Genetically Evolved Motion Controllers

Christopher Rayner
Masters Thesis

MSc Computer Animation and Visual Effects
Bournemouth University
NCCA

August 2014

Contents

List of Terms

GP	Genetic Programming
BVH	Biovision Hierachy - A Motion Capture File Format
ODE	Open Dynamics Engine - An Open Source physics engine
XML	Extensible Markup Language - A human readable file format common for storing data structures
Genome	An Individual Algorithm expressed as a string
LISP	"LISt Processing" - A high level programming language frequently used within the field of Artificial Intelligence
DOF	Degrees of Freedom - The available axes of motion of a figure

Abstract

Motion synthesis is an alternative approach to traditional key frame animation. It allows for natural motion gaits to be created or captured, reducing the need for handmade key frames and allows for dynamic content to be produced. Exploring previous work we look at the suitablility for genetically programmed motion control to be used as a tool within the animation field. Using evolutionary techniques inspired by nature, control systems are developed that often find solutions that are both efficient and surprisingly natural. We found that although the results of this work can be interesting and have visual appeal, the process lacks the efficiency needed to work within a real world context.

Chapter 1

Introduction

Problem statement

The process of animating an articulated figure is a long one, taking a large amount of talent and patience for the animator to create a stylised, yet fluid illusion of movement. The animator has full control of his/her work but by doing so gains the overheads of increasing complexity that is inflated with every added degree of freedom (DOF). The notion of complexity vs control has led to the development of both the procedural and physics-based animation domains. This paper and the works discussed within explore the development of procedurally defined artificial controllers for the control of articulated figures. This area of study is vast and thus we will contain our focus to the use of Genetic Programming (GP) as a tool for development of such controllers. The goal of this field is the automatic development of animated behaviour suitable for use in commercial projects.

Structure

1.0.1 Chapter 2: Background

A discussion of the background work in and around this area. The primary sources for this project are the works of Karl Sims and Larry Gritz, with their work doing much to establish the field. More recent work from both in and around this field have also been included for further reading.

1.0.2 Chapter 3: Evolutionary Computation Theory

A section detailing the theoretical elements of this field. This covers both the background influences of the genetic programming field as well as the theoretical workings

of many of the elements of the system.

1.0.3 Chapter 4: System Design

A discussion of the implementation details of this system, taking many references from the theory previously discussed. This chapter describes how the system was built and specific design choices that were made.

1.0.4 Chapter 5: Testing

A discussion of the early testing stages of the project used to establish a parameter setup suitable for further testing. This was required due to the long simulation times needed to gain results.

1.0.5 Chapter 6: Results

A discussion of the final simulation setups run for this project and their resulting outcomes.

1.0.6 Chapter 7: Conclusion

Closing thoughts of the results of this project including a detailing of the benefits and drawbacks of this system for use as a tool for commercial animation.

Chapter 2

Background

2.1 Overview

The field of Motion Synthesis consists of creating realistic motion without explicit user input (usually via key framing). Much of the work in this area is related to robotics with the intent of developing robotic controllers capable of adaptive learnt motion in favour of pre-computed walk cycles. Such controllers are notoriously difficult to design by hand as the complexity increases with each allowed degree of freedom. This creates a system embedded with co-dependencies that can be difficult for a human engineer to manage. Thus it could be said that the complexity of the system is restricted by the human element.
That human element can be removed from the equation by incorporating a system based on evolutionary design and allowing the system to identify the most efficient structure by itself based on the problem solving strategies found in nature.

Our interest here lies within an overlap of these two fields for use with digital or computer generated articulated figures.

2.2 Previous Work

The seminal paper by Karl Sims (Sims 1994), is one of the earliest in this area of research and has provided the grounding for many of the papers proceeding it. Sims approached the field with a focus more on the theoretical ideas of evolved behaviour and adaption than pure articulation. His work allowed for both behavioural and topological evolution through a system based on genetic design that enabled his virtual creatures to adapt both their motion and body structure to solve a given problem (Figure 2.1).

This work followed the principal ideas of evolution laid out by Darwin (Darwin 1869)

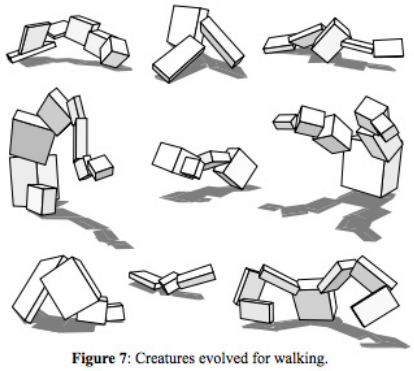

Figure 7: Creatures evolved for walking.

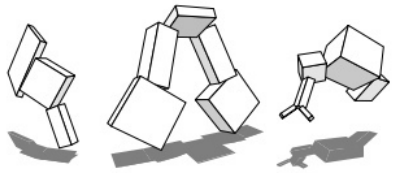

Figure 8: Creatures evolved for jumping.

Figure 2.1: *Karl Sims : Evolving Virtual Creatures (1994). Displaying a series of creatures that have learnt to traverse across a space by evolving both their topology and behaviour.*

and displayed evidence for the evolutionary process with his creatures displaying clear adapted abilities to walk, run, swim or fight for possession of an object.

The system Sims developed allowed evolutionary development via the use of parse trees to represent the layout (evocative of that of a DNA sequence which we will discuss later) of both the creature's body and brain. These trees act as single controllers for the character and propose one potential solution to the problem. A collection of these trees form a population for a single run.

Sims developed a system of evolution by simulating these characters within a physically correct environment and measuring the outcome of these controllers when given a specific task (the fitness value). The controllers not able to accomplish the task adequatly were slowly removed from the population, leaving only the current best solutions to continue to the next stage.

Sims discusses the practicalities of design including the problems of designing the fitness function used for testing the controllers - a common problem in this work due to the exploitive tendencies of the creatures during evolution.

Though essentially breaking new ground, Sims does little to offer any practical uses for this motion with any kind of user control. Such a task was later followed by Larry Gritz in (Gritz and Hahn 1995) & (Gritz and Hahn "1997").
Gritz chooses to promote practicality and usability over novelty, addressing the prob-

lem with a fixed topology model with emphasis on physically correct, believable yet stylised locomotion (Figure 2.2). The key aspect of the work accomplished by Gritz is his use of an explicit "style system" embodied within the fitness function. His model was not only tasked with finding a solution but doing so in a manner appropriate for use by an animator. This process heavily reduces the solution space each generation by highlighting un-wanted behaviour early.

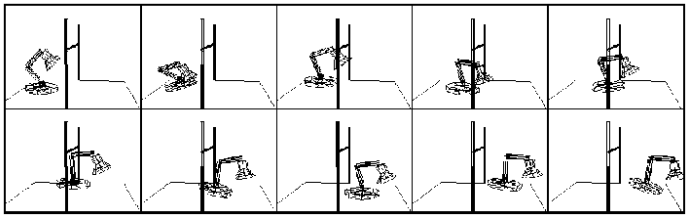

Figure 2.2: *Larry Gritz : Genetic Programming for Articulated Figure Motion (1995). Gritz used a fixed topology "Luxo Lamp" to run his simulations, keeping the focus on the evolution of behaviour.*

In his masters thesis Michael P Jones explores a hybrid of the works of both Sims and Gritz (Jones 2006) whilst also incorporating elements of the work of Spencer who amongst other aspects experimented with the use of single vs multiple controllers (Kinnear 1994). Jones took the visual cues and physical attributes of Sims' work and combined it with the more elegant program structure and the use of "Lisp-like" expressions used by Gritz, creating a system every bit as sophisticated as its predecesors.

2.3 Other Works

The recent work of Geijtenbeek proposes a further developed system that establishes evolutionary design as a serious player for animation of the complex figures desired by the industry today. Geijtenbeek (Geijtenbeek and van de Panne 2013) proposes a system based on "Co-variance Matrix Adaption" within highly developed bipedal characters with complex embedded muscle systems. The outcome of the system are highly believable motion gaits for autonomous bipedal characters that not only solve the task, but can do so dynamically whilst encountering a range of real-time disturbances. The results of this work are outstanding, displaying a series of kangaroo-like and bird-like creatures developing motion gaits that are remarkably similiar to their natural counterparts (Figure 2.3). Similar research can be found by Coros (Coros *et al.* 2011) and (Gehring *et al.* 2014) with a focus on the development of quadruped models with equally complex physiology.

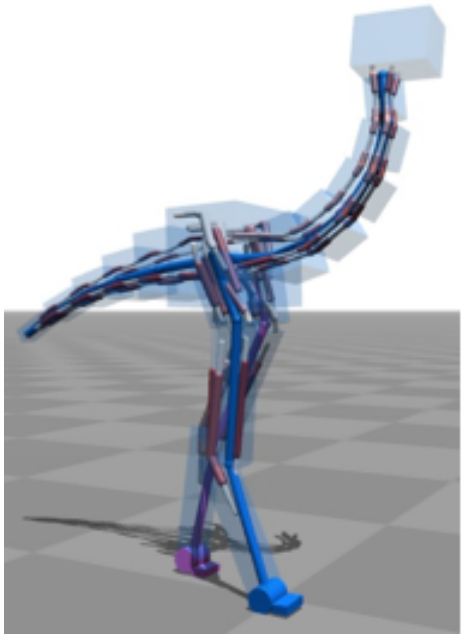

Figure 2.3: *Thomas Geijtenbeek : Flexible Muscle-based Locomotion for Bipedal Creatures(2013). Displaying the physical structure of a bird-like creature complete with a full muscle system.*

2.3.1 Neural Networks

Artificial Neural Networks are an optimization tool that attempts to mirror the behaviour of the brain, creating a system of adaptions and progression based on past experience. The network structure is traditionally hand-coded for the given problem but recent works have adopted the process of evolutionary design to create network structures whose topologies are evolved (Stanley 2002). The use of Artificial Neural Networks in combination with genetic programming has been investigated by Jeff Clunes (Clune *et al.* 2012) and Torsten Reil (Reil and Husbands 2002) (Reil and Massey 2011) amongst others. Clunes discusses the use of HYPERNEAT (an extension of Kenneth O'Stanleys' NEAT) a system designed to evolve neural networks in a modular fashion while attempting to remain as biologically realistic as possible and has displayed impressive results. The culmination of the work by Torsten Reil has led to the formation of Natural Motion, a company specialising in procedural animation, the emphasis however is on clean motion synthesis of pre-recorded motion capture data where the learning is focused on patterns between data sets.

Chapter 3

Evolutionary Computation Theory

Evolutionary Computation is the family name given to a group of programming techniques including Evolution Programming, Genetic Programming and Genetic Algorithms that combine the Darwinian principles of evolution with traditional machine searching and sampling problems.
The field has been in some degree of research throughout the 20th century but like most artificial intelligence research areas, spent a long time with limited progress until 1975 when John Holland proposed Genetic Algorithms (Holland 1975) (Later published in (Holland 1992)). The ideas behind evolutionary computation are drawn from the field of genetics and evolutionary theory.

3.0.2 Evolutionary Theory

The work of Charles Darwin during the 19th century and culminating in his published theory of evolution (Darwin 1869) was a turning point in the human understanding of the development of complexity in the natural world. Darwin proposed that nature over the course of millions of years allowed for the development of diverse and complex species without the need for an intelligent designer.
Darwin coined the term *Natural Selection* in reference to this process and stated that it is through entirely natural processes that complexity can arise from simple beginnings without any forsight or intelligent decision making.

Natural selection follows that within a species a small amount of natural diversity will exist within the genotypic makeup of each individual that may or may not be expressed in its phenotype (the physical expression of the genes). These differences combined with environmental factors have minimal effects on the behaviour of an individual but may result in a small survival or sexual advantage. For example, an individual with a slightly better lung capacity or muscle formation in the legs may be able to run a little faster than its companion. When being hunted by a predator, this slight advantage could mean the difference between life and death. The individuals

that survive pass on their slightly advantageous genes to the next generation whilst the individuals that die reduce the instances of their genetic makeup in the population. Over time the individuals that continue to die prematurely (i.e before mating) will lose their space in the gene pool altogether.
Needless to say, in contrast to Lamarkian Theory the "advantages" that an individual develops overtime and passes on to its children are genetic features. No physical attributes gained by the individual over its lifetime are passed on.

"The theory of evolution by cumulative natural selection is the only theory we know of that is in principle capable of explaining the existence of organized complexity."

Richard Dawkins- The Blind Watchmaker (Dawkins 1996)

The important part to take away from Darwin's theory is that natural selection is not a random process but a cumulative one. A random process for creating a complex individual would be an incredibly difficult task as each element of the complexity would equally have a random chance of being selected, the chance that all necessary parts would be selected in the correct order, though not technically impossible, is unfeasibly small. In the Blind Watchmaker(Dawkins 1996) Dawkins proposes a hypothetical task that displays just this: it involves monkeys, Shakespeare and typewriters and is worth investigating. However, it is suffice to say that cumulative selection reduces the space of possible decisions at each stage by removing inefficient choices from the population. Cumulative selection and by result evolution in general is a naturally progressive process, culling out inefficiencies from the system and favouring the further production of *fitter* individuals.
We can see from looking at the world around us that this process, over the course of millions of years has worked effectively in producing a vast variation of species including ourselves. We know this process works, but unfortunately a few million years is a little outside the life span of the average human to experiment. However, with the use of computers we are able to do just this, running complex evolutionary process within minutes.

3.0.3 Genetic Programming

The early work produced by John Holland (Holland 1992), focused on the evolution of a single algorithm. Genetic Algorithms are loosely based on the idea of a DNA string and are frequently expressed as a string of binary numbers, with each element representing a parameter of the problem much like each allelle of a DNA sequence (Figure 3.1). In this case each parameter of the problem must be known and have an element representing it in the string. For many cases this is not a problem, however difficulties arise when the potential solution is not necessarily known (partly or entirely) and thus a complete string cannot be formed.

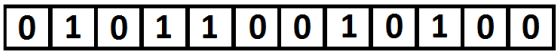

Figure 3.1: *Visual Representation of a Genetic Algorithm using Binary Elements*

3.0.3.1 Representation:

Genetic programming developed by Koza (Koza 1992) solves this problem by exploiting the concept of the evolutionary search space. Genetic programming does not rely on a pre-configured *template* of the solution and instead uses evolution to build the structure of the program dynamically, thus a single problem may have a number of solutions that all work equally as well whilst being algorithmically dissimmilar.

A genetic program can be represented in many forms including some linear forms that are visually similar to a genetic algorithm, however for the most part these programs are represented by tree-like structures with each node of the tree representing a single element of the algorithm. Though visually expressed as a tree the most common way to organise the system programmatically is through the use of LISP (or LISP-like) expressions (Figure 3.3).
A genetic program functions like so:
A set of possible nodes is chosen. This node set consists of:

Functions: + - * /
Terminals: 1 -1 x y

Figure 3.2: *A basic function and terminal set for a genetic program.*

Functions: These create branches of the tree. A function commonly takes one, two or three inputs and returns a single output.
Terminals: These create the leaves of the tree. Variable terminals retrieve real-time data from the system and pass it up the tree whilst constant terminals are preset with randomly selected values and pass these up the tree.

When these nodes are strung together a tree is formed (Figure 3.3). The tree takes in multiple values through its leaves, passing them up through the system. These values are modified accordingly until the root is reached and a final value is returned. This tree represents a single attempt at a solution.

At the beginning of the process a given number of these trees are randomly generated (using limits to prevent continual growth) and form the base generation. This generation typically provides very poor results, though a small chance does exist that a good or best solution could be generated, it is extremely unlikely. The evolutionary process is launched off of this base generation.

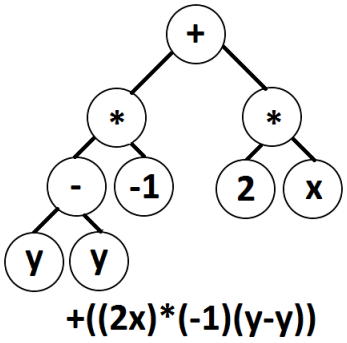

Figure 3.3: *A Visual Representation of a Genetic Program Tree. The Algorithm beneath is expressed in a Lisp-like format for human readability.*

3.0.3.2 Evolutionary Process:

Crossover: - Two trees are selected. A random element is selected in each tree and two copies of the original trees are created. Each child has the entire branch removed at this point and the two branches are swapped, creating two trees that represent the two variations of the original trees. (Figure 3.4)

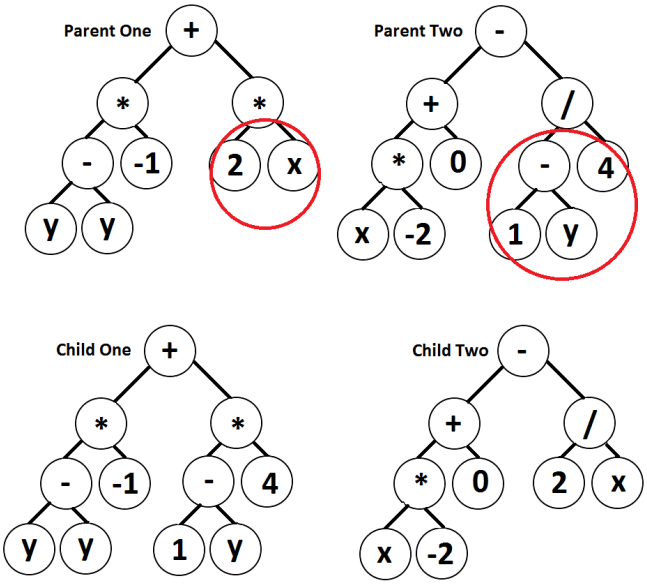

Figure 3.4: *Crossover with two children*

Mutation: This process comes after crossover and may or may not be used on the children based on a probability function.
- Single Point Mutation: A random element of the tree is selected and modified. A function will be given a new random chosen type, whilst a terminal will have its value modified.
- Branch Mutation: A random element of the tree is selected. This element and all

child elements are removed from the tree and a new branch is generated in its place. (Figure 3.5)

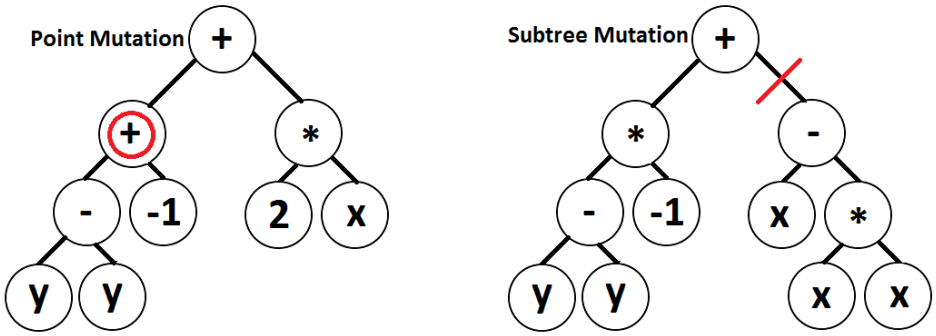

Figure 3.5: *Point & Subtree Mutation*

Reproduction: This process simply makes a genetically identical copy of the individual, essentially performing asexual reproduction. It is common practice to have a small element of the fittest individuals reproduced, allowing them to test their fitness against a more competitive population.

The process of crossover alone would be useless and create little more than an increased selection of random trees. However the selection of the trees that are to be combined come from a selection process themselves. Based on the idea of natural selection the process selects two of the fittest individuals for breeding. Two of the most common selection process are Tournament and Roulette selection.

Tournament Selection: In tournament selection a random group of individuals are chosen from the population (up to a given number) and form a competition. The fittest member of the selection is copied to a new gene pool and the process is repeated until the original population is exhausted and a new population has been constructed. It is important to state that the initial selection is random and thus the fittest member of the population is not guaranteed to be in the selection. This allows for a "current best" selection to occur and promotes genetic diversity as opposed to a single member dominating the population.

Roulette Selection: Roulette selection is a probability based selection in which each individual shares a *slice* of the selection wheel proportionate to their fitness. An individual with higher fitness will have a larger slice and be more likely to be chosen, however this does not remove the chance that a less fit member may be chosen, it only simply lowers the probability(Figure 3.6).

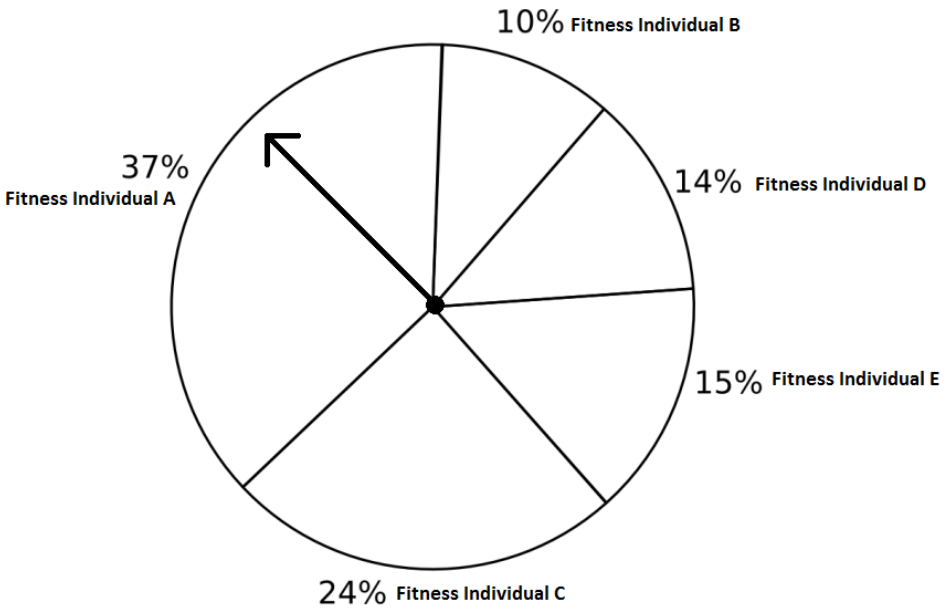

Figure 3.6: *Roulette Selection: Each individual is given a chance of selection based on its fitness with weaker individuals holding smaller sections of the wheel.*

Fitness: Measuring the fitness of an individual is a complex process. Being too restrictive can lose potentially good solutions whilst being too lenient will flood the population with individuals that will frequently be unfit for the task and will slow down the process. The fitness will naturally fluctuate across early generations with many of the individuals performing poorly at the given task. Over time these individuals will be removed from the population and the fitness will gradually rise and become more stable. A discussion in 4 explains the fitness measures used for this project.

Chapter 4

System Design

The implementation of the system was written in C++, incorporating OpenGL for visualisation purposes and various formats for external file sources. These will be touched upon briefly and examples of which can be found in the Appendices. The core system can be broken down into three distinct areas:

• A genetic subsystem capable of developing and evolving algorithmic strings forms the backbone of the system and creates the individual elements that will form a controller program.
• A simulation subsystem using the ODE physics engine is used to run simulations and gather data to test the controllers in a simulated environment.
• A rendering subsystem that is in charge of writing out the visual properties of the system to file in order to be rendered at a higher quality.

4.0.4 Genetic System

4.0.4.1 Node Network

The system has been designed with re-use in mind and attempts have been made to relieve the system of as much system specific data as possible. The configuration of the system is loaded at run-time from an external XML file (Appendix:D) containing data for which node types are to be used. A small selection of functional nodes have been included within the system and a sub-section of these have been initialised in the file. The system is configured to read and include any symbol given in the text file if its accompanying definition is included in the system. Any missing or incorrect symbols are discarded. A full list of functions implemented can be found in (Appendix:E).
The nodes used in this system were binary operators **+,-,*,/ (protected divide)**, unary operator **Absolute** and a ternary operator **IFLZ** (If Less than Zero) which

takes three inputs, returning the second if the first is less than zero or the third if the second is less than zero.
These nodes were chosen from the recommendations given by Gritz and Jones as a selection that offers enough variation for the system to be able to adequately solve the problem. The Absolute node was implemented as recommended by Koza (Koza 1992).

An inheritance model has been developed to incorporate the variation of different nodes, extending from a single base node class like so:

Node Class Hierarchy:
• **Node Class :** Base class providing a small range of concrete functions and a number of virtual functions.
• **Unary Class:** Takes a single input and returns an output.
• **Binary Class:** Takes two inputs and returns an output.
• **Ternary Class:** Takes three inputs and returns an output.
• **Variable Class:** Takes system input from function and relays its value.
• **Terminal Class:** Returns a a random floating point number within the given range (This system's range: -5,5).

Operator Class:
A singleton class containing the function operators for each node type. The default operator functions are implemented here as well as the variable data access for the system. These include:
• **Position:** Returns the current x, y or z position of the currently associated character.
• **Angle:** Returns the a joint angle of the currently associated character.

4.0.4.2 Genome

The Genome class represents a single designed solution or expression and has been named in reference to its biological inspiration. A single simple representation of a genome when complete in a LISP-like human-readable format looks like so:

$$+(-(2(-xy)-1)*(x-(21))) \tag{4.1}$$

The Genome maintains a recursive structure of nodes stored in a single root node. When evaluated the genome propogates through the branches of the tree to return a single value.

4.0.4.3 Controller

This implementation uses the multiple-controller setup as described by both Gritz and Sims. In this instance a controller consists of a single genome for each DOF of our model (Figure 4.1). By taking this approach the system allows for each genome to be associated with a specific function in our articulated character and this association remains throughout the evolution, as such the algorithm evolves to better perform its associated task.

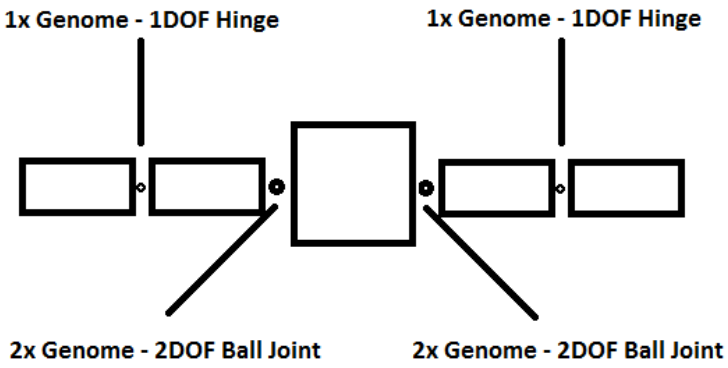

Figure 4.1: *A Controller for a model with Six Degrees of Freedom. A single Genome is generated for each DOF.*

This design setup is analogous to the biological use of homology within DNA strings. This natural system is used to *line up* the two DNA strings during crossover to ensure that the data transferred is like for like. For example any DNA string that swapped the gene for blue eyes with the gene for blonde hair would result in a redundant child gene being produced. As such, for our purposes a genome that swapped the expression that has evolved to control the right arm with the one to control the left leg would produce a child that would have lost a large chunk of its evolutionary design and may potentially become unfit to survive through the next generation as a result.

The use of a single controller to control the entire system was investigated by both Spencer & Jones with promising results, however was not investigated within this implementation due to time constraints.

4.0.4.4 Evolution

The evolutionary component of the system is comprised of two major classes:

• **Evolution Class:** Takes in the user defined parameters from file and runs a complete evolutionary cycle at the end of each generation, returning the newly evolved

population at the end.
• **Population Class:** Contains the collection of controllers within the system for a single population and the functions for manipulating controllers and or genomes using the crossover and mutation operators aswell as the selection process.

The evolutionary process maintained in these classes is further broken down into the following sub-components:

Initialisation:
• **Random Initialisation:** A random set of controllers can be generated to form a new generation. This is typically the system used to define the initial starting population (Generation 0).
When building this initial generation a distribution setup as described by Koza and known as **Ramped Half & Half** has been implemented to allow the system to create a larger variety of Genomes. Koza describes the process as an even-distribution model in which the population is separated into sub groups. Each group is given a size or depth limit that is governed from a range between the minimum and maximum. Half of the population is then generated using the *Full Method* Figure 4.3 and half with the *Grow Method* Figure 4.2.

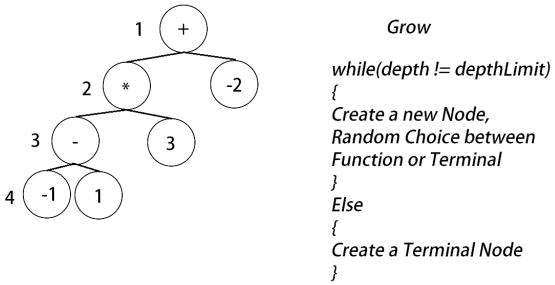

Figure 4.2: *Grow Method*

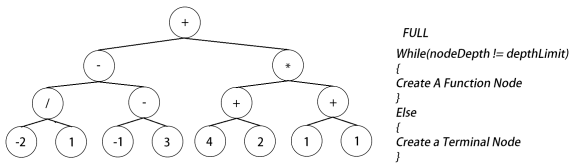

Figure 4.3: *Full Method*

The Full Method: States that only functions should be chosen until the depth or size limit is reached, then the leaves are ended in terminal nodes. This process guarantees to create full or complete trees.

The Grow Method: States that any node type can be selected up until the given limit in which it is terminated with a terminal node. This process produces trees that are misshapen and may produce a variation of long and short branches.

• **Initialisation from Sample:** The second type of population initialisation is to generate a population from another given population. This process is used at the end of every generation in which the new population is generated from the newly created offspring of the previous population following the evolutionary operators.

Operators:

• **Selection:** The selection process implemented for this system was the Tournament Selection process described in 3. This technique is recommended over Roulette Selection as the initial sub-group is randomly selected there is no bias towards better individuals, which allows weaker individuals to pass through the next generation as long as they are the best in the selected group. This process helps to eliminate the over-selection of super-individuals in population.

"In the valley of the blind, the one-eyed man is king"

A proverb suitable to describe the selection process

• **Mutation:** The mutation operator functions as described in 3 have a very minimal use in the population. The mutation operator is set up to function on the newly produced children produced via crossover however, as discussed previously, over using mutation can result in a population that resembles randomness and voids the evolutionary process. Therefore this operator has a very low probability of being called (we use 1.0%) reducing the amount of mutation in the system to a rare event.

• **Crossover:** The process of Crossover was described in 3 and is implemented in the system efficiently by making use of the pointer system defined by the inheritance structure of the nodes. Crossover simply becomes a matter of re-allocating the parent and child pointers of the chosen node to form a re-direction of the evaluation.

A decision was made to add a small buffer to the maximum size/depth limit. This is a percentage based on the current size (we use 10%) and allows for a small exceeding of the limits. Riccardo Poli discusses the limitations of hand-designed thresholds that may limit the potential solutions of the system (Poli *et al.* 2008), by adding a small buffer slightly longer algorithms are allowed to be designed which may lead to better solutions.

4.0.5 Physics Engine

The complete design of a system for physics simulations is an enormous task alone and beyond the scope of the project. As such a third party library was chosen to handle this aspect of the pipeline with this system handling the input and output data.

The physics engine chosen for this implementation was the Open Dynamics Engine (ODE) which is an open-source C library that is called via appropriate wrappers in the C++ classes and provides a stable physics environment capable of simulating multi-component articulated bodies via joints. Care must be taken to identify input data to the joints that evaluate in values regarded as *Not a Number (NAN)* as these values can lead to a system crash.

A single negative point to note is the sensitivity of the physics engine. Small changes to values can propagate through the system and affect other properties. This is apparent in the relationship between the time step, the mass of the objects and the gravity in the environment. A large number of simulations were run testing different variations to find the best combination and the final setup was not modified further. The common values of the physics engine are outlined in (Appendix:F).

4.0.6 Character Design

The characters designed for the simulation must have appropriate joints, joint restrictions and be configured of primitive shapes to reduce the complexity of the collision-detection in the simulated environment. This feature is prevalent in previous papers within this field in which the rigid construction of *box-like* characters is the standard output. For this implementation an attempt was made to capture the motion of the character as opposed to its visual features, using the original construction simply as a placeholder to make it a viable structure to be handled by the physics engine.

This system creates a class hierachy of wrappers for the following ODE Joint Types:
Joint Class Hierarchy:
• **Joint Class:** Base class providing a small range of concrete functions and a number of virtual functions.
• **HingeJoint Class:** Wrapper to implement a Hinge Joint, allowing rotation on a single axis.
• **BallJoint Class:** Wrapper to implement a Ball & Socket Joint, allowing rotation on all three axes.
• **MotorJointClass:** Wrapper to implement a MotorJoint. This must be incorporated alongside the Ball & Socket joint.
• **UniversalJoint Class:** Wrapper to implement the Universal Joint, allowing rotation on two axes.

Initial designs were implemented with hinge joints representing elbow and knee type joints and ball & socket joints for shoulder types. However, although ODE simulates the ball & socket joints well, errors were found when attempting to restrict the joint limits and so the universal joint was used as a replacement with no clear limitations.

Due to time constraints only a single character was tested for this implementation, however an import system has been constructed to read in the hierarchical structure from a .bvh file making it possible for further characters to be designed externally and read in.

4.0.6.1 Creature

The standard creature model as proposed by Sims has been been incorporated as the base test for the system. The creature is a simple five piece model with six available degrees of freedom. The large central body piece makes it an extremely stable character (Figure 4.4). The angle constraints (Table 4.1) keep the motion to a range that is biologically plausible with elbows that are constrained by a single hinge and cannot extend backwards and shoulders that only allow rotation to the front of the creature.

The weight of each body segment was set to 0.5 density units for the body and 0.25 for the arm segments. These numbers were arbitraily tested and found to be the best combination with the gravity and timestep of the simulation. Higher weights caused the character to become limp and unable to lift its arms whilst lower weights caused the "muscles" to become stronger resulting in the character throwing itself through the air.

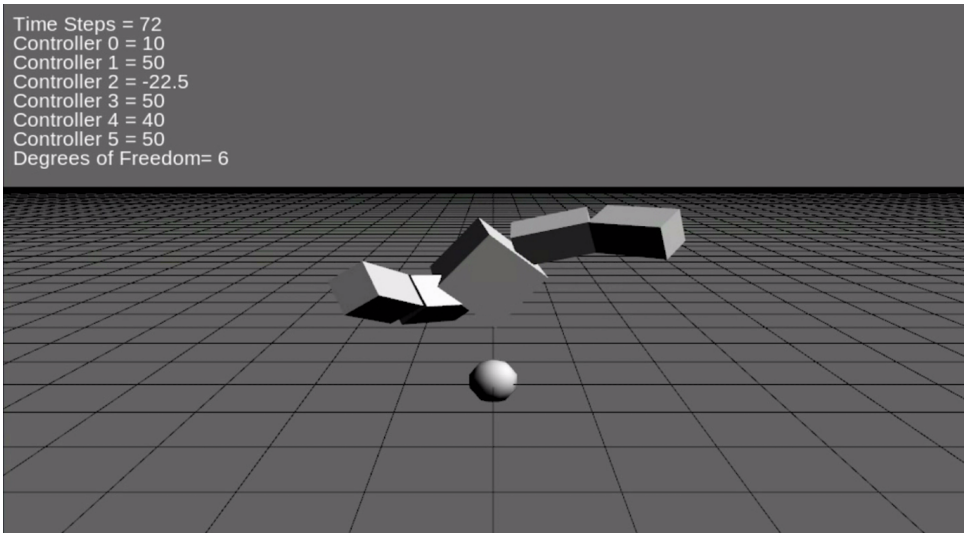

Figure 4.4: *Screenshot of the simulation running.*

Table 4.1: *Angle Constraints for Creature*

Joint	Type	Axis	MinAngle#1	MaxAngle#1	MinAngle#2	MaxAngle#2
Left Elbow	Hinge	Y	0	45	-	-
Right Elbow	Hinge	Y	0	45	-	-
Left Shoulder	Universal	Z,Y	-45	45	0	70
Right Shoulder	Universal	Z,Y	-45	45	0	70

4.0.7 Fitness Function

The design of the fitness function proved to be the most time-consuming process of developing the system. Due to the excessive time that the system takes to run the complete set of simulations it proved difficult to make subtle modifications and get feedback and just as difficult to isolate the feedback to a particular change. The fitness function is modelled upon the function given by Gritz and Jones in which a basic fitness test is extended to include a number of penalty factors (Gritz termed these *Style Points*). As recommended by Gritz these style points are slowly phased in over the course of the run to avoid over-constraining early generations with hard constraints. The phase factor (Equation 4.2) is calculated simply by the square root of the current generation over the total number of generations, resulting in a value that grows incrementally with each generation adding higher penatlies to more evolved characters.

$$Phase\ Factor = \sqrt{generations/maxGenerations} \tag{4.2}$$

4.0.8 Visualisation

The visualisation of the system has both a basic representation and exported output. The basic visualisation is given via a Open-GL context and displays a real-time interactive window of the character whilst the simulation is in progress. This display is a basic representation of the scene for quick viewing during runtime or review.

4.0.8.1 Export

A further export system was produced to write the character structure and all motion to the BVH (Bvh) motion capture file format. This data is read into Houdini (which nativly supports .bvh files) using the *mcbiovision* script to form a simple rig with the motion data attached. (Example Appendix:C)

A further Python script was written to allow user access and manipulation of the hierarchy implicity created by Houdini. This gives the user freedom to reconstruct the rig with primitive geometry (Figure 4.5), a chosen mesh (Figure 4.6) or dynamic components (Figure 4.7).

Figure 4.5: *An example render of the creature character reconstructed with primitive geometry and rendered with Mantra*

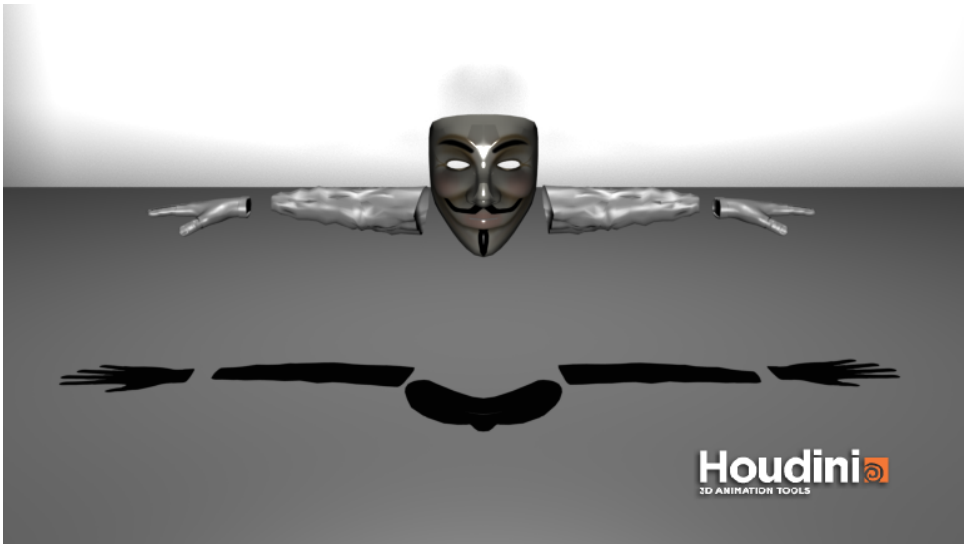

Figure 4.6: *An example render of the creature character reconstructed with user defined mesh elements for each section*

Figure 4.7: *An example render of the creature character reconstructed with components from the Houdini dynamics system.*

4.0.9 Command Line

The system is set up to initiate through a simple command line prompt. The user must input the path to the configuration file used to intialise the system and an output path for the final data files to be saved.
All backup files* are stored implicitly within the program's directory and are automatically overwritten with each run.

When the system intiates a simple OpenGL viewer is used for display with all relevant system information written to the screen. A graphical user interface was not made for this system due to the simple intialisation and the time taken for the simulation to run it was deemed unnecesary. (Figure 4.8)

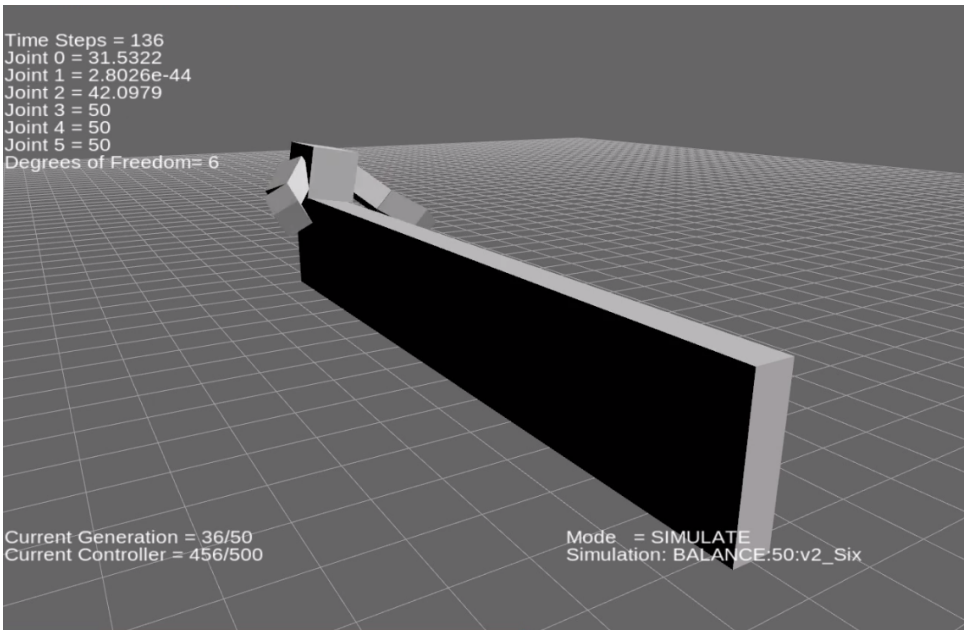

Figure 4.8: *OpenGL Runtime Interface. A simple interface to display key data whilst the simulation is running.*

**Because of the long length of the simulations the complete current generation of controllers are written to file at the end of each generation. This is very inefficient but provides a backup to avoid losing all current progress if the simulations are stopped for any reason, which happened on numerous occasions.*

Chapter 5

Testing

Due to the large number of variables within this system and potentially long simulation times a testing stage was established for this project to identify the best system setup that would be the most rewarding within the project time frame.

5.1 Identifying Time Scales

The common literature within this field argues that a balance is needed between the complexity of the system and the amount of time run. The simulation setup is directly connected to the overall exploration of the search space: too little time given and not enough solutions are explored, too much time and the system will stagnate after finding an optimal solution. This balance is identified by the population size and generation length parameters whilst the size of the search space is a direct consequence of the complexity of the character (Number of DOF).

The run time of the simulation is not affected by computational power, in fact the power needed for the each simulation is extremely minimal, as such we were able to run up to 10 simulations on a single machine with little sign of any deterioration. The run time of the simulation is a direct product of the intialisation parameters and the simulation length. A member must be run up to a time limit. This is repeated for each member of the population and this process is repeated for the given number of generations. A rough overview given that every member is run for the maximum amount of time is displayed in the table below(Figure 5.1).

Table 5.1: *Breakdown of the time scales for simulation run-time*

Population Size	Generations	Max Sim Length (Seconds)	Run Time (Days)
10000	100	40	462
1000	100	40	46
500	100	40	23
500	50	40	11

It is evident from these estimates that these timescales are far outside the range of time acceptable for this project and further measures were taken to reduce these times as much as possible.

5.2 Constraints

To reduce the simulation time of each member a number of constraints were added to end the simulation early, restricting the number of simulations that ran until the maximum time.

Intial tests were run with a variable to fail members that disobeyed the constraints, however this proved to be too restrictive and many of the simulations resulted in a minimal variation of members. As such this was removed and replaced simply with an early exit and the results of the parameters were included as soft constraints within the fitness function instead.

The hard contraints applied to the simulations were as follows:

• **Maximum Height:** Exits any simulation in which the characters Y position is over twice its bodies height, reducing members that use excess force to jump to the goal.
• **Inactivity Check:** Checks the simulation every 100 timesteps and compares the difference between its old position and new position. Reduces members that make no attempt to move within the simulation and stops simulations that may have performed well but have now come to a stop.
• **Forward Direction Check:** Checks the dot product of the characters Z-axis against the world Z-axis exiting any simulation that returns lower than 0.2. Reduces members that are facing backwards, but allows members to face sideways.
• **Horizontal Check:** Checks the dot product of the characters Y-axis against the world Y-axis to check if the character has become horizontal. This reduces members that have either fallen over and are unable to get back up or members that produce a rolling behaviour.

5.3 Parameters

5.3.1 Population & Generations

An initial set of simulations were run following the conditions given by Jones and using a population of 1000 with 100 generations (The maximum time was left at 40 seconds as it was rarely reached by any member). Even with the constraints included these simulations took a little over two weeks to run and as such were not suffcient as parameters to make regular tests.
Further tests were made with the parameters given by Gritz that established the ideal number of generations between 30-40 and the population size around 300-500. Tests were run with a fixed population of 500 and generations ranging between 20-60. Gritz claimed that these were adequate for his simulations and larger tests resulted in little improvement.
The results established that the generations lower than 50 showed little sign of improvement, with members making fairly good improvement at 50 generations whilst showing negligable improvement at 60 generations. Further testing of the these parameters is needed to establish the optimal setup but for this project a population of 500 with Generations set to 50 was chosen as the default for the final simulations.

The reduction in these parameters produced simulations that completed within 3-4 days, making them acceptable for the time frame for this project.

5.3.2 Other Parameters

Due to limited time to test the effects of varying the many combinations of parameters, all other parameters were fixed to given values as recommended by the following literature (Koza 1992) , (Kinnear 1994) & (Banzhaf *et al.* 1998). The Tournament size was set to 10, Mutation and Crossover rates were set to 0.1 and 0.9 respectively. The system is designed to handle both size and depth limits for Genome production but only the size parameter was tested with a value of 200.

5.4 Fitness Test

The main fitness function used for the final simulations is based on the function provided by Jones with small modifications. A small number of tests were run with varying fitness functions, reducing the tests including checks for forward direction, horizontal checks or height limits. As expected these produced extremely poor results with members producing behaviour that was not characteristically natural. The fitness function used can be found in (Appendix:A).

The Function includes checks for the following soft constraints:

• **Distance Travelled:** Checks the distance between the character and the target position providing the primary fitness for the character.
• **Maximum Height:** Members who exceed the height limit are exited early from the simulation and their maximum height is recorded as a penalty.
• **Off Course:** Penalises the member for excessive distance travelled out of its primary axis (In this case the x-axis).
• **Direction:** Penalty for not facing forward whilst travelling.
• **Early Exit:** Members who finish early are generally inadequate and so members are favoured who last longer as this generally means they are moving with some measure of efficiency.

5.5 Crossover Design

The original literature provided by Koza in (Koza 1992) and established as the traditional approach to genetic programming states that during crossover a test is performed on the children produced to ensure that they fall within the size parameters set for the genome. Koza states that if a child is outside of the acceptable range given then the child is discarded and replaced with a copy of one of the parents. The result of this is that the population becomes flooded with copies of identical genomes and breeding these genomes is likely to further the cycle.

When testing the simulations a version that followed Koza's principle was tested alongside a second version that simply created a new random genome upon failing the structure test. This second version attempts to add genetic diversity into the system, even if many of these new members will be poorly suited and removed at the end of the generation (especially during later generations).

This test resulted in a clear distinction between the two versions. The test based on Koza's method, whilst producing members that solved the task well, generated populations with many identical controllers, lacking in any kind of diversity once a local optimum had been found. The second version produced a far wider range of controllers establishing the attempt at genetic diversity as a minor success. However due to the larger variation the members that produced good behaviour, although clearly effective were not always as established as the Koza method.
It was decided that both features had positive elements and as such the final tests were ran with a samples of each type.

The following graph (Figure 5.1) presents data collected from the simulations displaying the average fitness values for both cases and as expected the average fitness increases over the course of the generations. A clear distinction can be made to the second version of the simulation displaying the average fitness improving at a faster rate than Koza's method before becoming relatively similar towards the final genera-

tions. This difference could possibly display the wider variation of individuals in the initial population using the second version resulting in an increased average fitness where the repeated controllers of the Koza approach provides a range and therefore limits the average.

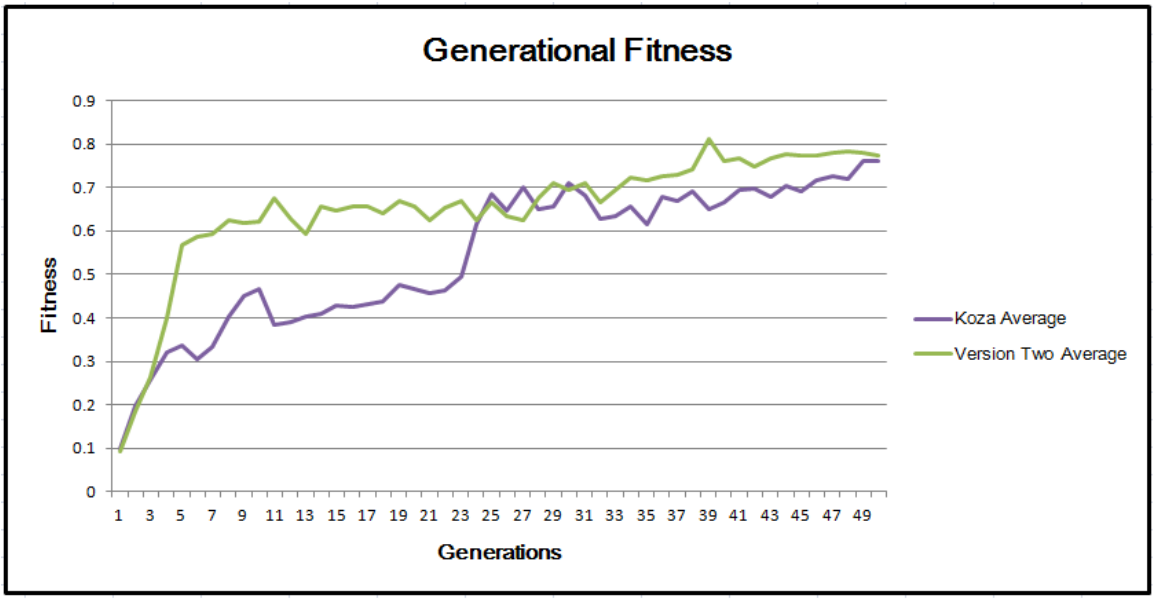

Figure 5.1: *Average Fitness across 50 generations for both crossover approaches*

Chapter 6

Results

Two tasks were designed for this project providing a slight variation in activity and challenge for the system to adapt to. The results of both tasks proved promising with a small range of motions generated that were at once surprising and unexpectedly natural.

For each test a small variation was made to test slightly varied system setups and these again caused surprising results including the clear exploitation of loop holes within the system. Whilst these may not be not visually interesting they provide clear evidence of a system that thoroughly explores the search space.

6.1 Walking Test

The basic task designed for the system simply required the character to reduce its proximity to a given point in space by any means possible. This is the simplest task and provided a base level indication of the effectiveness of the system. Though the evolution of an efficient walking behaviour would have been pleasing it was not expected within the limited simulation size and simulations were run in the hopes of efficient natural locomotion of any kind.

The accompanying videos display samples of both rendered and un-rendered* output.

6.1.1 Loop Hole

An interesting behaviour evolved during these runs in which in many cases the character simply moved forwards a small amount and stopped. This was a curious behaviour and it would be expected that this behaviour would be culled out early on. However, this controller propagated through the generations resulting in a small number of tests with little to no movement. These tests though visually uninteresting provide

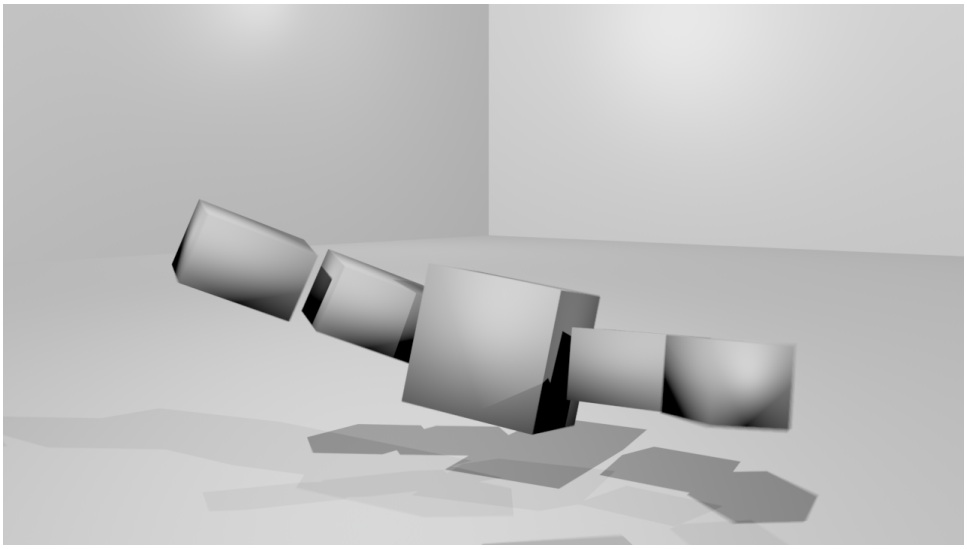

Figure 6.1: *The character is tasked with moving as far as possible across the plane*

behaviour that shows the exploitative properties of the system.

Characters that move a significant distance during their run are rewarded with a higher fitness value, however they are also highly susceptible to penalties from soft constraints that will balance out and reduce their fitness. A small number of simulations thus "discovered" that by moving only a small distance, enough to gain a small base fitness increase then stopping, they gained enough base fitness and reduced the amount of penalties issued to them. This enabled them to keep a relatively high fitness ranking and surviving the generation with little work!

6.2 Balance Test

A more challenging task was designed to test the adaptive properties of the system within a reduced solution space. This task involved the character reducing their distance to a target as before but instead of a large ground plane the character was placed on a thin beam in which ground contact was prohibited. The idea of this test was to encourage more intelligent behaviour, limiting the range of movements by reducing the contact area beneath the character.

The results display a small range of techniques developed to accomplish the task. As was expected no character moved across the entirety of the beam but nevertheless the range of techniques developed show promising behaviour that could be developed with more time.
An interesting walk-like behaviour became a popular choice across multiple simulations and clearly shows an attempt at locomotion that is both familiar and natural. Another popular behaviour was the vaulting behaviour, whilst relatively unstable

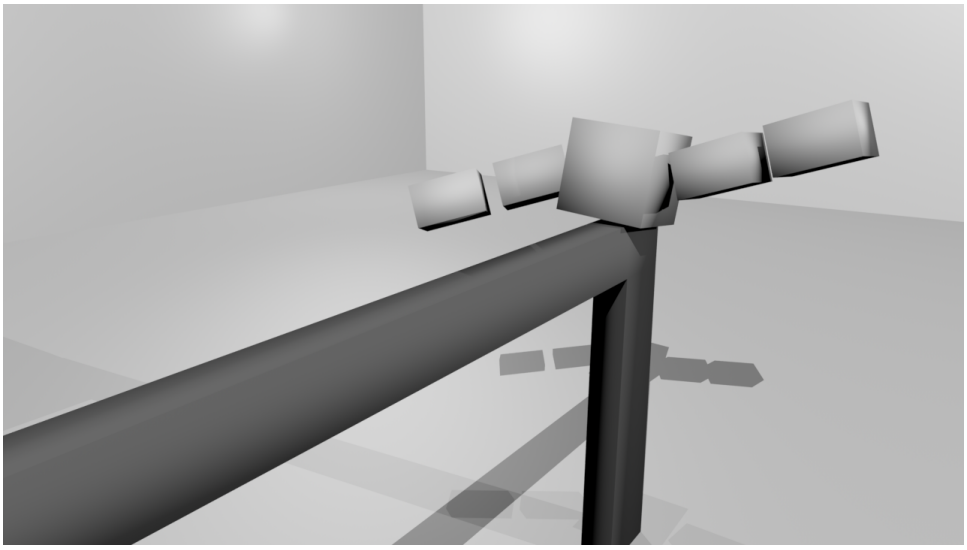

Figure 6.2: *The character is tasked with traversing across the beam as far a possible*

provided a large increase in the distance travelled with a single motion.

• Variation - Reduced Constraints: A small test was run to test the effectiveness of the constraints on the system and to compare a heavily constrained system against a loosely constrained one. As expected the character can be seen to simply vault itself through the air, although achieving the task, the naturalness of the performance is severely reduced.

6.3 Conclusions

The results gained from the series of simulations run clearly show a progression over the course of the simulation and a small variety of interesting behaviours were developed that both prove the capabilities of the system and the effectiveness of genetic programming in general. The main problem to conclude from these results is that the lack of time (generations) and diversity within the runs (population size) has clearly had a negative impact on the overall result. Whilst motions have developed, their progression is inhibited.
There is confidence that if this system was run with a larger parameter set and given sufficient time, behaviour would develop that would be considered more substantial. A system design that incorporated multi-threaded processing would also be a benefit in increasing the productivity of the system.

**Due to technical difficulties with some of the simulations a number of simulations were unable to be rendered out. A compilation of un-rendered footage has been provided to display some of these simulations that provided interesting behaviour as well as*

some of the outputs that were also later rendered.

Chapter 7

Conclusion

The aim of this study was to investigate the effectiveness of genetic programming and its applicability as a tool to the field of animation. This task was carried out through the further investigation of genetically evolved controllers, as proposed by Karl Sims and featured many of the key advancements established by Larry Gritz.

This work featured the development of control algorithms being developed in a fashion that is heavily influenced by natural/biological processes. This makes the design theory easy to understand and appreciate to readers with only a little knowledge of these computer systems, as much of the technical detail and resulting outcome can be described in pseudo-biological terms. However due to this close mirroring in design, the actual functionality of the system quickly becomes heavily convoluted with a vast number of variables and codependant elements that mean the smallest changes have drastic effects. Whilst a familarity with the system was gained during implementation, it is not an intuitive process and can be difficult to explain let alone run by an uninformed user.

A small number of behaviours were developed during this project that give strong evidence towards the power of genetic programming. These behaviours show intuitive solutions to the problems presented and with more processing time could be enhanced further with interesting results. This aspect is the first major flaw in the system. Much work was made to reduce the simulation time as much as possible but even so the resulting length of the simulations made the whole process extremely difficult with simulations taking days or even weeks to run. With small errors being difficult to identify until late in the simulation many long runs were forced to be cancelled and a significant amount of time was lost.

One of the major problems proposed by Gritz and further outlined by Craig Reynolds (Craig 1994) is the brittleness of the controllers developed. During this project it was found that whilst the behaviour within the simulation can be interesting, extracting that behaviour was frequently difficult. The repeatability of a simulation is governed on the fact of every variable being identical to the original, a

small bug in the system adding a error to the joint values makes the playback of the simulation entirely different from the original, effectively making the playback system redundant.

7.1 Summary

Genetic programming as a search technique was proven in this project to be more than capable of finding interesting solutions and even the small amounts of behaviour developed here were extremely rewarding. It is an enjoyable moment to see a completely random formation of elements creating seemingly intelligent motion and offering a glimpse of the artificial development of a natural locomotive pattern. Even the visually uninteresting events created by the system finding logical loop holes provide a thought-provoking notion of intelligence.

Unfortunately the process contains a large number of flaws as discussed that makes it a difficult process to work with. The extended run-time and fragility of the simulations severely stalled the progress made within this project and many of the planned features and extensions were removed due to time contraints. With the same time given to simulation and testing as many of the source papers there is confidence that some truly interesting outcomes could be developed. However there is a belief that even after this extended development time the controllers developed will still have the fundamental flaws described and though providing interesting visual stimulus in the simulation environment, the outcome would be completly un usable within any real animation environment.

Modern systems using advanced formats such as the ones discussed in the related work show promise for this area for use within industry. However the author concludes that from the process of this project and the results given the genetic process in this format provides an interesting demonstration of the evolution of behaviour and whilst extremely interesting in its own right and applicable to further research it has little application as a tool in commercial animation.

7.2 Future work

Though it is felt that little change can be made to improve the robustness and usability of this system in its current format there are a number of interesting investigations and extensions that could be applied to this system to enhance the behaviour developed.

7.2.1 Bloat Investigation

(Silva and Costa 2005) investigate the addition of a dynamic bloat system within the development of the genome. The paper reports on the inclusion of a dynamic genome size that adjusts itself accordingly to match the current best member of the population. This allows for the size of the genome to adapt accordingly extending the length of the algorithm generated if it is seen to be improving the system.

7.2.2 Intron Investigation

Introns are a biological element of the DNA structure and explored by Koza briefly in a computational context in (Koza 1992). An intron is an un-used element within the genome structure that does not explicity add any input to the system. However there is interest in investigating the part these introns play in structural design as it has been proven biologically that although they themselves add little, their inclusion performs a building block function in which they help to form the larger structure of the genome.

7.2.3 Predatory Behaviour

Many of the previous papers have explored this process with varying character designs and shown that the evolutionary process develops compelling behaviour when tasked with a problem. However in the natural world, simple problem solving ability is only a portion of the evolutionary pressures applied to an individual and in order to proceed to the next generation an individual must also survive predatorial pressures. This investigation may cause a more rapid form of development as even the fittest members of the population are susceptible to a predatory threat.

Bibliography

Banzhaf W., Francone F. D., Keller R. E. and Nordin P., 1998. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Bvh . Biovision hierarchy motion capture format.

Clune J., Mouret J.-B. and Lipson H., 2012. The evolutionary origins of modularity. *CoRR*, **abs/1207.2743**.

Coros S., Karpathy A., Jones B., Reveret L. and van de Panne M., 2011. Locomotion skills for simulated quadrupeds. *ACM Transactions on Graphics*, **30**(4), Article TBD.

Craig R., 1994. Advances in genetic programming. MIT Press, 221–241.

Darwin C., 1869. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life.* John Murray.

Dawkins R., 1996. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*. Norton.

Gehring C., Coros S., Hutter M., Bloesch M., Fankhauser P., Hoepflinger M. A. and Siegwart R., 2014. Towards Automatic Discovery of Agile Gaits for Quadrupedal Robots. In *IEEE International Conference on Robotics and Automation (ICRA 2014)*.

Geijtenbeek T. and van de Panne , November 2013. Flexible muscle-based locomotion for bipedal creatures. *ACM Trans. Graph.*, **32**(6), 206:1–206:11.

Gritz L. and Hahn J. K., 1995. Genetic programming for articulated figure motion. *Journal of Visualization and Computer Animation*, **6**, 129–142.

Gritz L. and Hahn J. K., "1997". "genetic programming evolution of controllers for 3d character animation". In *"Genetic Programming Annual Conference"*, "139–146".

Holland J. H., 1975. *Adaption in Natural and Artificial Systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press.

Holland J. H., 1992. *Adaption in Natural and Artificial Systems*. MIT Press.

Jones M. P. Evolving behaviours using genetic programming. Master's thesis, NCCA Bournemouth University, UK, 2006.

Kinnear K. E., Jr., editor, 1994. *Advances in Genetic Programming*. MIT Press, Cambridge, MA, USA.

Koza J. R., 1992. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press.

Poli R., Langdon W. B. and McPhee N. F., 2008. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`.

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

Reil T. and Massey C., 2011. Biologically inspired control of physically simulated bipeds. *Theory in Biosciences*, **120**(3-4), 327–339.

Silva S. and Costa E., 2005. Genetic programming and genetic and evolutionary computation conference (gecco).

Sims K., 1994. Evolving virtual creatures. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, New York, NY, USA. ACM, 15–22.

Stanley K. O., 2002. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, **10**, 99 – 127.

Appendix A

Fitness Function

This was the basic fitness function used, adapted from the function given by Jones(Jones 2006).

```
    void fitnessFunctionTwo(Controller* _controller,ODECharacter* _character,float _phase)
{
    ConfigParser* config = ConfigParser::instance();
    float fitness, penalty;
    float maxY = _character->getMaxPosition().m_y;
    float maxX = _character->getMaxPosition().m_x;
    float fdir = _character->getMinDirection().m_z;
    float characterHeight = _character->getRoot()->getSize().m_y;
    int steps = _character->getStepsTaken();
    int maxSteps = config->getMaxSimSteps();
    if(_controller->hasFailed())
    {
        fitness = 0;
        penalty = -5;
    }
    else
    {   //BASE FITNESS BASED ON DISTANCE TRAVELLED
        fitness = UTIL::mapValues(_character->getPosition().m_z,20.0f,0.0f,4.0f,0.0f);
        //PENALTIES
        float height = UTIL::mapValues(maxY,characterHeight+1,characterHeight,1.0f,0.0f);
        float axis = UTIL::mapValues(maxX,20.0f,-20.0f,1.0f,-1.0f);
        if(axis < 0.0f){axis*=-1.0f;} // Both Positive and Negative traversal in X
        float direction = UTIL::mapValues(fdir,1.0f,-1.0f,0.0f,1.0f);
        float earlyExit = (1.0f-(1.0f*steps)/maxSteps);
        penalty = -(height+axis+direction+earlyExit);
    }
    fitness = fitness + (penalty*_phase); //GRITZ PHASE FACTOR
    _controller->setFitness(fitness);
}
```

Appendix B

Controller

Example of an XML file gnerated to store the controller for later access.

```
<?xml version="1.0" encoding="UTF-8"?>
<CONTROLLER-FILE>
        <CONTROLLER>
                <Size>6</Size>
                <GENOME>
                        <NODE>
                                <Type>BINARY</Type>
                                <Parent>NULL</Parent>
                                <Function>*</Function>
                                <LeftChild>
                                        <NODE>
                                                <Type>UNARY</Type>
                                                <Parent>BINARY</Parent>
                                                <Function>ABS</Function>
                                                <Child>
                                                        <NODE>
                                                                <Type>VARIABLE</Type>
                                                                <Parent>UNARY</Parent>
                                                                <Function>POSITION</Function>
                                                                <Axis>1</Axis>
                                                                <Joint>0</Joint>
                                                        </NODE>
                                                </Child>
                                        </NODE>
                                </LeftChild>
                                <RightChild>
                                        <NODE>
                                                <Type>UNARY</Type>
                                                <Parent>BINARY</Parent>
                                                <Function>ABS</Function>
                                                <Child>
                                                        <NODE>
                                                                <Type>TERMINAL</Type>
                                                                <Parent>UNARY</Parent>
                                                                <Value>2</Value>
                                                        </NODE>
                                                </Child>
                                        </NODE>
                                </RightChild>
                        </NODE>
                </GENOME>
        </CONTROLLER>
</CONTROLLER-FILE>
```

Appendix C

BVH

Example of a .bvh file containing the character hierarchy and accompanying motion data.

```
HIERARCHY
ROOT Body
{
OFFSET 0 0 0
CHANNELS 6 Xposition Yposition Zposition Yrotation Zrotation Xrotation
JOINT LeftShoulder
{
OFFSET   0 -0.6 0
CHANNELS 3 Yrotation Xrotation Zrotation
JOINT LeftElbow
{
OFFSET  0 -1 0
CHANNELS 3 Yrotation Zrotation Xrotation
End Site
{
OFFSET  0 -1 0
}
}
}
JOINT RightShoulder
{
OFFSET 0 0.6 0
CHANNELS 3 Yrotation Xrotation Zrotation
JOINT RightElbow
{
OFFSET  0 1 0
CHANNELS 3 Yrotation Zrotation Xrotation
End Site
{
OFFSET  0 1 0
}
}
}
}
MOTION
Frames: 1772
Frame Time: 0.009
0 2 1 90 0 90  0 0 -0 0 0 0 0 -0 0 0 -0 0
0 1.99920535 1 90 0 90  0 0 -0 0 0 0 0 -0 0 0 -0 0
-7.83367381e-12 1.9979198 0.999580443 90 0.0201153085 89.9806061
2.94803726e-07 1.99614322 0.999241531 90.0000534 0.00781303644
...
```

Appendix D

Configuration

Example of the configuration file used to initialise the system.

```
<?xml version="1.0" encoding="UTF-8" ?>
<GENETIC-CONFIG-FILE>
  <GeneticVariables>
  <Generations>       5     </Generations><!-- The Number of Generations to Run for -->
  <PopulationSize>        10    </PopulationSize><!-- The Size of each Population -->
  <MaxNodeGrowth>     200   </MaxNodeGrowth><!--  The Maximum Nodes a Genome can grow for -->
  <GrowthLimitType>    SIZE  </GrowthLimitType> <!--  The Limit used for growth: DEPTH or SIZE-->
  <TournamentSize>      2      </TournamentSize><!-- The Number of Genomes selected for a Tournament -->
  <CrossOverRate>          0.9    </CrossOverRate><!-- The CrossoverRate during Re-Population -->
  <MutationRate>        0.1    </MutationRate><!-- The MutationRate during Re-Population -->
  </GeneticVariables>
    <GeneticOperators>
                <BinaryOperators>
                        <BinaryOperator>                +       </BinaryOperator>
                        <BinaryOperator>                -       </BinaryOperator>
                        <BinaryOperator>                *       </BinaryOperator>
                        <BinaryOperator>                /       </BinaryOperator>
                </BinaryOperators>
                <UnaryOperators>
                        <UnaryOperator>         ABS     </UnaryOperator>
                </UnaryOperators>
                <TernaryOperators>
                        <TernaryOperator> IFLZ </TernaryOperator>
                </TernaryOperators>
        </GeneticOperators>
        <Physics>
                <Gravity>       -9.81   </Gravity><!-- Default Gravity recommended by ODE -->
        </Physics>
        <Other>
                <MaxSimSteps>   2500    </MaxSimSteps>
        </Other>
        </GENETIC-CONFIG-FILE>
```

Appendix E

Function Set

Complete Set of Functions coded into the system

Function Set:
- + : Addition Operator
- - : Subtraction Operator
- * : Multiplication Operator
- / : Division Operator(Protected Divide)
- ABS : Absolute Value
- IFLZ : If Less than Zero
- <: Less than Operator
- >: Greater than Operator

[+, - , * , / , ABS, IFLZ, GT (>), LT(<)]

Appendix F

Physics Settings

The default settings of the ODE physics engine variables given for this project.

- Gravity = -9.81 (Y Axis)
- Simulation Stepsize = 0.009;
- LinearDamping= 0.0001;

World:
- AngularDamping= 0.005;
- MaxAngularSpeed=2 00;
- MaxCorrectVelocity= 0.1;
- ContactSurfaceLayer= 0.001;
- WorldCFM (1e-5);
- WorldERP(0.4);

Collision:
- SurfaceMode = dContactBounce;
- Surface.mu = dInfinity;
- Surface.mu2 = 0;
- Surface.bounce = 0.05;
- Surface.bounce_vel = 0.5;
- Surface.soft_cfm = 0.001;
- Surface.soft_erp = 0.6;