concepts implemented in a neural network controller. Section 4.1 describes the gait controller, in section 5 the evolution controller is presented.

The results of our experiments are described in Section 6. and the many things left for future work are described in Section 7.

2 The *Creeper* Physics

There have been two main classes of methods presented for dealing with dynamics equations: analytical and numerical. Analytical techniques conform to a mathematically rigorous theory [**?**], and produce quantitatively accurate results [**?**], however, they give rise to elaborate derivations and intricate algorithms [**?**]. Numerical methods have issues with inaccuracies [**?**], and may fail in some cases [**?**], but are more generally applicable and easier to comprehend [**?**]. There also do exist improved methods for numerical integration [**?**], [**?**], [**?**].

One method of using a numerical scheme to calculate and animate rigid body dynamics is presented in [**?**]. In this work, van Overveld states that the dynamical properties of a rigid body are completely determined by the tensor of inertia, the total mass, and location of the center of gravity. Thus it is possible to create a representative dynamical model for rigid objects. A variation of this method was used in *Creeper* to build a simulation that accurately and efficiently models the dynamics of semi-rigid structures.

2.1 Application of Dynamics to the *Creeper*

A model in *Creeper* consists of a set of points, or *particles*, each with a position in space, a velocity, and a mass. Some pairs of these points are connected by line segments. For each time division, the next position and velocity of each point is calculated from the sum of the accelerations applied to it during that time step, and its previous position and velocity,

$$p_{h+1} = p_h + v_h\, t + 0.5\, a_h\, t^2 \quad (1)$$

$$v_{h+1} = v_h + a_h\, t \quad (2)$$

where p is position, v is velocity, a is acceleration, h is the current time step, and t is time.

Line lengths are kept constant by applying a correcting acceleration parallel to the line, balanced to return the line to its proper length and taking into account the relative velocities of the two points,

$$a = \frac{r(d - vt)}{t^2} \quad (3)$$

where r is a rigidity constant, which can be used to set the elasticity of the lines. The simulation also has standard gravity and simple drag built-in, which can be set through control parameters.

3 Collision Detection

At present, most animation systems do not provide even minimal collision detection, but require the animator to visually inspect the scene for object interaction and respond accordingly. This is time-consuming and difficult, even for keyframe systems where the animator explicitly defines the motion. It is even worse for procedural or dynamical animation systems, where the motion is generated automatically.

Two methods for detecting collisions are presented by Matthew Moore et al. [**?**]. One is designed to test the *interpenetration of surfaces* modeling flexible objects, and the other is designed to test the *interpenetration of convex polyhedra*, where concave polyhedra can be decomposed into collections of convex ones. The penetration of a point through the surface of a triangle is tested for a fixed time increment. When the triangle is fixed in space, for example as a a component of the surface representing the ground, the calculation is $O(mn)$ for m triangles and n points. The system must also test for edge-surface penetrations [**?**]. In the case where two objects are in continuous contact, friction must also be considered to achieve realism [**?**].

3.1 Application of Detection

In the system presented here, the physical simulation is run for one time step and the predicted position of each point at the end of that time step is calculated. Collisions are detected by testing for penetration of each point through every line. A collision point can be found by solving the parametric vector equation,

$$\begin{aligned} P + (P' - P)t &= A + (B - A)u + (A' - A)t \\ &+ ((B' - B) - (A' - A))ut \end{aligned} \quad (4)$$

where P, A, B are points in the system, AB is the line, P', A', B' are the predicted position after the time step, u is a parametric variable, and t is time.

If $0 \leq t \leq 1$ and $0 \leq u \leq 1$, then the point has intersected the line at time t and position u along the line, see Figure 2.

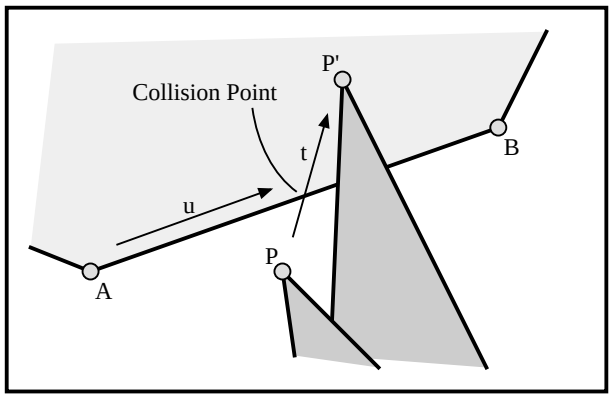

Figure 2: Visualization of a collision test, with a fixed line.

Equation 4 can be expanded to a second order polynomial equation in t, so there are two solutions for t, however, this can be a computationally expensive test, when done with each point/line combination [**?**]. To reduce the number of times the above calculations must be performed for the *Creeper* collision detection algorithm, a bounding box test is first done. In the case that the two bounding boxes, of the motion of the point and the motion of the line, do not intersect, the collision test can be omitted since no collision could have occurred.

3.2 Collision Response

In keyframed and procedural animation systems, collision detection is the main requirement, where collision response is left for the animator to perform. In animation systems using dynamics to generate motion, the system itself must respond to a collision.

The most intuitive way to handle collisions is with springs. Dynamic systems must have a method for applying external forces to objects. Thus, when a collision is detected, a very stiff spring is temporarily inserted to push the objects apart. The main problem with this method is that it can be computationally expensive for hard collisions. As the springs are compressed, smaller and smaller time steps are required for accurate numerical integration. An analytical solution for the collision of two arbitrarily articulated rigid objects is available as well. The analytical solution depends upon the conservation of momentum during a collision, and results in a new angular and linear velocity for each body.

Some combination of spring and analytical collision response may be desirable, since the case where an object is resting on another due to gravity would cause the analytical solution to be solved repeatedly, whereas a simple spring that counteracts gravity would be more stable in this case.

3.3 Application of Response

The first response after a collision is detected is to prevent interpenetration. This is done by modeling all collisions as non-elastic collisions, and setting the final position of points PAB to their position at the time of the collision. Although all collisions in *Creeper* are modeled as non-elastic collisions, elastic collisions are simulated realistically due to the elasticity of the models.

Once an initial collision has been dealt with, it is added to a list of active contact points. This contact point is kept track of until such time as the objects move apart again, and the contact point can be discarded. Where two objects are in contact, and as long as the force between the objects is greater than zero, a frictional force will hold the objects together through the contact point, and the points PAB are kept linear. This frictional force is infinitely strong in the current implementation, such that the contact point will not move along the surface of the line.

As long as two objects are in continuous contact, forces are exchanged through the contact point, and are balanced by position of the contact point along the line, see Figure 3.

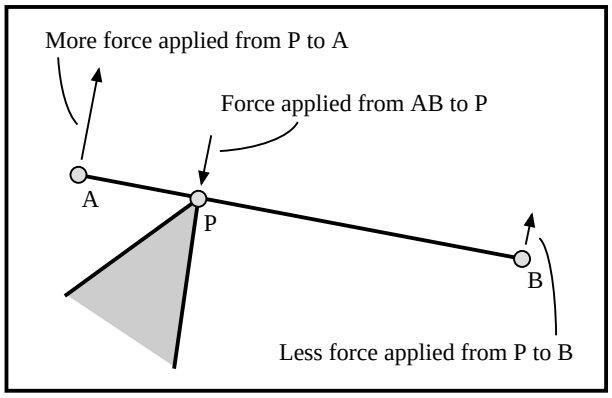

Figure 3: Visualization of a contact point.

4 Making the *Creeper*Move

For a physically-based figure to move with intent, it must move under its own power. Jihun Park et al [**?**] state, "A human body is complicated mechanically, and is very different from the robotic mechanisms. Most people use robotic mechanisms for their human body animation, we believe, because of the complexity of musculotendon ac-

tuators. But by simple muscle actuators, we can get more realistic results than simple joint torque actuation like a robot". In other words, for a biological animated figure to appear realistic in its movements, it should simulate the physical response of real muscles and tendons. How the muscles are positioned is a model design issue [**?**], and the effect of a muscle's force on an articulated figure is an aspect of the physical simulation. How a muscle behaves can be considered separately, and in terms of biological features [**?**], though the question remains how to control the muscles.

4.1 Walking Control With Neural Networks

Using a physical simulation for animation changes the animator's task from controlling the position of the model, to controlling the forces and torques applied to the model [**?**]. This is not only as demanding as using traditional animation, but also more difficult for a human to comprehend. Although there has been work in limiting this complexity by using goal-oriented control [**?**] [**?**], there has also been success in using lower-level artificial intelligence techniques to control model parameters [**?**], [**?**], [**?**], [**?**]. Michael van de Panne uses a sensor-actuator network [**?**] to control an articulated figure. In this work

One such technique is based on neural networks, which implement a network of interconnected simple processing elements, following the scheme of how brain cells process information [**?**]. Each cell in the network is called a neuron, which employs a simple function to map its weighted inputs to an output signal (Figure 4). The output $x_n^{(out)}$ of neuron n is calculated as:

$$x_n^{(out)} = \Omega(f(\sum_i w_i\, x_i)) \tag{5}$$

where f is the transfer function of the neuron, which is applied after the weighting of the input signals x_i, and Ω denotes the output function, which, in our case, is a unity filter function,

$$\Omega(x) = \begin{cases} 1 & : \quad x > 1 \\ -1 & : \quad x < -1 \\ x & : \quad otherwise \end{cases} \tag{6}$$

The transfer function f of a neuron may vary between implementations, depending upon the application it is intended for. Generally, simple functions are used, such as a threshold

$$f_\theta(x) = \begin{cases} 0 & : \quad x \leq \theta \\ 1 & : \quad x > \theta \end{cases} \tag{7}$$

or sigmoidal function

$$f_\theta(x) = \frac{1}{1 + e^{-\theta x}} \tag{8}$$

4.2 The *Creeper* Network Controller

The neural network controllers used by *Creeper* are organized as a standard feed-forward network structure, where signals are propagated from an input to an output layer, via one layer of hidden neurons, as shown in Figure 4. Although the network contains no explicit recursive connections, the loop through the environment acts as a recursive path [**?**].

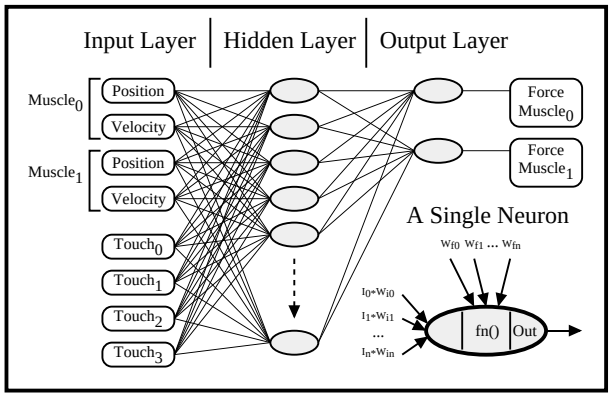

Figure 4: A neuron as used by Creeper

The inputs to the networks are the measured and filtered muscle length and delta length, (i.e the difference between the previous and current lengths), for each muscle, and a touch sensor for each vertex in the model. The outputs from the neural system control the change in length of each muscle. Each neuron is fed a weighted sum of its inputs, scaled by the sum of the weights, and computes output based on several possible, genetically chosen functions including:

$$f(x) = 2xa + s \tag{9}$$

$$f(x) = 1.0 - (\theta + 1)^{(2x(a+1))} \tag{10}$$

$$f(x) = \begin{cases} 1.0 & : \quad x > s + \theta \\ -1.0 & : \quad x < s - \theta \\ 0 & : \quad otherwise \end{cases} \tag{11}$$

$$f(x) = \frac{2a}{1 + 2^{-(6+3\theta)\times(x+s)}} - a \tag{12}$$

$$f(x) = a \times cos(0.63 \times t \times f + s \times \pi) + \theta \tag{13}$$

In these formulas above, θ denotes a threshold, a an amplitude, and s is a shift parameter. These sets of parameters (for each function) are included in the neural network chromosome, where they are treated like additional

weights that determine which of these functions are used and for which parameter settings.

The weights of all the connections are adjusted through a training session, which results in a functional mapping from input to output signals [**?**], [**?**].

5 Controller Evolution

Michiel van de Panne [**?**] points out that the top down approach assumes that we have knowledge of the type of motion that is desired, but it is often difficult to come up with the desired gait, or the motion desired may be physically impossible for the model to generate. Like van de Panne's creatures, the *Creeper's* gait is automatically generated. In this approach we use a genetic algorithm to evolve a gait to optimize certain fitness criteria.

A genetic algorithm makes use of Darwin's theory of evolution to transform a random population of individuals to a set of highly fit individuals, based on a functional definition of what makes an individual fit. The optimization is accomplished by repeatedly selecting the individuals with a higher fitness, and creating a new population based on these fit individuals.

The general scheme of a genetic algorithm, the details of which are explained in the following section, is as follows:

1. $t := 0$
2. A random population $P^{(t)}$ of individuals is generated
3. Each individual $x \in P^{(t)}$ is tested for its fitness $\sigma(x)$
4. Individuals are selected according to a fitness-proportinate selection scheme
5. Recombinations and mutations are applied to the selected individuals, which constitute the new population $P^{(t+1)}$
6. $t := t + 1$
7. Goto step 3, until a termination criterion is met

Hugo de Garis [**?**] successfully used a neural network to control the articulation of a simple stick figure, but in a non-dynamic environment. In order to train the neural network to walk, he used a genetic algorithm, where the calculated fitness was the distance the stick figure moved during a short simulation. Karl Sims [**?**] evolved not only the neural network controller for his blocky figures, but the structure of the actor as well. This produced actors that exhibited a selection of different behaviors, when trained with different fitness functions. Larry Gritz et al [**?**] and [**?**], used genetic programming in order to develop LISP controllers for an articulated lamp actor. There are other examples of this technique used successfully too, such as in [**?**], [**?**], wherein a genetic programming system is used to evolve the topology, the neuron functionality and the weights of a neural network controller.

5.1 Simulation Mode

Our gait animation system has two main modes for simulation and training. In simulation mode, either a single neural network with random initial weights is created, or a trained network is loaded from a file, which can then be run continuously in real time. No learning occurs during the simulation.

5.2 Training Mode

In training mode, a population P of N individual neural networks (with N chosen commonly in the range $[25, 100]$) are randomly generated, or loaded from a file, and the genetic algorithm is used to adjust the weights of the neurons. Each neural network is represented as a list of its neural weights. Each weight can take on values in the interval $[-1, 1]$ and have an accuracy of $K = 16$ bits.

Fitness Calculation

For every generation, each individual $n_k \in P$ is given 30 seconds in the simulation, and a fitness value $\sigma(n_k)$ is calculated based on how well the *Creeper* performs on a particular terrain. The shape of the terrain is parameterized and is randomly generated for each individual *Creeper* in order to promote more generalized neural controllers [**?**]. The fitness calculation takes two criteria into account: (1) the distance traveled from the starting point, and (2) the energy used:

$$\begin{aligned} \sigma(n_k) &= p_x^{(final)} - p_y^{(start)} \\ &- w_e \times \sum_{n \in Output} |x_n^{(h)} - x_n^{(h+1)}| \end{aligned} \quad (14)$$

where w_e is a user-defined constant for weighting the energy influence.

Offspring with Variation

After evaluating the whole population, pairs of individuals $(n_1, n_2) \in P^2$ are selected in a fitness-proportionate manner. That is, the probability $p(n_k)$ of an individual n_k to be selected for further promotion into the next generation is directly dependent on its fitness:

$$p(n_k) = \frac{\sigma(n_k)}{\sum_{i \in P} \sigma(n_i)} \quad (15)$$

Variation of the two neural network encodings, n_1 and n_2, occurs on a per-weight basis through the two genetic operators of multi-point crossover and point mutation. Multi-point crossover is used to recombine the

weight vectors of n_1 and n_2, where each vector has the form $n_i = (w_{1i}, \ldots w_{Mi})$, $i = 1, 2$, and a weight set of size M is assumed for each network. The recombination $rec(n_1, n_2)$ is performed with a probability of $\mu_{cross} = 0.001$ per chromosome as follows (see Figure 5):

$$rec(n_1, n_2) = n' = (w'_1, \ldots, w'_k, \ldots, w'_M) \qquad (16)$$

where the weights for the new chromosome are calculated by

$$w'_k = \begin{cases} mut(switch(w_{ki})) & : \quad \chi \leq \mu_{cross} \\ mut(w_{ki}) & : \quad \chi > \mu_{cross} \end{cases} \qquad (17)$$

$$switch(w_{ki}) = \begin{cases} w_{k1} & : \quad i = 2 \\ w_{k2} & : \quad i = 1 \end{cases} \qquad (18)$$

with the initial assumption that $i = 1$. Furthermore, χ denotes a function that generates uniformly distributed random numbers in the interval $[0, 1] \subset \Re$.

Subsequently, each weight, $w = (w_1, \ldots, w_K) \in \{0, 1\}^K$, is mutated with probability $\mu_{mut} = 0.1$, by flipping a single randomly chosen bit, where χ_K generates a random integer between 1 and K.:

$$mut(w) \quad = \quad (mut(w_1), \ldots, mut(w_K)) \qquad (19)$$

$$mut(w_i) \quad = \quad \begin{cases} 1 - w_i & : \quad \chi_K = i \\ w_i & : \quad otherwise \end{cases} \qquad (20)$$

Selection and subsequent variation through the two genetic operators, crossover and mutation, are iteratively applied until the a new offspring population is filled with the same number N of individuals of the parent population. This new population replaces the current generation, with the exception that the best $\mu_{elite} = 3$ individuals of a population survive into the next generation unaltered. This process of generating offspring populations is repeated continuously, until stopped by the user or a termination criterion is satisfied.

6 Results

Preliminary results have shown some successes and some failures. Figure 7 is the fitness graph of a successfully trained model, showing the fitness of each individual per generation of training. At the start of the training, the individuals in the population have a fitness average of approximately zero. The terrain then quickly rises, however, as high scoring individuals are better represented in following generations, in many mutated forms, while low scoring individuals are discarded. In the case shown in the figure, the first individuals reach a near maximum fitness occur around generation 75, but by generation 125, over half the population have reached this level, in various forms, as the successful genes propagate through the population. Note how the mutation and recombination operators, necessary for gene pool diversity, cause a percentage of low scoring individuals every generation. Although these individuals have little to no effect on the gene pool, there is a probability that some individuals with a higher fitness will be produced, so progress can be made [**?**].

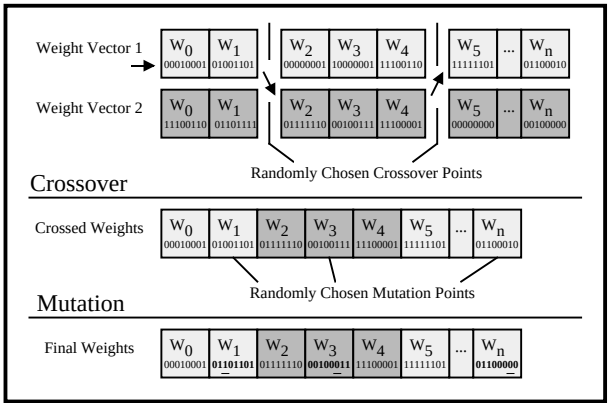

Figure 5: The Genetic Algorithm as used by Creeper

The walking animations produced look physically realistic within the constraints of the 2D system, and the neural network can deal with the uneven terrain. The gait generated is not always symmetric or efficient, in that some individuals walk with a pronounced limp, or kick their legs as they walk. This could be fixed in later versions by including *style* [**?**], [**?**] components to the fitness function, and so the techniques used in this system show promise for the automatic production of gaits. Providing the neural network with more information about its environment may help as well, since when the touch sensors were introduced into the system, the resultant walking animations looked more realistic. A few frames from a walking sequence after training are shown in Figure 6.

Model design is an important issue in *Creeper*. If a model is poorly designed, it might be difficult or impossible for a controlling neural network to produce a gait. Figure 8 shows a fitness graph where a model failed to learn. The graph shows three distinct cases, where the walking figure falls forward, stands upright, or falls backward. It seems to be important that a model is designed with enough strength and limb leverage such that it can easily lift a foot off the ground to take a step forward, but without so much strength that it can vibrate across the

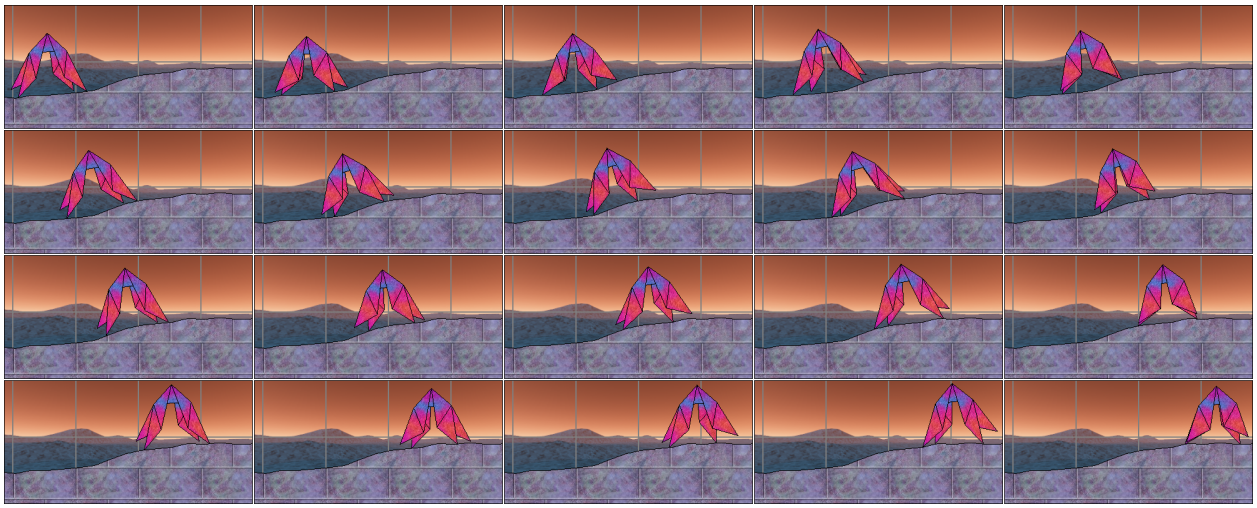

Figure 6: Some frames from an animation of the walk after training.

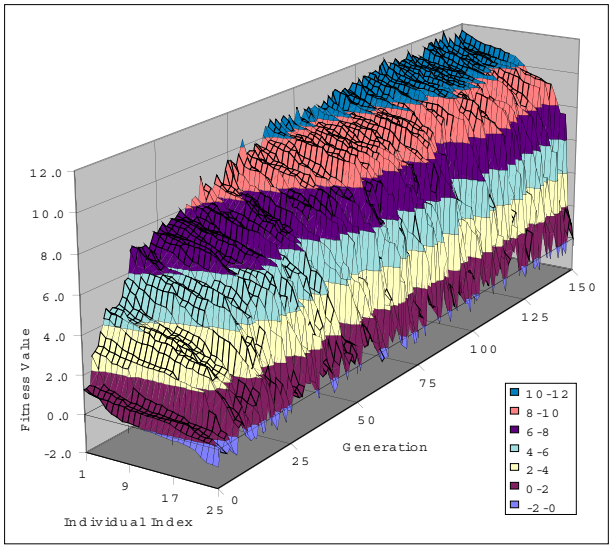

Figure 7: Fitness Graph for successful learning

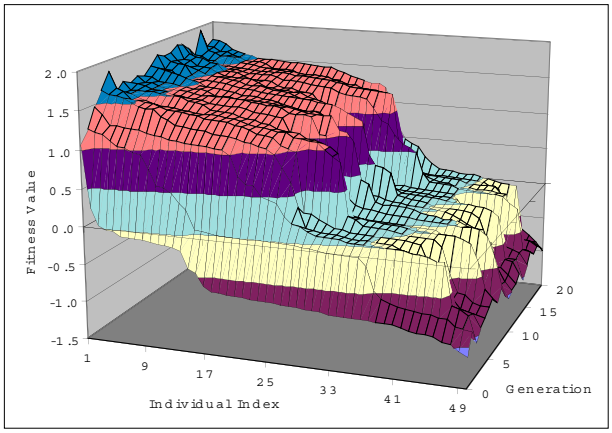

Figure 8: Fitness graph showing a failure to learn

terrain.

We are currently gathering statistics, on the effect of the various evolution parameters, that will be presented in the final version of this paper. In addition, a number of more complex models are being experimented with. A screenshot of the system, showing a simple walking creature, the textured background and uneven terrain is shown in Figure 9.

7 Conclusion and Future Work

This system is, for the most part, robust and accurate, and is able to automatically create a realistic animation of a walking figure. Collision detection and response, although not fully complete, greatly improve the realism of the simulation. The current implementation works well in the simulation of rigid body dynamics, however, there are some notable shortcomings that will be addressed in future versions of the *Creeper* program. Most notably, the implementation is in two dimensions, due to the fact that three dimensional collision detection is much more difficult to compute and more time consuming to calculate. In the move to 3D, there would also be additional requirements on the direction of walking. In order to become a true actor, the *Creeper* should be able to follow curved paths as well as straight lines. This would occur under the control of a higher mind which sets the amount of forward motion and left or right turns appropriately. Control decisions like these could be implemented as ad-

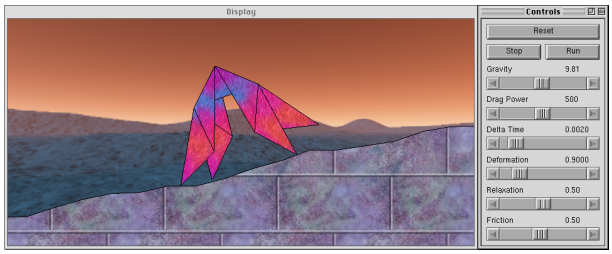

Figure 9: Screenshot of the walking creature.

ditional parameters of the neural network controller.

Having a model editing program, or being able to import 2D or 3D objects as models, would be a good thing to have. Currently, models for *Creeper* are designed on paper, then transcribed by hand to a text file, listing point coordinates and connecting lines. Having a built-in model editor would allow for quick changes to me made with little effort, and allow for free experimentation. Using a professional model design tool, then importing the model, would also make surfaces easier to apply, and perhaps allow for texture mapping the objects, for better visualization.

Regarding the neural network controller and its training, it may be beneficial to take advantage of the Baldwin effect [**?**], where individuals would employ some form of learning during the physical simulation. Another possibility for extension is to impose a higher degree of modularity onto the network, relating the topologies of the neural networks to the structures of the models [**?**]. Consequently, two symmetric limbs will have a symmetric network topology, so that a step forward with one leg will produce the same motion as a step forward for the other leg. It may also prove useful to evolve network topologies, to allow for the automatic generation of coordination subnets [**?**].

Acknowledgements

This work is supported in part by grants from the Natural Science and Engineering Research Council of Canada.