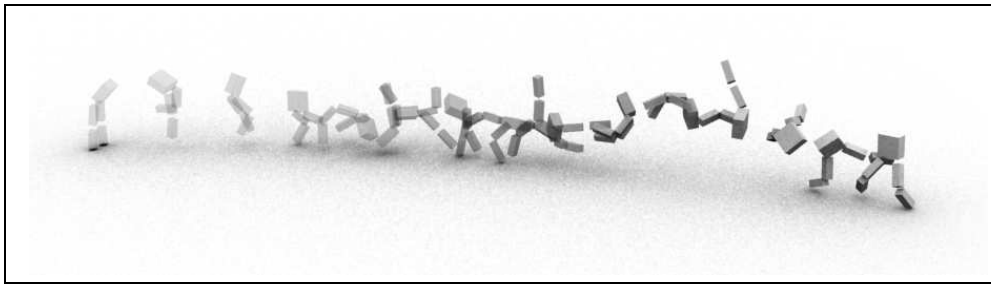


# EVOLVING BEHAVIOURS USING GENETIC PROGRAMMING



MASTERS THESIS

MICHAEL P. JONES

N.C.C.A BOURNEMOUTH UNIVERSITY

11th September 2006

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                          | <b>1</b>  |
| 1.1      | Thesis overview . . . . .                    | 1         |
| 1.2      | Nomenclature . . . . .                       | 2         |
| <b>2</b> | <b>Previous Work</b>                         | <b>3</b>  |
| 2.1      | Applications to Computer Animation . . . . . | 3         |
| 2.1.1    | Genetic Programming . . . . .                | 3         |
| 2.1.2    | Neural Networks . . . . .                    | 5         |
| 2.1.3    | Other Studies . . . . .                      | 5         |
| <b>3</b> | <b>Evolutionary Computation</b>              | <b>6</b>  |
| 3.1      | Genetic Programming . . . . .                | 7         |
| 3.1.1    | Recombination . . . . .                      | 8         |
| 3.1.2    | Mutation . . . . .                           | 8         |
| <b>4</b> | <b>Implementation</b>                        | <b>11</b> |
| 4.1      | Genetic Library . . . . .                    | 11        |
| 4.1.1    | Node Networks . . . . .                      | 12        |
| 4.1.2    | Controllers . . . . .                        | 12        |
| 4.1.3    | Expression Syntax . . . . .                  | 13        |
| 4.1.4    | Evolution . . . . .                          | 13        |
| 4.2      | Physics Engine . . . . .                     | 15        |
| 4.3      | Characters . . . . .                         | 15        |
| 4.3.1    | Creature . . . . .                           | 16        |
| 4.3.2    | Biped . . . . .                              | 16        |
| 4.3.3    | Specialised Nodes . . . . .                  | 16        |
| 4.4      | Components Library . . . . .                 | 17        |
| 4.5      | Fitness Function . . . . .                   | 18        |
| 4.6      | Visualisation . . . . .                      | 18        |
| 4.6.1    | Command line . . . . .                       | 18        |
| 4.6.2    | Graphical User Interface . . . . .           | 18        |
| <b>5</b> | <b>Pipeline</b>                              | <b>19</b> |
| 5.1      | File Structure . . . . .                     | 19        |
| 5.1.1    | Pipeline syntax . . . . .                    | 19        |
| 5.1.2    | Scripts . . . . .                            | 20        |
| 5.2      | Evolution . . . . .                          | 21        |
| 5.2.1    | Statistics . . . . .                         | 21        |
| 5.2.2    | Text User Interface . . . . .                | 22        |
| 5.3      | Visualisation . . . . .                      | 22        |
| 5.3.1    | Rendering . . . . .                          | 22        |
| <b>6</b> | <b>Results &amp; Analysis</b>                | <b>23</b> |

|          |  |           |
|----------|--|-----------|
| 6.1      | Learning Curve . . . . .                           | 23        |
| 6.2      | Fitness Function . . . . .                         | 23        |
| 6.2.1    | Constraints . . . . .                              | 24        |
| 6.3      | Evolutionary Parameters . . . . .                  | 25        |
| 6.4      | Character Design . . . . .                         | 27        |
| 6.5      | Specialised Nodes . . . . .                        | 27        |
| 6.6      | Creature . . . . .                                 | 28        |
| 6.6.1    | Multiple Controllers . . . . .                     | 28        |
| 6.6.2    | Single Controller . . . . .                        | 29        |
| 6.6.3    | Ball Interactions . . . . .                        | 30        |
| 6.7      | Biped . . . . .                                    | 31        |
| 6.7.1    | Walking . . . . .                                  | 31        |
| 6.8      | Blending . . . . .                                 | 35        |
| 6.8.1    | Balancing . . . . .                                | 37        |
| <b>7</b> | <b>Conclusion</b>                                  | <b>40</b> |
| 7.1      | Further Work . . . . .                             | 41        |
| 7.1.1    | Blending . . . . .                                 | 41        |
| 7.1.2    | Automatically Defined Functions . . . . .          | 41        |
| 7.1.3    | Other characters . . . . .                         | 41        |
| 7.1.4    | Matching motion capture . . . . .                  | 41        |
|          | <b>Bibliography</b>                                | <b>43</b> |
| <b>A</b> | <b>Correspondence</b>                              | <b>44</b> |
| A.1      | Natural Motion . . . . .                           | 44        |
| <b>B</b> | <b>Implementation Details</b>                      | <b>45</b> |
| B.1      | Fitness Functions . . . . .                        | 45        |
| B.2      | Evolution Settings . . . . .                       | 47        |
| B.3      | Features of the Graphical User Interface . . . . . | 48        |
| B.4      | Text User Interface . . . . .                      | 48        |
| <b>C</b> | <b>Scripts &amp; Aliases</b>                       | <b>50</b> |
| C.1      | Pipeline Aliases . . . . .                         | 50        |
| C.2      | Awk Statistics Script . . . . .                    | 50        |

# List of Tables

|     |   |    |
|-----|---|----|
| 6.1 | Joint limits for the biped walker . . . . . | 27 |
|-----|---|----|

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Virtual creatures evolved for walking and crawling . . . . .                                 | 4  |
| 2.2  | Biped figure used by Reil and Husbands . . . . .   | 5  |
| 3.1  | An example of a parse tree . . . . .   | 8  |
| 3.2  | Example of recombining parse trees . . . . .   | 9  |
| 4.1  | Creature and Biped Morphology . . . . .  | 16 |
| 6.1  | Graph to show the effect of population size on the average distance covered by biped walkers | 26 |
| 6.2  | Graph to show the effect of crowding on the average distance covered by biped walkers . .    | 27 |
| 6.3  | Example of a multi-controller creature’s motion . . . . .                                    | 29 |
| 6.4  | Example of a single-controller creature’s motion. . . . .                                    | 29 |
| 6.5  | Cartwheeling behaviour evolved by one creature. . . . .                                      | 30 |
| 6.6  | Examples of cyclic behaviours demonstrated by the creature. . . . .                          | 30 |
| 6.7  | Final position of a creature interacting with a ball . . . . .                               | 31 |
| 6.8  | A walker controlled by a single network . . . . .  | 32 |
| 6.9  | An acrobatic “walker” evolved with a single controller . . . . .                             | 32 |
| 6.10 | Examples of the multiple-controller biped walkers . . . . .                                  | 34 |
| 6.11 | First promising result for the biped with feet . . . . .                                     | 35 |
| 6.12 | Examples of the biped walker covering a large distance . . . . .                             | 36 |
| 6.13 | Balancing Behaviours . . . . .   | 37 |
| 6.14 | More extreme balancing behaviour . . . . .   | 38 |

# Chapter 1

## Introduction

The power of natural evolution is visible in every aspect of the living world. Over millions of years it has created organisms with a level of complexity far beyond what humans have managed. Evolution guides different species to find the best solution to living in their environments; in this sense it is a system of optimisation that has produced fantastically varied and interesting results.

Artificial evolution is the application of these ideas to a computational environment. The different aspects of evolution can be represented within a computer and an entire evolution witnessed in seconds instead of millions of years. Clearly the more complex the system becomes the longer such artificial evolutions take, however with the continuing improvement in computer resources and processing speeds, the field of computational evolution is an accessible one with intriguing ideas and results.

This thesis outlines an approach to the application of evolutionary techniques to the area of motion synthesis. Specifically it deals with the use of genetic programming to develop movements for articulated bodies in the three dimensions. The aim of the study is to create a natural walking behaviour for a bipedal model.

### 1.1 Thesis overview

**Chapter 2: Previous Work** Reviews the work of others who have used similar approaches in this field, including the pioneering work of Karl Sims. Special attention is paid to those who have used genetic programming.

**Chapter 3: Evolutionary Computation** Presents an introduction to the field of computational evolution and its applications. Describes some of the different approaches to artificial evolution, focusing on the genetic programming methods used within this study.

**Chapter 4: Implementation** Details the techniques used and how they were achieved. This section also provides algorithms for the main methods used and reports on any difficulties encountered and any design decisions that had to be made. Includes a description of the user interface elements developed for the study.

**Chapter 5: Pipeline** Describes the work flow and how the various elements fit together. Contains brief explanations of the data structures used and the scripts that automate the various tasks required.

**Chapter 6: Results & Analysis** Presents the results and describes the steps taken to improve them over the course of the study.

**Chapter 7: Conclusion** Discusses the results of the study and the advantages and disadvantages of using this approach to motion synthesis. Contains suggestions for future work in the field and possible extensions for the project.

The appendices include script examples and correspondence records. All source code and Doxygen documentation is provided on a CD at the end of this document.

## 1.2 Nomenclature

Throughout this thesis the individual members of an evolution will be referred to as characters or individuals. A generation refers to a single group of characters at one point in the evolution. A run refers to a complete evolutionary run of a number of generations with a population of characters. Finally a batch refers to a set of evolutionary runs with the same parameters but different seeds.

## Chapter 2

# Previous Work

The subject of evolving motion controllers has received the most attention in the field of robotics, where the results of the evolution can be used to control the robots reactions to different stimulus in the environment. Research within the field of computer animation is less prevalent but interest in the area has been growing over the last two decades.

The use of evolutionary techniques for motion synthesis is only a very small part of artificial evolution which otherwise receives a lot of attention for various different search and optimisation problems. Additionally motion synthesis is approached from many other areas. This study focuses only on the overlap between these two fields.

### 2.1 Applications to Computer Animation

Evolutionary computation within computer animation has been a subject of study since Karl Sims' pioneering paper in 1991, in which he investigates the application of evolutionary techniques to evolving plant morphology and images [Sim91]. In 1994, he went on to study the evolution of creatures and their behaviours using similar ideas and a method based on genetic programming [Sim94b]. Sims developed a language for directed graphs of nodes that could be used as the basis of the creatures physical and neural structure. The networks of these nodes were then subject to a co-evolutionary process to develop final morphologies and behaviours. As part of the evolution the creatures were tested in dynamic simulations to see how successful they were at different tasks. The results are intriguing to watch as the block-like creatures display organic looking motions and behaviours within different environments. Sims developed different sets of creatures for the tasks of swimming, jumping, running and goal following. Examples of the creatures developed to run or crawl are shown in figure 2.1.

Later Sims studied the effect of competition on the evolution of these virtual creatures [Sim94a]. His paper describes a competition which pitches two creatures against each other to fight for possession of a block placed between them. A number of interesting strategies were evolved for either grasping at the block or keeping the opposing creature away.

Sims mentions in his papers the difficulty specifying the fitness function for the evolution. This function is used to judge the creatures' performance but frequently the desired behaviour is difficult to translate into instructions for the system. The creatures tend to find effective strategies that exploit loop holes in the fitness tests, producing behaviours that are simultaneously optimal for the environment and yet undesired.

#### 2.1.1 Genetic Programming

Sims' work is based on a genetic programming approach though he develops his own directed graph networks, which are capable of recursion, rather than the less flexible parse trees that are generally used in the



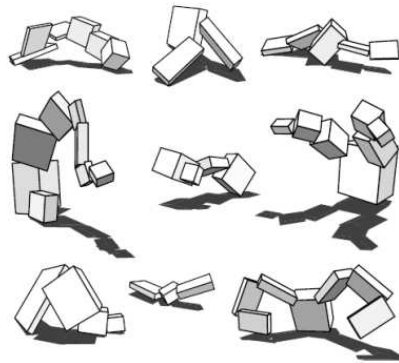


Figure 2.1: Virtual creatures evolved for walking and crawling by Sims [Sim94b]. The creatures are subject to a co-evolution of both morphology and controller networks with fascinating results.

field. More traditional work with parse trees was carried out by Larry Gritz who looked into the application of genetic programming to controlling 3-D articulated bodies [Gri99]. His PhD thesis paid particular attention to developing robust controllers and a system that could be scripted by non-technical users to produce animation data. Like with Sims, his evolutionary approach involved testing the controllers in a dynamic environment however the majority of his work was focused on producing behaviours for a fixed articulated structure: Luxor the Pixar lamp.

Both Sims and Gritz implement sensors for the specimens they evolve. This allows the individuals to monitor their own state, detect that of the environment and respond at a basic level. The sensors are nodes in the networks that return a value that varies based on the conditions in the simulation. For example, there are joint nodes that measure the angle of the joints in the body and contact nodes that detect if a particular element of the creature is in contact with anything else.

Spencer looks into using genetic programming to generate walking behaviours for a hexapod [Spe94]. His work focuses on a 2-dimensional top-down simulation that considers the forces applied by the legs but only in the plane of movement. Instability is accounted for by considering the position of the centre of mass with respect to the positions of the feet. If a certain instability condition is not met for a length of time then the hexapod is said to have fallen over. This approach allows for reasonably realistic behaviours without worrying about a full physical simulation. The author also believes that the work would extend well to 3-dimensions.

The study looks at how the availability of *a priori* information effects the success of the evolution. Useful functions and nodes were designed for the evolution, for example, an oscillator node or a function that reversed the leg motion dependent on whether the leg was in the up or down position. Experiments were then carried out to see how the evolutions performed with and without these task specific nodes and functions. Success was achieved at all levels but the results indicated the advantage of having the specialised nodes.

Spencer also used a single node network to govern the whole hexapod; a different approach to Sims and Gritz who evaluate one network per degree of freedom in the creature. Spencer's technique included side effecting nodes that could receive an input, use it to set the force on a joint and then pass the value on to the next node in the network. This allows the maximum possible information sharing within the network as implementing different networks limits the recombination options. If all the networks are grouped together a single recombination could effect a number of different degrees of freedom, whereas otherwise networks are normally paired off for recombination and so share information only with their partner and not with others in the group.

Another interesting feature of his work is that, as an indication of the success of the evolutionary technique, the results were compared against a hand designed controller. Whilst the design was quite rough it was interesting to note that the evolutionary methods found more efficient solutions in all cases.

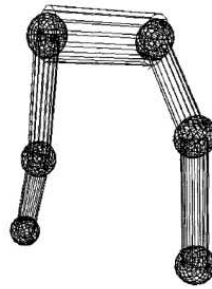


Figure 2.2: Biped figure used by Reil and Husbands [RH02]. The feet are modelled with spheres in order to simplify the collision handling for the physics system.

### 2.1.2 Neural Networks

Another common approach to this field is to use neural networks as controllers for articulated body. The concept is favoured in robotics and in a number of papers in computer animation. A possible advantage is that the neural network can be designed by hand and then optimised by evolving the network weights. This allows a high quality controller to be designed from known theory and then “tweaked” to maximum performance using evolution. Reil and Husbands look at recurrent neural networks as controllers for biped motion [RH02].

The authors report that 10% of the evolutions produce stable walkers, though due to a lack of constraints on the biped figure used, not all stable specimens exhibited natural looking gaits. Interestingly, and in contrast to a lot of literature, the authors claim success with a simple fitness function that rewarded the individuals based on how far they travel from the origin whilst keeping their “hips” above a certain height.

The biped they used was simplified to minimise the degrees of freedom and to make the physics simulation more stable (figure 2.2.) The spherical feet appear difficult to balance on however are important for simplifying the collision detection.

### 2.1.3 Other Studies

Genetic programming was also used by Craig Reynolds who published a paper on evolving controllers for obstacle avoidance [Rey94]. This paper focuses on the idea of introducing noise to the simulation to produce more robust controllers. The idea is that a controller that can act well in a noisy environment might be able handle changes more easily than one trained in a clean and clear simulation.

Unfortunately Reynolds does not report any success in his work, saying that the evolution of a robust controller through the use of noise may well be impossible but is hopefully just out of reach of the evolutionary techniques he tried during the investigation.

## Chapter 3

# Evolutionary Computation

Artificial evolution has been an active area of study since the 1950s when it was initially explored as a means to reproduce and study biological evolution on a simple scale. Some early uses focused on evolutionary techniques as a means of solving problems but it was not until 1975 that Holland's work on genetic algorithms really demonstrated the power of evolution for search and optimisation [Hol75].

The basis of computational evolution is that if a solution to a problem can be represented or encoded in a data structure, then an initial random population of these data structures can be created and evolved to find the optimum solution to the problem. The evolution requires iterating through a series of generations and at each stage measuring how well the solutions in the population solve the problem. This measure is known as fitness and the fittest members of the population are selected to continue to the next generation. Variations on the population are created through the evolutionary operators: recombination and mutation. When applied to the individuals in the population, these operators produce variations which will perform to a different level. Those that perform better can be preferentially selected for the next generation and so the process continues. The standard algorithm for artificial evolution is given in algorithm 1.

It might be tempting to believe that the process is no better than randomly generating a huge number of individuals and choosing the single fittest solution for the problem. Instead of having a hundred generations of three hundred individuals, one could create thirty thousand individuals and the best would probably be an adequate solution. This is not the case. Studies show that evolution guides the population to the optimum in a more efficient manner than random generation of solutions. As an example, Spencer reports that whilst none of the initial 28,000 hexapods that started his evolutions could walk, after twenty-eight generations with a population of a thousand every single run had produced a number of individuals capable of walking a reasonable distance [Spe94].

Normally the data structures that are evolved are simplified or encoded representations of the true solutions to the problem. The encoding translates the potential solution into a structure that can easily be acted upon by evolutionary operators. The fitness function must decode the structure, test the result and then assign a fitness value to that individual according to how well the solution performs.

Genetic algorithms use binary strings to encode solutions to problems, their main use is for situations that lend themselves to this representation. For other problems however it might be useful to have different representations. For this reason and others, a number of areas of computation evolution have formed over the last five decades each with their particular approach and encoding of the problem. An overview of the different areas is given below.

- **Genetic Algorithms:** As mentioned above, this approach to evolutionary computation involves representing the units of evolution as binary strings. These strings of 1s and 0s become the DNA of the individuals: a representation that is surprisingly true to life. Holland's initial work in the field was the first to put real emphasis on the recombination operator, with mutation being rarely used.

---

**Algorithm 1** Standard evolution process [BFM99b].
 

---

```

01  $t \leftarrow 0$ 
02  $P(t) \leftarrow$ initialise population
03  $F(t) \leftarrow$ evaluate population
04 while ( exit conditions not met ) do
05    $P'(t) \leftarrow$ recombine( P(t) )
06    $P''(t) \leftarrow$ mutate( P'(t) )
07    $F(t) \leftarrow$ evaluate( P''(t) )
08    $P(t + 1) \leftarrow$ select( P''(t), F(t), )
09    $t \leftarrow t + 1$ 
10 od

```

---

The recombination operation can be thought of as interchanging sections of the strings between two individuals, and mutation involves randomly flipping a 1 to a 0 or vice versa. Traditionally genetic algorithms deal in fixed length strings [BFM99b].

- **Genetic Programming:** The first implementations of genetic programming were at the start of the 1980s however the key publication in the field was by Koza in 1992 [Koz92]. Koza's work placed heavy emphasis on recombination and considers mutation a background operator. The key interest in genetic programming is that its representation is a node network of function and terminal nodes. This network, often with a tree-like structure, is then evaluated to produce a specific result. This means that the technique does not just optimise parameters on a predefined solution but instead can create the solution from scratch. The evolutionary units are themselves computer programs.
- **Evolutionary Strategies:** This area features techniques that focus on the optimisation of real-valued vectors with an emphasis on both mutation and recombination as evolutionary operators [BFM99b]. This form of evolution is ideal for tuning the weights of an otherwise fixed network or system. The different weights can be represented in a single vector and optimised with this approach.
- **Evolutionary Programming:** Initially developed for the optimisation of finite-state machines, evolutionary programming is now mostly used for evolving the values of real-valued vectors. The area has a strong emphasis on mutation and does not incorporate recombination at all [BFM99b].

### 3.1 Genetic Programming

As described above the power of genetic programming is in its ability to evolve computer programs. The evaluation of tree-like networks of nodes provides a very flexible context for generating a huge variety of programs and solutions. Node networks structured in this way are referred to as parse trees and are the main representation scheme in genetic programming. An example of a parse tree and the resulting expression is given in figure 3.1. Different representations can be used and some more recent studies implement networks without the constraint of maintaining a tree-like structure.

The networks created are composed of two different sets of nodes: function nodes and terminal nodes. The terminal nodes have either fixed values or values dependent on external factors and so take no inputs from the network and are consequently the terminus of a network branch. Functions nodes all take inputs, act on them in some way and output the result. The way in which the function node acts on its inputs is determined by the system designer however there are a number of traditional nodes that cover the basic mathematical operations of addition, subtraction, multiplication and division. Special care needs to be taken with the division node too avoid dividing by zero. Such an operation is normally referred to as a protected divide and the node normally returns unity if the denominator input is too close to zero. It is also common to include a constant node, referred to as an ephemeral constant, which is a terminal node with a floating point value randomly generated from a normal distribution with a mean and standard deviation that is suitable to the task.

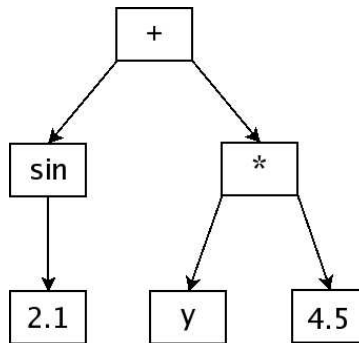


Figure 3.1: An example of a parse tree for the expression “ $\sin(2.1) + (y * 4.5)$ ”. Where “ $y$ ” is an input to the network.

In the evolutionary system each individual or member of the species is represented by single or multiple parse trees. Parse trees normally handle a single floating point data type with each node receiving and returning the same type, however more complicated systems can be implemented in which multiple data types are handled.

Once the node set is determined for an evolutionary run, an initial population must be created. This can be achieved using a stack storage system to keep track of the lower levels of the tree whilst the higher branches are created. As part of the evolution it is necessary to implement the necessary evolutionary operators. As discussed, both recombination and mutation operators are used with genetic programming, though the focus is on recombination.

### 3.1.1 Recombination

The recombination operator always acts on two members of the population at once and in genetic programming involves swapping parse tree branches between them. Due to the parse tree structure a branch can be removed by choosing a single node disconnecting it from its parent. This node then forms the head of a disconnected branch and can be moved to another parse tree and connected up in a suitable location. In recombination this process is carried out on each parse tree and the each branch is connected in the original place of the other. The technique is illustrated in figure 3.2.

Traditionally the two original parse trees are referred to as the parents and the recombined parse trees are called to the children. For practical purposes the networks are normally limited to either a set depth or a certain number of nodes, so additional checks must be carried out to make sure that the neither of the child networks exceed the limit.

### 3.1.2 Mutation

The mutation operator acts on a single individual in the population and alters their parse tree in some way. There are four main ways of mutating a parse tree [BFM99b]:

- **Grow:** Involves searching the network for a terminal node and replacing it with a newly grown branch. The branch can be randomly generated but should not force the network over the node limit.
- **Shrink:** The inverse of growth, involves finding a function node and replacing it with a terminal node thus removing the branch originating at that point.
- **Switch:** Acts like recombination within a single individual. Any two nodes in the network are selected and their positions swapped.

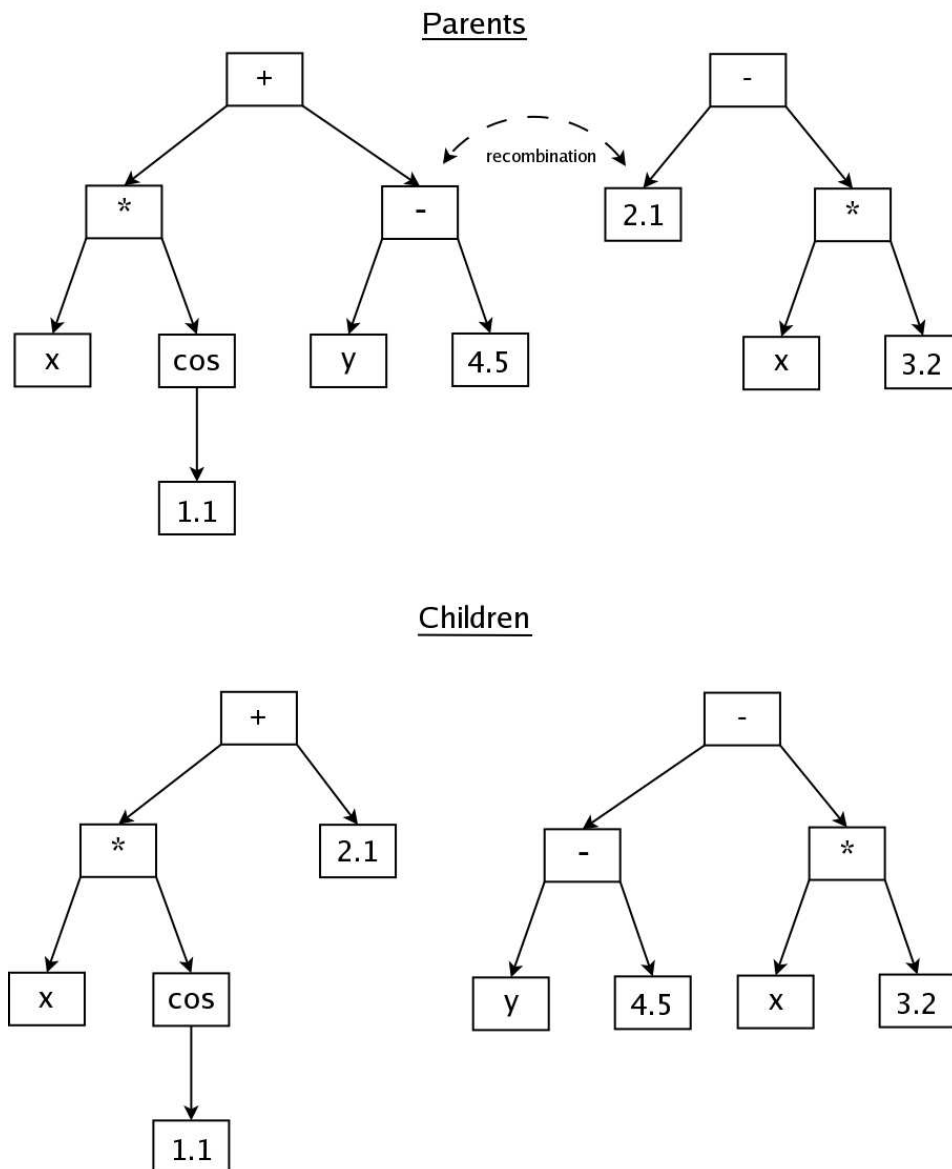


Figure 3.2: Example of recombining parse trees. Two parent parse trees exchange node branches to produce child parse trees. The operation is strongly emphasised in genetic programming systems.

- **Cycle:** A node is randomly selected and replaced with one of the same topology. So, for example, a node with two inputs can be replaced by any other node with two inputs. The rest of the network is left untouched.

For illustrations of the mutation techniques as well as a good overview of genetic programming and evolutionary computation in general, the author recommends “Evolutionary Computation 1: Basic Algorithms and Operators” by T Bäck, D B Fogel and T Michalewicz [BFM99b].

# Chapter 4

## Implementation

The implementation falls into two main areas: the evolution of the behaviours and the visualisation of the results. The evolution is controlled by genetic library which was implemented to contain a generic architecture for evolving solutions to a variety of problems. The visualisation system provides a command line viewer and a separate, more complete user-interface for viewing the results.

The core of both systems was implemented in C++. This is ideal for the evolutionary system as it is very fast, object-orientated language and evolution is a computationally expensive process that deals with units that can be represented well as instances of classes. The language also provides support for dynamic and static libraries so a class and function library can be built up for the main processes and the specimens that are to be evolved can be designed separately and dynamically loaded into the system.

Additionally the OpenGL API provides a fast way to visualise the results of the system, whilst geometry information can be exported from the simulations for presentation quality rendering of key examples [ope].

This chapter and the next detail the implementation of the various aspects of this study and the pipeline that handles the results.

### 4.1 Genetic Library

A library was written in C++ to contain methods and classes for all the areas associated with the evolution. The library handles the generation of the initial population of individuals and their evolution thereafter, and is designed to be as generic as possible with no particular emphasis on motion synthesis. The functionality covers that which needed by this study but in an open format that allows the user to design both their own specimens for evolution and the nodes to be included in the networks.

The main features are summarised below:

- **Evolution Class:** This oversees the evolutionary process. The initial values are set by the user and this class handles the rest.
- **Individual Class:** The base class for the characters in the evolution; it defines the necessary members and pure virtual methods for a class to be compatible with the system.
- **Controller Class:** Contains a node network to be evolved and the methods for querying it.
- **Node Factory:** A factory class for registering the nodes of the system. No nodes are registered by default though methods are provided for adding the standard set used in genetic programming.
- **Syntax Functions:** This group of functions provide the user with the means to generate the correct syntax for describing networks in the system.



- **Parse Tree Functions:** These functions handle operations directly on the node networks themselves. Evolutionary techniques such as recombination and mutation are carried out by these functions.

In addition the library contains classes for the traditional node set used for genetic programming.

### 4.1.1 Node Networks

The node networks are implemented as a set of node instances connected together by pointers. Each function node has pointers to its input nodes and every node has a pointer to its parent. Studies have been completed on how best to represent parse trees internally and whilst this system has the largest memory overhead and requires memory management it is also the fastest implementation discussed [KM94].

Each node inherits from the Node base class which allows a standard interface to all the nodes being used. The derived nodes then override the base class's virtual functions to provide methods specific to their inputs.

The node networks all use the same floating point data type. No other data types are used so all nodes can be connected to each other.

#### 4.1.1.1 Function Nodes

The standard function nodes implemented as part of the library were: +, -, \*, / and iflz. The first four represent the standard mathematical operations of addition, subtraction, multiplication and protected divide. The final function 'if-less-than-zero' takes three inputs, evaluates the first and if it is less than zero returns the second, else it returns the third. This allows conditional switching within the network with the advantage that the unused input is not evaluated.

#### 4.1.1.2 Terminal Nodes

The only terminal node included in the library is a constant node which is initialised with a value and returns that when evaluated.

#### 4.1.1.3 Node Factory

A node factory is used to register the nodes with the system and a set of functions are provided to make this task easy for the user. This functionality allows the user to design specific nodes for their evolution and incorporate them into the system. The new node should be derived from the base Node class. An example of a specialised node is one that measures a joint angle in an articulated figure and returns the result.

### 4.1.2 Controllers

The controllers built into the genetic libraries are wrapper classes for the node networks. The class is designed to be as generic as possible and so can be used for any purpose where the system has to return a floating point number upon evaluation. Multiple controllers can then be grouped together and implemented in any manner the user wishes.

**Multiple Controllers** The most intuitive approach to controlling an articulated figure with multiple degrees of freedom is to assign one controller per degree of freedom. In this case each controller could evolve to suit its particular task. The difficulty might be that the controllers perform independently but are judged as a group and so one misbehaving controller would bring the fitness of the whole group down.

**Single Controllers** Another approach to the problem would be to have one single controller overseeing the whole system. Different nodes in the controller would be responsible for different degrees of freedom and the entire network would evolve as a single entity. Substructures within the network could evolve to perform well at controlling different joints. A disadvantage to this approach is that successful structures within the network may, as a result of recombination, be moved to another area where their behaviour might be less appropriate. Whilst this could also happen with multiple controller configuration, the system is set so that only controllers in the same role are recombined. This means that the controller associated with the knee in one individual would only recombine with the knee controller in the other individual so successful substructures always remain linked to the same task.

### 4.1.3 Expression Syntax

In order to store the evolved controllers once the evolution has finished they are written out to a file. A syntax is used to store the controller layouts in a human readable format. The most commonly used syntax in genetic programming is the LISP S-expression syntax which is used to represent hierarchies of lists that are the basis of the language's structure.

The syntax uses brackets to enclose expressions and so represent branches in the structure. The first entry in a set of brackets is the function that governs that particular expression and any further entries acts as the parameters to that function. For example, the expression "1 + 2" is written as "( + 1 2 )". By nesting brackets additional branches can be added to the expression structure. For example, the expression "sin(3 + (4 \* 5))" is written as "( sin ( + 3 ( \* 4 5 ) ) )".

### 4.1.4 Evolution

The focus of the genetic library is the evolution class which handles the artificial evolution of the individual specified by the user. The class contains a number of members that act as parameters for the evolution dictating how the process is carried out. Fundamental parameters include the population size, the number of generations and the different weightings given to selection, recombination and mutation.

In order for the evolution to handle user designed individuals, it must be provided with appropriate creator functions and the fitness test for the individual. Once these parameters and initial settings are configured by the user, the evolution is handled by the instance of the class.

#### 4.1.4.1 Initialisation

There are two options for creating the initial population for an evolutionary run. The standard behaviour is to generate a random selection of individuals according to the population size specified by the user. The alternative is to initialise the population from a set of sample controllers; for example, promising results from one batch could be used to initialise the population of another.

**Even Distribution** The system uses a technique based on the "ramped half-and-half" method described by Koza as the best way to create an even distribution of control structures [Koz92]. The method Koza describes focuses on evenly distributing the size of the networks. He defines the sizes of his networks in terms of the levels or the network depths. If the minimum depth of the network is 2 and the maximum is 6, his method produces a range of networks 20% of which have a depth of 2, 20% with a depth of 3 and the same for 4, 5 and 6. Additionally he creates half of the networks using the "grow" method and half with the "full" method. The "grow" method naturally grows the networks letting them terminate when terminal nodes are selected or when they reach the maximum depth. The "full" method only selects function nodes for the levels which are not at the specified depth and then selects terminal nodes at the full depth, therefore guaranteeing a full tree.

The genetic library handles the limits on the node networks in terms of total number of nodes. This is preferred in some approaches over limiting the network depth, as it puts fewer constraints on the shape of the networks produced. The system creates a set of networks with an evenly distributed initial node limit between 3, which is equivalent to a depth of 2, and the maximum node limit for the evolution, which is specified by the user and is normally of the order of 100. Half of the networks are created with the “grow” method and half are created with the “full” method. In this case the “full” method is implemented by letting the network grow naturally and if it terminates before the node limit, a terminal node is randomly replaced with a function node and the tree is grown again. The replacement may be required several times, but the tree will eventually reach the node limit.

**Initialising from Samples** If the system is initialised from a previous set of individuals then the first generation is created by recombining and mutating the samples provided. The user can specify what fractions of the population are created by recombination and mutation. If the fractions sum to less than one then the remaining part of the population is randomly generated as described above. These randomly generated individuals will mostly provide a number of poor solutions that will weaken the population however they also provide much needed genetic diversity and might be useful in finding better solutions.

#### 4.1.4.2 Selection

The selection method chosen for this study was tournament selection. This is recommended in literature as a method well suited to maintaining a reasonable distribution in the selection whilst still favouring the better individuals. Importantly as the selection probability is dependent on but not proportional to the fitness, it does not overly favour “super-individuals” who happen to have an uncommonly large scores. It is best to avoid putting emphasis on the selection of these individuals as they may only represent a local maximum in the fitness space and so might cause premature convergence on a non-optimal solution.

The tournament selection procedure involves selecting a number of individuals at random from the population to form a small tournament. The winner of the tournament is the individual with the highest fitness and is selected for the next phase of the population. The tournament losers are placed back in the original population and may be selected again.

#### 4.1.4.3 Recombination & Mutation

Recombination was implemented as described in section 3.1.1. The advantage of the pointer based node networks is that the genetic operators can be applied by redirecting pointers at the base of a branch. This results in the whole branch being parented to another node thus performing the operation. Similarly the mutation operator was added as described previously.

#### 4.1.4.4 Multiple Groups

The sub-grouping or “island” method for evolution is technique which is aimed at maximising the efficiency of the search and avoiding premature convergence within the population. The full island method is normally a useful way of implementing the system in parallel across many computers however that approach was not explored in this study [BFM99a]. Instead a subgrouping method was used that splits the initial population in groups that are evolved separately from each other for a number of generations before being brought back together, processed and split off again as many times as the user specifies.

The concept behind this method is that populations can tend towards certain solutions from early in their evolution and this can lead the search away from other potential solutions in the problem space. If the population is divided into separate groups which do not affect each other then the groups may evolve in different directions and so the whole population performs a more complete search of the space. By bringing the groups back together the best solutions are shared out amongst the population and the better solutions can be used to seed several more groups for another stage of evolution.

#### 4.1.4.5 Crowding Method

The name is possibly misleading as the method discourages crowding as opposed to promoting it [BFM99a]. Like with the subgrouping method, the aim of the technique is to reduce early convergence of the population and so encourage a more thorough search of the problem space. The method is implemented as a replacement for the selection-recombination-mutation scheme described so far although it still uses these evolutionary operators under a different framework.

The approach involves randomly pairing members of the population together. These parent pairs are then recombined to form children which are in turn subjected to mutation. The children are then paired off, one to each parent, and the fittest member of each pair is let through to the next generation. This maintains a constant or improving fitness level within the population.

Diversity is encouraged by the way in which the children are paired off against the parents. A distance measure is performed between each child and each parent and the parent-child combination that results in the smallest total distance is used. The distance is a measure of the diversity of the individuals, with a larger distance representing a greater difference between those specimens. So by choosing the combination with the smallest total distance, the children are compared with and compete against the parent which is most similar to themselves. As only one member of each pair goes through to the next generation, the similarity is effectively removed as much as possible.

The distance between the individuals is measured as the Euclidean distance between their fitness vectors, which are the set of numbers relating to the different aspects of the fitness test. The vectors are  $n$ -dimensional where  $n$  is the number of reward or penalties assigned when measuring each individual's fitness.

## 4.2 Physics Engine

The physical simulation of the characters for the fitness tests is an essential part of the evolution. Implementing an accurate and stable physics engine is a difficult task in itself and outside the scope of this study. For this purpose the Open Dynamics Engine (ODE) was used for the physical simulations. This is a fast and reliable open source physics engine with support for joints [Smi]. The ODE provides a C-API for the creation and handling of the objects in the physics environment as well as a set of functions for querying their properties.

## 4.3 Characters

The characters used in the evolution were articulated figures with simple representations for simulation in the dynamic environment. The nature of the characters was in some ways limited by the capabilities of the physics engine which only provided support for certain collision shapes. One of the most stable collision objects was the cuboid and so the characters were built from a series of boxes.

The ODE also provides support for a number of joint types including hinge joints, universal joints<sup>1</sup> and ball-and-socket joints. Ideally an articulated figure should be made from a combination of hinge and ball-and-socket joints which would correctly simulate joints with single degrees of freedom like knees or elbows as well as joints with more freedom like hips and shoulder joints. Unfortunately whilst the simulation of these joints is excellent in the ODE, the control the API provides over their movement is limited and works best for the simpler joints.

Due to this constraint the articulated characters were implemented using hinge joints for knees and elbows and universal joints for the shoulders and hips. Whilst this reduces the realism of the simulation, the effect proved minor and still allowed reasonable motions to be generated.

---

<sup>1</sup>Universal joints are double hinge joints where the axes of the hinges are perpendicular.

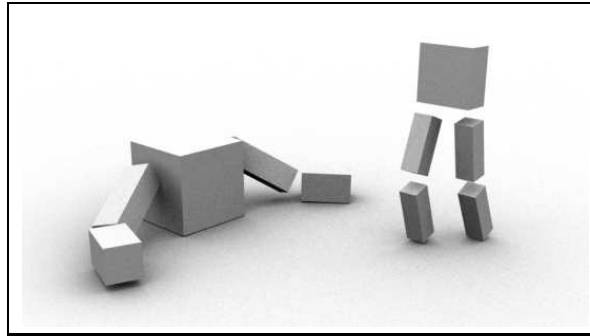


Figure 4.1: Creature and biped morphology. The character geometries are basic in order to simplify the simulation.

The two articulated characters that were used are described below. Both characters were tested with a single controller governing the motion of all their joints and with multiple controllers for which each degree of freedom was influenced by a separate controller.

Due to the nature of the physics engine, joint torques can only be added to a particular joint and not set to a particular value. This behaviour can lead to multiple torques being added to the same joint which may result in an unnatural joint strength. As such an advantage of the multiple controller approach is that the results can be clamped to within a certain range before being applied to a network. This allows additional control over the system that may be desired.

### 4.3.1 Creature

Initial evolutions were carried out with a “virtual creature” of the type that one might expect to find in Sims’ work [Sim94b]. The creature was designed to be highly stable so that solutions would not need to have a fine sense of balance in order to be successful. However, as the final aim of the project was to work with a biped figure, the same level of complexity was included in this creature in order to properly test the system.

The final creature appears as a simplified torso with arms but no legs or head. Made from five sections to represent the body and upper and lower arms on each side, the creature was connected with universal joints for the shoulders and hinge joints for the elbows.

### 4.3.2 Biped

The biped was designed to be the simplest figure that could display bipedal locomotion. As such the first version was made from five cuboids: one for the torso or pelvic region and two for each leg. Universal joints were used for the hips and hinge joints for the knees.

Later stages of testing experimented with the addition of feet to the biped model. They were initially left out in order to keep the complexity of the figure to a minimum however they were included later as two additional cuboids attached with hinge joints to the end of the lower legs. Again a more complete representation of the ankle is avoided to reduce the complexity of the simulation.

### 4.3.3 Specialised Nodes

Additional nodes were created for use with particular characters acting as their senses in the simulated world.

- **Joint measure node:** The joint measure node gives the character information about the joints in its body. Each node is assigned to a particular joint angle within the body and so would either measure the angle of a hinge joint or one of the two angles in a universal joint.
- **Joint set node:** The joint set node applies a force to a particular joint. Like the joint measure node, it is initialised to work with a particular joint angle within the body. The node is an example of a side effecting node as described by Spencer [Spe94].
- **Contact node:** The contact node deals with the character's contact with the ground plane. If contact is detected between the creature's hands and the ground or between the bipeds feet and the ground then the node returns one, otherwise it evaluates minus one.
- **Balance node:** This node is aimed to give the biped character some sense of balance. The node finds the positions of the character's feet and returns a component of the vector between the average of the positions of the feet and the centre of mass of the characters main body segment.
- **Time node:** The time node contains an internal counter that is incremented and returned each time it is evaluated. The main purpose of such a node is that in combination with the sine and cosine nodes it would produce a steady oscillating signal which would be beneficial for developing repetitive motion.

Further to these specialised nodes several additional mathematical nodes were implemented as recommended in the literature [Spe94, Sim94b]. These include sin, cos, max, min and fmod (floating point modulus.)

## 4.4 Components Library

Character creation for this study involves a certain amount of duplication of code, so to avoid this a components library was created that contained the main classes used for creating the characters and running the simulations. The library was used both for character design and the visualisation systems implemented for viewing the results of the evolutions. The main contents of the library is briefly described below.

- **Box Class:** A structure for grouping together the box ODE physics object and methods for drawing and querying the cuboids that were used as body segments for the characters in the simulation. Contains methods for drawing to OpenGL and exporting as OBJ and RIB.
- **ICharacter Class:** The pure virtual interface for the characters in the simulation. This class is derived from the Individual class in the genetic library and acts as the interface for dealing the characters classes that are dynamically loaded into the visualisation systems from the character plugins.
- **Character Class:** Derived from the ICharacter class, this provides a number of default methods that are common to all the child classes.
- **Input File Class:** The system settings are most easily configured by reading in an input file at run time. This class is implemented with all the necessary parameters for the evolution and the physics engine.
- **Option Parser Class:** Command line options are useful of configuring a program at start up and are necessary for being able to specify configuration files and character files to use in the system. This class implements a command line option parser using the GNU getopt function with default values for all the necessary parameters.

## 4.5 Fitness Function

The fitness function for a particular problem can be easy to program but hard to get right. Motion synthesis is no exception to that. A number of papers refer to the difficulty of specifying a suitable function that describes to the computer what the user is after [Gri99, TM00, Rey94, Sim94b]. Ironically it is power of the evolutionary techniques that make this part of the implementation so difficult. A function that the user might think describes the solution accurately will frequently contain a number of loop holes that the system will exploit.

The correct weighting of the fitness function can also be a difficult task. Massey and Taylor emphasise the problem of having fitness functions that are too strict early in the evolution and so cull out many potential solutions that may evolve to give promising results later on [TM00].

In this study the fitness function can be divide into two sections. The first is the simulation and the second is the calculation of the fitness value from the results of the simulation. The calculation is the weighted sum of the statistics gathered during the simulation. Gritz recommends phasing in some of the weights over the course of the evolution using a phase factor,  $\phi$ , which is given as:

$$\phi = \sqrt{\frac{g}{G}} \quad (4.1)$$

Where  $g$  is the current generation and  $G$  is the total number of generations in the evolution [Gri99]. This allows some of the factors in the fitness function to be introduced slowly throughout the evolution so that the individuals are slowly guided in the right direction rather than being forced there from the first generation.

## 4.6 Visualisation

Two approaches to viewing the results of the evolutions were implemented. Firstly a command line program that runs an OpenGL viewer and secondly a more complete multi-screened user interface with navigation tools for loading up different results. The two systems are described below.

### 4.6.1 Command line

The command line tool uses OpenGL and Glut to create a window and render a view of the character's actions. The result is a small and efficient program that can be quickly loaded to examine a single result from the evolution. The program statically links to the necessary libraries and loads the character plug-in specified in the configuration file.

The OpenGL viewer provides camera control for examining the character's motion and command line options allow for exporting OBJs. The advantage of a command line utility is that, whilst it is less user friendly, it can be wrapped in a script and integrated smoothly in the system's pipeline.

### 4.6.2 Graphical User Interface

The GUI consists of nine OpenGL view ports that each display a member of the selected run. The layout allows quick review of a number of different characters and provides a easy interface for selecting different results. The GUI was implemented using the wxWidgets cross-platform development API [Sma]. A list of the program's features is given in appendix B.3.

# Chapter 5

## Pipeline

Evolutionary systems process a large amount of information and in order to track and review this information it is best to set up a system or pipeline to ease the task. The basic pipeline used for this study and the pipeline syntax concept was designed by Peter Bowmar. The pipeline is designed with Houdini in mind and so compliments this software package setting its environment correctly for best possible use.

The main pipelining tasks for this project involve the organisation of the data produced by the evolutions for storage, review and visualisation. This part of the pipeline was scripted by the author of this study.

### 5.1 File Structure

The file structure consists of two main sections called the job root and data root. The job root contains the research, tools, source code and programs for the project and the data root contains everything created by the programs in the job root. The job section represents the content produced by the system designer and the data section contains the content produced by the computer.

An advantage of this set up is that the file structures for the two sections can be designed separately and with their particular roles in mind. This allows the job section to be divided into an intuitive and easily accessible structure based on the separate areas of the project whilst the data root can be a much more systematic and complex structure which is dealt with entirely through scripts and programs which allows the user to easily navigate the information.

#### 5.1.1 Pipeline syntax

The data section of the file structure has an organised format with a hierarchy of folders to divide the information into useful sections. The hierarchy has eight levels and files are only stored in the lowest level. The structure is designed for an animation and so normally the levels are: job, sequence, shot, element, sub-element, version, resolution and format. However more specific levels were chosen for this study and are described below.

- **Job:** The name of the project (motionSynthesis).
- **Sequence:** A section of the project (research, development, analysis, characters).
- **Category:** An output category for the project (evolution or visualisation).
- **Character:** The character type featured (biped, bipedbalance, creature, etc)



- **Batch:** The evolutionary batch (batch-001, batch-002, etc).
- **Run:** The run within the batch, also includes “data” which contains the information relevant to all the runs in the batch (data, run-001, run-002, etc).
- **Section:** Specifies the type of output in the evolutionary folders (syntax, statistics, raw output) and the type of data within the data folder (program or config).
- **Format:** The file type for the data contained (binary, text, tiff, ifd, obj).

A rigidly specified folder structure allows a pipeline to be developed with programs and scripts that read and write to this structure. To simplify the handling of the structure a syntax is developed that allows the information to be searched and handled efficiently. The syntax provides a representation for the structure that all the scripts and programs can handle and process. The “pipeline syntax” sets out the structure as follows.

Job:Sequence:Category:Character.Batch:Run:Section:Format

The syntax offers a more generic specification than hard coded directory paths and only requires the relevant scripts and programs to be aware of the root of the data structure which can be made available to them in an environment variable. This also means the data section can be moved to a different location and only the environment variable needs to be updated for the system to work.

Another key part of the syntax is the use of the wild-card character. The “+” symbol is interpreted as a wild-card by the scripts that use the syntax and so multiple sections of the data structure can be matched by a single expression.

## 5.1.2 Scripts

A number of scripts were created to smooth the work flow based on the pipeline and its syntax. They vary from short scripts that automate the processing of simple tasks to longer more in-depth tools for analysing the data produced. A few small scripts were implemented using the tcsh language however the majority of tools for this project were scripted using Python [vR].

The basis for the three pipeline scripts described below is a custom made Python module which provides a “pParser” class to aid in parsing the pipeline syntax. An instance of the class can be created within a Python script by initialising it with the desired syntax. The pParse object can then be queried for information about the syntax and directories it matches.

### 5.1.2.1 Pls

This script is the pipeline’s equivalent of Unix’s “ls” command. Run by itself it will list, in syntax format, every directory in the data structure. However if run with an syntax formatted argument it will list all the directories in the data structure that match the string provided. If no wild-cards are used then it will return the single directory that the string matches, if one exists. If wild-cards are used then it will return all the directories that match the input syntax expression. The command:

```
pls motionSynthesis:+:visualisation:+.+:+:+obj
```

Returns a list of all the directories in the visualisation part of the job’s data structure that have a format specified as “obj”. This allows fast access to all the OBJ exports from the simulations. The command includes several flags including: “-p” which returns full file paths instead of pipeline syntax, “-f” which

also lists all the files in each of the returned directories, “-s” which gives the size of each directory listed and “-e” which indicates if the returned directories are empty.

This is useful but given that the syntax is tiresome to type out every time the productivity is greatly increased by using Unix’s functionality for creating aliases. The aliases used for this project are listed in appendix C.1, and include amongst others “pcd” which allows the user to change to a directory specified in pipeline syntax.

### 5.1.2.2 Pcp & Prm

Some commands require more functionality that can be achieved by aliasing others. For this reason “pcp” and “prm” commands were scripted that allow similar functionality to the Unix “cp” and “rm” commands. Prm can be used to remove directories matching a syntax expression; giving the user options for interactive and verbose outputs. Pcp is used to copy the contents of directories to new locations. If the format specification of the new location is different to the original then the script attempts to convert the contents of the directory to the appropriate file format using the Houdini iconvert and gconvert tools for convert images and geometry.

## 5.2 Evolution

A wrapper script was created for the initialisation of multiple evolutions. The script sets up the relevant batch and run directories as well as copying over the current configuration file, libraries and character source code corresponding to that used in the evolution. An argument specifies the number of evolutions to run, each of which is run with the appropriate command line settings. Additionally the output from the various evolutions is pipelined to the correct place.

The script creates the specified number of child processes, one per evolution so that the script can execute multiple commands without waiting for each to finish. Each child process waits for the end of the evolution and then filters the output using the “grep” command line utility placing all the statistic information from the evolution into a separate text file. The full statistics gathering process is described below.

### 5.2.1 Statistics

Whilst a clear indication of the success of a run can be seen from the results it produces, further information can be attained by gathering statistics from the evolution. The parameters of the fitness function are of particular interest, especially if one can find out the affect of different constraints on the final fitness. To this end, sets of results are printed out from the evolutionary system and collected in a file under the headings of “Stats” and “Penalty”. These headings allow the information in the final output file to be run through the “grep” Unix utility and sorted into separate files which are then processed with another Unix program “awk”.

#### 5.2.1.1 Awk

This utility provides a script-based interface for extracting information from text files. It works by treating each line of the file as a record and each word in the line as a field. Scripts can then be built up that manipulate these records and fields to extract useful information such as maximum values or averages. The scripting language enables the user to build up a complex analysis of the text file if desired. The Awk script used for processing the statistical output of the evolutions is included in appendix C.2.

## 5.2.2 Text User Interface

A text user interface (TUI) was scripted in Python using the Curses module. The TUI allows for quick navigation of the data structures used to store the evolutionary information. Early versions were used to launch the character viewer program for reviewing the results of the evolutions however that functionality was depreciated in favour of a separate program that allowed more options. The TUI continued to be a fast and effective way of tracking progress and checking information on previous batches. A screen shot of the interface is given in appendix B.4.

## 5.3 Visualisation

Quick visualisation of the results from evolutions can be achieved with the characterVis script. This accepts a pipeline syntax expression as an argument, which should specify a directory that contains the character information from a particular evolutionary run. The script runs the command line visualisation program, described in section 4.6.1, on the top result of the last generation in that run.

The script is necessary to oversee the creation of relevant directories in the data structure for OBJ exports from the simulation. It also sets the execution environment for the visualisation program which includes setting the library path environment variable so that the appropriate library for that evolution is linked to the program when viewing the results. This is an important consideration as new version of the library were developed throughout the study and, whilst every effort is made to prevent it, sometimes the changes are not backwardly compatible. To limit problems all evolutions were stored with the relevant version of the libraries which are then used to run the program in the future.

### 5.3.1 Rendering

In order to produce high quality renders of the results it was necessary to use a commercial renderer. Initial investigations into exporting data from the simulations explored RIB exports for rendering with Pixar's Photorealistic RenderMan and OBJ exports for importing into a standard animation package and rendering from there. Due to the functionality provided by an animation package, OBJ exports were used that were imported into Houdini for rendering with Mantra.

#### 5.3.1.1 Houdini Digital Asset

Houdini allows for the streamlining of pipelines with its digital asset functionality. This allows a network of nodes with the software to be grouped together and implemented as a single node asset with user defined controls. In this case an asset was created that allows the user to specify a character, batch and run and it would automatically link to the correct OBJ files and provide a specific camera, light and rendering setup for that result.

Further to these basic requirements the asset provides a ghosting mode for simultaneously displaying a series of stills of the character at specific points in its motion. The separation of these ghosts can be specified and the asset calculates which ones should be visible based on the characters current position and progress along the time line. Additionally a ground plane is provided for the visualisation that can be toggled on or off.

# Chapter 6

## Results & Analysis

The results of the evolution are an intriguing and often amusing part of the process. The technique produces natural looking movements however their efficiency and purpose are frequent not as desired.

The early stages of the study focused on the virtual creature with aim to evolve a motion controller for moving forward as far as possible within the simulation time. The knowledge that was gained was then transferred to the case of a biped and an attempt was made to produce controllers for a walking behaviour.

Towards the end of the study additional work was done to look into balancing behaviours for the biped and interactions between the creature and a ball with varying results.

All demonstration videos that are referenced can be found in the /Demos directory of the accompanying CD.

**A note about units:** The ODE does not use units and so no meaningful parallel can be drawn to proper SI units for this study. As a result, distances and forces are given as unitless numbers.

### 6.1 Learning Curve

As discussed before, designing the fitness function for any problem is a difficult task. Whilst over time a certain level of skill is developed, a large number of batches were processed before significant progress was made. The discussion presented in this section focuses on the successful results and tries to give some idea of the learning process involved.

A number of factors need to be considered when attempting to evolve a behaviour. The main considerations are the fitness function, the evolutionary parameters and the design of the character being used. The fitness function is the factor with the most obvious effect but the rest of the system must also be properly configured in order to achieve optimum results.

### 6.2 Fitness Function

The fitness function, as discussed in section 4.5, is the method by which an individual in the population is judged and assigned a fitness value dependent on their performance. The functions used in this study are not complex in form; they required a section for running the simulation and then a calculation of the fitness based on the measurements made during the simulation. The fitness is calculated using the equation:

$$fitness = goal - (\phi \times penalty)$$

Where the *goal* factor is a measure of the success of the individual at achieving the main goal of the test,  $\phi$  is the phase factor, given by equ. 4.1, and *penalty* is a weighted sum of various constraints applied to the simulation. The constraints are used to reward and penalise different features of the behaviour depending on what is desired. Full examples of the fitness functions used are given in the appendices.

As the simulations required for the fitness tests are the most computationally expensive part of the evolution it is important to make them as short as possible. The fitness tests in this study were run with a maximum time limit of 40 seconds, however it is safe to say no fitness test reached this limit. The reason for this is that efforts are made to exit the simulation as soon as the characters start displaying unacceptable behaviour. Examples of such behaviour are facing the wrong direction, falling over, jumping to high or remaining inactive for a number of seconds. These checks all help to minimise the processing time required to test the individuals. Some of the events will result in the character failing the test whilst others, like being inactive for a period, are less serious but can still be used as an indication that the simulation should be stopped as the characters are unlikely to display further desirable behaviour.

### 6.2.1 Constraints

Two forms of constraints were used within the fitness functions of this study. Hard constraints represent limits on the behaviour that must be satisfied in order to pass the fitness test, whilst soft constraints are used to guide the behaviours in the right direction by proportionally rewarding actions that display the desired features and penalising those that do not. If a character fails a test then they are given a fitness score which is below the minimum achievable by a character that passes the fitness test, making them the least fit individuals in the population. The hard constraints are used sparsely and only to define regions of the solution space that exhibit unacceptable behaviour; often called infeasible solutions in the literature. Soft constraints should be used to indicate which feasible solutions are more preferable than others.

As an example, it is desirable that a character that is meant to be walking maintains a height at which its feet can reach the ground, it is unacceptable if the character reaches a height greater than twice that which it needs to touch to the ground. So a hard limit would be placed at twice the characters height, and a soft constraint would be implemented that rewarded it for staying closer to the ground.

If a hard constraint is specified that it too harsh on the character then sometimes it will cull out potentially good solutions. However the presence of hard constraints is good way to narrow down the number of feasible solutions to the problem which in turn helps to focus the search to those solutions with more potential.

The soft constraints were formulated by tracking a particular parameter of the simulation, such as the height of the character, and then penalising the character based on the value of that parameter. The penalties were calculated using a mapping of the recorded value into a penalty range. The mapping function is of the form:

$$penalty = Map(parameter, parameter_{max}, parameter_{min}, penalty_{max}, penalty_{min})$$

Where  $parameter_{max}$  and  $parameter_{min}$  define the range over which the recorded value is considered and  $penalty_{max}$  and  $penalty_{min}$  define the range of the penalty values. The function linearly maps the parameter range to the penalty range and returns the point in the penalty range that the value corresponds to within the parameter range. The value is clamped to within the parameter range so that the penalty is limited. To illustrate, the maximum x-component of the character's position during the simulation might be recorded and then mapped with the following function:

$$x_{penalty} = Map(x_{max}, 8.0, 3.0, 1.0, 0.0)$$

The results would be that if the  $x_{max}$  value exceeded 8.0 it would be clamped to 8.0 and would return a penalty of 1.0. If the value of  $x_{max}$  was less than 3.0 then it would be clamped to 3.0 and would return a penalty of 0.0. If however the value of  $x_{max}$  was within the parameter region for example, 4.3 it would then be mapped to the corresponding penalty, in this case 0.26.

### 6.3 Evolutionary Parameters

The evolutionary system has a number of settings and parameters that can be configured to change the nature of the evolution. The parameters are set using a configuration file, an example of which is given in appendix B.2. The different settings and their effects are summarised below.

It must be noted that the duration of this study is not enough to perform a proper analysis of these parameters. The evolutionary settings and other factors were frequently quite different from batch to batch as every possible effort was made to improve the results. No time was available for the careful testing of the effect of each parameter in isolation. This would be appropriate research for a longer project as discovering the optimum values would greatly improve the results of the evolution.

**Population Size** The population size directly relates to how well the evolution explores the problem space. The larger the population, the more complete the search and the more likely the evolution is to produce the optimum solution. However the processing time for an evolution is also proportional to the population and so a compromise must be reached when deciding on the population.

Sims uses a population of 300 for his work. Gritz performed a series of studies and concluded that the optimal population for his work was 700, with only negligible benefit from exceeding this value. Spencer used a population of 1,000 though does not provide any discussion on the subject.

The first evolutions for this study were initialised with a population of 300. The results were not promising but a number of other factors may have effected them. As the study progressed the population was increased in the hope of achieving better results. The creature evolutions were largely completed with a population of 1,000. Using significantly more than this did not appear to produce more promising results whilst using less often resulted in a lower quality batch.

For the biped evolutions the population was increased to 5,000 for the walking behaviours and 3,000 for the balancing behaviours. These produced good results whilst taking an acceptable time. Initial tests run at lower populations and later studies of the effects of reducing the population evolved much poorer individuals. Figure 6.1 contains an analysis of the results from batches with different populations. The graph indicates that the average distance covered by the bipeds was strongly effected by the population size, showing a steady increase up to a size of 5,000. A single extra test was performed for a population of 10,000 and the results returned were in fact worse than the batch with 5,000. This should not be the case and is more an indication of the role that chance plays in the evolutionary process than any other possible trend. However it illustrates that using a massively larger population might not have been that beneficial to the results of this study. The test was not repeated due to the extreme computational expensive of such an evolution.

**Generations** All evolutions were run for 100 generations. This value is much the same as examples given in literature. Larger values were not tried due to time constraints and whilst lower values were not attempted as studying examples from lower generations and analysing the statistics from the evolutions indicated that the fitness of the individuals increased through the majority of the 100 generations even if only by a little towards the end.

**Tournament Size** A tournament size of 6 was used for initial evolutions. Genetic programming literature indicates a size between 6 and 10 is most appropriate. Analysis of the final generation of evolutions shows that the top ten individuals behaved almost identically even for the smaller populations, this was taken as an indication that the population had converged considerably by the final generation and so the tournament size was kept relatively low to minimise the selection pressure and convergence rate. However in later stages a larger tournament size of 12 was tested with very promising results. A further test with a tournament size of 18 did not show any more improvement so the size was left at 12.

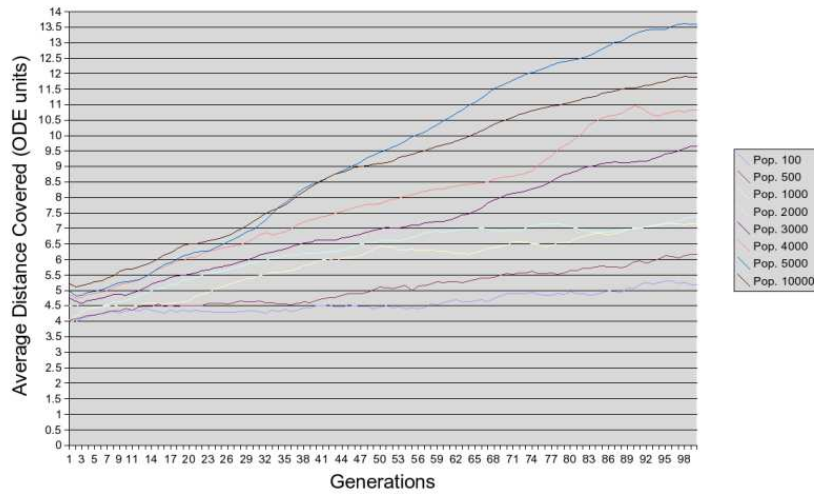


Figure 6.1: Graph to show the effect of population size on the average distance covered by biped walkers

**Node Limit** No information was found in the literature for recommending the number of nodes to allow in the controller networks. Given the complexity of the desired actions an estimation was made and the node limit was set to 150. This was not varied during the study.

**Recombination & Mutation Fractions** Koza's work emphasises the importance of the recombination operator and so between generations he uses 10% reproduction and 90% recombination with mutation playing a very minor role. The mutation operator is generally applied to the whole generation after reproduction and recombination and has a small chance of effecting each individual. Gritz reports that he uses these values as well. The implementation of this system involves specifying a mutation fraction as well which was kept small in accordance with the literature. The final levels used for all the evolutions were 10% reproduction, 80% recombination and 10% mutation.

**Island Method** The island, or multiple group, method was testing for a series of evolutions and did not have any noticeable effect on the quality of the results.

**Crowding Method** The crowding method had a clear impact on the results as it successfully promotes diversity in the population. The first test with the method used crowding for all 100 generations and the top ten individuals produced displayed very different behaviours. This is notably different to the standard evolutions in which the top ten individuals exhibited the same behaviour. This impressive diversity was not matched by high performance as shown in figure 6.2. The crowding technique seems to halt the evolutionary progress for the generations that it is used for and so only serves to delay the evolution.

It should be explained that whilst the crowding method should only increase the average fitness of the population, due to the penalty factors being phased in during the evolution the fitness can appear to fall if no significant progress is made, as consecutive generations are judged more and more harshly. These results should not be seen as conclusive of the effectiveness of the crowding method in evolutionary systems however for the particular framework of this study nothing is gained from the technique. This is possibly because the crowding method lacks the selection pressure of the more traditional selection-recombination-mutation approach and so does not promote the stronger individuals in way that is of great benefit to the whole population.

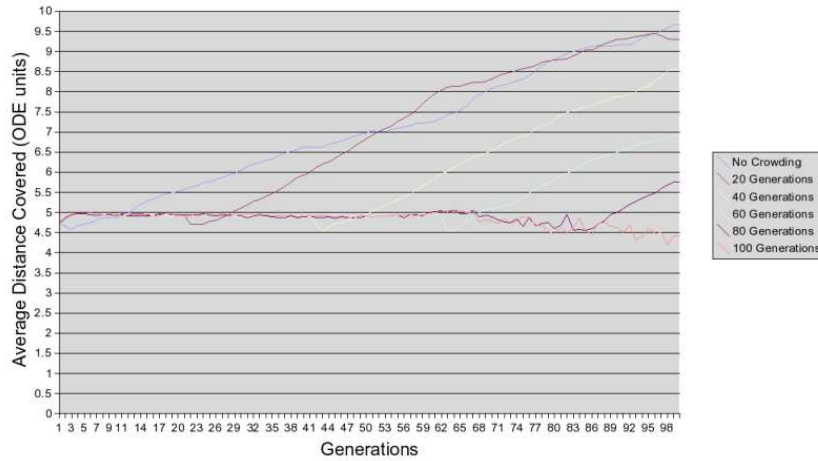


Figure 6.2: Graph to show the effect of crowding on the average distance covered by biped walkers

## 6.4 Character Design

In this case the character design refers to the number of degrees of freedom in the character and how they are restricted. Initially both the creature and the biped had a large amount of freedom in their joints. The joint angles were limited by the collisions of the character's geometry and necessary joint restrictions such as only allowing the knees to bend in one direction. This gives the characters a realistic amount of freedom however it allows them to move in ways that are not needed for the desired behaviours. This means the evolutionary system must search through a much larger number of movements and allows it to evolve behaviours that lie outside the desired range.

Whilst joints limits were not initially implemented for the characters, when they were tried the experiments showed a large improvement in the quality of the results. The joints were limited to only the minimal ranges needed for the desired behaviour. Table 6.1 gives the joint restrictions used in the case of the walking behaviour for the biped with feet.

## 6.5 Specialised Nodes

The custom nodes designed for the character controllers were described in section 4.3.3. During the study various node combinations were tested to see the effects on the success of the evolutions. Different approaches were taken in literature: Sims uses a large number of custom node types including oscillating nodes and derivative functions, Spencer also investigates the use of oscillating nodes and concludes that they are beneficial, whilst Gritz focuses on sensory nodes for judging different aspects of the simulation and only uses one extra function node: if-less-than-zero. All studies include the ephemeral constant node.

| Joint               | Limit (Degrees)           | Limit (Radians)               |
|---------------------|---------------------------|-------------------------------|
| Hip - Front to back | $30 \geq \theta \geq -30$ | $0.52 \geq \theta \geq -0.52$ |
| Hip - Lateral       | $5 \geq \theta \geq -5$   | $0.09 \geq \theta \geq -0.09$ |
| Knee                | $0 \geq \theta \geq -40$  | $0 \geq \theta \geq -0.70$    |
| Ankle               | $20 \geq \theta \geq -30$ | $0.35 \geq \theta \geq -0.52$ |

Table 6.1: Joint limits for the biped walker



This study found that Gritz's approach was the most successful for this problem. The best results were evolved using the standard function set with the if-less-than-zero node in combination with the contact, joint measure and balance sensory nodes. The inclusion of oscillatory nodes such as sine and cosine was tested at the recommendation of industry professionals, see appendix A.1, however only reduced the quality of results. It might be reasonable to believe that such nodes need a steadily increasing inputs to allow them to oscillate properly however additional tests were completed with a node that return the sine of the simulation time and still the results were significantly worse than with just the standard node set. Further nodes such as min, max and fmod also failed to aid the evolutions so the function set was kept to that described by Gritz.

## 6.6 Creature

The virtual creature tests were designed to be simple and relatively achievable for the evolutionary system. The creature was designed with stability as a focus so the controller would not have to learn any precise movements in order to be fit. Instead, any motion that in some way propelled the creature in the desired direction would be appropriate.

### 6.6.1 Multiple Controllers

Initial tests were completed with multiple controllers for each creature. The output from each controller was originally unclamped allowing the raw value to be applied to the joints. An outline of the progress is given below where each set refers to a group of one or more batches that share similar traits.

**Set 1** A simple fitness function that focused only on the motion in the z-direction resulted in a bizarre set of behaviours. The creatures generally flailed their arms in a seemingly random fashion which they had learnt would move them in the z-direction. Whilst this demonstrates an evolutionary success the behaviour should be discouraged as it does not appear natural. The resulting motion would normally involve leaping off the ground for long periods at a time.

**Set 2** In order to curb the random behaviour a hard constraint was placed on the height so that if the creatures exceeded a certain height they were failed. This successfully culled out the jumping behaviours that plagued the first set. Additionally the joints were clamped to a torque of 500 units which stops the creatures from generating huge forces with their limbs.

This set moved in a much more constrained manner, keeping close to the ground but with largely random motions. Noticeably the creatures had no tendency to face the direction of travel.

**Set 3** An additional check was added to the fitness function to test the creature's local forward direction against the world's z-axis. The check involved taking the vector dot product of the two directions and a hard constraint was placed on the results so that the creature failed if the dot product returned less than zero. This removes any creatures that turn to face more than  $90^\circ$  from the z-direction.

To improve the behaviour a further soft constraint was added to benefit creatures that tended to face the correct direction. At various intervals the direction check was performed and the smallest value, indicating the largest angle found during the simulation was stored. The value could then be used to reward those creatures that varied only a little from the desired direction.

**Set 4** The final additions to the fitness function were implemented to try to stop the creature from straying too far from the z-axis along which they were meant to travel. The greatest distance reached from the z-axis was recorded and subjected to a soft constraint that penalised large deviations.

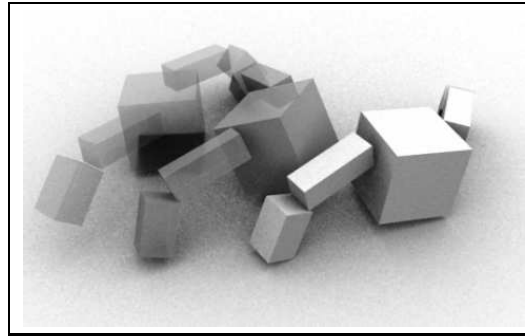


Figure 6.3: Example of a multi-controller creature's motion. The distance covered is not large however the movements are realistic and show potential for further development.

**Result** These constraints all help to guide the behaviour of the creature to an acceptable form of movement down the z-axis. The result is an intriguing motion that looks reasonably natural for the model. No creature evolved a continuous crawling behaviour or covered a large distance however by the end a large fraction showed a clear intention to move forward and were able to cover a short distance comparable to their arm span. Examples of the behaviour are illustrated in figure 6.3 and the fitness calculation is included in appendix B.1.

The knowledge gained from these experiments was used to evolve a creature that was controlled by a single network to explore a different approach.

### 6.6.2 Single Controller

In order to allow more freedom for the evolutionary operators, experiments were carried out using only one controller that governed the behaviour of the whole body using side effecting joints as described in section 4.3.3. Evolutions were completed with the same fitness function as was developed for the multiple controller setup.

Also as discussed previously, limiting the joints in the body to the desired ranges can help to improve the results. In the case of the creature the main limitation was placed on the shoulder joints so that the creature could not lift its arms above shoulder height. This constrains the arms to moving in the region close to the ground therefore improving the chance that any motion will push against the ground plane and move the creature. The creature's elbow joints were only limited by collisions between the geometry of its body.

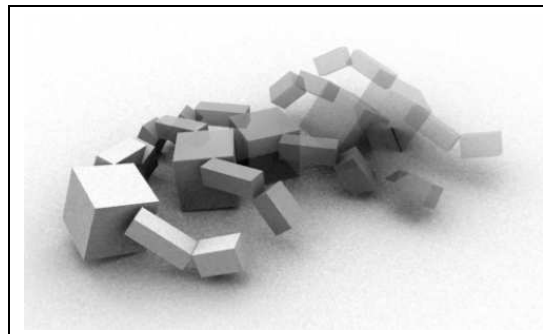


Figure 6.4: Example of a single-controller creature's motion. The creature covers a much large distance than its multi-controller counter parts whilst still using steady and relatively natural motions.

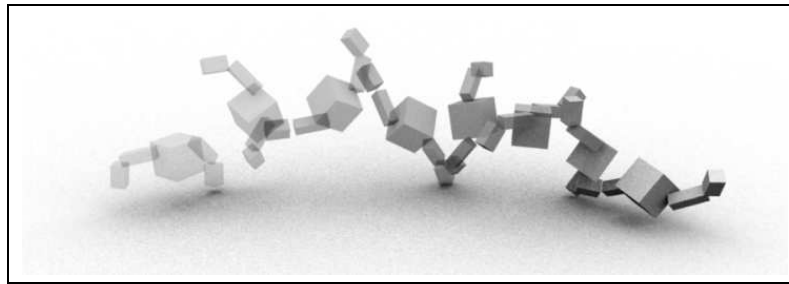


Figure 6.5: Cartwheeling behaviour evolved by one creature.

The results were promising and led to a series of more successful crawlers than were previously evolved. Examples of the creature's motion are given in figure 6.4<sup>1</sup>. One strategy of particular interest was a cartwheeling behaviour that allowed the creature to always face in roughly the correct direction perform a series of flips rotating around its forward facing axis. Such behaviour is amusing to watch and once more clearly demonstrates the success of the evolutionary system in finding flaws in the fitness function specified by the author. The behaviour is corrected by adding a check that compares the creature's local y-axis ("up" axis) with the world y-axis, the angle between them is subject to both soft and hard constraints as with the forward direction check.

Another interesting set of results were the creatures who developed cyclic behaviours. Examples can be seen in figure 6.6<sup>2</sup> in which the creatures clearly return to a similar position at regular intervals throughout the simulation. Whilst the first of those two stopped after a period, the second continued its slow progress for the length of the simulation, showing no signs of coming to rest. Unfortunately neither behaviour could be viewed as efficient or natural but were still interesting specimens.

### 6.6.3 Ball Interactions

Later in the study, further tests were completed to try to produce more interesting simulations in which the creature interacts with a ball. The fitness test was based around minimising the distance between the creature and the ball which was placed to its left. The results show the creature moving towards the ball, stopping next to it and occasionally knocking the ball towards itself with one of its arms.

The surprising feature of this set of behaviours was the accuracy with which a number of the creatures stopped next to the ball. The relatively jerky movements were brought to a complete stop when the creature was next to the ball. Whilst such a behaviour is understandable considering the fitness function, as any

<sup>1</sup>Full video: /Demos/Creature/SingleController/Sample4.mp2

<sup>2</sup>Full videos: /Demos/Creature/SingleController/Sample2.mp2 and /Demos/Creature/SingleController/Sample3.mp2

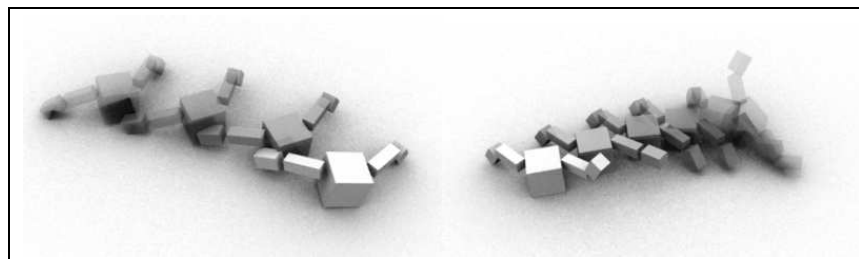


Figure 6.6: Examples of cyclic behaviours demonstrated by the creature, The ghosted motion shows the creature's position at various stages clearly indicating a trend. The movements were slow and only the right hand example continued for a long period.

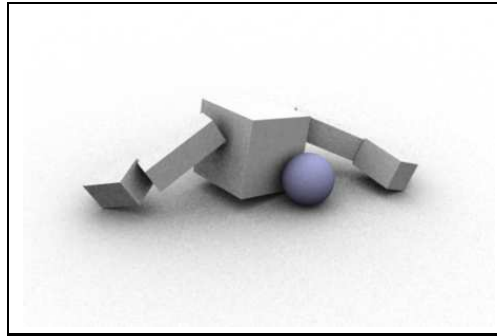


Figure 6.7: Final position of a creature interacting with a ball

movement close to the ball would likely knock it away; it was still intriguing to see the creatures demonstrate such control of their actions. An example of the rest position of one of the creatures is given in figure 6.7<sup>3</sup>.

## 6.7 Biped

The biped character was initially built, as described in section 4.3.2, as a simple five component articulated body. Feet were left out of the initial design in order to minimise the complexity of the problem. Later experiments included the feet to test what effect they would have on the success of the bipeds.

### 6.7.1 Walking

The primary behaviour investigated was a walking behaviour. The aim of the experiments, which was to produce a natural looking walk cycle, was not achieved however significant and promising progress was made. The results of the various approaches are described in the following sections.

#### 6.7.1.1 Single Controller

Drawing on the experience of working with the creature, the biped model was initially implemented with a single controller. However as discussed this can lead to large torques being generated at the joints as a network can evolve multiple nodes to effect the same joint. The result was a tendency to heroic acts of strength that propelled the biped a long distance but were unrealistic.

It is unclear as to why this problem did not occur with the creature behaviours. It is possible that the hard constraint on the creature's height was low enough to deter any jumping behaviours or it might be that large jumps often left the creatures unstable enough to fall over and fail a direction check.

The early fitness function for the biped was kept to a minimum so that a feeling for the range of movement could be gained before narrowing the scope to the desired behaviours. The only features of the initial tests were a reward for the distance covered and generous hard and soft constraints on the maximum height reached by the biped. This would allow the bipeds the freedom to perform many more interesting manoeuvres.

Further constraints were added to the fitness function and this helped to curb some of the more extreme behaviours however the results were still not promising. Consequently the multiple controller approach was soon favoured in which more control could be exercised over the torques applied. An example of the best "walk" for the single controller biped can be seen in figure 6.8, whilst figure 6.9<sup>4</sup> demonstrates one of

<sup>3</sup>Full video: /Demos/Creature/WithBall/Sample1.mp2

<sup>4</sup>Full video: /Demos/Biped/SingleController/Sample1.mp2

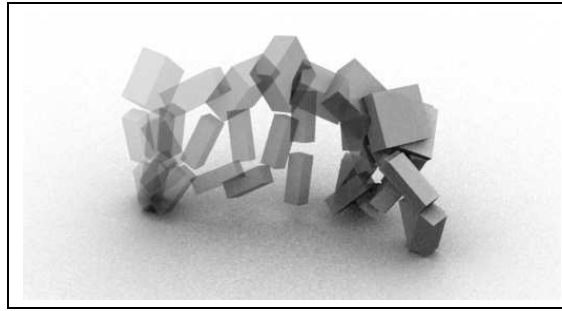


Figure 6.8: A walker controlled by a single network

the more extreme and quite acrobatic forms of movement.

### 6.7.1.2 Multiple Controllers

Evolving the biped with multiple controllers and no joint effecting nodes requires each controller to be evaluated and applied to the corresponding joint angle. The torques were clamped to within an acceptable range,  $50 \geq \tau \geq -50$ , before being applied to the joints. This range allows the bipeds enough strength to walk around but not enough to perform any exaggerated movements.

Progress was made in a similar fashion to that described for the creature. The initial fitness function is specified and then additional constraints are slowly added to counter the undesirable behaviours that are evolved. Given the past experience reasonable progress was made early on but many further investigations failed to improve on the early results. Often what seems like the wise choice for an additional constraint will significantly reduce the success of the evolution or else fail to make any impact on the results.

The process becomes quite a frustrating one at times which frequently requires recent attempts to be aborted and the fitness function reverted to the last successful state. In the end a relatively successful fitness function was developed that mostly focused on the features developed for the creature and so included the following elements.

- Distance reward: The bipeds were rewarded for moving between 2 and 20 units along the z-axis, receiving a proportional fitness increase with a maximum of  $4.0 + (2.0 \times \phi)$ , where  $\phi$  is the phase factor for that generation. This distance was chosen as it was safely beyond that which was covered by even the fittest individuals whilst still being small enough to reward slight improvements in the progress of the characters. The reward is specified using a mapping function, as described in section 6.2.1, which was:

$$distance_{reward} = Map(z_{max}, 20.0, 2.0, 4.0 + (2.0 \times \phi), 0.0)$$

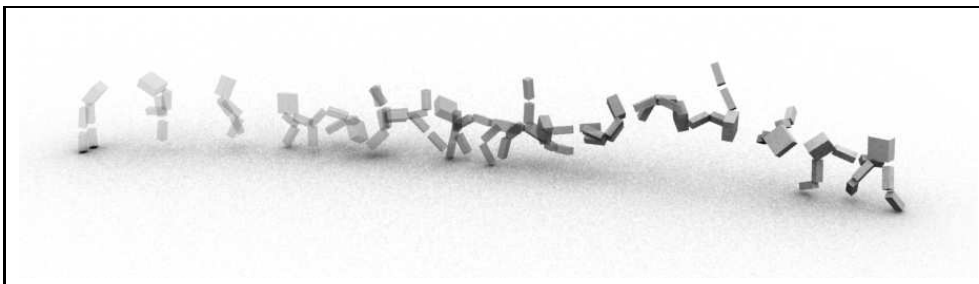


Figure 6.9: An acrobatic “walker” evolved with a single controller

**Algorithm 2** Hard constraint on the z-direction of the biped

---

```

01  $dirValue = \hat{z}_{local} \cdot \hat{z}_{world}$ 
02 if ( $dirValue < (-0.5 + \phi)$ ) then
03    $failed = true$ 
04    $exit = true$ 
05 endif

```

---

- *x*-position penalty: If the biped deviated from the line of the *z*-axis it was penalised according to the mapping:

$$x_{penalty} = Map(x_{max}, 6.0, 1.0, 1.0, 0.0)$$

- Height penalties: The torso section of the biped was initially set at a height of 6.5 units and was subject to a hard constraint at 12.0 units and an exit check at 3.5 units to stop the simulation if the biped fell over. Additionally the height of the torso was subject to a further two soft constraints with the mappings:

$$high_{penalty} = Map(height_{max}, 7.0, 5.5, 1.0, 0.0)$$

$$low_{penalty} = Map(height_{min}, 5.0, 3.5, 0.0, 1.0)$$

These help to keep the biped from going too high or stooping too low. The second penalty would actually have no effect in this study as no biped stayed upright for the entire simulation time and so all would have a minimum value of 3.5 or below as they fell over and triggered the exit condition.

- Direction penalties: As discussed before, direction checks were performed by taking the dot product of the characters local *y* and *z* axes with the corresponding world axes. The minimum recorded values were stored and used to calculate the penalty. Hard constraints on the direction were also included and their severity was phased in as the evolution progressed. The final hard constraint is given in algorithm 2. Soft constraints were also applied with the following mappings:

$$zDir_{penalty} = Map(zDir_{min}, 1.0, 0.0, 0.0, 1.0)$$

$$yDir_{penalty} = Map(yDir_{min}, 1.0, 0.0, 0.0, 1.0)$$

- Early exit penalty: The final feature of the fitness function is a penalty imposed for exiting the simulation early. The penalty uses the mapping:

$$earlyExit_{penalty} = Map\left(\frac{lastStep}{maxStep}, 1.0, 0.0, 0.0, 2.0\right)$$

The evolutions with this fitness function produced bipeds who could successfully stagger a few steps before collapsing. The full fitness calculation is included in appendix B.1. The results are convincing in their realism but did not feature smooth movements. Examples of the behaviours are shown in figure 6.10<sup>5</sup>. Whilst these results did not reach the level of a continuous walk cycle, or even close to that goal, a significant amount of time had been spent with this character and so it was decided to expand the investigation to include a model with feet to see if any improvement could be made.

### 6.7.1.3 Feet

The realism of the biped's structure was enhanced by the addition of blocks to represent feet. The feet were attached to the end of the legs with hinge joints to represent a simplistic ankle and additional controllers were added to govern the behaviour of the new joints. The first batch of these walkers appeared promising though of a similar standard to the models without feet, figure 6.11. The approach used to various parts of the evolution is described below.

---

<sup>5</sup>Example videos: /Demos/Biped/MultipleControllers

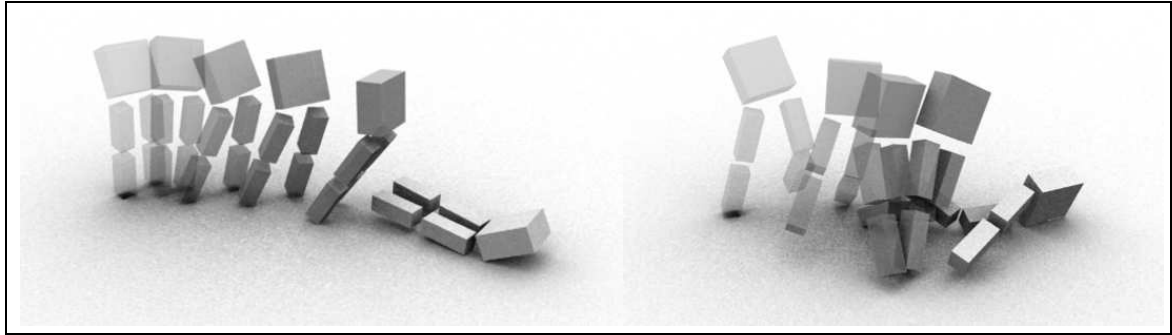


Figure 6.10: Examples of the multiple-controller biped walkers

**Fitness Function** This was kept very much the same as the case described above. The only changes involved the adjustments of the height parameters to accommodate the feet and an additional one off penalty for not lasting longer than 200 time steps in the simulation. This was added to hopefully discourage the frequent but poor behaviour of simply inching forward with a step or two and then falling over. This behaviour plagued a number of batches despite other far more successful behaviours being evolved by runs in the same set.

**Joint Strengths** One noticeable flaw in the initial batch was the ability for the bipeds to move themselves by only moving their ankle joints. The joints were strong enough to lift the entire character and move it along. This produced a series of shuffling behaviours that, whilst unsuccessful in moving large distance, appeared to be the preferred behaviour for the bipeds to evolve. Additionally if they did lift their feet from the ground they would tend to wiggle them excessively.

These two undesirable behaviours were successfully countered by clamping the joint torques for the ankles at a much lower level than the other joints. The other joints remained being clamped to the range,  $50 \geq \tau \geq -50$ , whilst the ankle joints were reduced to the range,  $20 \geq \tau \geq -20$ .

**Joint Limits** The first batches with the new model evolved promising behaviours in which the biped appeared to take a few controlled steps with appropriate foot movement to walk forward a little. However a number of these steps were quite exaggerated actions often leaving the biped stretched out and unable to return to an upright position. These behaviours were not frequent with the other model and so no action was taken to counter them however counter measures were clearly needed for this design.

In order to prevent these undesirable movements much stricter joint limits were implemented for the model. A normal human walk cycle involves only a small range of movement in the joints and so the joints were restricted to appropriately small ranges. However this proved to limit the movement too much and many individuals never managed to properly co-ordinate lifting the foot and bending the knee as the leg swings through to take a new step. The margin for error with the new joint limits was so small that the effects were counter productive. So the ranges were opened up a little more to provide scope for the individuals to learn.

#### 6.7.1.4 Final Results

The behaviours developed for the biped with feet proved to be the most successful of the project. Whilst no results displayed truly natural walking gaits a number of individuals capable of covering a large distance in a reasonable manner were evolved, figure 6.12<sup>6</sup>.

<sup>6</sup>Example videos: /Demos/Biped/WithFeet

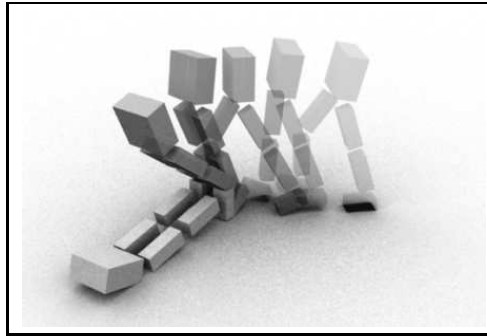


Figure 6.11: First promising result for the biped with feet

One limitation of the behaviours evolved was that the bipeds tended to move one foot forward at the start of the simulation and then shuffle forward in small steps where that one foot always remained out front and the other behind. Despite the prevalence of this behaviour the author is confident that a more accurate walking behaviour could be evolved if given time enough to tune the fitness function and evolutionary parameters.

At this point further investigation was carried out into the possibility of transitioning between behaviours and so be able to evolve a behaviour that would compliment the walking if properly blended.

#### 6.7.1.5 Further Evolution

As discussed in section ??, the evolutionary system developed allows an evolution to be initialised based on samples from another evolution. There is potential for the best of one batch to be evolved further to produce better results. However this was attempted and the system failed to improve on the initial samples. It is unclear as to why this happened though it is possible that the examples used did not include enough genetic diversity to evolve better solutions. Additionally all of the 14 runs evolved with this approach produced near identical results. Normally some runs are successful and others are not and the successful ones show a large degree of variety in the behaviours they evolve, however in this case all runs produced the same behaviour. This indicates a convergence of the evolutions onto a good solution from the initial samples and a failure in all cases to improve on it.

Further to these problems, such evolutions are much more computationally expensive as fit individuals are begin tested from the first generation, instead of the usual set of initially weaker solutions that all exit early from the simulations. Due to these poor results and the extra computational expense involved, this technique was not explored further despite the possible potential behind the idea.

## 6.8 Blending

In order to aid the walking behaviour, it would be useful to shift to a behaviour that corrected the instabilities in the cycle. Once a stable position had been reached the biped could transition back to the walking behaviour again. Whilst not producing an altogether smooth motion this might still help the walker and would demonstrate a more intelligent behaviour that responded to unstable situations.

To this end a balancing behaviour was investigated and the results are described in the following sections. In order to test the blending procedure a version of the biped model was implemented that had two separate sets of controllers. The character is initialised with the expressions for a walking controller and the expressions for balancing controller. Each time the controllers are evaluated as assessment of the current state of the character is made and if it is deemed to be stable then the walking controller is used and if it appears to



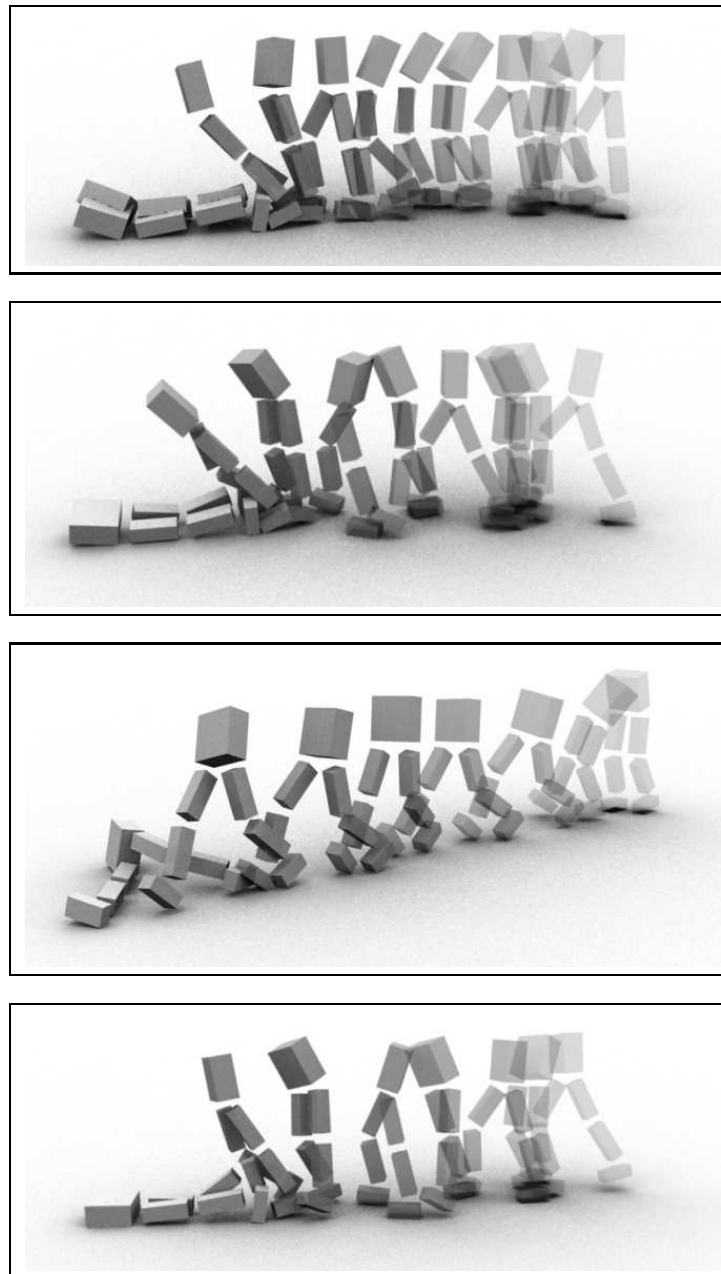


Figure 6.12: Examples of the biped walker covering a large distance

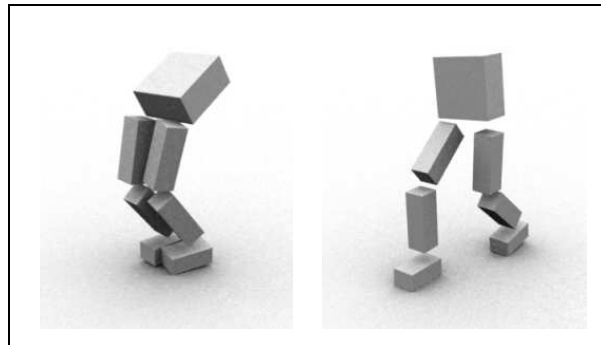


Figure 6.13: Balancing behaviours showing a simple strategy that allows the biped to sway under a small force (left) and a more natural response that requires the biped to step forward to absorb a stronger force (right)

be in an unstable state then the balancing controller is used instead. For any region of uncertainty between these two states a linear interpolation of the two controllers is used accordingly.

## 6.8.1 Balancing

In previous experiments, mistakes in the fitness function have already let to bipeds with the ability to stay standing still for long periods of time. So the ability to balance when left alone in a relatively stable position was definitely achievable. However this is not a useful behaviour as the biped must be able to recover from an unstable position, not just maintain a stable one. So instability should be forced upon the biped with the aim to evolve behaviours that counter it.

### 6.8.1.1 Single force

Initial tests were carried out with the biped being pushed from behind. The pushes were evenly spaced at two second intervals and were of constant strength. These first tests required the ideal biped to maintain its position at the origin with the hope that the biped would step forward to stop the fall and then step back to its original position.

The strength of the force was initially set to 100 units in the simulation, but this magnitude turned out to be too small as the bipeds could easily maintain there position by bending backwards in an awkward position. The stance allowed them to absorb the force and stay upright with only a gentle sway in their motion, figure 6.13a.

This test also highlighted a flaw in the fitness function, by requesting that the biped maintain the initial position of its main body section as closely as possible it evolved to arch backwards and thus reduce the distance to its origin as much as possible. Consequently the function was updated to reward the biped for maintaining the same height and not moving to either side, however it was free to move forwards if necessary.

A force of 150 units was settled on and a batch was produced that showed active attempts to stay balanced, most often by bracing the legs far apart along the line of the force to provide a stable base with which to absorb the impact. To within acceptable margins these short, simple behaviours appeared realistic and demonstrated a similar approach to what a human would do in the same circumstances, figure 6.13b<sup>7</sup>. A more extreme example is pictured in figure 6.14<sup>8</sup>.

<sup>7</sup>Full video: /Demos/Biped/Balance/Step.mp2

<sup>8</sup>Full video: /Demos/Biped/Balance/Splits.mp2

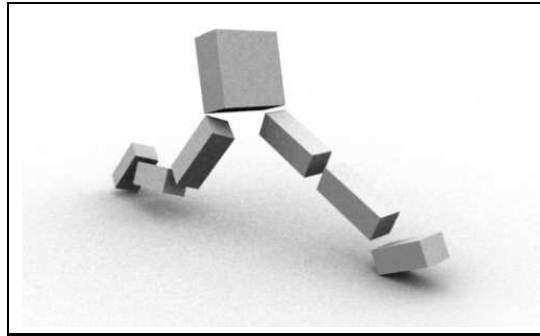


Figure 6.14: More extreme balancing behaviour

**Blending** Given this initial success an attempt was made to blend this balance behaviour with a previously evolved walking behaviour. The results were poor. It became immediately obvious that the balancing behaviour evolved was a very rigid behaviour; one that was specific to the environment created for that particular evolution and not robust enough to be used in other simulations. The result is that if the biped starts to fall and so blends to the balance behaviour the controller attempts to move the same leg that it did in the original simulation and to an extent that is not necessary or well judged. The example tested caused the biped to fall faster as the controller attempted to raise the wrong leg.

This result leads to two options. Either a large number of controllers are implemented in the same biped and an effort is made to dynamically choose the correct controller for the task, or preferably a more robust controller is evolved that can deal with different situations. The first option is unattractive as it would require a much larger and more complex analysis of the character's state in order to decide which controller to use and so would be moving too far from ideal situation of evolving the character's controllers with minimal input from the user.

The second option, creating a more robust controller, is an important concept in artificial evolution. For any problem which is not a rigidly defined situation but is subject to subtle changes it is important to develop a solution that can cope with these variations. Two methods have been proposed in the literature. As discussed in the review, Reynolds has tried an approach of adding noise to the fitness tests in order to produce individuals that can survive a more turbulent or ill defined environment [Rey94]. There is also a method described by Koza and used by Gritz which is to evaluate each individual under different conditions in several tests and assign an overall fitness which is an average of the performance across the all the tests. The choice is clear as Reynolds reports no particular success with his method and Gritz claims to have achieved reasonably robust controllers with the technique he used.

### 6.8.1.2 Multiple Tests

A fitness function was configured so that each individual would be tested four times, each time with a push in a different direction and the overall fitness was set as the average of the performance over the different tests. Due to the added complexity of the task the initial forces were reduced to 100 units each.

The majority of individuals evolved to cope well with the pushes though could handle them all with relative ease and minimal movement. The force was increased to 150 for the second batch which produced lower levels of success but still evolved number of individuals who could remain standing. Unfortunately two undesirable features of the individuals become apparent: firstly that the best way to account for all cases was to move as little as possible and just try to sway with the force, and secondly that due to the regularity of the force the bipeds learnt to rely on it for balance. This latter counter productive behaviour evolved as the bipeds could be pushed forward by the force and lean back to compensate, safe in the knowledge that if they lean back too far the next push would help to keep them upright.

A further batch was tested in which only a single force was applied to the character during the simulation. The magnitude of the force was increased to 200 so that the character should be forced to take a more active approach to balancing. Additionally the timing of the force was changed so that it was applied at a randomly chosen point in the first 80 steps of the simulation. This was designed to improve the robustness of the controller so that the characters would have to learn to respond to the force rather than immediately moving from the start of the simulation.

Unfortunately this set did not produce any more promising results, with the majority of individuals falling over when pushed in the front or back and remaining relatively stable when pushed side to side. The bipeds continued to favour staying still and swaying with the force over actively stepping to avoid falling over.

**Blending** An attempt was made to see how the more successful individuals would act when blended with one of the walkers from the previous section. The results were not acceptable, and much like before most led to the walker falling over faster than when it is treated separately.

Given the difficulties encountered it was decided that developing a successful mechanism for blending behaviours was beyond the scope of this study. The exercise was still considered worthwhile as it indicates flaws in the evolutionary techniques that were not previously considered.

## Chapter 7

# Conclusion

The aim of this study was to investigate the application of genetic programming to motion synthesis for three dimensional articulated characters in a dynamic environment.

Behaviours were evolved by describing the desired actions in a fitness function and then allowing the evolutionary process to search for the best solutions to the problem. This method is relatively unintuitive and difficult at first but once a degree of familiarity has been developed for the task it becomes significantly easier. Whilst the results are not comparable to a key frame animated figure and do not appear as natural or smooth as would be desired for a final animation, they prove that the technique has promise and that improving the quality is only a matter of time and patience.

Dynamic behaviours were evolved for a simple virtual creature and a biped model. The early stages of the study produced locomotion controllers for the creature that allowed it to cover a reasonable distance using a variety of interesting and organic actions. Single and multiple controller systems were investigated and compared. The greatest success was had with a single controller system using the standard genetic programming node set extended with a small number of sensory nodes.

The knowledge gained from working with the virtual creature was used to evolve behaviours for the biped model. The aim of this part of the study was to develop a walking behaviour and whilst no completely stable walkers were evolved, a number of behaviours that allowed the biped to cover a considerable distance were developed.

Further studies explored the possibility of blending controller systems to improve the behaviours. Experiments were completed to evolve a balancing behaviour with some success. Several natural responses were evolved that allowed the bipeds to stabilise themselves after a firm push. However on further investigation it was discovered that these behaviours proved too rigid to successfully blend with another controller set. Efforts were made to evolve a more flexible and robust controller by using techniques recommended in the literature however they met with little success and none were suitable for blending.

The work on balancing controllers and blending highlighted an important failing of the evolutionary method which is its tendency to produce brittle controllers that cannot be easily used in other situations. Considerably more work would have to be done to evolve more robust controllers for the behaviours investigated.

This study has proved a success in that it has evolved walking behaviours for a biped character and demonstrated that evolutionary systems have significant promise in this particular area. Gritz concludes in his work that genetic programming can be used to develop behaviours for relatively simple articulated figures. This work has extended that idea to include more complicated figures than Gritz investigated and clearly shows that the technique is capable of handling the additional complexity.

## 7.1 Further Work

A large number of possibilities open up when this field is examined. This study made an effort to explore the basic material and to look into one further area of interest: the blending of behaviours. However there are many areas that the author would like to study given the time and resources.

### 7.1.1 Blending

The work completed on the area of blending in this study was only a basic and unsuccessful introduction to the subject. A more thorough investigation might unveil interesting and useful results which significantly improve on the single behaviours developed here. The key to the area is certainly the development of more robust controllers which are capable of producing accurate motions under different conditions.

### 7.1.2 Automatically Defined Functions

Koza describes the use of automatically defined functions (ADFs) as building blocks for networks in genetic programming. The concept is that an evolutionary operator can be defined that selects a branch in a network and compresses it down to a single node, an ADF, that contains the branch as a subnetwork. As a consequence this subnetwork is never split up due to other evolutionary operators. This is advantageous if the subnetwork contains a useful combination of nodes that would be best kept together and so the ADF can be treated as a building block for good solutions. If a number of useful building blocks are created with this method then it becomes easier for the evolutionary process to find good combinations of these elements without the risk of splitting them up.

ADFs were not implemented for this evolutionary system so it would be interesting to investigate the effects they have on the efficiency of the technique.

### 7.1.3 Other characters

This study covers only two different character designs. It would be interesting to explore that application of the method to other designs, especially ones which lend themselves to other forms of movement. Investigations into more complex character structures would examine any possible limits on the evolutionary technique. Complicated characters were avoided in this study however it is possible that behaviours could be successfully evolved for much more intricate designs that were used here. Such studies could also include behaviours for a full biped character with legs, arms and a head.

Additionally it would be interesting to use a simulation of an underwater environment to test the effectiveness of the technique for developing realistic swimming motions. Sims' work explores the area but not with a fixed morphology. It would be interesting to see, under a physically realistic simulation, what swimming motion a biped would evolve and how it would compare to the styles used in competitions in the real world.

### 7.1.4 Matching motion capture

A final area that could be explored is using an example animation as a target for the evolution. Motion capture data of a human walking could be used to provide the ideal behaviour and the fitness of individuals could be judged by how well they match to the example data. The advantage of this approach is that the actions would not have to be so carefully described in terms of a fitness function, instead the user could rely on prerecorded data to provide the ideal solution. A disadvantage would be that the resulting behaviour would probably be very brittle and only suited to reproducing that particular animation. However once the animation had been matched well by the simulation then variations of the behaviour could be evolved

using the system. This might allow the more rigid motion capture data to be used as a seed for a number of slightly different behaviours.

# Bibliography

- [BFM99a] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Advanced Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, 1999.
- [BFM99b] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, 1999.
- [Gri99] Larry Israel Gritz. *Evolutionary controller synthesis for 3-d character animation*. PhD thesis, 1999. Director-James K. Hahn.
- [Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [KM94] Mike J. Keith and Martin C. Martin. Genetic programming in c++: Implementation issues. *Advances in Genetic Programming*, pages 285–310, 1994.
- [Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992.
- [ope] OpenGL. <http://opengl.org>.
- [Rey94] Craig W. Reynolds. Evolution of obstacle avoidance behavior: using noise to promote robust solutions. pages 221–241, 1994.
- [RH02] Torsten Reil and Phil Husbands. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Trans. Evolutionary Computation*, 6(2):159–168, 2002.
- [Sim91] Karl Sims. Artificial evolution for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, New York, NY, USA, 1991. ACM Press.
- [Sim94a] Karl Sims. Evolving 3d morphology and behavior by competition. *Artif. Life*, 1(4):353–372, 1994.
- [Sim94b] Karl Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22, New York, NY, USA, 1994. ACM Press.
- [Sma] Julian Smart. wxWidgets Cross-Platform Development Library. <http://www.wxwidgets.org/>.
- [Smi] Russel Smith. ODE: Open Dynamics Engine. <http://ode.org>.
- [Spe94] Graham Spencer. Automatic generation of programs for crawling and walking. *Advances in Genetic Programming*, pages 335–353, 1994.
- [TM00] Tim Taylor and Colm Massey. Recent developments in the evolution of morphologies and controllers for physically simulated creatures. *Artif. Life*, 7(1):77–87, 2000.
- [vR] Guido van Rossum. Python Scriting Language. <http://www.python.org>.



# Appendix A

## Correspondence

### A.1 Natural Motion

The paper mentioned is “Recent developments in the evolution of morphologies and controllers for physically simulated creatures” by Tim Taylor and Colm Massey [TM00].

---

Subject: **Project**

From: "**Rob** ——" <rob.——@naturalmotion.com>

Date: **Tue, August 22, 2006 2:52 pm**

To: **mike@———**

Got some feedback from a couple of guys here, most of our behaviour engineers are arriving back in Heathrow later today, so I'll forward you any other comments i get from them.....

Yes, Karl Sims (Evolving Virtual Creatures) used sine wave and saw wave nodes. I think they will help a lot. Also consider carefully the inputs and outputs to the node networks: Do you want the output to be straight torques, or as extension values in a spring model, or as velocity values to drive towards using torque? Try all 3 perhaps. What information do you expect the controller to need as input? Angles of limbs? Joint angles? Contact with ground information? Joint angle velocities? Linear limb velocities? An averaged orientation for the creature? There's lots of input values to chose between. But you can't use too many simultaneously as inputs or the rate of evolution will grind to a halt. Thomas Lowe.

Colm Massey recommended a paper which i've attached, I believe he wrote it when he worked at Math Engine - it's about the very early work he did just prior to him starting NaturalMotion.

**Rob**

## Appendix B

# Implementation Details

### B.1 Fitness Functions

Examples of the fitness functions used for the this study are given in the following to algorithms. The biped fitness function, algorithm 3, stores the penalty values as members of the class which are used for both the crowding method, described in section 4.1.4.5, and for the statistical output of the program. The creature tests were performed before statistical analysis was included in the study. The code used is C++.

---

**Algorithm 3** Biped fitness function

---

```
float maxFitness = 4.0 + 2.0*phase;
float xMax = 1.0;
float highMax = 1.0;
float lowMax = 1.0;
float dirZMax = 1.0;
float dirYMax = 1.0;
float eeMax = 6.0;
float extra = 3.0;
float minFitness = -((xMax + highMax + lowMax + dirZMax + dirYMax + eeMax + extra) * phase);
// Characters must perform exceptionally badly to fail
// Other wise it is best to hone their performance rather than fail them.
if(failed) {
    fitness = -5.0;
    penalty = -15.0;
} else {
    // Initial fitness value
    character->distance = fitness = Fit(bestPos.z, 20.0, 2.0, maxFitness, 0.0);
    // "Penalty" considerations
    character->maxX = Map(maxX, 6.0, 1.0, xMax, 0.0);
    // Keep near to ground. Penalise if max height is far from ground.
    character->high = Map(maxY, 7.7, 6.3, highMax, 0.0);
    character->low = Map(minY, 5.7, 4.2, lowMax, 1.0);
    // Direction Penalty
    character->directionZ = Map(minZDir, 1.0, 0.0, 0.0, dirZMax);
    character->directionY = Map(minYDir, 1.0, -1.0, 0.0, dirYMax);
    character->earlyExit = eeMax - eeMax * ((1.0*step)/maxSteps) * ((1.0*step)/maxSteps);
    penalty = -character->maxX - character->high - character->directionZ - character->directionY
- character->low - character->earlyExit;
    if(step < 200) {
        penalty -= extra;
        character->extra = 1.0;
    }
}
character->fitness = fitness + (penalty * phase); // Gritz addition: - 5.0 * (1.0 - phase);
```

---

---

**Algorithm 4** Creature fitness function

---

```
// Characters must perform exceptionally badly to fail
// Other wise it is best to hone their performance rather than fail them.
if(FAILED) {
    fitness = 0.0;
    penalty = -5.0;
} else {
    // Initial fitness value
    fitness = Map(pos.z, 30.0, 0.0, 4.0, 0.0);
    // Penalty considerations
    // Keep to the Z axis. Large journeys into X are penalised
    float xPos = Map(maxX, 15.0, 5.0, 2.0, 0.0);
    // Keep near to ground. Penalise if max height is far from ground.
    float height = Map(maxY, 6.0, 2.0, 1.0, 0.0);
    // Direction Penalty
    float direction = Map(minDir, 1.0, -1.0, 0.0, 1.0);
    float earlyExit = (1.0 - (1.0*step)/maxSteps);
    // Total up the style
    penalty = - xPos - height - direction - earlyExit;
}
// Final fitness - Phase in the penalties. Initially just focus on being able to move
forward.
character->fitness = fitness + (penalty * phase); // Gritz addition: - 5.0 * (1.0 -
phase);
```

---

## B.2 Evolution Settings

The following is an example of the configuration file used to initialise an evolution. The different options are parsed by the Input File class in the components library and the resulting values are used to set different parameters for the evolution. The bottom settings are for the Open Dynamics Engine used for the physics simulation and once they were properly tested they remained the same for the whole study.

```
# BipedStats
    character BipedStats
# Evolution Settings
# -----
    population 4000
    generations 100
    tournamentSize 12
    nodeLimit 150

    crowding 0
    crowdGenerations 70

    evolutionType singleGroup
    groupGenerations 50
    numGroups 10

    mutationFraction 0.1
    recombinationFraction 0.8

    initMutationFraction 0.2
    initRecombinationFraction 0.6

    perGenerationOutput 9
    finalGenerationOutput 20

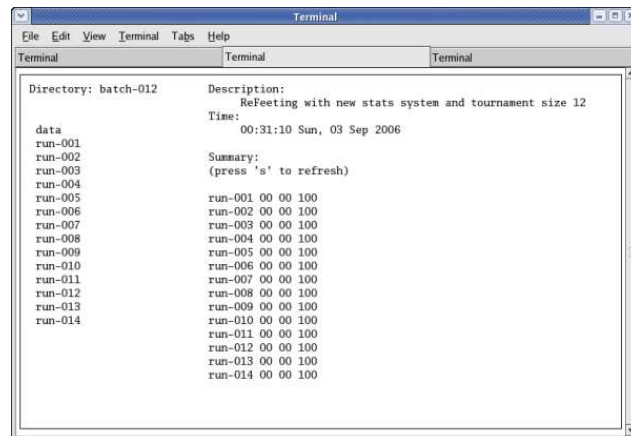
# Register Nodes
# -----
    Sin 0
    Cos 0
# Fitness Function Settings
# -----
    heightLimit 10.0
    maxSteps 1000
# ODE Settings
# -----
    worldGravity -9.81
    worldCFM 0.0001
    worldERP 0.4
    worldSurfaceLayer 0.001
    worldMCVel infinite
    cSoftCFM 0.001
    cSoftERP 0.4
    cMu infinite
    cBounce 0.05
    cBounceVel 0.5
```

### B.3 Features of the Graphical User Interface

- **Drop down menus:** The top of the main window contains a three drop down menus for selecting the character, batch and run that the user would like to view. When menu option is selected the other menus update automatically to show the relevant available options. A “Load” button loads the selected run into the OpenGL view ports.
- **Nine screens:** The program loads and runs nine results at once from the selected run. The user can enlarge any of the screens by hovering the mouse over it and tapping the space bar. Pressing space bar again will return to the nine view.
- **Lockable screens:** The user can lock any of the nine screens so that they retain the current example within them and are not effected by the user loading a new run. This means that examples can be compared between different runs and batches with ease. Hovering the mouse over the viewer will display the character, batch and run in the status bar to remind the user which selection it is. Locked screens are indicated by a border displayed within that view port.
- **Quick Reset:** The simulations in the view ports can be reset at any time without loading the run again. A “Refresh” button on the interface resets all the viewers, whilst individual resetting can be achieved with the “Reset” option on the right-click menu in the view ports.
- **Export Characters:** If one or more desirable results are found the character information can be exported to a file. The user must lock the viewers of the results they wish to export and then select the “Export to file” option from the File menu. Additional results can be appended to the same file with the “Append to file” options from the same menu. This option is particularly useful for creating samples for initialising a new evolution. The exported files can be read in by the evolution program and used to populate the first generation of a new evolution.
- **Blend:** Controller blending can also be achieved with the GUI. The Blending sub-menu in the view ports’ pop-up menu is used to specify two characters to blend between. The blending is designed for the balance and walking controllers for the biped model with feet. It is controlled automatically and is designed to transfer from the walking behaviour to the balance behaviour when the model appears unstable. The change between the controllers signified by the character model changing colour in the viewer.
- **Export and Import Blends:** For ease of use, any example of the blending behaviour can be exported using the right-click menu in the viewer that is displaying the blended character. For later review, the controllers can then be imported again using the “Import Blend” option in the File menu which automatically loads the selected example in to the first available view port.
- **Export OBJs:** Any character can be exported as a sequence of OBJs. The user must view the character’s behaviour up to the point at which they wish to stop the sequence and then select “Export OBJs” from the right-click menu in that viewer. The sequence will then be replayed with a border around the viewer which indicates the export is underway. The export will stop when it reaches the point which the user specified.

### B.4 Text User Interface

A screen shot of the text user interface is given below.



```
Terminal
File Edit View Terminal Tabs Help
Terminal Terminal Terminal
Directory: batch-012
Description:
ReFeeting with new stats system and tournament size 12
Time:
00:31:10 Sun, 03 Sep 2006
Summary:
(prompt 's' to refresh)
data
run-001
run-002
run-003
run-004
run-005
run-006
run-007
run-008
run-009
run-010
run-011
run-012
run-013
run-014
run-001 00 00 100
run-002 00 00 100
run-003 00 00 100
run-004 00 00 100
run-005 00 00 100
run-006 00 00 100
run-007 00 00 100
run-008 00 00 100
run-009 00 00 100
run-010 00 00 100
run-011 00 00 100
run-012 00 00 100
run-013 00 00 100
run-014 00 00 100
```

## Appendix C

# Scripts & Aliases

### C.1 Pipeline Aliases

The following aliases enhance the power of the pipeline and its syntax.

```
alias plsbatch 'pls +:{\!:1}+.batch--{\!:2}:+:+:+'
alias plsVis 'pls masters:+:visualise:{\!:1}+.+:+:+'
alias plsTiff 'pls -e masters:+:visualise:{\!:1}+.+:+:+:tiff'
alias plsObj 'pls -e masters:+:visualise:{\!:1}+.+:+:+:obj'
alias plsMpeg 'pls -e masters:+:visualise:{\!:1}+.+:+:+:mpeg'
alias plsConfig 'pls masters:+:+.+:+:config:+'
alias pcd 'cd 'pls -p \!:1''
```

### C.2 Awk Statistics Script

**Algorithm 5** Awk Script for processing the evolutionary statistics

---

```

#!/usr/bin/gawk -f
# Penalty cycle group generation rank distance high low exit dirZ dirY
BEGIN { SUBSEP = "@"; }
# only grab the Statistics lines and not the ones with the header information
($1 == "Penalty") && ( $2 == "cycle") && ( titles == 0 ) {
    titles = 1;
    banner = $0;
    for(field=1; field<=NF; field++) {
        bannerList[field] = $field;
    }
    bannerRecords = NF;
}
($1 == "Penalty") && ( $2!~/cycle/) && ( $0!~/nan/) && (( $6 < 100000 ) && ( $6 > 0 )) {
    # Count the number of statistic lines
    statNum += 1;
    numRecords = NF;
    for ( field=6; field<=NF; field++ ) {
        sum[field] += $field;
        if($field > max[field]) max[field] = $field;
    }
    # count up the number of generations by incrementing a value everytime a new
generation is found.
    if( !($4 in genCount) ) numGen += 1;
    # Count the number of lines in each generation by using the generations field as an
index.
    genCount[$4] += 1;
    for ( field=6; field<=NF; field++ ) {
        genSum[field, $4] += $field;
        if($field > genMax[field, $4]) genMax[field, $4] = $field;
    }
}
# Print the max and averages for each generation
END {
    printf "Gen\t"
    for(field=6; field<=bannerRecords; field++) {
        printf "Max.%s\tAv.%s\t", bannerList[field], bannerList[field];
    }
    printf "\n"
    for ( gen=1; gen<=numGen; gen++ ) {
        printf "%d", gen;
        for( field=6; field<=numRecords; field++) {
            printf "\t%.3f\t%.3f", genMax[field, gen], genSum[field,
gen]/genCount[gen];
        }
        printf "\n"
    }
    printf "Gen\t"
    for(field=6; field<=bannerRecords; field++) {
        printf "Max.%s\tAv.%s\t", bannerList[field], bannerList[field];
    }
    printf "\n"
}

```

---