Demonstrating Darwinian Evolution Using Swarm

Lorcan Coyle

Lorcan.Coyle@cs.tcd.ie

http://www.cs.tcd.ie/Lorcan.Coyle/FYP/DESwarm.pdf

B.A.I in Computer Engineering

Final Year Project May 2001

Supervisor: Mike Brady

Abstract

This report details the design and implementation of an artificial world that demonstrates Darwinian evolution. This is done using an agent based modelling tool called Swarm. This world is populated with hundreds of agents that compete with one another to survive. Their behavioural attributes are coded into their genes. The initial population consists of individuals with randomly coded genes. It is hoped that by exerting Darwinian evolution on this, a population of agents, optimally suited for survival in the world will emerge.

Acknowledgements

I would like to start by thanking Mike Brady for agreeing to supervise this project. His time and patience were very much appreciated. Without his direction there is a good chance that a sizable part of this report would have been a description of the behaviour of *Atta Rodona* (the leaf cutter ants), which was originally what I wanted to model. Happily I listened to him and I believe the project is better for it.

Many thanks also to Adrian Fitzpatrick, Suzanne Grogan, Ciara Coyle, Gordon Power and Barry Brady for their help in the editing process.

Table of Contents

Chapter 1 - Introduction

1.1 Introduction to the Report

This report describes the design of model that demonstrates Darwinian evolution. "Darwin's *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, made several points that defined what we know as Darwinian evolution:

- That types or species do not have a fixed, static existence but exist in permanent states of change and flux
- That all life, takes the form of a struggle to exist -- more exactly, to exist and produce the greatest number of offspring
- That this struggle for existence culls out those organisms less well adapted to any particular ecology and allows those better adapted to flourish -- a process called Natural Selection"[1]

This model to demonstrate this takes the form of an artificial world called the heatspace, which is inhabited by bugs who struggle to survive on it. These bugs evolve over time into a population optimally developed for survival on the heatspace. The model is implemented using the agent based modelling tool Swarm. This report outlines the design and implementation of this world and explains the information that was gathered in running the model.

1.2 Intended Audience

This report is intended for readers at sophister level in a Computer Science degree. It assumes some knowledge on the part of the reader in the areas of Java programming, intelligent agents, cellular automata, evolutionary algorithms and genetics.

[1] From a course by David Lindsay at the University of Texas available online at http://www.cwrl.utexas.edu/~syverson/worldsfair/exhibits/hall2/lindsay/page1.htm

1.3 Motivation

The motivation behind this project came from a number of projects in the ALife field. These demonstrated different aspects of physical and societal evolution. It was decided that this project should demonstrate physical evolution. It was hoped that this would make it possible to evolve solutions for problems involving interactions between large numbers of individuals (e.g. rule systems for economics or crowd control).
Swarm is a simulation and modelling tool available free online. This was found to meet the requirements of this project. It was important to master this tool to implement the project fully.

1.4 Project Goals

The main goals of this project are as follows:

- To create a model that demonstrates Darwinian Evolution.
- To gain proficiency in the use of Swarm.

1.5 Report Outline

This report is divided into seven chapters structured as follows:

- Chapter One gives an introduction to the report, stating project aims and outlining the layout of the report.
- Chapter Two deals with background information relevant to the project and other projects in the general area.
- Chapter Three is concerned with the design of the application.
- Chapter Four introduces Swarm and explains why it was used.
- Chapter Five details the implementation of the project.
- Chapter Six analyses the information garnered from the runs of the simulation.
- Chapter Seven summarises the report, and contains the conclusions of the report, and what subsequent future work could be done in the project area.

1.6 A Note on Web References

Some of the references listed are to web pages. Given the nature of the Internet there is no guarantee that these will still be there in the future. All the web pages listed in the bibliography were to established sites (such as University web sites). Even so, they have all been archived on the attached CD.

1.7 The Attached CD

The attached CD contains all the swarm libraries and instructions to install as well as archived versions of all the web pages referenced in this report. The full code listing is given as well as instructions as to how to run the model and use the built-in debugger. A list of the contents of the CD is given in Appendix C.

1.8 Summary

In this chapter, the project and report were introduced. The motivation behind the project was discussed and the structure of the report was presented by stating the content of each chapter. Finally, the attached CD and its contents were outlined.

Chapter 2 - Background

2.1 Introduction

There are a number of groups working on similar projects to this. The main one is the ALife community. There is also the Sugarscape Project and Gene Pool on which this project is loosely based. This chapter introduces each of these.

2.2 ALife

"Artificial Life (ALife) is a field of study devoted to understanding life by attempting to abstract the fundamental dynamical principles underlying biological phenomena, and recreating these dynamics in other physical media -- such as computers -- making them accessible to new kinds of experimental manipulation and testing." [2] The ALife community has a large presence on the web and overlap with a number of other groups, such as the Santa Fe Institute and the Swarm Development Group. The techniques being developed in the ALife community are being applied to problems in fields totally unrelated to the synthesis of life.

2.3 Sugarscape

Joshua M. Epstein and Robert L. Axtell, two members of the Santa Fe Institute, set out to apply agent-based computer modelling techniques to the study of complex human social phenomena, including trade, migration, group formation, combat, interaction with an environment, transmission of culture, propagation of disease, and population dynamics. Their broad aim was to begin the development of a computational approach that permits the study of these diverse spheres of human activity from an evolutionary perspective as a single social science. They created Sugarscape, a model that achieved this aim, and wrote a book (*Growing Artificial Societies Social Science from the Bottom Up*) detailing how they designed it. A screenshot from one of the Sugarscape preview movies is shown in figure 2(a) below. The agents are represented

[2] C. G. Langton. "Preface." In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, Volume X of *SFI Studies in the Sciences of Complexity*, pages xiii-xviii, Addison-Wesley, Redwood City, CA, 1992

by red dots and food by yellow ones. This screenshot hides all the underlying information analysis.

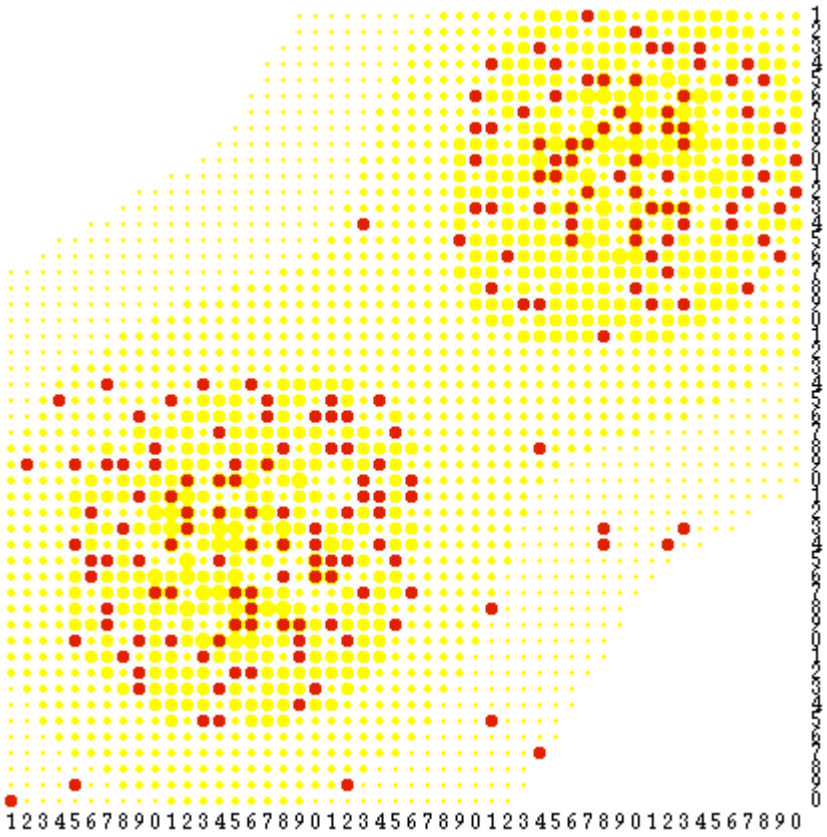

Figure 2(a): A Screenshot of Sugarscape in Action

This project is an extension of their idea; it models a simple system loosely based on Sugarscape. It ignores the higher-level social phenomena and focuses on the evolution of agent physical attributes. It leaves open the possibility of implementing other dynamics.

2.4 Gene Pool

Gene Pool is an artificial life simulation where populations of physics-based organisms evolve over time. These organisms are called "swimbots". The user can change mate preference criteria and thus influence what kinds of swimbots get chosen most often to breed new swimbots. Over time, they get better at pursuing each other and competing for food. It is best appreciated as a virtual Darwinian aquarium. The main differences between this project and *Gene Pool* is the fact

the later is physics based (the speed of the swimbots is its fitness factor – and this is defined using physics), and that it offers the user no idea what is going on under the surface of the model, no analysis of why the model does what it does. Its main strength is its simplicity; it is has a great graphical interface, which makes it easy to watch for hours. A screenshot of it in action is shown below in figure 2(b). This is zoomed in on an area where a group of multi-coloured swimbots are racing towards the two yellow dots, which represent food. The ones who reach the food first will be the ones who can shake their limbs (to propel themselves through the water) most efficiently.

Figure 2(b): A Screenshot of Gene Pool in Action

2.5 Summary

This chapter discussed the state of the art in ALife systems. It gave a brief introduction of two of the more famous models produced by this community. This project borrowed ideas from both of these models and extended them.

Chapter 3 - Designing the Model

3.1 Introduction

This chapter discusses the design of the different aspects of the project. It describes the model and its component parts. These are the environment, the population and the use of genetics. It then discusses the analysis tools and data required to give value to the model.

3.2 The Model

The model is made up of two main parts, the heatspace and the bugs. The bugs are the agents, the inhabitants of the model, and the heatspace is the environment in which they live. Every time-increment, or *day*, the state of the heatspace and the dynamic attributes (such as position) of the bugs are updated. The genetic makeup of the bug population evolves over time until the population is made up of bugs well adapted for survival on the heatspace.

3.3 The Heatspace

The heatspace is an n*m grid of cells, each of which is defined in terms of its x and y coordinates. Each cell has a heat level and is capable of being occupied by, at most, one bug. Cells can be harvested of their heat by any bug that occupies them. The heat levels grow back at a rate based on the time of the year. This is to give the appearance of seasons in the model.
The heatspace is designed using the theories of *Cellular Automata* (introduced by John Von Neumann in the late 1940s). They are discrete dynamical systems whose behaviour is completely specified in terms of a local relation. Space is represented by a uniform grid of cells each one containing data; time advances in discrete steps. Cells use the laws of the system and the states of its close neighbours to determine its next state. Thus, the system's laws are local and uniform. Neighbourhoods define the local interactions between cells. Each cell can only affect or be affected by its neighbouring cells. There are a number of neighbourhoods available for use. The one chosen for this project was the Moore Neighbourhood. This is an eight-direction where each cell interacts with its neighbours as shown in figure 3(a).

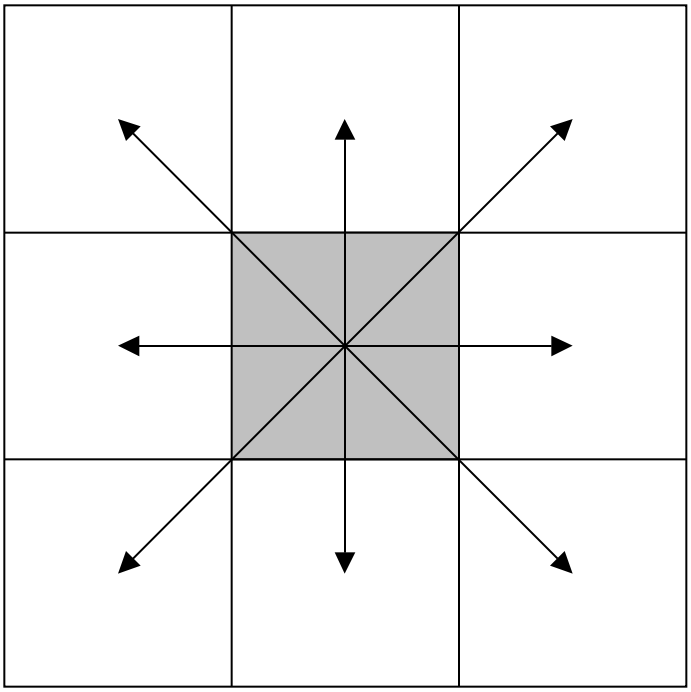

Figure 3(a): The Moore neighbourhood.

The grid is not infinite in this model and so boundary conditions had to be defined. The type chosen was a wrap-around boundary. This folded the chessboard shaped grid into a "toroidal" (or doughnut shaped) world. Any object passing the north-most cell on the chessboard would arrive at the corresponding south-most cell and vice versa. The same would happen an object crossing the East or West boundary. By this condition an object could continue travelling north (or any of the other cardinal directions) along the grid until it eventually arrived back where it started.

3.4 The Bugs

The bugs are the inhabitants of the heatspace. They harvest their "environment" for energy to survive, and die when this runs out. All bug interactions in the model are purely local. The bugs have no global view of the model and cannot interact with any bug or cell on the heatspace outside their local neighbourhood. They have a number of dynamic attributes; these are updated every day and are not directly dependent on the genetic makeup of the bug. These dynamic attributes (with their variable names in brackets) are:

- The current energy level (*energy*): the current energy level of the bug. If this falls below zero the bug will die.
- The bug's age (*age*): measured in number of days
- Current position on the heatspace (*x* and *y*. This information is not available to the bug itself, and is used only by the modeller.
- The bug's colour (*colour*): to be used as a diagnostic tool. This will give the user a view of the state of the population as the model runs.

They are also born with a number of static attributes that is, attributes that do not change during their lifetime. These attributes are stored in the genes of the bug. When a bug is born these attributes are decoded from its genes. These genetic attributes are:

- Metabolic Rate (*metaRate*): the amount of energy that the bug expends living each day, i.e. this energy toll is extracted from the bug every day; no matter what activity the bug is performing.
- Sight Radius (*sight*): The distance the bug can see dictates the size of its neighbourhood. Bugs with greater sight have more choice in the interactions they can make.
- Appetite (*appetite*): The maximum amount of energy that the bug can consume in a single day. It should be noted that there is no guarantee that there is this amount of energy in a cell.
- Energy received at Birth (*birthEnergy*): The energy the bug is born with. Its mother donates this at birth.
- Fertility Energy Level (*fertileEnergy*): The energy the bug needs to attain before it can bear children.
- Maximum Age (*maxAge*): The bug's maximum possible age. If the bug reaches this age it dies.
- The type of gene splicing used (*spliceOrCrossover*) and junk genetic (*junkCoefficient*) data: These are explained in the next section.

There are a number of other dynamic attributes used. These are used for diagnostic purposes only and have no effect on bug behaviour:

- The age (*firstChildAge*) the bug had its first child at (0 if this bug has no children).
- The number of children the bug has had so far in its life (*noOfChildren*).
- A unique number given to the bug on creation by the model (*index*). This tells us the total number of bugs created in the model.

Bugs follow a number of simple rules every day of their lifetime. A flowchart of these rules is shown in figure 3(b):

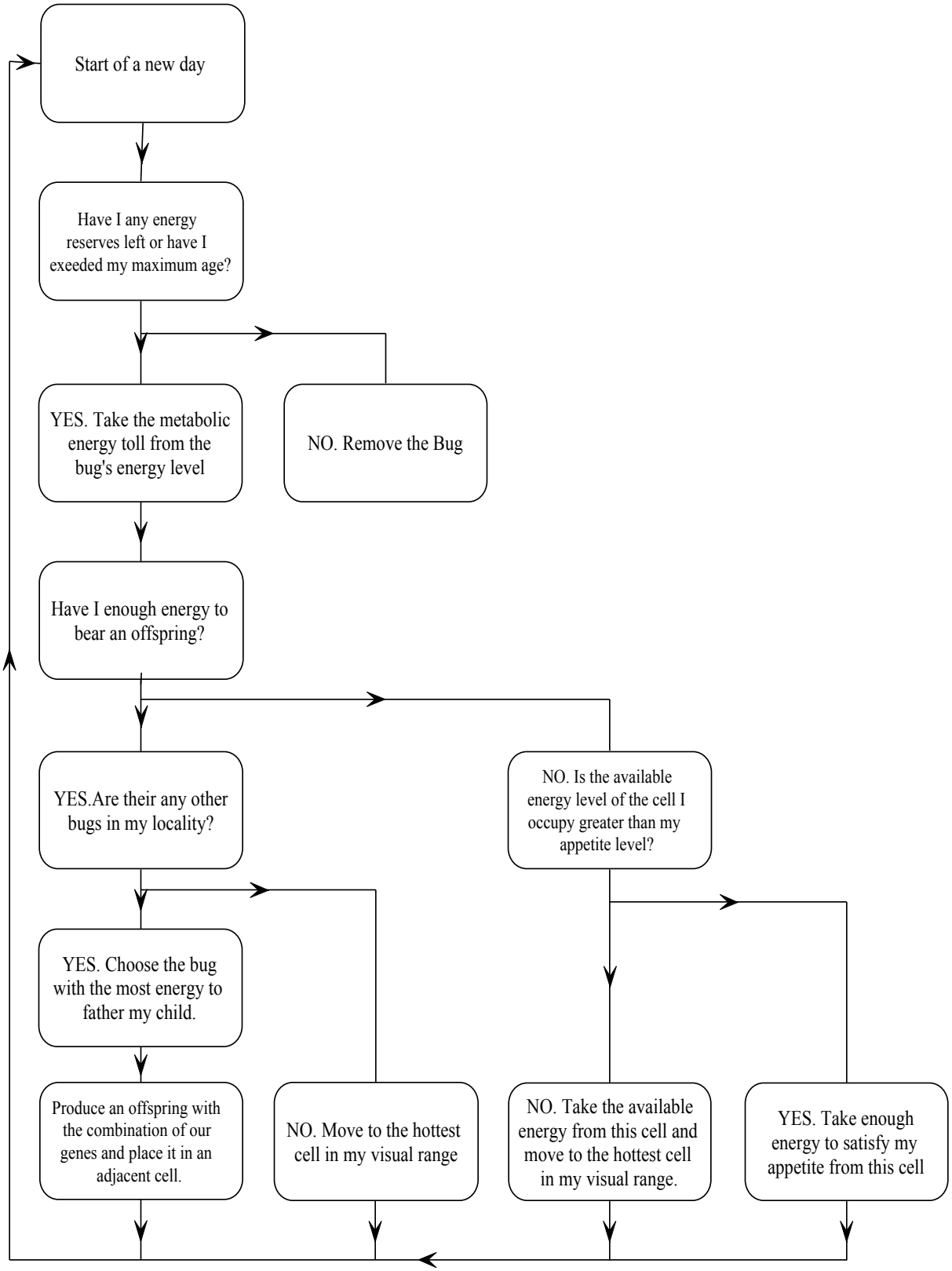

Figure 3(b): Flowchart of Bug Behaviour, Repeated Once Every Day.

3.5 Reproduction and Genetics

The main goal of the project was to demonstrate Darwinian Evolution. As such it was necessary to model the reproduction and genetics correctly. Reproduction in this model is sexual. This means that two participants are required for genetic combination. As such there are a number of possibilities for the population's sexual composition:

- That the population should consist of two sexes, one to play the roll of father and the other to play the roll of mother, that is to say, one should play a relatively passive role – the father, and an active participant - the mother who would lay the eggs, or carry the child to term (e.g. mammals or reptiles).
- That the population should consist of one sex who would play an equal role in reproducing, that would each play both rolls, thus producing one offspring each (e.g. snails)
- A combination of the two; the population consists of one sex, and individuals are capable of performing both roles. The role of mother carries an energy toll and is only undertaken by if she can afford it. Since the roll of father carries no such toll, any individual can play it.

The third option was chosen because it offered the most flexibility for the bugs. With a single-sex model there was less likelihood that a population crash would occur since any two bugs could form a new family group rather than a male and female. It enabled every bug to produce offspring. This was viewed as an advantage for the survival of a population over time.

Bugs are born with the same abilities and have the same behaviour as adults from birth. They do not begin their lives executing a "child" routine and compete with their parents as they would strangers. As such, there is no preferential treatment given by parents to children or vice versa.

If a mother finds a number of potential mates, she will chose the one with the highest energy level. It is possible for two mothers to mutually father the other's child in one day. That is to say that two bugs play both parental rolls, and two offspring are produced.

As was already mentioned, the static attributes of the bugs are stored in their genes. Since gene combination is used, a way must be found to combine two genes to produce a third that conforms to the theories of genetics. This was done in one of two ways:

- The common elements of the genes are copied and transferred to the child. Where elements differ, one of the parents' elements is chosen at random.
- A crossover point is chosen in the gene, and all genetic information up to that point is transferred from one parent, and from the other parent after that point.

To determine which of the two methods was superior, both were used side by side. If a bug became a mother, it would use the form of splicing coded in its genes to produce its offspring. In this way it is possible that a mother could have children that splice their genes differently than she does (if their father used different splicing and they inherited this from him). The type of gene splicing used was itself coded in the genes (*spliceOrCrossover*). It is hoped that by coding this into the genes the better method should emerge dominant over time.

It is important in genetics to ensure that the gene pool does not turn stagnant. That is to say that the entire population does not end up having identical genes. A distinction must be drawn here between achieving a population with a high fitness, which implies having more or less the same genetic makeup across the population, and a situation whereby every member of the population is related closely to each other. This is where the use of junk genes came in. Junk attributes were added to the genes of all bugs. This was random bits spread between the genetic attributes that didn't contribute in any way to the bug's physical makeup. Since the junk data in the genes does not contribute in any way to the attributes of the individual it should remain random across the population. Bugs closely related to each other would be expected to have similar junk genes (e.g. bugs who shared the same mother and father). Therefore if the entire population had identical junk data, it would be fair to say they shared a common ancestry. This is a question of statistics more than anything else; it is quite possible that a population could form with identical junk data but quite different attributes, so this too was monitored.

3.6 The Analysis

The model is useless unless we can get information from it and draw conclusions from this. Since the characteristics of the population and environment are constantly changing, a number of probes must be set up to monitor these. Graphs of the following attributes against time could prove to be useful:

- The average energy level across the heatspace. This will give an idea of the effect of seasons and the bugs themselves on the heatspace.
- The current population. By comparing this to the energy level of the heatspace, it will be possible to see the influence each will have on the other (e.g. predicting famines due to overpopulation). This information will also be important in determining the effective size of the gene pool.
- The average genetic attributes across the population. Which will allow the realisation of the main goal of the project.
- Other demographic information (such as birth and death rates or the age profile of the population) that may also be useful.

3.7 Summary

This chapter outlined the design of the project. It began by explaining the format of the model and then went on to discuss its components, the heatspace (the environment), bugs (the inhabitants), and the types of genetics used. Finally the important characteristics of the model that are to be analysed were outlined.

Chapter 4 – Chosen Technology

4.1 Introduction

This chapter introduces Agent Based Modelling as the technology to implement the project and Swarm as the tool. It gives reasons for the choice of Java as the implementation language and introduces some of the components of Swarm's Activity library. Finally it looks at some of the areas where Swarm is currently being used, the key advantages of using it and its future.

4.2 Agent Based Modelling (ABM)

The main reason for using ABM is to get an idea of the global dynamics of a system by building it from the bottom-up. That is to say, using synthesis rather than analysis as a means of determining the global dynamics. With synthesis, the modeller aims to accurately describe a system's components and plausible interactions, and then use a realisation of that description as an empirical basis for study of the systems' global dynamics. Traditional mathematical methods (Ordinary Differential Equations, Partial Differential Equations and statistical approaches) can describe macroscopic properties of a system that are already known, but they don't explain the origin of those properties or handle discontinuous systems or heterogeneity in populations well. ABM complements and enhances rather than supplants these traditional approaches.

"An *agent* is the colloquial term for pretty much any component in an ABM that has extent. Agents have:

- Internal data representations (memory or state)
- Means for modifying their internal data representations (perceptions)
- Means for modifying their environment (behaviours)" [3]

Agents should have no global knowledge (i.e. no absolute co-ordinates). They are only capable of interacting with their local environment and neighbours. This makes for a more realistic and flexible model.

Agent-based modelling is generally interested in the dynamics of collections of agents. Agent-based models usually involve taking a one or more agent types and making many copies (or

[3] From a tutorial offered at the Santa Fe Institute's 2000 Complex Systems Summer School (slide 6). Available online at http://www.swarm.org/csss-tutorial/frames.html

slight variants), enabling the study of heterogeneous populations. Swarm is the premier ABM tool available today and was chosen to implement this project.

4.3 An Introduction to Swarm

Swarm is a set of libraries that facilitate implementation of multi-agent simulation of complex systems, originally developed at the Santa Fe Institute (It was initiated by Chris Langton in 1994 and the first version was available by 1996). The Swarm software is available to the general public under GNU licensing terms.

The primary feature of Swarm is the virtual machine. The virtual machine allows the researcher to describe agent behaviours one by one, agent-by-agent, context-by-context, all while keeping an exact notion of time and concurrency in the world. Swarm also makes it possible to nest hierarchies of agents. Each hierarchy can treat those below as "black boxes". This is useful because it often isn't clear where to begin a modelling effort. Rather than seeking denotation on how the system should work and looking for deviations, one can build independent model components from many perspectives and then combine them. This bottom-up approach has the advantage of documenting the both unexpected bad and good things in the system, as well as contextual sensitivities.

4.4 Objective C versus Java

Swarm can be coded in Objective C or Java (Scheme is also available but has not been documented adequately yet) and these were weighed up against one other. Java is the more recent addition to the System. It encourages static typing, while Objective C encourages dynamic typing. Objective C gives more power to the programmer and allows him to do exactly what he wants (however this works against the beginner). The fact that Java has garbage collection built in is a huge advantage, as the modeller no longer has to keep track of freeing up transitory object memory. Java is also the more popular language, and is better documented.

In summary, while Objective C has advantages, the cultural and practical advantages of Java outweigh them in for the purposes of getting started with Swarm. Therefore, this was chosen as

the language of this project. As an aside it should be noted that the swarm itself is tending to go in this direction also. [4]

The purpose of the Java layer of Swarm is to mirror the protocols of the Swarm libraries as Java interfaces. The fact that Java is a more of a statically typed language is handled by introducing new types such as Selector. The Selector loads up type information so that the Objective C virtual machine can talk to Java on Java's terms. Even if the user writes swarm programs in Java, which is compiled into Java byte-code, Swarm itself is compiled into native code. This approach has precedent in the Java community. For example, Java3D uses OpenGL or DirectX native code libraries for performance.

4.5 The Activity Library

The Activity library is the conceptual core of Swarm. It provides the mechanisms to take local, relativistic descriptions of agent behaviour and run them on a common space. There follows a brief description into some of its more important components (Note that because the names of some of the components are similar to their descriptions, objects have been marked in italics. Note also that a *Swarm* is an important component of the library and should not be confused with Swarm the overall package).

An *Action* is a potential behaviour in some context. Usually, it is a function call or message to be sent to a target or some set of targets. It is different from a simple method or function in the sense that it is an object representing a potential call, not just the mechanism for realizing such a call.

ActionGroups are for groups of events that happen together. These are preferable to linear code because there is the potential for meaningless phenomena to be seen by the modeller if all concurrent behaviours actually run in some fixed sequential pattern. An *ActionGroup* allows the policy toward concurrent events to be changed, i.e. to shuffle the concurrent events according to different random seeds in different runs. When creating an *ActionGroup* to send a message to a set of agents, it is useful to indicate to Swarm whether that set is intended to be homogeneous or heterogeneous. Faced with a heterogeneous set of agents, the process of method dispatch must

[4] From the Swarm FAQ in response to the question “Why Objective C?” (Question 1.3) available online at http://lark.cc.ukans.edu/~pauljohn/SwarmFaq/SwarmOnlineFaq.html#1.6

occur over and over again. However, since the method implementation is the same for a homogeneous set, Swarm can exploit that.

A *Schedule* is a means to order events. They are a kind of Map where the key is time, and the value is an *Action*. *Schedules* can be given a policy on how to handle concurrent events. They are active data structures. *Actions* can be added or removed in an ongoing fashion (dynamic scheduling). The act of activating a *Schedule* or *ActionGroup* in a *Swarm* creates an object called an *Activity*. This is a way of segregating *Actions*; a number of them may be grouped together into each *Activity*. If an *Activity* is dropped, that behaviour (which may be a repeating behaviour) will cease in a *Swarm*. A colloquial description of a *Swarm* is that it is a way to represent a community of agents. Communities can be embedded in other communities, or they can be independent. A *Swarm* serves two purposes:

- A container: Agents or components that are a part of a Swarm typically will use that Swarm as a container. Thus, when a *Swarm* is dropped all the components in that *Swarm* are also dropped.
- A common temporal space: At a given instant in time, many things may be scheduled to happen in a simulation. A *Swarm* keeps track of the ordering of events, translating the agents' subjective experience of time into a single objective sequence of events.

4.6 Current Applications of Swarm

Every year the Swarm community gets together for a conference called SwarmFest. To get an idea of some of the current applications of swarm here some of the featured applications on show this year:

- Creation of Lager Scale Urban Models using cellular automaton simulation
- Modelling the migration of juvenile salmon using hydrological and topographical features of known locations. Game theory is used to define fish interactions.
- Applying Complex Adaptive System (CAS) ABMs to investigate multifaceted systems such as electricity markets (this is of interest at the moment due to the electricity shortages in California).
- The non-adaptive inflammatory response is a complex system that protects the human body against injury. However, severe perturbations to this system may lead to pathologic conditions in which the system behaves in paradoxical fashion. The clinical

manifestation of this is Systemic Inflammatory Response Syndrome (SIRS). It is hoped that more sophisticated models will be used to generate state-space maps of the inflammatory response, enabling identification of phase transitions between the "normal", dynamically stable state and its unstable, pathologic state (SIRS). Swarm and Agent-Based Computer Simulation are envisioned as engineering tools in designing treatment regimes and as an aid in the understanding of how the various components of the system interact to produce organ-level physiology.

4.7 Reasons for using Swarm

The main reasons for choosing swarm were

- To leverage the experience of others.
- To avoid work that had already been done.
- The Swarm diagnostics and graphing libraries are perfect for the analysis needs of this project.
- Swarm scales well for systems in which components are very simple.

4.8 The Future of Swarm

Swarm itself is evolving and further versions promise new advances such as:

- Integrating COM with Swarm
- Supporting XML as another language in Swarm
- Creating a graphical model builder for Swarm

4.9 Summary

This chapter introduced Swarm and outlined the choices made in what version of Swarm was used. It went into some detail on the important scheduling components used in the project and it looked at some of the areas where Swarm is currently being used and the key advantages of using it. The alternative to using Swarm would have been designing an ABM tool myself. While this would have been possible it could not have come close to the power offered by Swarm.

Chapter 5 - Implementation

5.1 Introduction

The entire model is implemented using Swarm. There are four tiers controlling all aspects of the model. This was done to simplify the debugging process by allowing each of these to be tested individually. They call each other in series up and down the hierarchy. Each tier has certain areas of responsibility and jurisdictions. If a bug reproduces, it can place its child on the heatspace, but it does not have the authority to activate it. A request to do this is thrown up a level to the Model swarm, whose responsibility it is to do this. The Observer Swarm is responsible for displaying the graphing widgets, but it is unable to access information from the model so it "asks" the Model Swarm to compile and return this information to it. The hierarchy is shown below in figure 5(a):

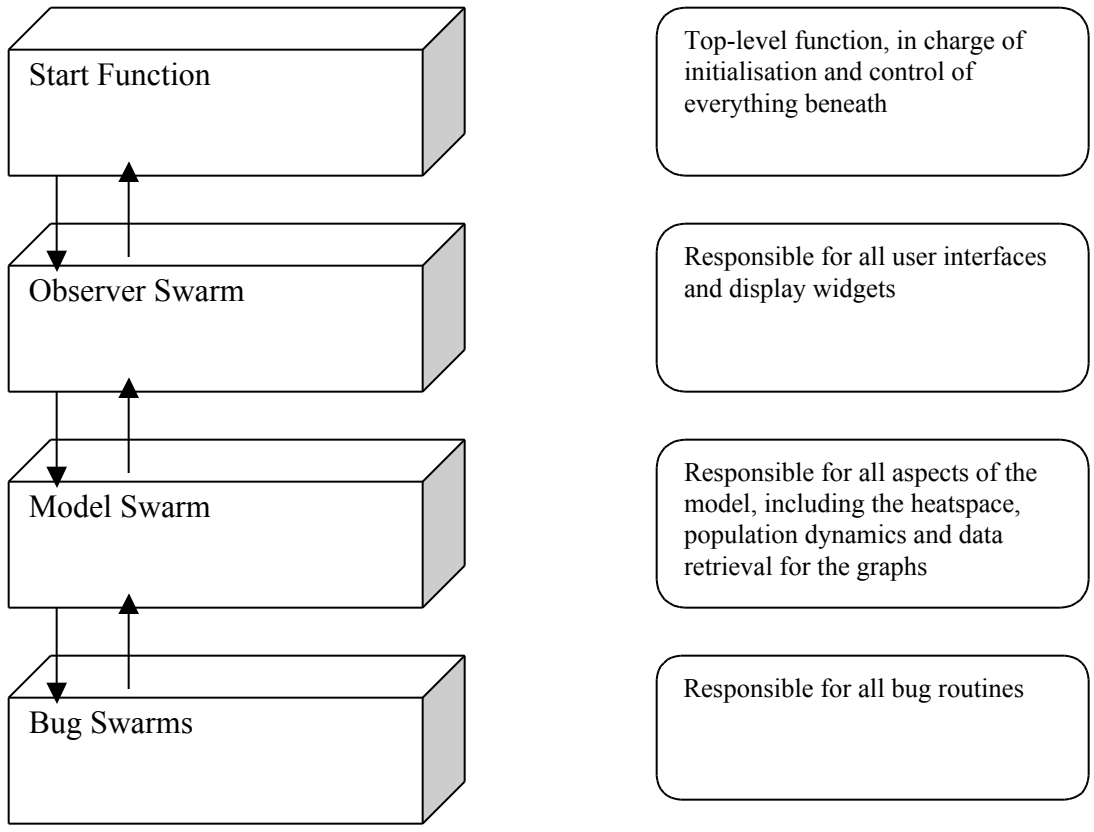

Figure 5(a): The Four-Tier Model Hierarchy as Implemented with Swarm

5.2 Implementation of the Heatspace

The heatspace is implemented as a Swarm Diffuse2d Object. Because of the way this operates, it was necessary to draw a distinction between the heat values used on the heatspace and the energy levels used by bugs. The Swarm Diffuse2d Object is a discrete second order approximation to 2-d diffusion with evaporation. Maths is done in this object using integers on the range 0 to 32767. Values are therefore capped at 32767 (*maxHeat* in code). To keep the energy levels of bugs simple it was decided to map these heat values to energy values of 0 to 10 (i.e. a value of 1 on the energy scale corresponds to *maxHeat / 10*). Therefore a heat value of 4000 would correspond to an energy value of 1 (4000 / 3276.8).

Bugs can access the heatspace in two ways, they can find the available level of energy of the cells in their neighbourhood, and they can take a certain amount of energy from the cell they currently occupy. As far as bugs can tell there is no such concept as heat since all accesses they make involve energy. They cannot access the bottom energy level of the heatspace, i.e. they cannot take energy from the board below *minHeat* (*minHeat = maxHeat / 10*). Further details of how bugs interface the heatspace are given in appendix A.

The heatspace alters itself based on two variables, the evaporation rate and the diffusion constant. The evaporation rate is a variable by which the energy level of every cell on the heatspace is multiplied. This is varied seasonally about 1 with a period of 100 days (a year). The diffusion constant spreads the energy across the space locally. This gives the heatspace a smoother and more realistic appearance. Details of how these are done are also given in appendix A.

The heatspace is displayed on a GUI as shown in figure 5(b) below. The brighter the green level of the cells, the higher its energy level. This heatspace looks discontinuous because the bugs are invisible and the model is only a few days old. This is deliberate done to illustrate the colouration of the heatspace. A smoother and more realistic screenshot, with visible bugs, is shown later in the chapter in figure 5(c).

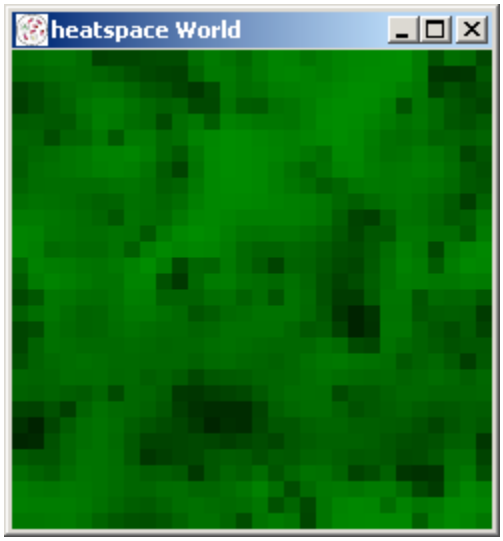

*Figure 5(b): A view of a 30*30 heatspace (bugs are present but invisible).*

5.3 The Bugs

In order to make it easy for extra bug-like agents to be added to the model in the future, a super class for all agents, *agent* was created. The *bug* class was a sub class of this. Common methods that would be accessed by all types of agents are implemented in the *agent* class. These include the methods for movement, interfaces with the heatspace and methods relating to agent display (how the agent is viewed by the user). All methods and variables associated with bug-specific behaviour, such as the *bugStep* routine (the method that determines all bug behaviour, detailed in the next section) and methods associated with mating. This two-tiered approach proved to be useful as it abstracted the functionality that would be common to new agents.

The bugs themselves are displayed on the heatspace GUI as coloured squares. They are capable of changing their colour as a diagnostic aid to the user. The colours used and their meanings are listed below in table 5(i), and a screenshot of a heatspace with bugs showing all the colorations is shown in figure 5(c).

Table 5(i): Diagnostic Colours and their Meanings

Colour Name	Colour	Meaning
PositiveBug	Orange	This bug is harvesting more energy than it is expending
NegativeBug	Blue	This bug is expanding more energy than it is harvesting
NeutralBug	Red	This bug is breaking-even on energy
ParentBug	Pink	This bug has just become a parent
NewBornBug	White	This bug has just been born
AvailableBug	Grey	This bug is looking for a mate

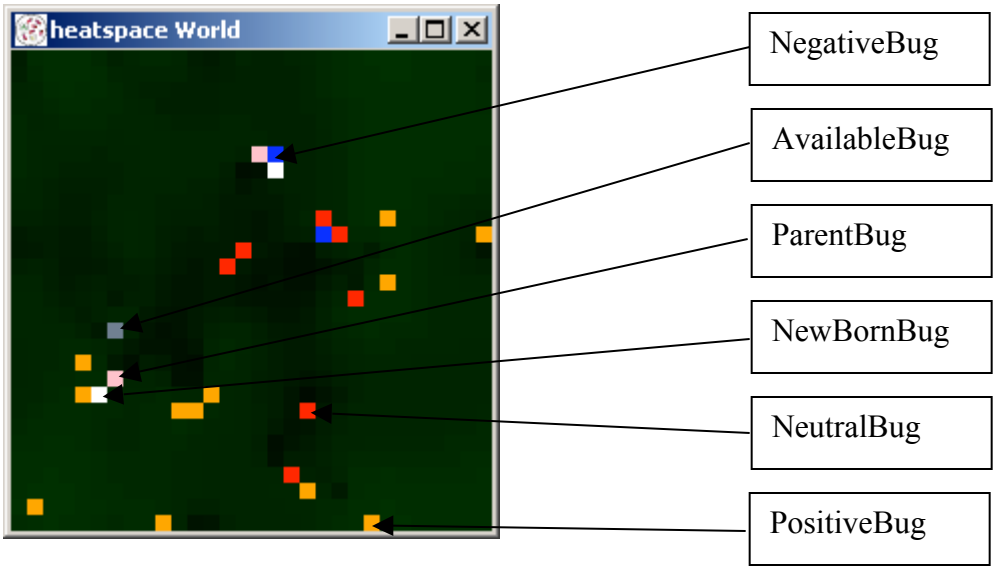

Figure 5(c): Screenshot of the Heatspace showing Bug Colouration

5.4 The BugStep Routine

Every day every bug in the model runs its *BugStep* routine. This implements the flowchart shown in figure 3(a) and determines all bug behaviour. It does this by looking at the energy level and age of the bug, based on these it determines what the bug can do each day. Every bug has a *Schedule* built into it. The single A*ction* that is allocated to this is the *bugStep* routine. This is done on initialisation and activated by the model Swarm. This schedule is dropped when the bug dies (more of this in the next section). A stripped down version of the *bugStep* routine is shown below (all internal diagnostics and concurrency code have been removed).

```
01    public void BugStep(){
02          int startEnergy = energy;
03          Age++;
04          if(energy < 1 || Age > maxAge){
05                world.putObject$atX$Y (null, x, y);
06                ObserverSwarm.modelSwarm.removeBug(this);
07                return;
08          }
09          energy = energy - metaRate;
10          if(energy > fertileEnergy){
11                if(LookToMate()){
12                      noOfChildren++;
13                      setColour(ObserverSwarm.ParentBug);
14                      if(firstChildAge == 0)
15                            firstChildAge = Age;
16                            return;
17                      }
18                }
19                setColour(ObserverSwarm.AvailableBug);
20                GoToHottestCell();
21                return;
22          }
23          energyHere = heat.getEnergy(x, y);
24          if(energyHere > appetite){
25                energy = energy + appetite;
26                heat.takeEnergy(appetite, x, y);
27          }
28          else{
29                energy = energy + energyHere;
30                heat.takeEnergy(energyHere, x, y);
31                GoToHottestCell();
32          }
33          if(energy < startEnergy)
34                setColour (ObserverSwarm.NegativeBug);
35          else if(energy == startEnergy)
36                setColour(ObserverSwarm.NeutralBug);
37          else
38                setColour(ObserverSwarm.PositiveBug);
39    }
```

Step by step this translates to:

- Line 02: a record is kept of the energy of the bug at the start of the day (*startEnergy)*
- Line 03: the age of the bug is incremented
- Lines 04-08: if the bug's energy level is below zero or its age is above *maxAge* it is removed from the model
- Line 09: the metabolic cost of remaining alive is charged to the bug's energy level
- Lines 11-22: If the energy level is higher *fertileEnergy* the bug looks for a mate. If one is found, a child is born (line 12, `LookToMate()`), the number of children that bug has had

is incremented, and the routine ends. Otherwise the bug changes its colour to *AvailableBug* and moves to the hottest cell in its visual range.

- Line 23: the energy value of the current cell is measured and stored (*energyHere*)
- Lines 24-32: If this *energyHere* is less than or equal to the bugs appetite the bug takes it from the heatspace and moves to the hottest cell in its visual range. Otherwise the bug takes in its appetite worth of energy and remains there.
- Lines 33-38: The bug changes its colouration based on the net inflow of energy as outlined in table 5(i).

5.5 Dealing with the Dynamics of the Population

The speed of the model depends on a number of factors; the cellular automata of the heatspace, the maintenance of the GUIs, the data retrieval of the diagnostic tools and the execution of the Bug schedules. With the exception of the last one, all are fairly independent of the size and turnover of the population. Since every bug in this model has a schedule to control its actions, problems arise for the swarm scheduler. The speed of the model is dependent on two things relating to this:

- The number of bugs (and consequently schedules) in the model.
- The population turnover, since new bugs need to have their schedules activated and dead bugs their schedules dropped

During population spikes this problem becomes acute. The simulation virtually grinds to a halt. There are therefore two ways to increase the speed of the model. Maintaining a large static population and recycling dead bugs yields one solution. Dead bugs could have their *bugStep* activities dropped from their schedules in favour of a trivial one that takes no time, and then reactivate the *bugStep* activity when they are "reborn". This would carry a small overhead in changing the schedules. This solution is quite easy to code. However it comes with its drawbacks: The static population must be high to accommodate the occasions when the population spikes above normal levels (this happens every few years), and we have traded one scheduling overhead for another (albeit smaller one).

The second solution was to maintain only the currently alive bugs and live with the scheduling overhead. Since this turnover is usually low and the population is usually far below the peaks this solution was chosen. Some combination of the two would possibly have been a better

solution, where a buffer of "undead" agents is maintained to deal with times where the population was stable but there was a low population turnover and then changing the size of this buffer as needed. This was not investigated fully because the second solution proved to be adequate.

The purely dynamic solution was implemented by creating extra schedules in the model Swarm, one for creation of agents (the born method) and one for removal (the kill method). The population was stored in a Linked List, the alive-list. The schedules involving the actions of the alive-list agents were run as usual. If a member of the population was killed, it was removed from the alive-list and added to a new list of dead agents, the dead-list. Every day the *Kill* method would deactivate the agents in the dead-list. All memory taken up by these would be de-allocated. Likewise when a new bug was born it would be placed on the heatspace, and added to another list, the newborn list. The *Born* method activates the schedules of these agents, and adds them to the alive-list. After each method, the dead and newborn lists were cleared. The *Kill* method was scheduled to run before the agents' schedules, and the *Born* method to run after. This order ensures that there are no conflicts between the schedules.

5.6 Genetics

The genes are implemented as a bit stream into which the bug's static attributes are coded. At birth, each bug decodes this to get the values of its static attributes. The position and lengths of each attribute is predefined and hard coded into the bug's makeup. Each one is assigned a weight. The area of the array assigned to each one is searched and the 1s are summed. The result is multiplied by the weight. All attributes, with the exception of *JunkCoefficient*, *spliceOrCrossover* and *sight* have their weights added to them to prevent zero values. It is possible therefore for bugs to be blind (these bugs move about randomly and are capable of surviving and having children like other bugs, although this usually proves to be an overwhelming handicap). The genetic attributes, their length, weight and positions are listed in table 5(ii) below.

Table 5(ii): Details of Genetic Attributes

Attribute Name	Position	Length	Weight
JunkCoefficient	1, 3, 12, 16, 21, 34, 38, 59	8 bits	N/a
SpliceOrCrossover	2	1 bit	1
Appetite	4-11	8 bits	1
MetaRate	13-15	3 bits	1
BirthEnergy	17-20	4 bits	4
FertileEnergy	22-33	12 bits	5
Sight	35-37	3 bits	1
MaxAge	39-58	20 bits	25

To ensure genetic diversity and allow bugs with different genetic attributes a fair chance at the beginning of the model, it is important to randomise the genes of all bugs (i.e. randomise every bit in the bit stream). Since each bit stream is fifty-nine bits long it is impossible to represent all possible combinations of genes (this would require 2^{59} bugs), however a good approximation of this is made. The majority (>80%) of bugs are born with crippling disadvantages (high metabolic rates, blind, low maximum age) and these die in the first year or so. The certainty that this initial cull will happen allows us to initially overpopulate the model substantially.

Over time the fitter bugs will tend to have children that survive and go on to produce more offspring and so over time the average characteristics tend to approach certain (usually predictable) levels. As such the fitness of the population is increased by Darwinian evolution and not by an explicit fitness function.

Another source of genetic diversity is mutation. This is also at work in this system. Having decided the value of a bit in the offspring's gene there is a small chance (based on the mutation factor) that this bit will be flipped. The possibility is the same for every bit and is small (it was decided to standardise the percentage chance of a bit being flipped to 0.02%). This was the same for both modes of gene splicing.

As was previously explained, it is essential to monitor the genetic diversity of the population. To do this, the standard deviation of the junk genes across the population was measured every day. If it dropped below a predefined level, a warning would be issued to the user, and the simulation would be halted.

Further implementation details of the Genetic coding and splicing, including code listings for important functions are given in appendix B.

5.7 The Data Analysis

Swarm has a number of built in diagnostic aids. Data probes and graphing widgets were used extensively in this project. Data probes allow the user to see certain variables in objects. These variables can be specified at initialisation or all variables in an object can be probed. A probe of selected variables from the model Swarm class is shown below in figure 5(d).

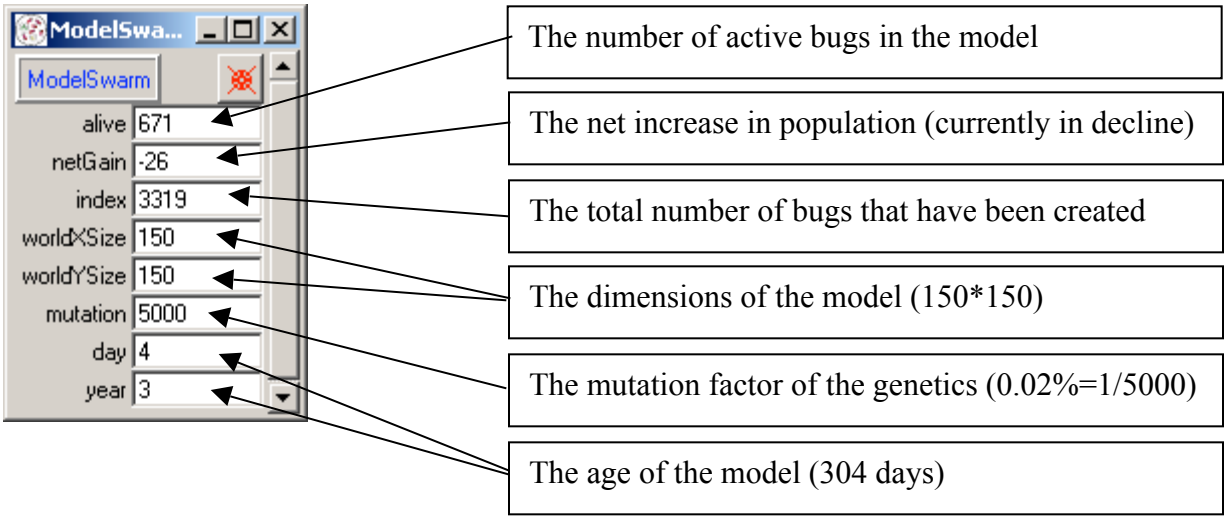

Figure 5(d): The Model Swarm Data Probe with Explanations of the Variables:

It is possible to probe individual bugs on the heatspace simply by right clicking on them on the GUI. This brings up a probe as shown in figure 5(e). It is possible by probing the number of bugs to get a microscopic view of an area of the heatspace. The probe disappears when the bug dies. It was also useful during debugging to ensure that the bugs ate behaving as expected.

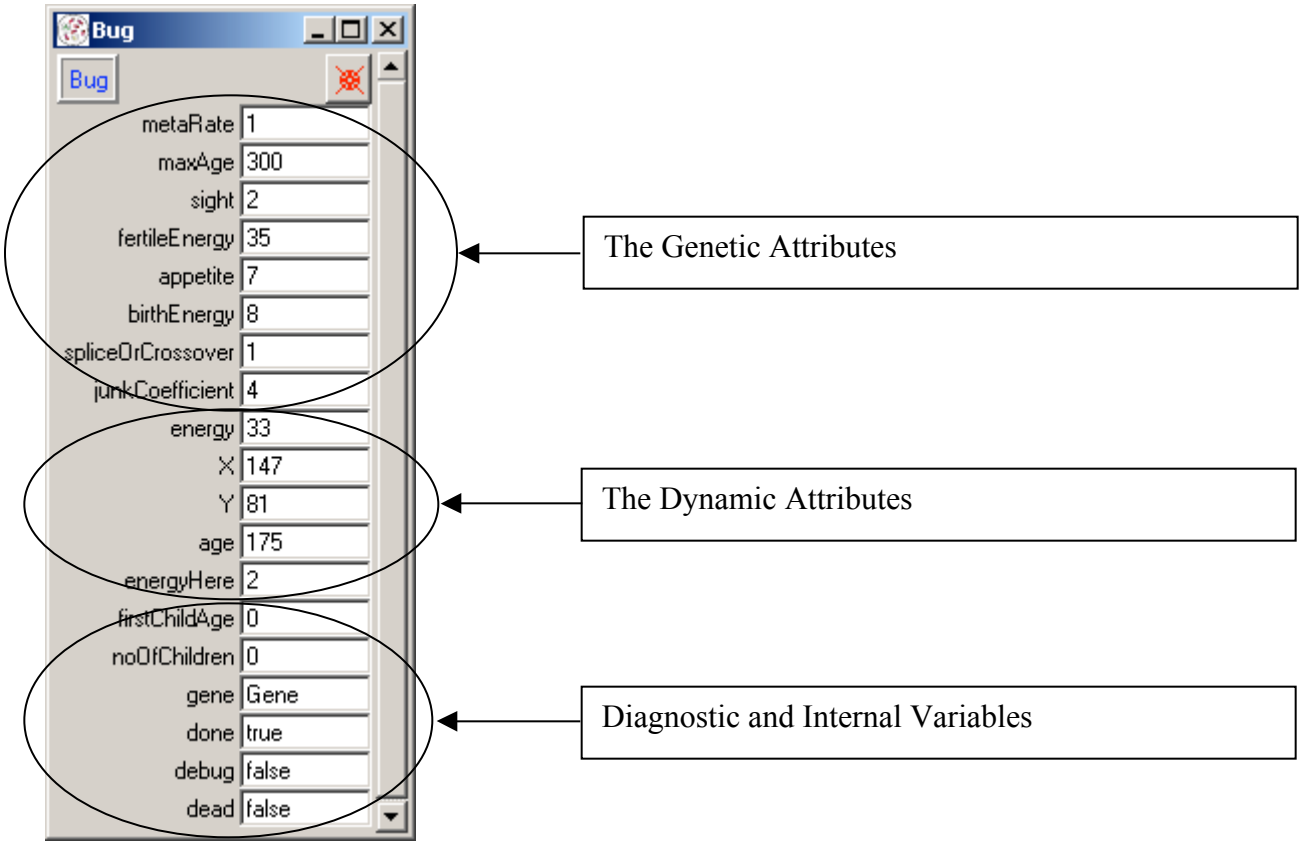

Figure 5(e): A Data Probe on a Bug with Descriptions of some of the Variables.

Swarm's graphing widgets were used to extract data from the model and display it in a useful way as the model ran. Information about the heatspace, population size and genetics were extracted and sent to these. Four of the important graphs were created. These are shown below in Table 5(iii). It is possible to zoom into any area in the graph to find exact values at population spikes for example. A population graph is shown below in figure 5(f).

Table 5(iii): Details of the Commonly Used Graphs

Graph Name	Purpose
The Heatspace Graph	Measures the average energy level across the heatspace
The Population Graph	Measures the current population of bugs, as shown in figure 5(f)
The Genetic Attributes Graph	Measures the normalised average attribute levels across the population of bugs
The Age Graph	Measures the average age of the bug population, the average age at which bugs give birth for the first time, and the age of the oldest bug

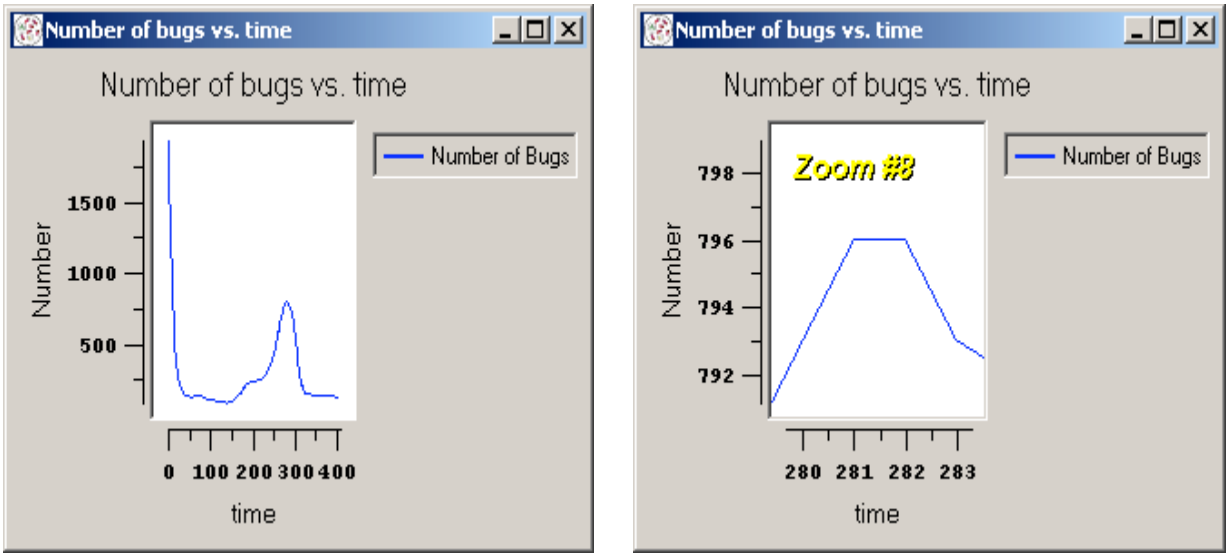

Figure 5(f): A Graphing Widget in Action – Zooming in on a Spike on the Population Graph

From the first shot it is impossible to determine the exact value of the population spike, but after zooming in on the area it is possible to pinpoint its value and location (the population spikes at 796 between days 81 to 82 in year 2).

5.8 The Model Schedules

Since all methods in the model had to be repeated every day, *Actions* and *ActionGroups* were assigned to them and these were all attached to schedules. All schedules had to be ordered in such a way that they would all run, and not interfere with each other. The data probing should be done after the bugs have moved to ensure that information garnered from these was as up-to-date as possible. The ordering of heatspace related events were especially important. The bugs had to harvest the heatspace and the seasonal effects had to be done after the heatspace updated itself otherwise they would have no effect on the heatspace. The ordering is shown below in table 5(iv).

Table 5(iv): The Ordering and Description of the Model Schedules

Function Call	Description	Order
ObserverSwarm._update_	Refreshes all GUIs, draws heatspace and bugs	1
ObserverSwarm._updateGraphs_	Updates all graphs	2
ProbeDisplayManager.update	Updates all Object Probes	3
Heatspace.updateLattice	Performs the Cellular Automata on the Heatspace	4
ModelSwarm.kill	Removes dead bugs from the Bug Scheduler	5
ModelSwarm.born	Activates newborn bugs	6
ModelSwarm.Seasons	Modifies the Heatspace settings based on the day of the year	7
All Bug.BugStep calls	Runs all Bug routines	8

The first three schedules were part of the observer Swarm and the remainder were under the control of the model Swarm.

5.9 Summary

This chapter detailed the implementation of the project. It discussed the way the heatspace, bugs and genetics were implemented and goes into some detail on the main routine executed by the bugs. It discussed scheduling requirements and outlines why the segregation of the system into *Swarms* was necessary. It went into some detail on the data probes and graphing widgets. These are used in the next chapter to further the understanding of the behaviour of the model.

Chapter 6 - Analysis

6.1 Introduction

This chapter seeks to explain some of the data that was extracted from the models created. First it explains the standardising of the initial size of the heatspace, to optimise the time constraints and stability of the model (i.e. that it will run long enough to get useful results from it). It analyses the evolution of genetic characteristics and comments on its importance.

6.2 Selecting a Standard Heatspace Size

An important limit on the value of this simulation is the size of its environment and population. Because the evolution of the population happens over time, the model must run for as long as possible. If the population crashes before this happens, the run is useless. This is less likely to happen when the environment is large and the initial population is as genetically diverse as possible. These requirements must be balanced with the computationally intensive nature of the problem. A balance was struck between the size of the model and the time taken to perform a run. To test the limits of the simulation size, a large model was run off, with a heatspace size of 200*16000. This took more than an hour to get through 150 days of the model before the program crashed. It was decided that the standard heatspace size would be 150*150. This proved to be large enough to ensure that most populations survived and small enough that 20 years of modelling could be done in just over an hour and so, all further analysis is on models with this size. Note that all simulations were performed on an 800 MHz Pentium III with 128Mb RAM.

6.3 Stability of the Model

A stable population and a high carrying factor of the heatspace are advantageous for a model to be considered valuable. Since bug interactions are all local and quite random the models implemented in this project are an unstable models. Due to the chaotic nature of the model, small changes in the initial model variables have huge and unpredictable effects on the way the attributes of the population change over time. Since the size of the heatspace and the mutation coefficient had already been standardised this left the initial population as the only obvious

variable to change in studying the chaotic nature of the program. This is further illustrated further in the next section.

The more global view of the model that the bugs have, the more stable the model itself would be. For example, if bugs could be notified of an impending famine they could practice birth control and build up their energy reserves in anticipation. Special warning agents could be created that do not need to harvest and travel around the heatspace giving guidelines on birth control and harvesting practices to all bugs in their sphere of influence. They could coordinate with each other and relax controls where necessary. The use of these agents would take from the purely capitalistic nature of the model and introduce a kind of "government control".

The carrying capacity of the heatspace is the population of bugs that can be supported indefinitely on the heatspace. This is directly proportional to the physical size and regenerative properties of the heatspace and to the harvesting efficiency of the bugs inhabiting it. The carrying capacity of the heatspace was calculated by using a moving average of the population. Moving averages are used to get an idea of the overall trends in chaotic data. A 500-day moving average of population is the average population of the last 5 years. An example of this is shown in figure 6(a) along with the population. It is clear from the graph that the moving average is a good way to average out the spikes. The larger the moving average the smoother the graph.

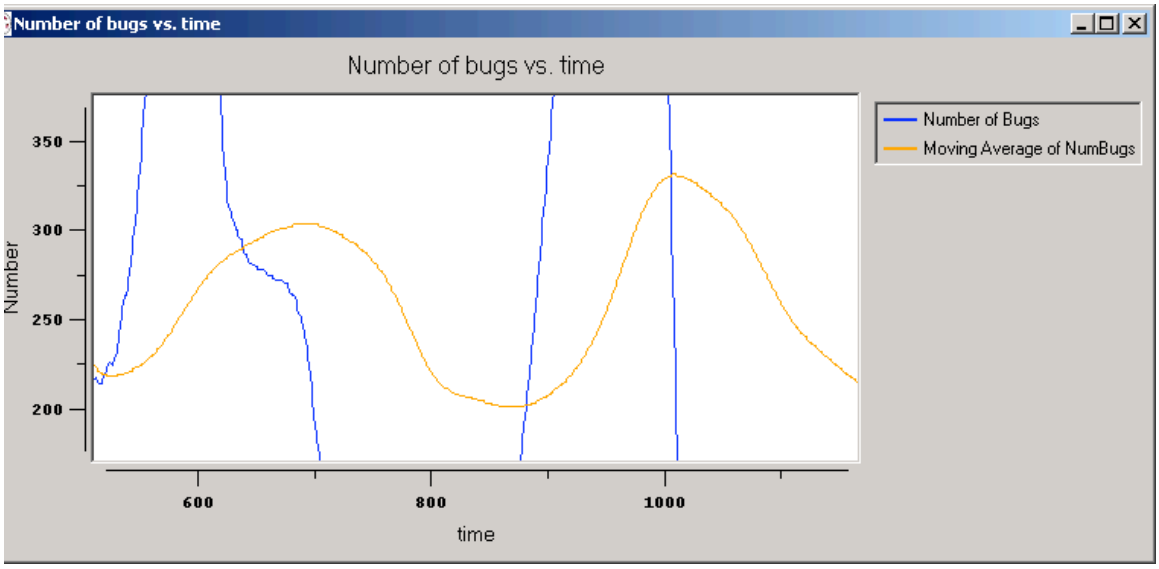

Figure 6(a): 500-Day Moving Average, yielding a carrying capacity of about 250.

6.4 Analysis of the Genetics

One of the most important aspects of genetics is the way mutants dominate a population over time, and the reasons behind this. Given an initially homogenous population with mutants randomly being introduced over time there is the possibility of a mutant eventually dominating, that is to say that the population will eventually reach a new level of homogeneity consisting of individuals with the mutant characteristics. The probability of this occurring is directly related to the relative advantage of the mutant characteristics (much like the old saying – "In the land of the blind, the one eyed man is king").

To test this, a model was set up with an initial population with the usual randomly coded attributes except for sight. That is to say, every member of the population has the exact same bit stream in the area of its gene associated with sight (every bug has sight 1 coded as 001). From the graph in figure 6(b) it can be seen that the population starts off with average sight being 33% of maximum (i.e. sight is 1 out of a possible 3) across the population. The level of mutation is five times higher (at 0.1%) than that usually used, to kick start the process. The first mutant (with vision of 2) is introduced on day 171. This failed to gain a foothold into the population and by day 420 this mutation had died out again. A second mutant was introduced on day 625; this mutant gained a foothold and after a further 30 years (and the introduction of further mutants) the average sight of the population had risen to over 95% (an average vision of more than 2.8).

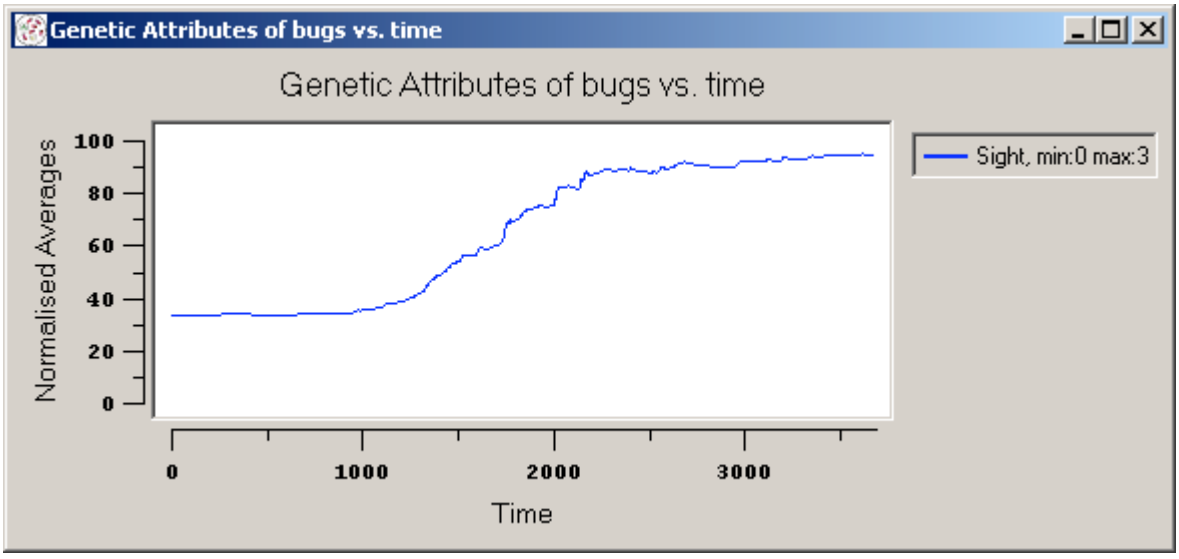

Figure 6(b): The Dominance of Mutant Genes Over Time on Introduction into a Homogenous Population

The dominance of a new mutant is dependent mostly on its relative advantage. Chance also has a part to play (this is why the first mutant in the above example did not dominate despite having a selective advantage). A fit bug born just before a famine will be unlikely to survive it without a healthy energy reserve (unfortunately, bugs are always born with a relatively low level of energy).

Five simulations using the standard settings with differing initial population size were run. The six graphs shown in Figures 6(c) to 6(g) show the evolution of genetic attributes to the end of the run (which came either as a result of a population crash, of because the gene pool had shrunk below acceptable limits). The attributes are normalised and expressed as percentages. Because the genes are randomised at the beginning of each run the plots all start at 50%.

As was already mentioned, the majority of the initial population is made up of bugs with genes that offer a serious disadvantage against the rest of the population. The initial cull of these individuals can be seen at the beginning of each of the graphs. Spikes in the plots of metabolic rate and sight are obvious in the first 100 days or so of the graphs for each run.

It is interesting to note that although a bug with a high metabolic rate is at a severe disadvantage, mutation throws these individuals back into the population periodically. It is clear from figures 6(c) to 6(f) (although not in figure 6(g)) that some individuals with above-average metabolism have been re-introduced. However these never gained a foothold in the population.

Table 6(i) lists the genetic attributes and details of their convergence. These details were extrapolated from the graphs in figures 6(c) to 6(g). From this table it is clear that metabolism and sight are of critical importance to the survival of bugs since they are quickest to converge.

In the last section it was mentioned that due to the chaotic nature of the model, small changes in the initial setting have the potential to drastically change the outcome of the run. This is illustrated in figures 6(d) (initial population: 1998) and 6(e) (initial population: 2000). The population of run 6(d) crashes a little after 10 years, before all attributes have been fully optimised. Run 6(e) ran for over 40 years proving to have been far more valuable.

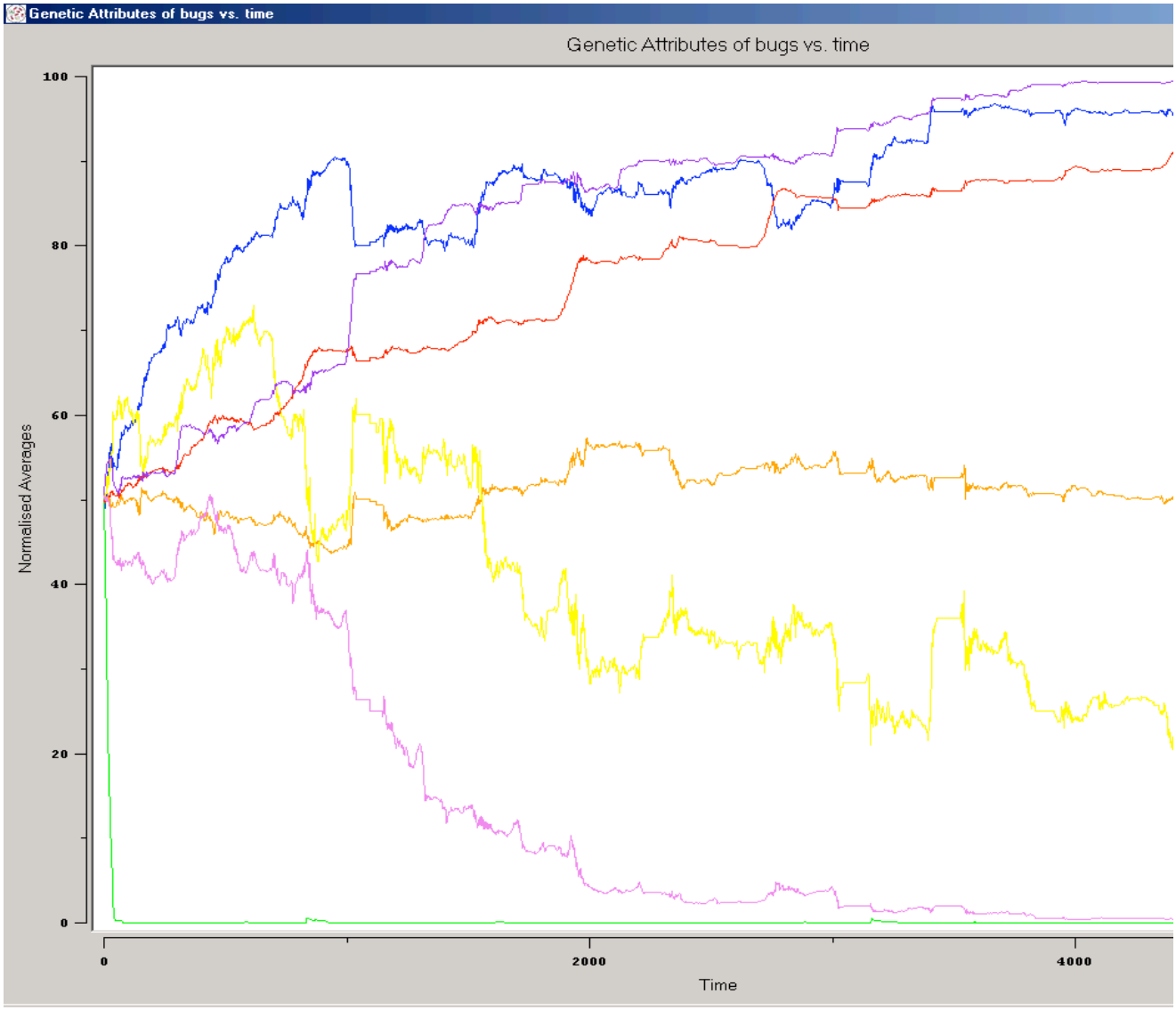

Figure 6(c): Genetic Attributes of Bugs over Time - Initial Population = 1500

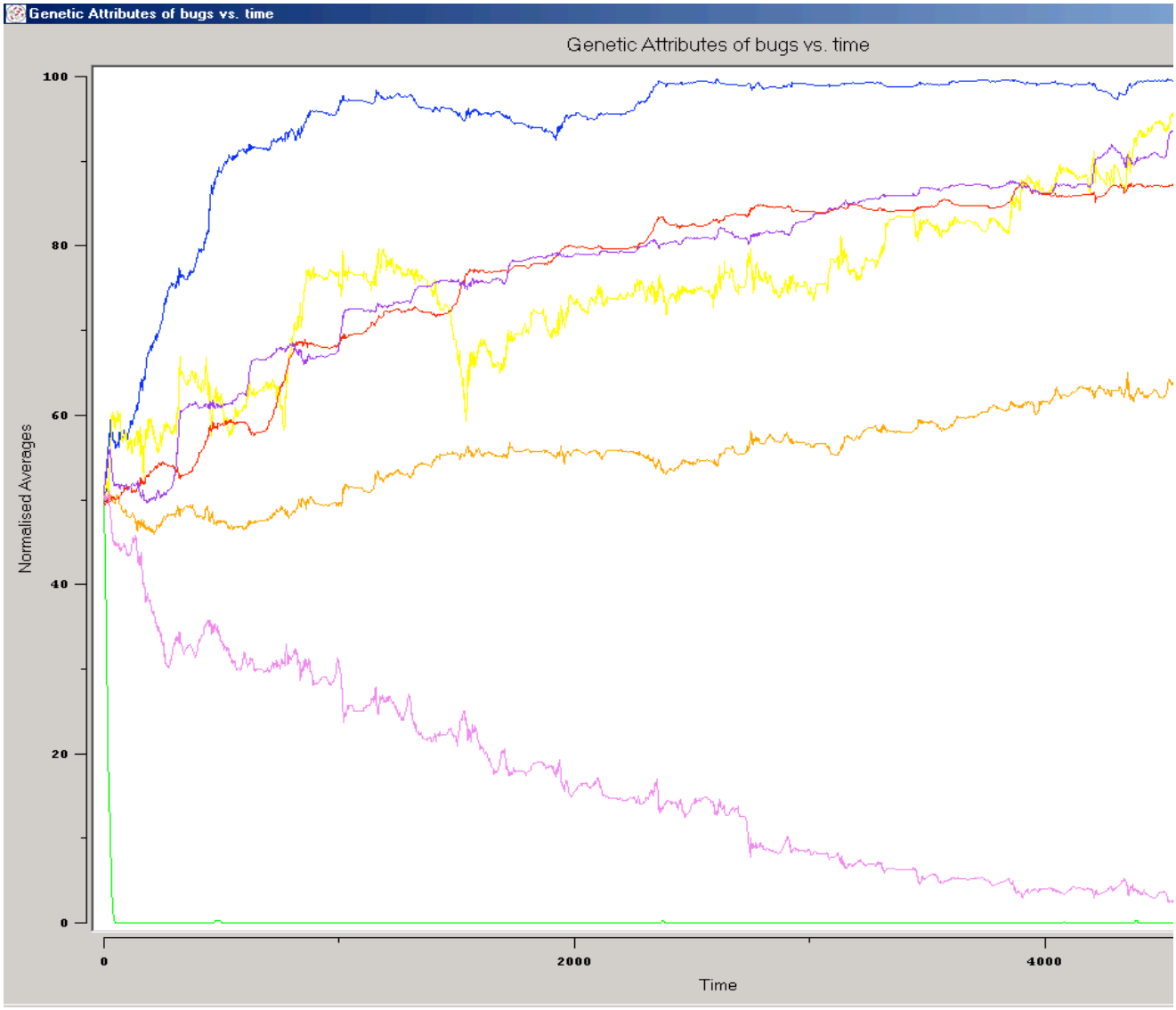

Figure 6(d): Genetic Attributes of Bugs over Time - Initial Population = 1998

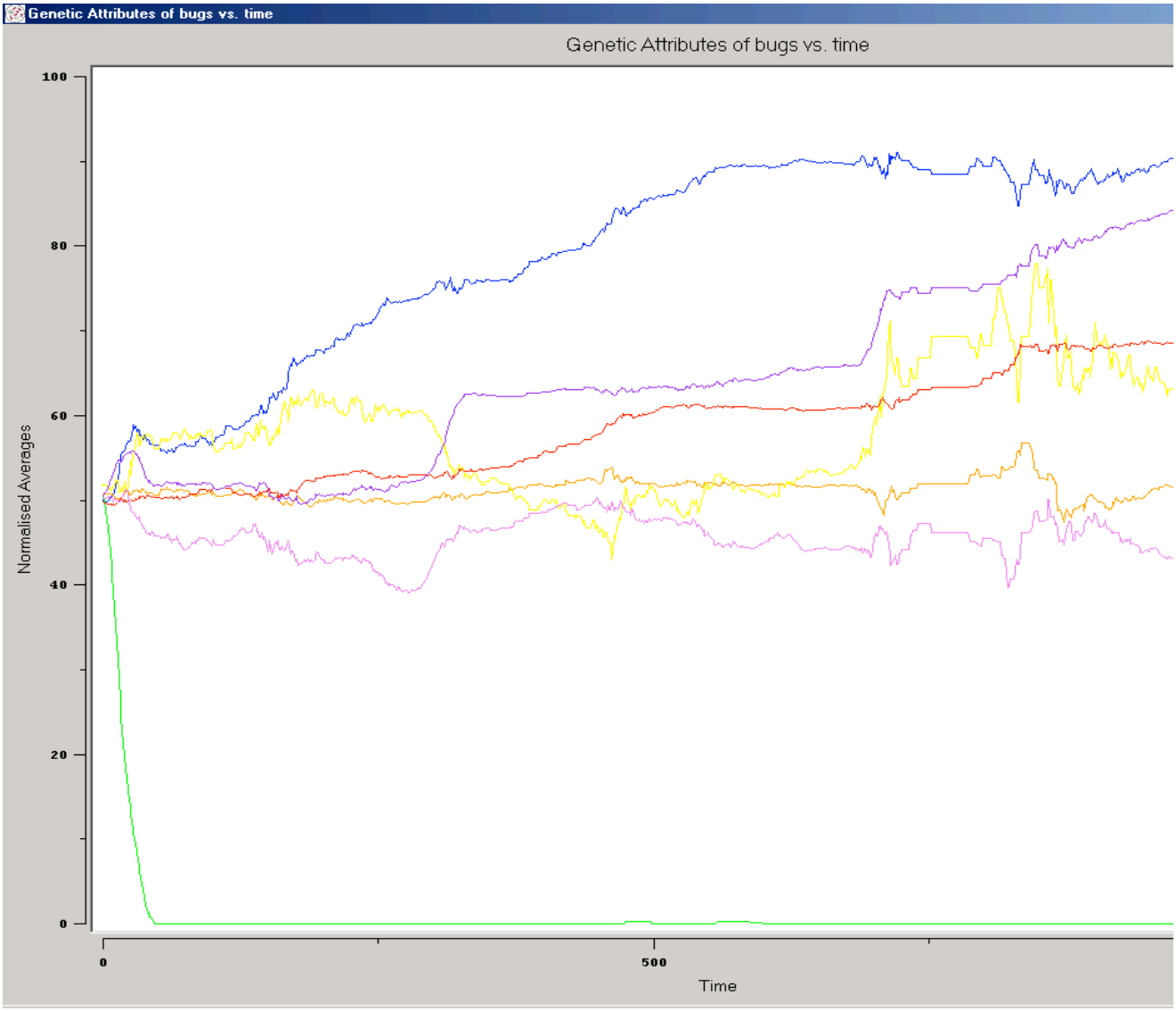

Figure 6(e): Genetic Attributes of Bugs over Time - Initial Population = 2000

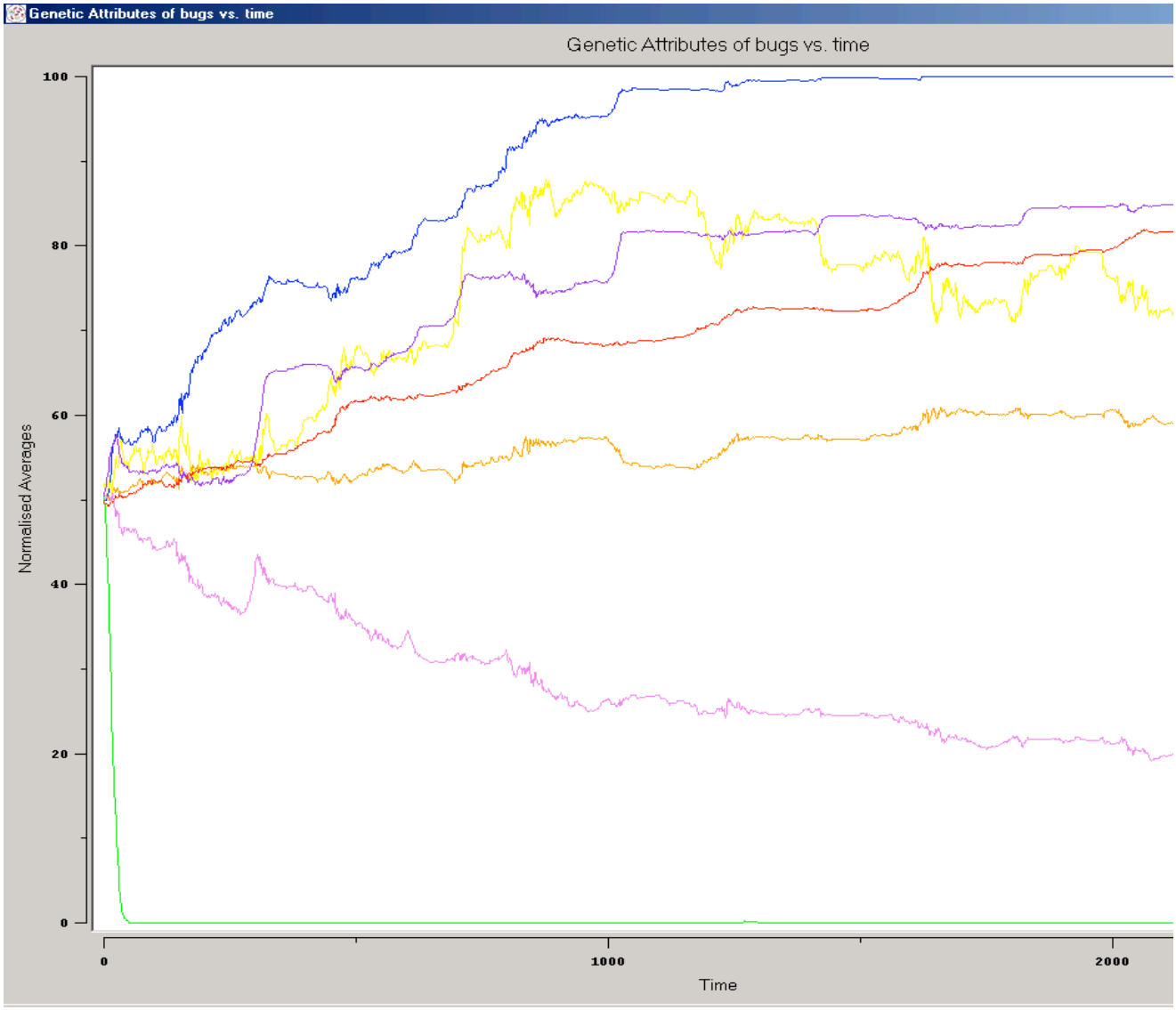

Figure 6(f): Genetic Attributes of Bugs over Time - Initial Population = 2500

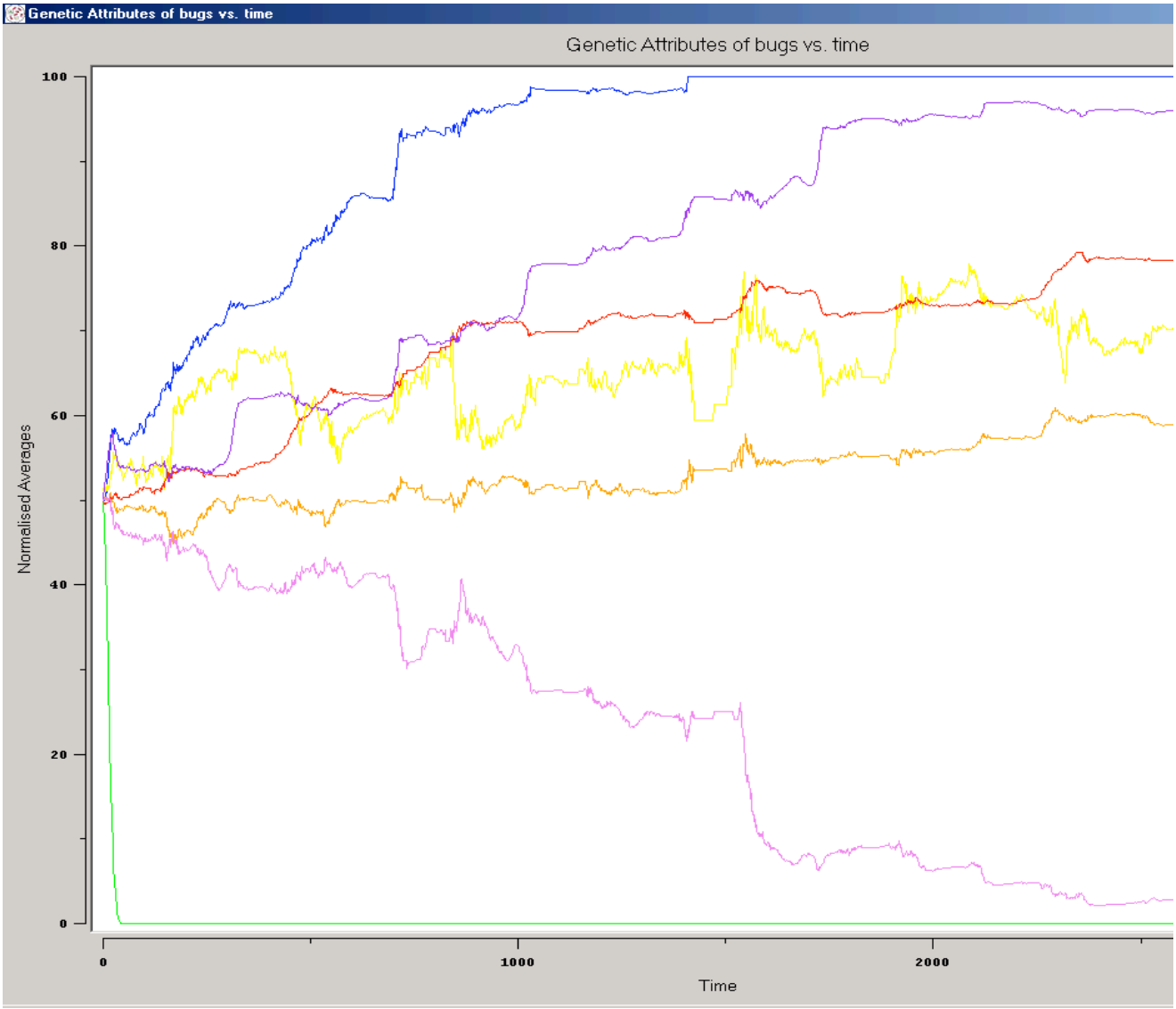

Figure 6(g): Genetic Attributes of Bugs over Time - Initial Population = 3000

Table 6(i): Convergence Speeds of Genetic Variables - Determined from Figures 6(c) to 6(g)

Attribute Name	Converges Towards	Speed of Convergence
Sight	100%	Quickly
Appetite	Not convergent	Not at all
% Of bugs using crossover	Not convergent	Not at all
MetaRate	0%	Very quickly
MaxAge	>75%	Slowly
FertileEnergy	100%	Moderately
BirthEnergy	0%	Moderately

The convergence of the other attributes was not as quick and their directions not as immediately obvious as sight or metabolism. *FertileEnergy* converged upwards but *BirthEnergy* converged downwards (both at about the same rate). This is because having a child can endanger the life of the mother (since her energy level is depleted). This risk is reduced if the amount of energy required to have children (*fertileEnergy*) is increased and the energy toll she pays per child (2**birthEnergy*) is reduced. *MaxAge* converged towards 100% but never quite made it, this could be because it was made up of more bits than any of the other attributes (20 bits) and so required models to run longer then these. Appetite does not appear to converge to either extreme. It seems unusual that an attribute that contributes to bug behaviour should not converge definitely. This is until we note that the *available* level of energy on the heatspace is rarely above 2. This means that bugs with an appetite of 9 have no selective advantage over bugs with appetite of 3. Both have only a small advantage over bugs with appetite of 2. The only bugs to suffer are those with an appetite of 1. These bugs cannot ever harvest a surplus of energy and are doomed to extinction.

One of the lesser goals of this project was to determine the best way splicing genes, by using a single crossover point or by comparing the genes bit by bit. From figures 6(c) to 6(g) we can see that the *spliceOrCrossover* gene doesn't converge to a definite value. It is the most erratic of the attributes and it would be fair to say that neither type of gene splicing offers a relative advantage over the other. The reasons for this probably lie in the makeup of the bit streams in the genes. If sequences in the gene were important, then we could expect crossover points to have a selective

advantage. This is not the case in the form of genetic coding used in this model since attribute values are valued by the number of 1s in the sequence not their ordering.
Initially the attributes were coded using binary encoding, e.g. 430 coded into a bit stream of length 9 would be 110101110. This method valued certain bit positions more than others and slowed the convergence of attributes. The value of the first bit is 256 and the value of the last bit is 1, this concentrates the fitness of the gene in certain bit positions, which was viewed as a disadvantage.

6.5 Analysis of the Diagnostics Information

It was explained earlier that the colouration of the bugs change daily as their conditions do, but it is interesting to note that changes can occur globally within a short period of time (often as quickly as 5 days). An example of this is when the vast majority of bugs change from orange to red and then to blue as the average level of the heatspace falls below certain levels (the orange-red transition level is a little under 30% and the blue-red transition level is about 25%).
This also happens to a lesser extent at the advent of winter and the opposite happens as spring gives way to summer. The fact that this happens regularly (although not every year) at seasonally important time indicates that the seasonal aspects of the program are operating realistically. The fact that the bugs react to the seasons indicates that seasons help force evolution by creating a measure of hardship for bugs to overcome.

Reading the information from individual data probes on bugs it is possible to tell something about the model as a whole. Looking at the information in the probe shown in figure 5(e) (in the last chapter) we see that the bug is 175 days old and still has not had children. Its genetic attributes are quite good; metabolism, sight, appetite are all fairly optimal. For a bug with this kind of fitness to not have had children would suggest that the heatspace could have been going through a bad patch for most of this bug's life, which was indeed the case. Shortly after this snapshot was taken, the situation improved and the bug had a number of offspring. It should be noted that not too much emphasis should be placed on these speculations due to the fact that the information is based on the immediate surroundings of the bug and not on the global system.

6.6 Summary

This chapter discussed the standardisation of initial variables and explained why this was necessary. The inherent instability of the models was explained. The value of non-graphical modes of data extraction (data probes and colouration of bugs) was also investigated.

Most importantly, this chapter analysed the data retrieved on the evolution of genetic attributes over time, and verified that Darwinian evolution was indeed taking place. The dominance of mutants over time, and reasons behind this are investigated and demonstrated. Five graphs of average bug genetic attributes over time are included, and these are analysed. The two modes of gene splicing were compared and found to be equally powerful for the type of genetics used in this model.

Chapter 7 - Summary and Concluding Remarks

7.1 Demonstrating Darwinian Evolution

The heatspace model demonstrated Darwinian evolution. The average genetic attributes of the population changed constantly towards optimal levels for existence on the heatspace. This was done from two angles simultaneously; natural selection removed the weaker bugs, and the fitter bugs tended to have more offspring, thus increasing the average fitness of the population. The value of genetic mutation was demonstrated as a necessary means of ensuring diversity. Each of these factors are recognised in Darwinian evolution.

7.2 Assessment of Swarm

This project provided an excellent test for the capabilities of Swarm as a modelling tool. While it took quite a while to learn how to use it, the rewards in doing so were vast. Once a basic initial working model was created, most of the basic scheduling, graphing and built-in data structures had been encountered and so it was much easier to design further models and implement them.
Already Swarm dominates Agent based modelling and is by all accounts the number one tool in use in the ALife community. The only viable alternative to using Swarm is to create a custom tool as part of any project and the overhead in doing this probably wouldn't be worth the benefits.

7.3 Problems Encountered

The only real unsolved problem in this project was in getting large-scale models to run for long periods of time. They tend to crash the memory management system of the computer. Although Java does all the garbage collection in Swarm modelling this does not solve all the memory leaks. The graphing widgets store all the information taken in by them and over time this accumulates such that it becomes a noticeable drain on memory after a time. Even if the graphing and information collection elements of the model were disabled the program continued to take up more and more memory as the model progressed until eventually the memory management system crashed with a memory allocation error. This could have been a programming error on my part, a failure to account for all schedules, it could have been a

Swarm problem (less likely) or it could have been an unavoidable side effect of using large populations of agents with a high turnover rate. At any rate this problem was never fully tracked down and really only affected larger scale models, not the standard ones detailed here.

7.4 Future Direction of the Project

This project lays a broad foundation for further development in a number of areas. More components could be added to this project with relative ease given a familiarity with Swarm. One example of a further development would be to attempt to synthesise some of the theories of economics. This could be done by introducing the concepts of money and introducing extra resources to harvest. Extra agents such as bankers and economic regulators could be introduced. By defining a number of extra bug interactions as rules and integrating these into the bugs' genes an optimal rule system should emerge.

7.5 Conclusion

There were two main goals to this project. They were as follows:

- To gain proficiency in the use of Swarm.
- To create a model that demonstrated Darwinian Evolution.

This project proved to be the perfect test of the Swarm tools, at this point I would feel confident in being able to use Swarm to apply the techniques used in this project to solve problems in areas outside the ALife field.

A model for Darwinian Evolution was successfully developed (though inherently chaotic). Provided the model ran long enough an optimal solution always emerged. The only two issues that prevented this were the occasions where the population crashed early in the simulation (or the gene pool shrunk) or in the case of large-scale models where the memory leaks crashed the program before enough time passed for a solution to emerge.

Two different types of gene splicing were compared using the same techniques as those used to emerge an optimal set of genetic attributes. Results were inconsistent and it was discovered that neither way offered a selective advantage over the other.

In doing this project, it is hoped that I made a positive contribution to the ALife and Swarm communities. This report will be submitted to the Swarm Development Group, and hopefully, sooner or later, someone else will pick this up and look at it as step towards implementing a more complex model.

Appendices

Appendix A: Implementation Details of the Heatspace

Bug interfaces to the Heatspace

The following function returns the available energy in a cell specified by its coordinates to the calling bug. The function `getValueAtX$Y(x, y)` on line 02 is the Diffuse2d interface to get the heat level at position x, y. The heat to energy conversion is performed on line 02.

```
01    public int getEnergy(int x, int y){
02          int energy = (((getValueAtX$Y(x, y) - minheat) * 10) / maxheat);
03          if(energy > 0)
04                return energy;
05          else
06                return 0;
07    }
```

The following function allows a bug to take energy from the cell specified by x, y. The interfaces to the Diffuse 2d are `getValueAtX$Y(x, y)` in line 03 and `putValue$atX$Y(heatHere, x, y)` in line 08. The energy to heat conversion is performed on line 02.

```
01    public void takeEnergy(int energy, int x, int y){
02          int moreheat = (energy * maxheat) / divisor;
03          int heatHere = getValueAtX$Y (x, y);
04          if(heatHere - moreheat < 0)
05                heatHere = 0;
06          else
07                heatHere = heatHere - moreheat;
08          putValue$atX$Y (heatHere, x, y);
09    }
```

The Seasonal Variance of the Heatspace

The following function deals with implementing seasons in the model. As mentioned before, the seasons vary throughout the year. A year is made up of 100 days. This in divided into four seasons. The evaporation rate is the heatspace multiplier, every day every cell's heat value is multiplied by this. At day 0 the evaporation rate is 1 (i.e. in day 1 the heat level of the heatspace is in equilibrium). From this point until day 65 the evaporation rate of the heatspace is greater

than 1 (i.e. the heat level of the heatspace is expanding). For the remainder of the year the evaporation rate is below 1.

The important thing to note is that the evaporation rate spends more time above 1 than below and that the peak value (1.01875) is higher above 1 than the trough value (0.9875) is below it. The combination of these factors means that there is a substantial net gain of heat every year. This gain is what enables bugs to thrive on the heatspace.

```
01    public void Seasons(){
02          if(day < 25)
03                evaporationRate = evaporationRate + .0005;
04          else if(day < 50)
05                evaporationRate = evaporationRate + .00025;
06          else if(day < 75)
07                evaporationRate = evaporationRate - .00125;
08          else
09                evaporationRate = evaporationRate + .0005;
10          heatspace.setEvaporationRate(evaporationRate);
11          if(++day == 100){
12                year++;
13                day = 0;
14          }
15    }
```

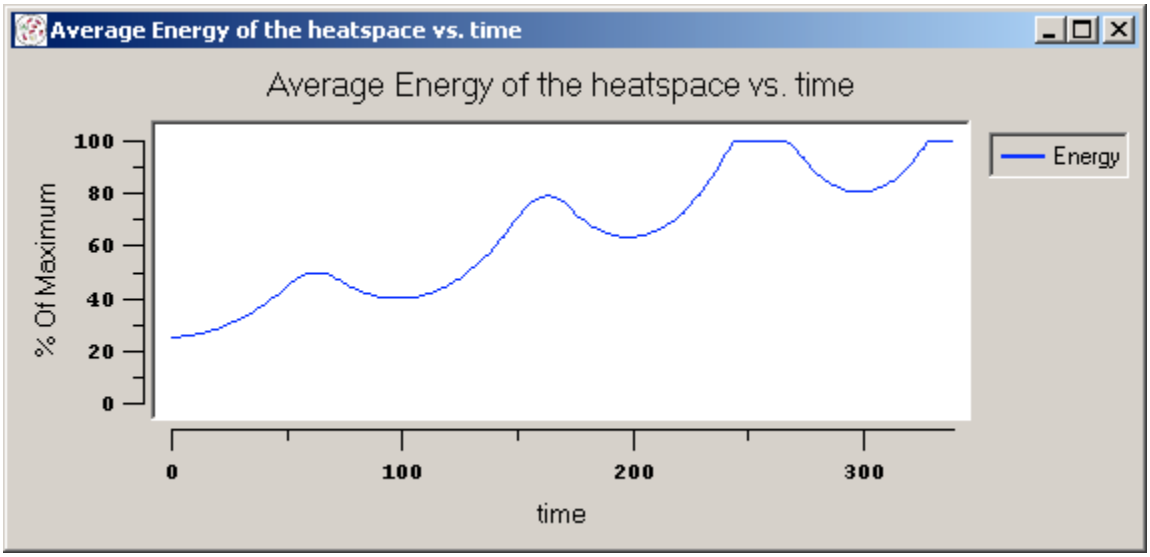

Figure A(a): The Seasonal Variance of the Energy Level of the Heatspace (Without Bugs)

From the graph it is clear that the heat levels peak on day 65 and reach their trough at day 0 of each year. In the absence of bugs the heat level will spiral up to its maximum level in a matter of 3 years from an initial level of 25%. A more long-term view of this widget in action (with bugs)

is shown in figure A(b). In practise the bugs prove to be efficient harvesters and the average level across the heatspace will rarely rise above 40%. Since the bugs cannot harvest the bottom 10% the average level will rarely drop below 20%.

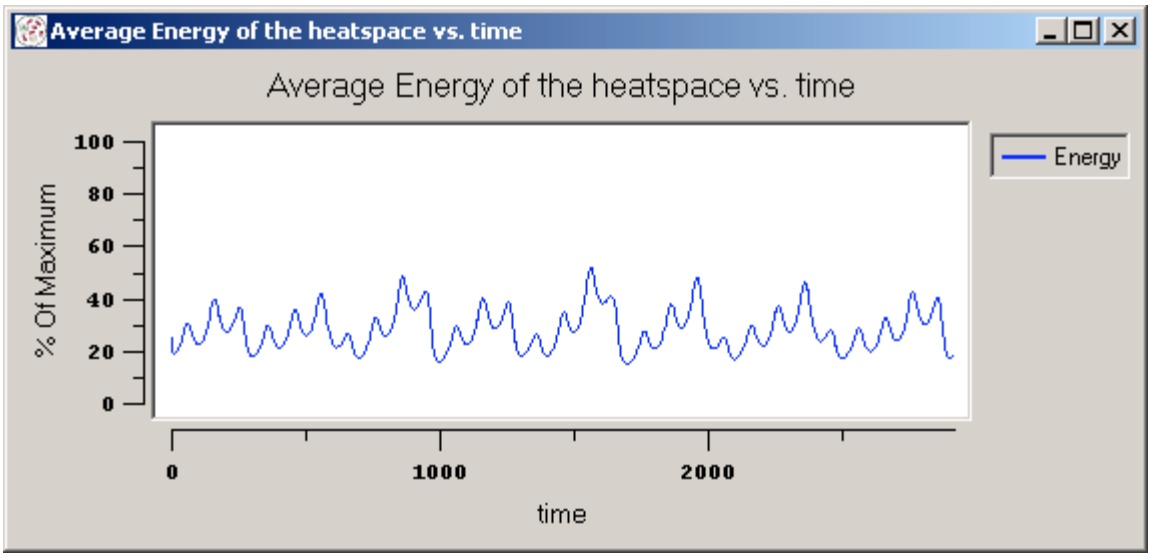

Figure A(b): The Seasonal Variance of the Energy Level of the Heatspace (With Bugs)

Implementation of Diffusion on the Heatspace:

Diffusion is implemented using cellular automata with a simple Moore neighbourhood. For each cell on the Diffuse2d Grid this roughly amounts to:

```
newHeat  = evaporationRate * (myHeat + diffuseConstant * (nbdavg - myHeat));
```

Where: *nbdavg* is the weighted average of the 8 Moore neighbours
newHeat and *myHeat* are the updated and old heat levels of the cell respectively

Appendix B: Implementation of the Genetics

The following block of code is a function performed by the mother to splice her gene with another using bit comparison. It takes in a gene from the father (*a* in line 01) and splices it with its own gene (*gene* in lines 5 and 6) to form the gene *child*. This is returned to the caller (the mother) and passed onto her newborn child. Every bit in the mother's gene is compared with the father's corresponding bit. If they are equal the child gets this bit, otherwise the child gets one or the other randomly. There is a possibility of the bit getting flipped based on the mutation coefficient (lines 8 and 9).

```
01    public boolean[] SpliceGenes(Gene a){
02          boolean [] b = a.GetGene();
03          boolean [] child = new boolean [length];
04          for(int index = 0; index < length; index++){
05                if(gene[index] == b[index]){
06                      child[index] = gene[index];
07                      if(Random(0, mutation) == 0){
09                            child[index] = !child[index];
10                      }
11                else{
13                      child[index] = (Random(0, 1) == 0) ?
                        false : true;
14                }
15          }
17          return child;
18    }
```

The following block of code is a function performed by the mother to splice her gene with another using single point crossover. A crossover point is chosen somewhere along the length of the gene (line 2) and the order of the crossover (line 5) is chosen (i.e. which parent's bits are chosen first). The child is given the bits up to the crossover point from the first parent and the bits after the crossover point from the second parent. There is the same possibility of the bit getting flipped as in the previous scheme.

```
01    public boolean[] CrossOver(Gene a){
02          crossover = Random(0, length);
03          boolean[] b = a.GetGene();
04          boolean[] child = new boolean [length];
05          boolean first = (Random(0, 1) == 0);
06          for(int index = 0; index < length; index++){
07                if(first) child[index] =
                        (index < crossover) ? gene[index] : b[index];
```

```
08                else   child[index] =
                         (index > crossover) ? gene[index] : b[index];
09                if(Random(0, mutation) == 0)
10                       child[index] = !child[index];
11          }
12          gene = child;
13          return child;
14    }
```

Note: In the above to methods, the function `Random(x, y)` replaces `Globals.env.uniformIntRand.getIntegerWithMin$withMax(x, y)` for aesthetic purposes.

The following block of code is a function that takes in the starting (*start*) and finishing points (*fin*) of a genetic attribute as well as its weight (*mul*) and returns it in its integer form (*total*). Because each bug knows the positions and weights of its genetic attributes if can use this function to decode its genes.

```
01    public int DecodeGene(int start, int fin, int mul){
02          int total = 0;
03          while(start < fin){
04                if(gene[start++])
05                      total += mul;
06          }
07          return total;
08    }
```

Appendix C: Contents of the Attached CD.

Here is a brief description of the contents of the attached CD-ROM. If you downloaded this document from the web it may be possible to get further information by sending e-mail to Lorcan.Coyle@cs.tcd.ie with detailed requests.

This CD contains seven directories:

- Demonstration: Contains all the information presented in the Project Demonstration.
- Final Year Report: Contains the Final Year Report, Title Page and Diagrams.
- Gene Pool: Contains a version of the Gene Pool program, as well as the website it came from.
- Source Code: Contains the source code for this project.
- Sugarscape: Contains the Sugarscape movies and the Brooks website for it.
- Swarm: Contains all the necessary files and instructions to install and run Swarm.
- Swarm Documentation: Contains instructions to use the Swarm Java API
- Web Bibliography: Contains archived copies of all the web pages contained in the bibliography of the report

Appendix D: Tables and Figures

Bibliography

Growing Artificial Societies: Social Science from the Bottom Up - Joshua M. Epstein and Robert L. Axtell

The Sugarscape movies and information about the model are available from http://www.brook.edu/SUGARSCAPE/

Integrating Simulation Technologies with Swarm a paper delivered to the "Agent Simulation: Applications, Models and Tools" conference held at University of Chicago in October of 1999. http://www.santafe.edu/~mgd/anl/anlchicago.html

C. G. Langton. "Preface." In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, Volume X of *SFI Studies in the Sciences of Complexity*, pages xiii-xviii, Addison-Wesley, Redwood City, CA, 1992. http://alife.org/index.php?page=alife&context=alife

Fundamentals of Molecular Evolution – Wen Hsiung Li and Dan Graur

Gene Pool available for download from http://www.ventrella.com/GenePool/gene_pool.html

For a definition of Cellular Automata: http://www.brunel.ac.uk:8080/depts/AI/alife/al-ca.htm

The Official Swarm FAQ page: http://lark.cc.ukans.edu/~pauljohn/SwarmFaq/SwarmOnlineFaq.html

The Java Reference Guide to Swarm 2.1.1 is available with frames from http://www.santafe.edu/projects/swarm/swarmdocs/refbook-java/index.html

A tutorial offered at the Santa Fe Institute's 2000 Complex Systems Summer School. Available online at http://www.swarm.org/csss-tutorial/frames.html