Constructing Human-Like Architecture with Swarm Intelligence

Dennis Plougman Buus

August 2, 2006

Department of Computer Science

Aalborg University

Department of Computer Science

TITLE:
Constructing Human-Like Architecture with Swarm Intelligence

PROJECT PERIOD:
Dat6,
February 1, 2006 -
August 1st, 2006

PROJECT GROUP:
d643a

GROUP MEMBERS:
Dennis Plougman Buus

SUPERVISOR:
Yifeng Zeng

ABSTRACT:

This thesis documents the development of a collective building algorithm based upon the principles of swarm intelligence. We detail a design that uses branching rules to map a configuration of building blocks to an appropriate building action in a manner that can produce large scale architectural features indicative of human construction. Furthermore, we examine the feasibility of using a genetic algorithm to evolve rules for use with our collective building algorithm.

Preface

This report documents the work of Dennis Plougman Buus at Aalborg University, Department of Computer Science during the spring semester of 2006.

I would like to thank Yifeng Zeng for support and guidance as supervisor for this project and Kim Plougman Gunvald for his assistance in the implementation of the DirectX 3D rendering framework used in the Test & Visualisation Environment.

Aalborg, Denmark, August 1st, 2006.

Dennis Plougman Buus
dbuus@cs.aau.dk

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 Swarm Intelligence 3

2.1.1 Self-organisation 4

2.1.2 Stigmergy 4

2.1.3 Selected Works 5

2.2 Collective Building 7

3 SwarmArchitect **11**

3.1 General Architecture 11

3.1.1 World and Agent Architecture 11

3.2 Pheromone 13

3.2.1 Diffusion and Decay 14

3.2.2 Repulsion Pheromone 16

3.2.3 Purpose of Pheromone 16

3.3 Agent Perception 17

3.3.1 Density Maps 18

3.4 Agent Movement 19

3.4.1 Build Rules 20

3.5 Branching Rules 21

3.6 SwarmArchitect Algorithm 24

3.7 Summary 26

4 Evolving SwarmArchitect Rules 27

4.1 Genetic Algorithms 27

4.2 Fitness Function 29

4.2.1 Image Comparison of Density Maps 30

4.2.2 Performance 31

4.3 Selection and Reproduction 32

4.3.1 Selection 32

4.3.2 Population and Encoding 33

4.3.3 Genetic Variation 33

4.4 Algorithm Listing 34

4.5 Summary 35

5 Test & Visualisation Environment 37

5.1 Data Visualisation 37

5.2 Interactivity 39

5.3 Summary 39

6 Experimental Results 41

6.1 SwarmArchitect 41

6.1.1 Cornering Problem 41

6.1.2 Building Performance 45

6.2 Genetic Algorithm 47

6.2.1 Experiments Setup 47

6.2.2 Experiment 1 49

6.2.3 Experiment 2 52

6.2.4 Experiment 3 53

6.3 Summary 54

6.3.1 SwarmArchitect 54

6.3.2 Genetic Algorithm 55

7 Conclusion **57**

7.1 SwarmArchitect . . . 57

7.2 Genetic Algorithm . . . 58

7.3 Summary of Contributions . . . 58

7.4 Future Work . . . 59

7.4.1 SwarmArchitect . . . 59

7.4.2 Genetic Algorithm . . . 60

7.4.3 Test & Visualisation Environment . . . 60

A Summary **61**

Chapter 1

Introduction

Swarm Intelligence is a relatively new research area that takes a computational approach to decision making, optimisation problems, and multi agent cooperation strategies. By observing the behaviors of naturally occuring swarm systems exemplified primarily by social insect societies, swarm intelligence seeks to mimic the robustness, adaptivity and efficiency of these systems, which emerges as a result of multiple interactions between simple components.

One specific field within swarm intelligence deals with the collective building behaviors that result in complex structures such as termite nests and wasps hives. Applications of collective building algorithms typically involve simulations of a number of agents that act according to very simple rules, governing the construction of large architectures.

Purpose

The purpose of the work presented in this report is to examine the feasibility of applying the concepts of swarm intelligence to the problem of constructing human-like architecture through the use of a collective building algorithm. We will develop an algorithm for this purpose and examine a method of supporting the algorithm by evolving the necessary computational rules that control agent behavior and decision making.

Motivation

Research into collective building algorithms in swarm intelligence has so far been focused mainly on simulating natural construction such as those con-

structed by termites. There has been hardly any work done into the application of swarm based algorithms to the construction of human-like architecture. This makes the topic of this document an interesting proposition. Moreover, prior work within this field has not fully explored the combination of both quantitative and qualitative stigmergy within the same algorithm, and this presents another interesting challenge.

Structure of this report

The following chapter reviews a number of important concepts particular to swarm intelligence. Concepts are introduced and explained, with references to seminal works in the area.

Chapter 3 details the development of the SwarmArchitect algorithm which we propose as a potential method of extending the reach of collective building algorithms into the realm of human-like construction.

Chapter 4 goes on to examine a strategy for utilising a genetic algorithm to evolve the rules used by SwarmArchitect agents.

In chapter 5 we describe the implementation of a graphical environment meant to aid the development of SwarmArchitect.

Chapter 6 looks at a number of experiments that demonstrate some of the capabilities of SwarmArchitect and the genetic algorithm.

We finish off with chapter 7, in which we summarise and discuss the results of our work, and propose topics for future work.

Chapter 2

Preliminaries

This chapter introduces some of the underlying principles that provide the basis for the SwarmArchitect algorithm and which we shall refer to throughout this report.

The first section reviews some of the fundamental concepts of swarm intelligence and presents some seminal works in this area. In the section following that, we introduce two approaches to collective building algorithms.

2.1 Swarm Intelligence

In their book, Bonabeau et al. [2] define the term *swarm intelligence* as "any attempt to design algorithms or distributed problem-solving devices inspired by the collective behavior of social insect colonies and other animals". When observing social insects, the question arises of how they manage to perform elaborate construction, division of labor, path-finding and much more without hierarchies and command structures. Societies consisting of several thousand unremarkable individuals collaborate to perform remarkably complex tasks and exhibit flexibility, adaptability and fault-tolerance in doing so. With swarm intelligence, we attempt to leverage these highly desirable properties and put them to use in the design of algorithms.

The approach taken in swarm intelligence is to start by observing specific aspects of, typically, social insect behaviors. Based upon observations we attempt to construct a computational model that captures these properties, in order to learn more about the complex interplay that makes them possible. Based upon such a model, we can then refine and apply these techniques to problem-solving and decision-making.

Swarm intelligence, as well as the natural systems we base it on, relies heavily upon two important concepts: *self-organisation* and *stigmergy*. These are what enables swarms of very simple individuals acting and interacting according to simple rules to exhibit highly complex collective behaviors, such as the path-finding and trail-following behavior of ant colonies.

2.1.1 Self-organisation

Self-organisation is based upon four important mechanisms.

- Positive feedback
  Positive feedback mechanisms influence actions in a feedback loop. In the example of ant trail following the positive feedback loop consists of individuals depositing pheromone whereever they walk, which attracts more individuals who in turn deposit more pheromone.

- Negative feedback
  Negative feedback provides a counter to positive feedback. In the example of ant trail following behavior, gradual evaporation of pheromone ensures that a trail will disappear when it is no longer relevant.

- Randomness
  In self-organising systems, randomness is an essential requirement, ensuring the exploration of the problem space that is necessary to discover new solutions. In the ant example, the slight tendency to random walks away from established trails are what lead to the discovery of new food sources or shorter paths between nests.

- Multiple interactions
  Self-organisation is highly dependent on multiple interactions between component parts. Multiple individuals may explore, share knowledge and build upon the actions of others.

2.1.2 Stigmergy

Stigmergy is the concept of communication by way of changing one's environment. Stigmergy works by an individual acting upon the configuration of it's local environment, performing an action which changes the environment into another configuration. This, in turn, affects the decision of the next individual that senses this new configuration.

In swarm intelligence we often talk of two kinds of stigmergy: quantitative and qualitative. Pheromone trails are the archetypical example of quantitative stigmergy. They influence decisions in proportion to their intensity. Qualitative stigmergy involves coupling specific stimuli with specific actions, such as in the case of wasp nest construction. Wasps returning with building materials sense and recognise the configuration of the nest which influences their choice of where to place their mouthful of paper.

Stigmergy as a communication strategy provides a great deal of benefits. Particularly, information is stored locally, in the environment where it is pertinent, so deciding when and to whom to transmit information becomes a non-issue. We do not need to theorise about which individual might benefit from a certain piece of knowledge, nor take steps to avoid redundant retransmission.

2.1.3 Selected Works

In this section we provide brief examinations of selected works within the area of swarm intelligence. These works serve as examples of the many models of insect behavior that have influenced later research, and we shall refer back to these examples in later chapters.

Trail Following

In experiments with ant path-finding, Deneubourg et al. [3] devised a model of ant trail-following behavior. The experiment involved placing a paper bridge between an ant colony and a food source as shown in figure 2.1.

From this experiment, Deneubourg et al. came up with an equation describing the trail following behavior. If an ant has a choice between two paths A and B, then the probability P_A of chosing path A is given by:

$$P_A = \frac{(k + A_i)^n}{(k + A_i)^n + (k + B_i)^n} = 1 - P_B, \tag{2.1}$$

where A_i and B_i are the amounts of pheromone on paths A and B, respectively. The parameter k controls the randomness of the function, as a high k increases the likelyhood that an ant chooses it's path regardless of pheromone intensities. The parameter n describes the linearity of the function, such that a high n will make even a small difference in pheromone between the two paths influence the decision to a large degree. The authors found that setting $n = 2$ and $k = 20$ provided the best fit of the experimental data.

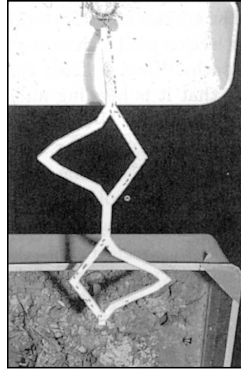

Figure 2.1: Live ant experiment. After Bonabeau et al. [2].

This equation has since been generalised and adapted to many different swarm intelligence applications in later works.

Clustering

In another article, Deneubourg et al. [4] describe a model for the observed clustering behavior of certain ant species that arrange corpses of dead ants in cemetaries. Figure 2.2 shows this observed behavior.

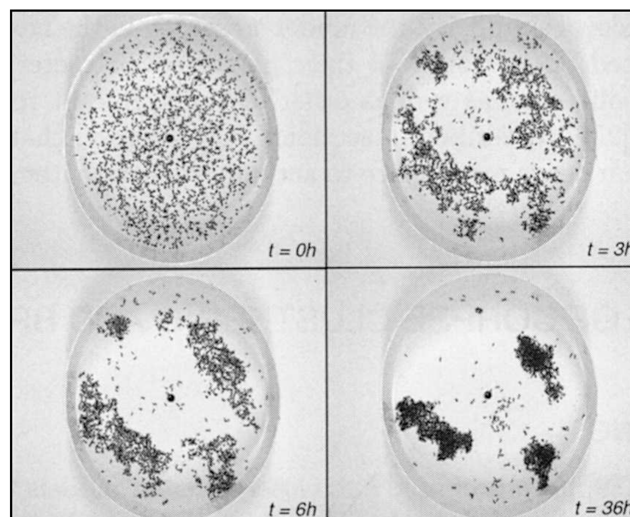

Figure 2.2: Ant workers form clusters from 1500 scattered corpses. After Bonabeau et al. [2].

The model consists of a number of agents moving randomly in a space, deciding upon an action to take after every step. If an unladen agent encounters a corpse it must decide whether or not to pick it up as a function of the perceived density of corpses in the immediate area. The probability p_p that the agent will pick up the corpse is given by:

$$p_p = \left(\frac{k_1}{k_1 + f}\right)^2, \tag{2.2}$$

where k_1 is a threshold value and f is a measure of the perceived fraction of corpses in the area. As is evident from this equation, when the density of corpses f increases far above the threshold, the probability that the agent will pick up the corpse approaches 0, thus making agents less likely to remove corpses from clusters that have already formed.

Conversely, after every step an agent carrying a corpse must decide whether or not to drop it. The probability p_d that the agent drops the corpse is given by:

$$p_d = \left(\frac{f}{k_2 + f}\right)^2, \tag{2.3}$$

where k_2 is another threshold value. The effect of this equation is opposite of the former in that an increasing density of corpses makes it more likely that the agent will drop the one it is carrying.

2.2 Collective Building

This section describes some of the previous works dealing with collective building algorithms.

Quantitative Stigmergy

A large body of research exists which deals with quantitative stigmergy based collective building algorithms which mimic termite nest construction. An example of such an approach is examined in [9]. It consist of a three-dimensional world where pheromone concentrations influence agents' decision to place building blocks. Different kinds of pheromone are deposited and diffuse out across the environment. The gradient of pheromone affects the probability of an agent placing a block.

In [9], termite foragers are spawned into the world with the sole purpose of travelling from one point to the other, depositing pheromone along the way.

Builder termites then place blocks with a certain probability within a limited band of pheromone concentration such that a building neither takes place in areas of very high concentrations (where traffic is high), nor areas with very low consentrations. The result is the formation of tunnels which cover the trails travelled by foraging termites, as shown in figure 2.3.

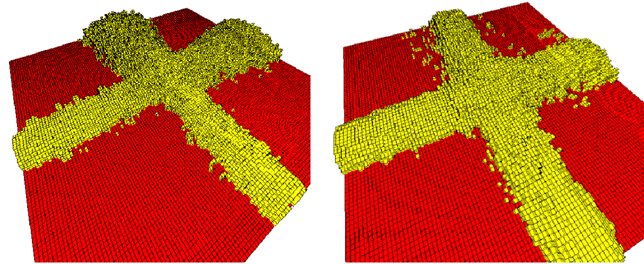

Figure 2.3: Tunnels are built covering the paths travelled by forager termites. After Ladley and Bullock [9].

Another source of pheromone, a simulated termite queen emitting a steady stream acts as template function, the gradient of which guides the building of the royal chamber. This is shown in figure 2.4.

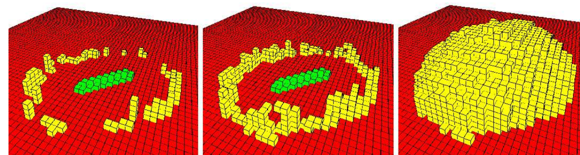

Figure 2.4: A royal chamber is formed around a queen emitting a pheromone template guiding construction. After Ladley and Bullock [9].

Qualitative Stigmergy

Theraulaz and Bonabeau [11, 10] introduced a simple algorithm for the purposes of experimenting with collective building controlled by qualitative stigmergy. In their approach, a number of agents move about randomly in a three-dimensional lattice and deposit building blocks when they encounter a triggering configuration.

Agents posess a library of rules which they consult every time they move. If the local configuration matches a rule in it's library, the agent performs the

specified building action. Using randomly generated rule sets, occasionally a structure would be constructed which resembled a wasps nest, examples of which are shown in figure 2.5.

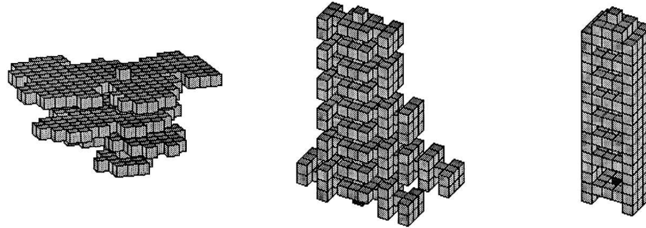

Figure 2.5: Examples of structures built by agents using qualitative stigmergy. After Theraulaz and Bonabeau [11].

In a later article, Bonabeau et al. [1] applied a genetic algorithm to the problem of generation rule sets. Deeming it problematic to quantify structuredness of a generated wasps nest in an algorithmic fitness function, they used human evaluation to ascertain a fitness value.

Chapter 3

SwarmArchitect

This chapter examines the development of SwarmArchitect; a collective construction algorithm based upon the principles of swarm intelligence. Each section presents the challenges of a specific problem area and examines the different solution strategies considered during development of SwarmArchitect.

The first section summarises the overall architecture of the algorithm, describing the basic concepts and stating some of the important constraints imposed by our problem specification. Each following section examines the details of a specific area of implementation, describing and analysing the challenges posed and a number of different possible implementation choices.

3.1 General Architecture

In [11, 10], one of the first experiments with qualitative stigmergy in swarm construction, Theraulaz and Bonabeau described a very simple swarm building architecture where agents moved randomly without restriction in a 3-dimensional space, placing a building block at their current position when they encountered a triggering configuration. This work serves as the inspiration for SwarmArchitect.

3.1.1 World and Agent Architecture

The SwarmArchitect algorithm makes use of a number of simple agents that move about in a discrete, 3-dimensional lattice, depositing building materials

according to a set of stimulus-response rules. The lattice is constructed as a 3-dimensional array of objects containing information about the state of each cell in the lattice, such as the presence of a building block and the intensity of pheromone. One cell may contain 1 or 0 building blocks.

When a building block is placed in the world, a certain amount of pheromone is deposited along with it. These pheromone concentrations diffuse and decay as the simulation proceeds.

It is important to note that the concept of an 'agent' in most swarm intelligence applications bears no resemblance to the common notion of independent, intelligent agents. When we refer to agents in this document, we merely use the term as a useful way to visualise the concepts of insect-inspired construction.

In SwarmArchitect, an agent is represented by a set of coordinates (x, y, z), designating a position in the lattice. A looping construct takes the role of performing all actions and decisions on behalf of our imagined agents, visiting each of these coordinates in turn, performing the appropriate building actions and updating pheromone intensities in the surrounding area. An agent is a means of keeping track of which point in the world we are operating on. As the algorithm runs, agents are moved about the world by changing their coordinates. Agents move in turn, selecting their direction stochastically taking into account the pheromone intensities in the world. Agents will tend to move towards areas with fresh building activity.

Individual agents are able to directly sense the environment in a small area around their position. Agents match this direct sensory perception to a library containing a limited number of triggering configurations, acting upon these in different ways when they are encountered. Agents may also refer to three different aggregate views of the densities of building material in their current position.

Agents do not sense each other, nor do they communicate or act according to any explicit cooperation strategies. All communication and coordination is achieved by means of stigmergy. When an agent changes the environment, this change, in turn, affects subsequent decisions of other agents.

SwarmArchitect employs a combination of quantitative and qualitative stigmergy. While an approach based purely on qualitative stigmergy could theoretically allow the expression of human-like architecture on its own, the space of possible stimulating configurations would be enourmous, since large scale architectural features would require a very large sensory range. By limiting the sensory range of agents to a $3 \times 3 \times 3$ cube, the size of this space is reduced. Provisions for large scale features are made in the form of quantitative

stigmergy which influences the choice of action when an agent encounters a stimulating configuration.

The algorithm stops after a predetermined number of iterations.

3.2 Pheromone

In this section we examine the role of pheromone in SwarmArchitect. Pheromone is one example of a stigmergic messaging medium, and in many swarm intelligence applications, it is the sole means of communication and coordination between individual agents. Chapter 2 described a number of swarm intelligence applications that put pheromone to great use. There is, however, no reason that the use of pheromone deposits must always take such an exclusive role in swarm intelligence applications.

In SwarmArchitect, pheromone is used to guide agents towards areas of intensive building activity, in order to achieve faster convergence towards a solution. Pheromone is deposited in the world along with building blocks as they are placed, in effect pointing out areas where building activity has taken place recently. In SwarmArchitect, pheromone takes a slightly lesser role than what is the norm for many swarm intelligence algorithms. However, pheromone deposits are an important measure which facilitates cooperation between agents. This mechanism takes the place of more sophisticated coordination and communication strategies, becoming what is effectively a call for aid. Figure 3.1 shows an example of pheromone deposits around newly placed building blocks.

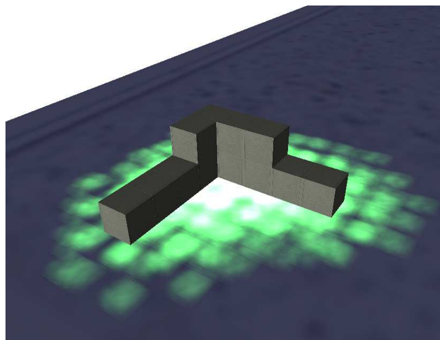

Figure 3.1: Pheromone deposits formed around newly placed blocks in the SwarmArchitect world.

Swarm intelligence applications often rely on the ability of the actions of an individual to recruit other individuals to assist in whatever effort is being carried out. In SwarmArchitect, we exploit the fact that the occurrence of multiple pheromone deposits is an indication that an area has the potential of additional rule matching possibilities and thus could benefit from attracting attention from several agents. In what Bonabeau and Theraulaz ([11]) name a *coordinated algorithm*, the building action associated with the matching of one rule will often result in a new configuration which is also present in the library of building rules. It is therefore reasonable to suggest that there is a higher chance that agents will locate buildable configurations in areas where building activity has recently taken place.

Since SwarmArchitect relies on other measures for guiding the exact placement of building blocks, it is only necessary to track pheromone concentrations in two dimensions. Pheromone deposits will have fulfilled their role if they just have the effect of attracting agents to the general area; thus, when we apply pheromone we can simply ignore the height dimension and apply all pheromone deposits and updates to the 'floor' of the environment. Therefore, unlike other algorithms which have to perform pheromone update operations in every cell in a three-dimensional environment, SwarmArchitect is able to process this operational step fairly quickly.

3.2.1 Diffusion and Decay

In order for pheromone deposits to serve their purpose as a means to recruit agents to areas where building activity is happening, it is necessary for the deposits to spread over a larger area beyond the cell where a fresh block has been placed.

One possible solution would be to simply deposit pheromone in a larger area around each building block, adding to the pheromone intensities already present in these cells. Pheromone would thus reach out in a predetermined radius around a building block.

Another approach would be to implement a mechanism for diffusing pheromone deposits, allowing them to spread further and further out as the simulation progresses. One simple approach to diffusion involves moving part of the pheromone of a cell to its neighboring cells. Assuming each cell diffuses pheromone to each of its eight neigbors, the amount of pheromone $\Delta\tau_{c_i}$ that each neighbor receives from cell c_i can be expressed as:

$$\Delta\tau_{c_i} = \frac{\tau_{c_i} \cdot d}{8}, \tag{3.1}$$

where $0 < d < 1$ is the *diffusion coefficient*, which regulates the percentage of pheromone moved from a cell to its neighbors and τ_{c_i} is the amount of pheromone in cell c_i. A higher d results in more rapid diffusion.

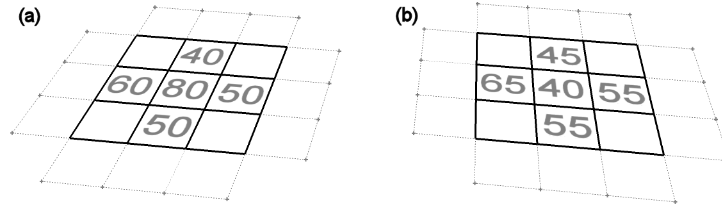

Figure 3.2: Example of diffusion of pheromone from the middle cell to its neighbors (only 4 shown) with $d = 0.5$. (a) Before. (b) After.

In SwarmArchitect, the spread of pheromone is achieved by means of diffusion, since this allows for pheromone to spread dynamically and reach out as far as necessary to attract agents, eliminating the need to theorise about which fixed radius is sufficient.

While the choice to utilise diffusion does involve more computation, the fact that the operation is only applied to a limited number of cells in the environment means that it does not increase overall computational complexity by any significant amount.

Once an area has been built up to the point where rules are no longer being matched, we must ensure that agents do not continue to be recruited to the area. In order to ensure this, pheromone will evaporate at a steady rate, such that recruitment ends fairly quickly after building actions cease, and agents may be directed to another area.

Pheromone evaporation can be applied to a cell c_i with a simple procedure as shown below:

$$\tau_{c_i} \leftarrow (1 - \rho) \cdot \tau_{c_i}, \tag{3.2}$$

where τ_{c_i} is the amount of pheromone in cell c_i and $0 < \rho < 1$ is a coefficient dictating the speed of evaporation.

At the end of every iteration the evaporation rule of Equation 3.2 is applied in one pass to every cell of the environment floor. Then, a diffusion pass visits each cell once more and for every cell moves an amount of pheromone according to Equation 3.1 to each neighbor. The diffusion procedure does not incur any loss in the overall amount of pheromone present in the environment, except when pheromone is diffused out beyond the edges of the world.

3.2.2 Repulsion Pheromone

In the discussion above, we have only mentioned one type of pheromone; the building pheromone which attracts agents to areas with recent building activity. However, SwarmArchitect makes use of another type of pheromone as well. A small amount of *repulsion pheromone* is placed by every agent each time it enters a new cell. The purpose of repulsion pheromone is to ensure that agents do not clump together in a small area unless there is a reason to do so; that is, when the area has recently been built on. As the name suggests, agents are repulsed by this type of pheromone and will tend to move away from areas with high concentrations. This causes agents to distribute more evenly during their search for suitable building sites, and to scatter more quickly once building in an area has ceased.

In order that repulsion pheromone does not become counterproductive, it is deposited in much smaller quantities than building pheromone so that it's influence becomes negligible when building activity is occurring. It's primary use is during periods where little or no building is taking place and agents need to distribute their search efforts over a larger portion of the world space. Therefore, repulsion pheromone is deposited at a fixed fraction of 5-10% of the amount of building pheromone deposited with a single block. Repulsion pheromone is diffused and evaporated in the same way as building pheromone.

Another mechanism of repulsion was utilised in the Ant Colony System (ACS) algorithm [6, 5] for the *travelling salesman problem* in order to prevent agents from clustering around a single path. In that algorithm, the presence of an agent simply cause the amount of attractive pheromone to decrease. In SwarmArchitect we use a seperate repulsion pheromone since we require a repulsion mechanism that also works when no attractive pheromone is present.

3.2.3 Purpose of Pheromone

These measures combined provide a robust and easily adjustable means for distributing pheromone in a way that can support an effective distribution of agents across the virtual building site, recruiting the help of additional agents in areas that are ready to receive further building efforts. Indirectly, agents construct and share a dynamic, real-time knowledge base of desirable building locations, leading to more efficient utilisation of agents and thus quicker convergence of the algorithm towards a solution.

3.3 Agent Perception

Agents must be able to perceive some features of their environment in order to make decisions about where to focus building efforts and where to place blocks. As in the work by Theraulaz and Bonabeau [11], SwarmArchitect agents have a direct perception range which is limited to a cube of $3 \times 3 \times 3$ cells in the environment surrounding its position. Additionally, in SwarmArchitect, agents are able to sense concentrations of pheromone on the ground in a small area in each of the directions they can travel. The range of pheromone perception is 2 squares ahead, behind, left and right of the agent. Being able to sense pheromone beyond their visible range aids the agent in travelling in the right direction towards the source of pheromone being diffused outwards.

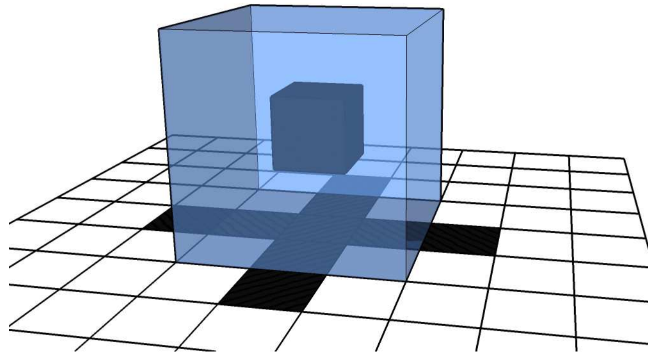

Figure 3.3: An illustration showing the perception ranges of the SwarmArchitect agents. The large transparent box represents the visual range and the darkened areas on the ground represent the agents' range of smell. The black box in the middle represents the agent itself.

Since the construction of human-like architecture requires the formation of features that are much larger than this very small perception range, additional sensory information must be made available to agents. The following section details a novel approach that provides agents with immediate sensation of block density as it pertains to the cell where the placement of a building block is being considered.

3.3.1 Density Maps

As building progresses, SwarmArchitect maintains three aggregate views of the overall density of building blocks, which we call *density maps*. These density maps may be likened to an x-ray photograph of the simulated environment, taken from three different positions outside the simulated environment; front, side and top. Figure 3.4 illustrates this concept.

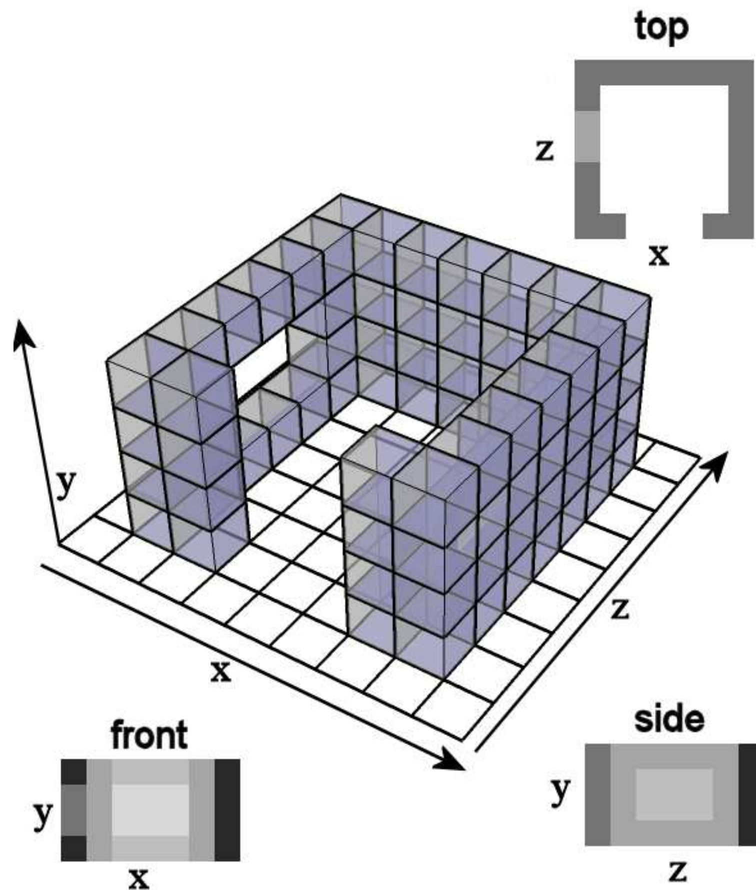

Figure 3.4: An illustration of a small building with the corresponding front, side and top density maps shown as grey-scale images

The maps are implemented as 2-dimensional arrays, with each cell containing a count of blocks. Consider the top map in figure 3.4. We construct this map by placing our viewpoint directly above the model. To determine the 'colour' of each 'pixel' in the map, we count the number of blocks as we look down one coloumn in the environment. We use the same approach to construct

the other two maps, but simply repositioning our viewpoint. In the actual implementation, we construct and update all density maps simultaneously, when a building block is placed in the world. We map the 3-dimensional world coordinates to the 2-dimensional map coordinates as follows:

		World Coordinate
Front Map	x	x
	y	y
Side Map	x	z
	y	y
Top Map	x	x
	y	z

As the density maps can be updated in constant time every time a building action occurs, they provide a valuable means of large scale indirect perception at the cost of a negligible increase in computational complexity. Agents can immediately access aggregate density values and include this information in their decision making process. We shall see exactly how agents make use of density maps in section 3.4.1.

3.4 Agent Movement

The work by Ladley and Bullock [9] on a simulation of termite nest construction, shows that placing constraints on agent movement may be useful in achieving faster convergence towards a solution. The authors reported a significant improvement in running time of their algorithm when logistic constraints were utilised. The addition of such constraints is another measure by which the space of possible solutions is reduced, as agents become much less likely to spend time travelling in areas that are unlikely to exhibit any sort of triggering stimulus. When agents are unable to move randomly in empty space, but instead restricted to movement along existing architecture, more time is dedicated to finding ways of expanding the architecture already in place and less on meaningless travel. For this reason, SwarmArchitect also places restrictions on agent movement.

Agents may not move diagonally, nor may they move in free space. Agents may only move into an empty cell (containing no building block) and then only if it shares a face with at least one non-empty cell. Intuitively, agents are subject to the laws of gravity, but they have the ability to climb existing architecture.

When an agent moves, it chooses a direction by stochastic selection, taking into account the amount of pheromone of the surrounding cells. Agents are attracted to high concentrations of pheromone deposited as a result of building action.

We utilise an adapted form of the movement selection equation from the work by Deneubourg et al. [3], which we described in section 2.1.3 as equation 2.1. We generalise this equation so that it becomes suitable for an arbitrary number of directions. Let C be the set of all allowable target cells. η_{c_i} is the desirability of the target cell $c_i \in C$. The probability p_{c_i} that an agent will move to cell $c_i \in C$ is given by:

$$p_{c_i} = \frac{(r + \eta_{c_i})^\alpha}{\Sigma_{c_j \in C}(r + \eta_{c_j})^\alpha}. \tag{3.3}$$

Note that instead of basing the decision upon raw pheromone concentration values, we calculate a desirability η_{c_i} of moving to cell c_i, in order to support the increased range of smell as well as the effects of repulsion pheromone. The desirability η_{c_i} is calculated according to pheromone concentrations, by adding the concentrations of building pheromone in the two squares in the direction of cell c_i and the concentration of repulsion pheromone present in the opposite direction. Thus, a high concentration of repulsion pheromone to the south has a positive impact on the probability of the agent choosing to move north.

The parameters r and α make it possible to adjust the pheromone attraction behavior of agents. The parameter α controls the linearity of the function. A high value of α will make even small differences in pheromone concentrations have a high impact on the choice of the agent. Thus, α is an important way of affecting the path that an agent takes toward a source of pheromone, with a high α making agents follow the pheromone cloud directly to the strongest source. The parameter r adjusts the tendency of the agent to choose it's direction randomly. It serves a different purpose than α in that it allows us to adjust the proportion of exploration vs. exploitation.

3.4.1 Build Rules

Mapping a perceived configuration to an appropriate building action could conceivably be done in a number of different ways. In this section we examine some of the issues involved in making such decisions.

In the simple agent architecture described in [11] agents place a building block in the cell they are occupying at the time that a triggering configuration is encountered. A triggering configuration is a specific placement of building blocks in a $3 \times 3 \times 3$ cube surrounding the agent. Every time an agent moves, it checks its surroundings and tries to match this information to one of the triggering configurations in its rule set. A rule is a mapping from exactly one triggering configuration to exactly one building action which in Theraulaz and Bonabeau's [11] system is the placement of a block of one of two colours in the cell that the agent is occupying.

Another possible approach would entail sensing a configuration and mapping this to an entirely different configuration, in effect performing multiple building actions in a single turn.

SwarmArchitect allows only one block to be placed after a triggering configuration has been encountered, but the block may be placed in any position within the perception range of the agent. SwarmArchitect rules therefore map a triggering configuration to the placement of a block in a specific location.

3.5 Branching Rules

The goal of SwarmArchitect is different from existing work on swarm construction, in that we wish to produce human-like architecture. If we view the SwarmArchitect algorithm as a language, then our goal is to develop a language that is powerful enough to express features that are indicative of human-like architecture. These include long straight walls, right-angled corners, and openings within walls, such as doors and windows.

If the swarm is to be able to construct a corner, for instance, certain conditions must be met. At some point during the construction of a straight wall, an agent must be able to decide to no longer continue extending the wall, and instead place a block off to the side in order to start a corner. It is clear that such a decision is simply not possible if we map a single triggering configuration in the form of a block placement pattern in a $3 \times 3 \times 3$ cube to exactly one resulting configuration. In this case, the rule that allows the building of a long straight wall would always result in walls being built all the way to the edge of the world or until they hit existing structures.

Two novel approaches in SwarmArchitect make it possible for agents to build features such as corners. First, *branching rules* allow the mapping of a single triggering configuration to several different building actions. When an agent

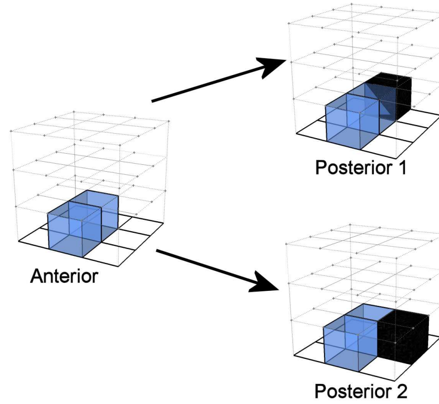

Figure 3.5: An illustration of a SwarmArchitect rule. The darkened blocks show the two possible block placements.

encounters a triggering configuration, it must then decide which building action to take. Second, the aforementioned *density maps* add a layer of large scale information about the environment which can be taken into account when deciding between different possible courses of action. Figure 3.5 shows an example of a SwarmArchitect rule.

Let A be the set of all possible actions that an agent has to choose from when encountering a specific triggering configuration. The probability p_{a_i} that an agent choses build action $a_i \in A$ is given by:

$$p_{a_i} = \frac{\eta_{Fi} + \eta_{Si} + \eta_{Ti}}{\sum_{a_j \in A} \eta_{Fj} + \eta_{Sj} + \eta_{Tj}}, \tag{3.4}$$

where η_{Fi}, η_{Si}, and η_{Ti} are the front, side, and top desirability values for the cell being considered for block placement by action a_i.

We wish to make our decision on which building action to take dependent on the density values read from the density maps. In order to do so, we calculate desirabilities using an adaptation of the equations from the Basic Model of clustering behavior from Deneubourg et al. [4] described in section 2.1.3. We use equations 2.2 and 2.3 as follows:

Let $M = \{F, S, T\}$ be the set of density maps; front, side and top. The desirability value η_{mi} for action a_i and map $m \in M$ is given by:

$$\eta_{mi} = \begin{cases} \left(\frac{D_{mi}}{\delta_{mi}+D_{mi}}\right)^2, & \text{if } \delta_{mi} > 0 \\ \left(\frac{|\delta_{mi}|}{|\delta_{mi}|+D_{mi}}\right)^2, & \text{if } \delta_{mi} < 0 \\ 0, & \text{if } \delta_{mi} = 0 \end{cases}, \tag{3.5}$$

where δ_{mi} is the density threshold and D_{mi} is the density value read from density map m for the cell being considered for block placement by action a_i. The density thresholds associated with each action (or posterior) in a rule make it possible to adjust the influence of building densities on our choice. If we wish for the desirability of an action to drop as the density of blocks from the front map increases, we set the density threshold D_{Fi} to a negative value. If we want the desirability to increase, we set the threshold to a positive value. If we want a particular density reading to have no effect on the desirability, then we set the threshold to 0.

In the equation above, we calculate a probability for each building action, however, in the implementation of SwarmArchitect, the actual selection is done deterministically. The action with the highest calculated p_{a_i} is the one we select. It is entirely possible and straight-forward to make Swarm-Architect use stochastic selection such as basic *roulette wheel selection* but deterministic selection makes it easier to guarantee that certain large scale features can be consistently produced. For instance, with the right coefficients for densities set, it becomes possible to ensure that walls forming the outside shell of a building can join together, for instance, by making a corner-producing action always take place if a high density is detected in the direction perpendicular to the wall. This would indicate that a parallel wall has already formed a corner, which we would want to meet up with.

Despite using deterministic rules, we can still produce different buildings from the same set of rules due to initial perturbations, such as the distribution of agents and pre-built material and the stochastic movement selection.

To lessen the risk of non-convergence, SwarmArchitect does not allow removal of building blocks once they have been placed. While removal of blocks might provide the means for cleaning up misplaced blocks, this is not implemented in SwarmArchitect, since one of the major goals of the algorithm is quick convergence. In terms of the expressiveness of the language that the algorithm represents, there is nothing to suggest that allowing removal of blocks would make this language any more powerful. The intuitive notion is that it is more efficient to place blocks in the correct positions from the beginning, rather than relying on cleaning up mistakes afterwards.

In order to facilitate fast and efficient matching, rules are encoded as strings of bits. Comparing the anterior of the rules to the local configuration of the worlds is straight-forward. First, a similarly encoded bit string representation of the $3 \times 3 \times 3$ area around the agent is constructed. Then, each anterior rule may be matched against this string.

As an additional measure to encourage human-like architecture, SwarmArchitect excludes certain types of rules. Diagonal placement of blocks is not allowed, so any building block placed in the world must share a face with at least one other block.

3.6 SwarmArchitect Algorithm

Algorithm 1 provides a high-level description of the SwarmArchitect algorithm.

The algorithm progresses as follows.

First, the world data structures are initialised, a rule set is taken as input and a number k of agents are created and scattered randomly across the world.

In the main loop, the simulation runs for a predetermined number of interations.

In each iteration, every agent performs a number of actions. First, the agent senses the configuration of the environment within it's perception range and looks up this configuration in the rule set. If the configuration matches a rule, then the agent must decide which action (posterior) to take. It does so by utilising equation 3.4. Once an action has been decided upon, the agent places a building block in the appropriate position, along with an amount of building pheromone at ground level.

The agent must then decide which direction to move. It examines each possible cell it can move to and calculates the probability of moving there using equation 3.3. Using roulette wheel selection, the agent then decides upon a cell and moves there.

At the end of an iteration, all agents have performed their actions. The world then applies diffusion and evaporation to the pheromone concentrations and once these operations have completed, a new iteration can begin.

Algorithm 1 SwarmArchitect Algorithm

```
/* Initialisation */
Input: A set of rules
Input: Simulation parameters including max_iterations and k
Initialise world
Construct initial density maps
for each agent k do
  set random (x_k, y_k, z_k)
end for
/* Main loop */
for 0 to max_iterations do
  /* Agent loop */
  for each agent k do
    Construct sensory information for (x_k, y_k, z_k)
    for each anterior rule do
      if (sensory information matches rule) then
        for each posterior rule a_i ∈ A do
          Calculate p_{a_i} according to equation 3.4
        end for
        Place building block according to the rule with the highest p_{a_i}
        Deposit pheromone in the appropriate floor cell beneath the
        newly placed building block
      end if
    end for
    for all allowable target cells c_i ∈ C do
      Calculate p_{c_i} according to equation 3.3
    end for
    Select target cell with Roulette Wheel selection
    Move agent to chosen cell
  end for
  for each cell c_i in world do
    Evaporate pheromone according to equation 3.2
  end for
  for each cell c_i in world do
    Move an amount of pheromone from c_i according to equation 3.1 to
    each neighboring cell
  end for
end for
```

3.7 Summary

The SwarmArchitect algorithm combines many concepts from previous work in collective building and swarm intelligence and introduces entirely new ones as well. It demonstrates a way to combine both quantitative and qualitative stigmergy in the same agent architecture through the use of both pheromone deposits and the mapping of local configurations to building actions. Agents are able to sense both their immediate environment as well as refer to large scale aggregate views of building densities. Combined with branching rules, this makes it possible for agents to build architectural features that are indicative of human construction and which are much larger than their immediate visual range.

Chapter 4

Evolving SwarmArchitect Rules

The quality of the architecture that is output by SwarmArchitect is dependent on the set of rules it receives as input. Agents can only perform a building action when their library of building rules contains a rule that matches the local configuration of building blocks in the immediate vicinity of the agent.

The highly complex interplay between rules is not easily understood nor predicted and while one can manually construct rule sets that together form simple shapes, this is not feasible if we wish to construct rule sets capable of more advanced architecture. Even with efforts to reduce the space of possible solutions, the number of different possible rules and combinations thereof is still immense. Therefore it is necessary to rely on other means to explore this space in order to discover rule sets that make it possible to achieve more complex architecture.

This chapter examines the design of a genetic algorithm which is tasked with generating rule sets for use with SwarmArchitect in order to examine the feasibility of such an approach. The following sections introduce the concept of genetic algorithms and details a number of key points that must be considered when one wishes to use a genetic algorithm to solve a given problem. Subsequent sections detail the implementation of the genetic algorithm for SwarmArchitect, including a discussion of the all important *fitness function*.

4.1 Genetic Algorithms

Genetic algorithms [7, 8] provide a useful means of searching a very large problem space in order to discover a “good” solution to a specific problem.

A genetic algorithm applies the principles of natural evolution to a computational problem, whereby a population of potential solutions evolve from one generation to the next. Natural selection influenced by the quality of genetic properties determines the chances of reproduction.

The implementation of a simple genetic algorithm generally requires a number of key features:

- A population of potential solutions (individuals).
- A method for determining the relative *fitness* of each individual.
- A strategy for selecting individuals for reproduction.
- Methods of applying evolutionary variation such as mutation and crossover.

A population typically consists of a number of potential solutions encoded as strings of bits, but other representation schemes exist. The simple genetic algorithm starts with a population of randomly generated solutions or *genomes*. A genome includes all the chromosomes that make up a solution, and it is these that we are interested in evolving.

Constructing an appropriate fitness function that is able to evaluate and quantify the fitness of each individual in the population is vital to the implementation of a genetic algorithm. The fitness function must be able to measure how well suited an individual is as a solution to the problem in question, in order that the population can gradually move towards higher and higher fitness values and a good solution. Genetic algorithms cannot be used to solve problems where it is impossible to construct a fitness function.

After all individuals in a population have been evaluated for fitness, a selection process selects pairs of genomes that will be used to produce the new generation. Selection favours individuals with higher fitness.

When a pair of genomes has been selected, there is a chance that *crossover* will occur. This involves choosing a random crossover point in the genome sequence, and exchanging the chromosomes of the two parents at that point to produce two children. If crossover does not occur, the parent genomes are simply copied into the new generation. In practical implementations of genetic algorithms, the probability of crossover is typically fairly high.

After a pair of offspring has been produced, a mutation pass is applied. During the mutation phase, each bit in a genome has a very small chance of

being reversed. In the case of bit string encoded chromosomes, the parameter for mutation is usually a very low number, such as 0.001.

In order to leverage the benefits that a genetic algorithm affords us in terms of an evolutionary approach to producing effective rule sets for SwarmArchitect, these implementation details must be carefully considered and tailored to this unique problem space. The population must consist of a number of appropriately encoded potential solutions to the problem and a method for evaluating the fitness of different rule sets must be developed.

4.2 Fitness Function

The major challenge in developing the genetic algorithm for SwarmArchitect is designing an appropriate fitness function. Our ultimate goals are mainly subjective, as we are searching for ways to construct architecture with features that are functional and aesthetically pleasing from a human point of view. The problem then becomes one of extracting an objective fitness score for use in natural selection, from information that, to a large extent, is subjective.

Theraulaz and Bonabeau [11] suggested using human evaluation as the fitness function for a genetic algorithm. A generation consisting of a number of rule sets would be generated, and the simulation would be run with each of these sets to produce a number of structures. Human observers would rank each resulting structure on it's subjective qualities, such as 'structuredness'. Note, however, that this fitness function does not operate directly on the rule sets, but rather on the structures they produce, which will differ from one simulation run to the next, due to the stochastic nature of agent movement.

This approach was also considered for SwarmArchitect, but subsequently rejected. In order to avoid the need to recruit human observers and due to the expected number of generations needed to evolve useful rule sets, the decision was made to look for an algorithmic approach.

The solution is an algorithm which can compare architecture built by SwarmArchitect to hand crafted architecture and return an objective measure of their similarity. By comparing generated architecture to human built architecture, we are still able to include subjective qualities in the fitness calculation. The hand crafted architecture captures the subjective qualities in a representation that may be compared directly to the generated architecture. However, as with the approach developed by Theraulaz and Bonabeau [11], this strategy does not provide a 1:1 mapping between a single individual

and a fitness score. The score achieved by a rule set may differ from one simulation run to the next, due to stochastic agent movement.

4.2.1 Image Comparison of Density Maps

Devising a method for comparing two 3-dimensional structures is by no means straight-forward. The fitness function for SwarmArchitect makes use of the density maps which influence agent decisions. The three density maps together represent a complete representation of a struture. Since the maps are equivalent to gray-scale images, we can use image comparison algorithms to calculate an objective measure of the similarity between two structures. In fact, the original representation of the structures is no longer important, as all that is needed in order to compare two structures is to compare their associated density maps.

One of the most commonly used methods for grey-scale image comparison is the root-mean-squared error (RMS) calculation [12]. The RMS method is simple to implement and processing even large images is nearly instantaneous.

In order to compare two grey-scale images, a measure of their dissimilarity, the root-mean-squared error, is calculated. Let f and g be two grey-scale images. The root-mean-squared error is then given by:

$$RMS(f, g) = \sqrt{\frac{1}{n(X)} \sum_{x \in X} (f(x) - g(x))^2}, \tag{4.1}$$

where $n(X)$ is the number of pixels in an image X and $f(x)$ is a single pixel in the image. The method is based on examining the intensity of each pixel in the first image and comparing it to the corresponding pixel in the second image.

The genetic algorithm for SwarmArchitect utilises the RMS measure by comparing each of the three density maps associated with some generated architecture, with the density maps of some hand crafted architecture. This, then, gives us an objective measure of the dissimilarity between the two pieces of architecture, which we can then use as a base for the fitness score of a rule set as follows:

$$\text{fitness} = \left(\frac{RMS(f_F, g_F) + RMS(f_S, g_S) + RMS(f_T, g_T)}{3} + 1 \right)^{-1}, \tag{4.2}$$

where f_F, f_S and f_T are the front, side and top density maps of the generated architecture and g_F, g_S and g_T are the front, side and top density maps of the hand crafted architecture. The fitness is thus a number between 0 and 1. The highest possible fitness, 1, can only be achieved in the case of a perfect match between the density maps where the RMS would be 0.

The RMS method for grey-scale image comparison has some limitations, however, since it matches each pixel in one image to the corresponding pixel in the next. The biggest concern for the purposes of SwarmArchitect is the fact the RMS is unable to recognize two pieces of identical architecture if they are merely placed in different locations. In this case the RMS algorithm can produce a rather large value for the dissimilarity. In order to alleviate this problem we can assist the SwarmArchitect algorithm in positioning construction in the right place, by strategically placing one or more corner stones from with building can start.

Although other more accurate and specialised image comparison methods exist, such as the Δ_g measure developed by Wilson et al. [12], the RMS approach is forms the basis for the fitness function for SwarmArchitect, particularly because of its low computational complexity and ease of implementation.

4.2.2 Performance

In implementing the genetic algorithm it is important to consider certain performance concerns. In an attempt to speed up the evolutionary process, one could choose to increase the probability of mutation, so that each generation can potentially take larger steps. However, such an approach carries with it the risk of uncontrollable evolution where good solutions may be ruined by large mutations. The better choice is to attempt to make as many small steps as possible, starting from a large initial population.

However, the fitness function relies on running the full SwarmArchitect algorithm to generate one instance of architecture for every rule set in population. This is the biggest drawback to this fitness function, since it limits the speed with which we can evolve good rule sets.

One possible approach to achieving faster performance of the genetic algorithm is to limit the SwarmArchitect simulation runs to a relatively small number of iterations so as to make fitness evaluation quicker. This also has the added benefit of rewarding rule sets that result in fast construction, since they will be more likely have produced a sufficiently large construction before the maximum number of iterations is reached.

As another way to improve performance, the number of rules in each rule set can be dramatically increased. SwarmArchitect uses an efficient method for matching a sensed configuration against rules, and adding several thousand rules will not increase the running time by any significant amount. Therefore, it is possible to search a much larger portion of the search space in every generation by increasing the number of rules in a rule set. The number of rules in a rule set may be decreased each generation, in order to favour wild exploration during the early phases and gradually shift the focus towards smaller refinement in later generations.

As a final performance improving measure, the search space can be reduced in the early generations by only producing rules that ignore the top layer of the agents' sight range. This will allow the genetic algorithm to focus on producing rules that can build the 'floor plan' of the architecture before considering rules that build upwards.

4.3 Selection and Reproduction

This section examines the details of the selection and reproduction methods implemented in the genetic algorithm for SwarmArchitect. We will discuss the selection strategy and review the constituent parts of a SwarmArchitect rule, examining the way rules are encoded as a genome in this implementation.

4.3.1 Selection

Many different methods and combinations of methods exist for selection of individuals in genetic algorithms. For the purposes of this work, we opt for a simple approach that may be implemented fairly quickly. The genetic algorithm for SwarmArchitect uses basic roulette wheel selection combined with elitist selection of the first two genomes. At the beginning of each new generation, the two genomes with the highest fitness are chosen immediately for reproduction through elitist selection, before roulette wheel selection is applied to chose the parents of the rest of the generation.

Roulette wheel selection provides a simple means of selecting parent pairs, and elitist selection of the first pair ensures that we never risk discarding the best solution during a selection pass.

4.3.2 Population and Encoding

The purpose of the genetic algorithm is to evolve a rule set which agents can act upon to produce architecture that is similar to the hand crafted architecture we use for comparison. A population is therefore a number of different rule sets. Each rule set contains a number of building rules. A single rule is made up of the following parts:

- One anterior rule for matching against a world configuration.
  The anterior is encoded as a string of 27 bits, one bit for each of the 27 spaces in an agent's perception range. A bit set to 1 signifies the presence of a building block in that space.
- Two or more posterior rules, designating the placement of a block
  Each posterior rule is encoded as a string of 9 bits, designating the coordinates for placing a block, relative to the agent's position. This encoding is only used by the agents, however, as the genetic algorithm does not mutate the bits directly, but rather operates on a list of valid positions.
- For each posterior i; front, side and top density thresholds D_{Fi}, D_{Si} and D_{Ti}.
  These are encoded as strings of 8 bits when the genetic algorithm operates on them, and normalised to a range of [-1:1] during run-time.

Note that the definition of the SwarmArchitect building rules allow for several possible block placements in response to a local configuration, but throughout this report, we have assumed only two posteriors in each rule.

4.3.3 Genetic Variation

In genetic algorithms an entire genome is commonly encoded as a single string of bits, and the genetic variation operations such as crossover and mutation are applied to this single string. However, rather than serializing each ruleset as one very long bit-string and attempting to recover from the invalid rules that would invariably result from indiscriminate genetic variation, we maintain the encoding of rules used in the simulation and ensure validity of the resulting rule sets as we apply the variation operations.

A rule set is implemented as a vector of rule objects. When we apply crossover on a pair of parent rule sets we select a random number as an index into this

vector. Every rule that occurs after this position in the vector is exchanged with the other parent rule set. This ensures that crossover produces new genomes with every rule intact.

Mutation requires careful thought in order to maintain the constraints on SwarmArchitect rules. As mentioned above, a rule has three different types of data and each of these need special consideration when applying mutation.

First, the 27 bit anterior rule is mutated simply by visiting each bit and flipping it with a small probability. The integrity of the resulting anterior is ensured by checking that it is not already present in the genome of the new individual. If this is found to be the case, the anterior is copied over without mutation.

Second, from the mutated anterior, a list of valid building positions is constructed. Each posterior rule is mutated with a probability of $6 \times$ the mutation parameter by chosing a new building position randomly from this list. This also occurs if a posterior is made invalid by the mutation of the anterior.

Lastly, the density thresholds are mutated in the same way as the anterior, by simply flipping each bit with a small probability and then normalising the resulting integer to the range of [-1:1].

4.4 Algorithm Listing

Algorithm 2 provides a high-level description of the genetic algorithm for SwarmArchitect.

When the genetic algorithm begins, a population of random rule sets is generated. These form the initial population.

The main loop runs for a specified maximum number of generations or until the algorithm is terminated. For each generation, the algorithm looks at every individual in turn.

In the simulation loop, a single individual – a rule set – is fed to the SwarmArchitect algorithm which is allowed to run for a specified number of iterations. When SwarmArchitect returns, it will have generated a structure and from those a set of density maps which are used in the fitness evaluation of the rule set. The genetic algorithm runs the fitness function described in equation 4.2 to determine the fitness of the rule set and saves the result in a temporary data structure.

Once every rule set has had it's fitness determined the genetic algorithm moves on to the selection and variation stage where a number of parent pairs

must be selected. First, the two highest scoring rule sets are chosen through elitist selection. Then, the remainder of the parent pairs are chosen through normal roulette wheel selection.

Finally, each parent pair produces two offspring for the next generation, through crossover and mutation as described in section 4.3.3.

Algorithm 2 Genetic Algorithm for SwarmArchitect

```
/* Initialisation */ Generate population of random rule sets
/* Main loop */
for 0 to max_generations do
  /* Simulation loop */
  for each rule set R in population do
    Run SwarmArchitect for max_iterations with Rule Set R
    Determine fitness F from resulting density maps using equation 4.2
    Save (R, F) pair
  end for
  /* Selection and variation stage */
  Save parent pair (R1, R2) of rules with highest fitness
  for 0 to max_population/2 − 2 do
    Save parent pair (R1, R2) of rules with Roulette Wheel selection based
    on fitness.
  end for
  Empty population
  for each saved pair (R1, R2) do
    Apply crossover and mutation as described in section 4.3.3
    Save children to population
  end for
end for
```

4.5 Summary

The genetic algorithm developed for use with SwarmArchitect may provide a solution to the difficult problem of producing a rule set that can construct architecture with the subjective features that we desire. It does so using a novel approach based on aggregate views, or *density maps*, which provide three seperate 2-dimensional 'x-ray' photographs of buildings. These maps may then be compared to maps of hand crafted buildings in order to gain an objective measure of the dissimilarity, which is used as the basis for the fitness score.

There are, however, certain problematic areas. First, the RMS method of grey-scale image comparions has shortcomings that make it unable to cope with, for instance, situations where a building very similar to the desired result has been constructed in a different position from the hand crafted architecture. Second, the fact remains that in order to evaluate the fitness of a rule set, it is necessary to first run a simulation on the set and construct some architecture. This architecture is then evaluated by the fitness function rather than the rules that were used in it's construction. Due to the stochastic nature of the simulation, the resulting architecture will differ from one simulation to the next. This lack of a 1:1 correspondence between a rule set and it's fitness value means that certain favourable genetic properties may be lost due to an 'unlucky' run of the simulation.

Chapter 5

Test & Visualisation Environment

When working with swarm intelligence, part of the process is discovering the unpredictable, emergent patterns that arise as a consequence of the myriad of simple interactions. Therefore, it became necessary to develop a special purpose tool for SwarmArchitect to allow not only visualisation of pertinent data, but also online experimentation with various parameter adjustments. The Test & Visualisation Environment was developed for SwarmArchitect with the purpose of aiding in the design of the algorithm and represents a significant investment in development time.

This chapter examines the features of the Test & Visualisation Environment and highlights some of the different ways in which the tool supports the development of and experimentation with the SwarmArchitect algorithm.

The first section describes the different options for visualising data during the running of SwarmArchitect, and the next section goes into detail about the interactive features, such as the ability to build structures and adjust parameters on the fly.

5.1 Data Visualisation

The basic requirement of the Test & Visualisation Environment was to provide a view of the three-dimensional world that the SwarmArchitect agents inhabit. The main portion of the screen is dedicated to this 3D view. It is possible to move about freely in the environment, in order to inspect the building process and behavior of agents from any angle.

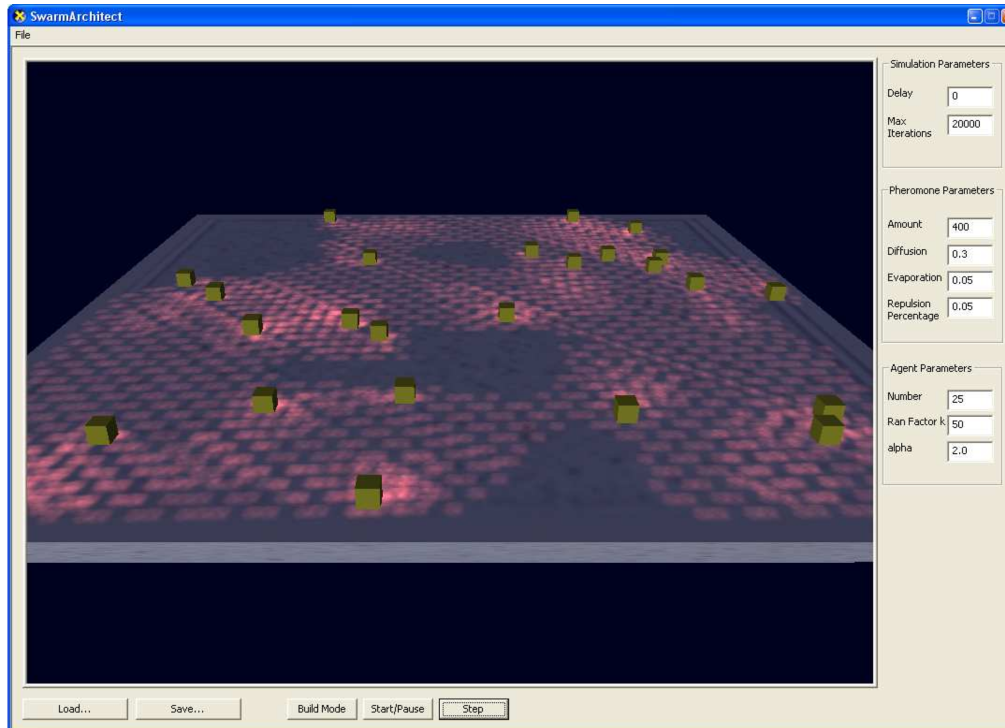

Figure 5.1: Main view of the Test & Visualisation Environment for SwarmArchitect. A number of agents moving about in an empty world with repulsion pheromone visible.

One very important feature is the ability to view the presence and concentrations of pheromone desposits in the world as the simulation progresses. It is difficult, if not impossible, to predict the complex interplay between the many parameters that control the distribution of pheromone and agents' reaction towards it. The Test & Visualisation Environment makes the information about these emerging patterns available at a glance, and the effects of parameters becomes immediately visible. Figure 5.1 shows an instance of a number of agents moving about in an empty world space, with their trails of repulsion pheromone visible at ground level. The brightness of pheromone colours is directly proportional to the concentration present in the world. By making pheromone concentrations immediately visible, the Test & Visualisation Environment supports the researcher in experimenting with and setting these parameters to their optimal values for the task at hand.

Through shortcut keys, every visible component in the three-dimensional view may be switched on or off. Ground, building blocks, agents and both

kinds of pheromone may be hidden in order to provide unobstructed views of specific aspects of the simulation.

5.2 Interactivity

The Test & Visualisation Environment provides the means to interact with the simulation in a number of ways.

Almost every parameter of the simulation may be set up and changed through controls available on screen, allowing for easy experimentation.

Under most circumstances, the SwarmArchitect simulations run so quickly that it becomes impossible to perceive the actions of individual agents. Therefore, the Test & Visualisation Environment implements controls that make it possible to start and pause the simulation as well as step through it, a single iteration at a time, or even specify a delay between each iteration.

In order to easily set up experiments and test agents' reactions to specific configurations of blocks, the Test & Visualisation Environment may be put into build mode, where the researcher can place building blocks, pheromone or both in the environment while the simulation is paused. Once a structure has been built, the configuration of the world may be saved in a file and used later as the basis for further testing or as the comparison structure for the genetic algorithm.

5.3 Summary

The Test & Visualisation Environment developed for SwarmArchitect is a vital tool in experimenting with the algorithm. As with most swarm intelligence applications, the unpredictable and complex interplay between many simple components makes a highly experimental approach to setting up parameters and developing the algorithms a necessity. Implementing a robust and flexible visualisation and experimentation tool was a vital first step in the work presented in this report.

Chapter 6

Experimental Results

This chapter presents the results of experiments performed with SwarmArchitect and the genetic algorithm. In the first section we examine experiments that highlight and explain some of the details of the SwarmArchitect algorithm and the expressiveness of its language. In the following section we will present results from the genetic algorithm, showing the progression of the agent genome over time and the resulting structures after training.

All experiments were carried out on a 2.5 GHz AMD Athlon64 processor.

6.1 SwarmArchitect

When examining the SwarmArchitect algorithm, perhaps the most important aspect to show is the expressiveness of the language. If it can be shown that the algorithm is capable of expressing the kinds of features we are interested in, then it follows that given enough time, and a sensible evolution strategy, a genetic algorithm will be able to evolve a set of rules that meets our requirements. We will start this section with some small scale experiments designed to demonstrate these points.

6.1.1 Cornering Problem

Straight walls and right angled corners are very important and ubiquitous features of human architecture, so therefore, the ability to express these together in the same rule set becomes a primary concern. Breaking the problem down further, if we have a rule that allows agents to place consecutive blocks,

forming a wall, then it is necessary to have a mechanism in place that will allow agents to stop following this rule, and take a different action. This is the reason that SwarmArchitect uses branching rules, where a single triggering configuration may result in different building actions. We will conduct a number of experiments related to the cornering problem in order to show the expressiveness of the SwarmArchitect language.

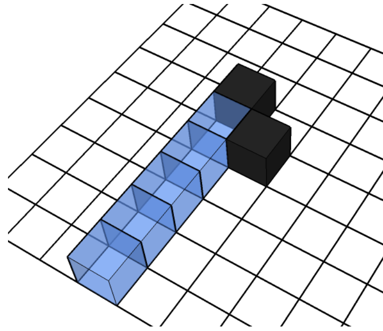

Figure 6.1: An illustration of the cornering problem. The darkened blocks show the block placements under consideration.

Figure 6.1 shows the setup of this problem. An agent has placed several bricks in a row, according to a rule that triggers a building action once a specific configuration is encountered. If we did not have branching rules, then the wall would invariably be built all the way the edge of the world, since agents do not have a choice to stop performing the building action.

With branching rules affected by density map readings, we are able to make agents stop building straight ahead, and instead place a block to the side, thus forming a corner. Figure 6.3 shows how the probability of continuing the wall gradually falls as a function of the block density. When the probability of building a corner becomes higher, the next time an agent encounters the triggering configuration, the other building action will be carried out.

Experiment 1 - Cornering at a specific density

The first experiment related to the cornering problem aims to show that we can create a rule which will recursively place blocks in a straight line until a certain density has been achieved. After this, the rule must place a block off to the side in order to start a corner. It is fairly straight-forward to create the anterior and posteriors of such a rule as figure 6.2 shows.

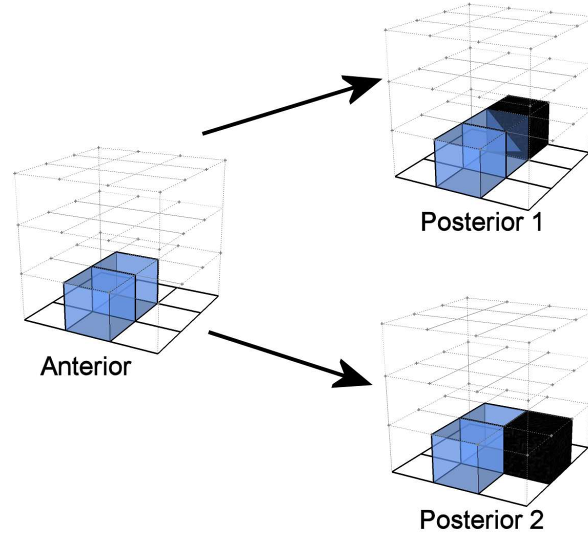

Figure 6.2: An illustration of the SwarmArchitect rule for the cornering problem. The darkened blocks show the two possible block placements.

For this experiment we wish to build a corner once the wall is 10 squares long.

In order to demonstrate the exact working of the choice mechanism we refer once again to Equation 3.4 repeated below. Recall that A is the set of all possible actions that an agent may choose from when encountering a specific triggering configuration. The probability p_{a_i} that an agent choses action $a_i \in A$ is given by:

$$p_{a_i} = \frac{\eta_{Fi} + \eta_{Si} + \eta_{Ti}}{\sum_{a_j \in A} \eta_{Fj} + \eta_{Sj} + \eta_{Tj}},$$

where η_{Fi}, η_{Si}, and η_{Ti} are the front, side, and top desirability values for the cell being considered for block placement by action a_i.

Let $M = \{F, S, T\}$ be the set of density maps; front, side and top. The desirability value η_{mi} for action a_i and map $m \in M$ is given by:

$$\eta_{mi} = \begin{cases} \left(\frac{D_{mi}}{\delta_{mi}+D_{mi}}\right)^2, & \text{if } \delta_{mi} > 0 \\ \left(\frac{|\delta_{mi}|}{|\delta_{mi}|+D_{mi}}\right)^2, & \text{if } \delta_{mi} < 0 \\ 0, & \text{if } \delta_{mi} = 0 \end{cases},$$

where δ_{mi} is the density threshold and D_{mi} is the density value read from density map m for the cell being considered for block placement by action a_i.

For this experiment, we shall set the following density thresholds:

$$\begin{array}{lll} \delta_{F1} = -0.24; & \delta_{S1} = 0; & \delta_{T1} = 0; \\ \delta_{F2} = 0; & \delta_{S2} = 0.1; & \delta_{T2} = 0; \end{array}$$

δ_{F1} is set to -0.24 so that the desirability of Posterior 1 drops as the front density D_{F1} increases, ie. as the wall becomes longer.

By setting most of the thresholds to 0, we can disregard their corresponding desirabilities. Thus, when considering Posterior 1, we are only concerned with the density value read from the front density map, and for Posterior 2 we are only concerned with the side density. Furthermore, it is immediately apparent that the side density for Posterior 2 remains constant in this example, as in this position we will always be influenced by a single piece of the wall already in place.

Also, with these settings, we can simply observe the desirabilities η_{F1} and η_{S2}. As long as $\eta_{F1} > \eta_{S2}$ we extend the wall.

Figure 6.3 shows the progression of the desirabilities in this example. Recall that the density value is defined as $\frac{\text{number of blocks}}{\text{world dimension}}$. The straight dashed line is the desirability of Posterior 2, which remains constant. We see that as the front density D_{F1} for Posterior 1 reads 0.45 – 9 blocks divided by world dimension of 20 – the desirability of Posterior 1 is still greater than that of Posterior 2. When the 10th block is placed – when $D_{F1} = 0.5$ – the desirability of Posterior 1 drops below that of Posterior 2, meaning that the wall will no longer be extended ahead, but instead a corner will be built.

Figure 6.4 shows the progression of the simulation in SwarmArchitect. We start off with two blocks placed in the world, from which the agents will begin building. Simulation continues until the wall has been extended to a length of 10 squares and a corner has been built.

This experiment demonstrates the ability of the SwarmArchitect rules to allow recursive building of a wall until the desired length has been achieved, after which the agents place a block off to the side, producing a corner.

Experiment 2 - Cornering early to meet perpendicular wall

This example uses the same parameters as the one above. In this case, however, a perpendicular wall is placed in the initial setup. Figure 6.5 shows the experiment.

In this example, using the same rule as the one in example 1, we produce a corner before the wall has been extended to 10 in length. Instead, the

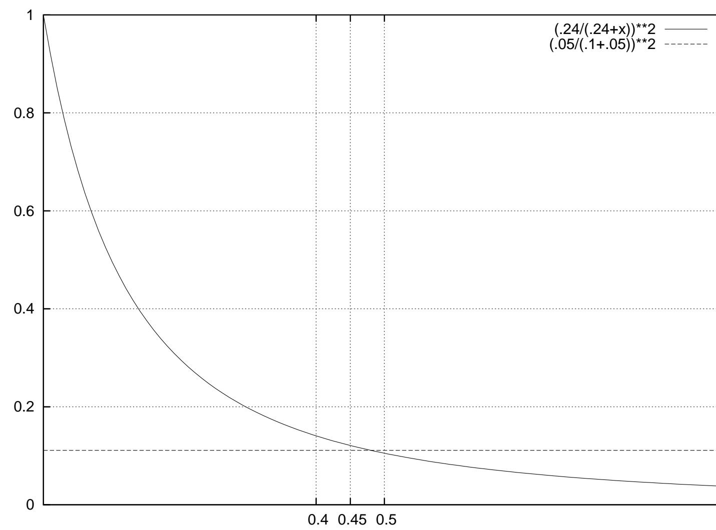

Figure 6.3: Experiment 1 – Graph showing the desirability of Posterior 1 as a function of its front density reading. Solid line: η_{F1}. Dashed line: η_{S2}

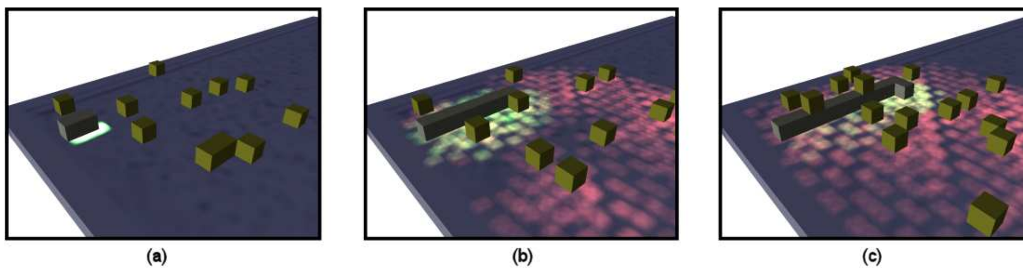

Figure 6.4: Experiment 1 – (a) Initial setup. (b) Building underway. (c) Wall has been extended to 10 squares and a corner has been built.

presence of the perpendicular wall influences the desirability of Posterior 2, resulting in a corner being built which lines up with this wall.

6.1.2 Building Performance

Experiment 3 - Pheromone

In this experiment we wish to show the effects of pheromone. Pheromone plays an important part in guiding agents towards sites that are ready for building. In this experiment, agents will be tasked with building a fairly large

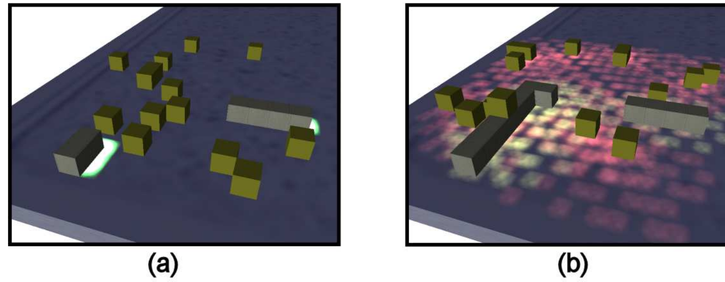

Figure 6.5: Experiment 2 – (a) Initial setup. (b) A corner has been built in order to line up with the perpendicular wall

construction. The simple rule used in this experiment has been crafted so as to ensure that the there can be no variation in the finished construction from one simulation run to the next. We will run the simulation several times, with pheromone deposits both on and off and record the number of iterations needed before the construction is complete. When no pheromone is present in the world, the movement decisions of agents will be entirely random.

The world is set to $50 \times 50 \times 50$ and the number of agents is 40. We will use the same rule as in the previous two experiments, except with thresholds altered so that corners are never built. The world is initially setup with 10 different sites where building may start from. Each of these sites must be extended all the way to the end of the world before the task is considered to be finished.

Figure 6.6: Experiment 3 – The world is setup with a number of building sites.

The simulation was run 10 times both with and without pheromone. The table below shows the number of iterations needed to finish construction in each simulation run.

Run #	1	2	3	4	5	6	7	8	9	10	Avg.
Pheromone	658	678	644	731	407	658	787	1008	610	890	**707.1**
No pheromone	3279	4634	6235	5785	6357	4283	4569	4370	5213	3755	**4848**

These results show quite clearly that the pheromone strategy implemented in SwarmArchitect has a large impact on the effectiveness of the builder agents. When pheromone is switched off and agents move completely at random, the number of iterations needed is an order of magnitude greater than when pheromone is being utilised.

Repulsion pheromone ensures that agents search a wide area for suitable building sites and building pheromone ensures that once a building site has been found, additional help is recruited to the area, resulting in sustained building activity.

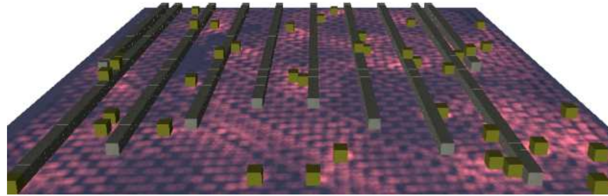

Figure 6.7: Experiment 3 – All building sites have been expanded into walls that run to the edge of the world.

6.2 Genetic Algorithm

As is evident from the discussion above, the branching rules of the Swarm-Architect algorithm are not easily constructed by hand. This complexity becomes even more pronounced when several rules are required to work together to form larger structures. The exact settings of the density thresholds is of vital concern if the set of rules are to produce satisfactory results.

In this section we detail the results of experiments conducted using the genetic algorithm for SwarmArchitect in order to develop individual rules and rule sets that are capable of producing the kinds of architectural features we wish to express.

6.2.1 Experiments Setup

It is common for genetic algorithms to start with very large initial random populations, in order to cover the entire search space. However, because of

performance concerns in regards to the fitness function, we will limit the population to 20 randomly generated rule sets.

As discussed in chapter 4, the major downside the genetic algorithm for SwarmArchitect is the fact that in order to evaluate the fitness of an individual in the population – a set of rules – we must run the simulation with the rule set in question and build a structure which then forms the basis for the evaluation. The process of performing several simulation runs for every generation can be very time consuming, and as such not well suited for gaining results with a genetic algorithm in a timely fashion. Therefore, every effort must be taken to ensure that simulation runs complete as quickly as possible, and this necessitates certain compromises.

The running time of the SwarmArchitect algorithm is mostly dependent on the number of agents. At every iteration, every agent must sense its environment, look for a match in its library of rules, decide upon a building action if relevant, and then decide which direction to move. In order to ensure fast running times of every simulation run, we must limit the number of agents as well as the number of iterations. This, however, means that we must take steps to ensure that we are able to build our structures with few agents and before we have used all of our allotted iterations.

For these experiments, we will limit ourselves to testing small scale architectural features. This means that agents can complete building in fewer iterations, especially if we also limit the total area of the simulation. If we limit the area to just cover the architectural features we are trying to develop, agents spend less time moving in empty areas and are able to find building sites quicker.

Using the RMS image comparison algorithm as the basis of the fitness function, we must contend with the fact that two identical pieces of architecture placed in different positions can show a large root-mean-squared error, since the algorithm only compares each pixel in one image to the corresponding pixel in the other. However, strategic placement of one or more cornerstones can remedy this shortcoming. A cornerstone can be a small structure cut out of the hand crafted architecture that we are using for comparison in the fitness function.

An ideal fitness function will reward every small step in the right direction, but for many problems, such a fitness function does not exist. This is certainly true for the problem of generating rules for SwarmArchitect, and this point brings up another potential problem in using image comparison as the basis for the fitness function. During the early stages, there is a high risk that the evolutionary process could stall because rule sets which do not yield

any construction at all would recieve a higher fitness score than sets which actually do place down building blocks, albeit in the wrong positions.

At the very least, a rule set should be able to match a configuration of the seed structure and be able to build something. Rule sets which do not meet this elementary requirement of being able to start building from the seed structure can obviously not evolve in any reasonable way. The fitness score that they do receive from simply not building anything would not be indicative of the worth of their genetic properties. Since we do not wish to reward inactivity, this problem must be addressed. In the genetic algorithm for SwarmArchitect we will keep track of whether or not a rule set has managed to build anything at all, and if not, reduce its fitness score by a large percentage. The justification is that a rule set which has achieved it's fitness score through building rather than inaction is far more valuable, as the basic ability to build something from the seed structure is a step in the right direction. We will not discard the inactive rule sets completely, since they will still hold some value as a source of genetic variation.

The genetic algorithm parameters common for all experiments are set at:
Mutation factor: 0.05
. Crossover probability: 0.7
Fitness reduction for inactive sets: 80%

6.2.2 Experiment 1

The first experiment will demonstrate the basic function of the genetic algorithm. The objective is for the genetic algorithm to produce a single corner-building rule, much like the one described above. The rule must, from a seed structure of two blocks, extend a wall until it is 5 squares long, and then make a corner. Figure 6.8 shows the seed structure and the comparison structure for this experiment.

This experiment uses fairly tight parameters, with 8 agents, world dimensions of $12 \times 12 \times 12$ and maximum iterations of 200. Moreover, the top layer of the rules is not available.

Figure 6.9 shows the progression of the experiment. After 739 generations, the genetic algorithm had developed a rule which yields a perfect fit of our comparison structure, thereby earning the maximum fitness score of 1. This was achieved in approximately 25 minutes. Visual observation of the early stages of the process revealed that after the first individual discovered an anterior matching the seed structure, this highly desirable genetic property quickly spread throughout the population.

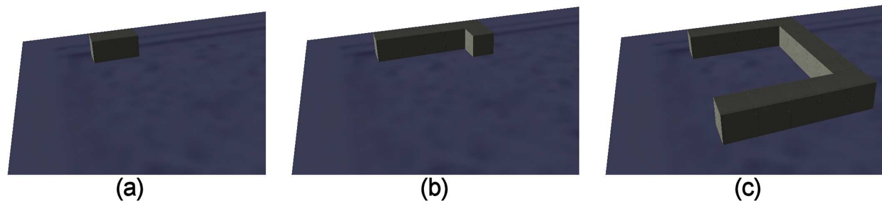

Figure 6.8: GA Experiments 1 and 2 – (a) Seed structure for both experiments. (b) Comparison structure for experiment 1. (c) Comparison structure for experiment 2.

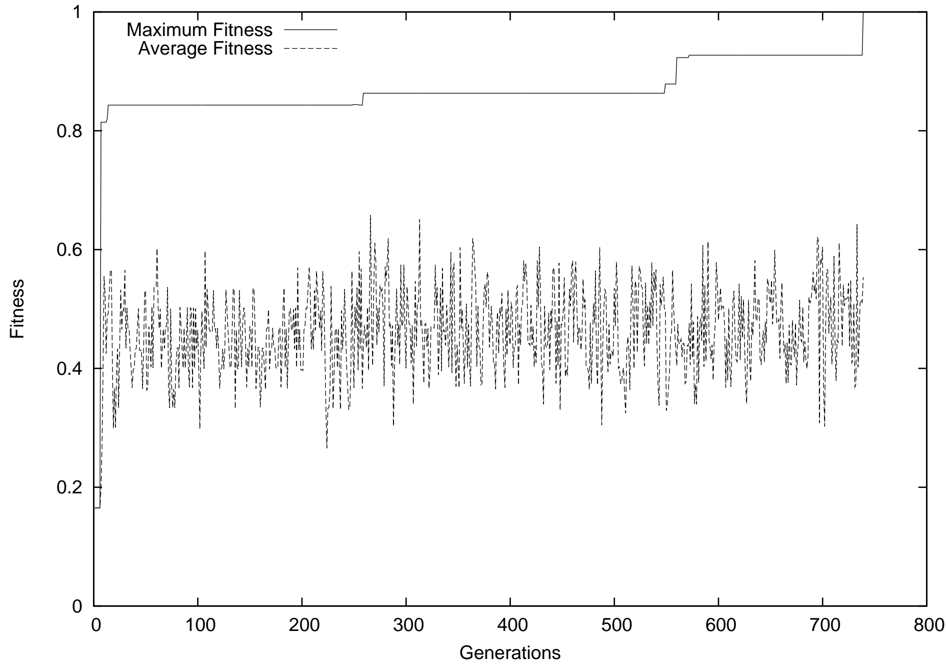

Figure 6.9: GA Experiment 1 – The graph shows the gradual improvement of the maximum fitness over time.

Figure 6.10 is an illustration of the rule developed by the genetic algorithm in this experiment. The density thresholds of the rule were as follows:

$$\delta_{F1} = -0.345; \quad \delta_{S1} = 0.125; \quad \delta_{T1} = -0.973;$$
$$\delta_{F2} = -0.267; \quad \delta_{S2} = -0.533; \quad \delta_{T2} = -0.839;$$

Note that most of the thresholds in this rule are irrelevant and could just as well be set to 0 as in the hand crafted rule in section 6.1.1, since the related

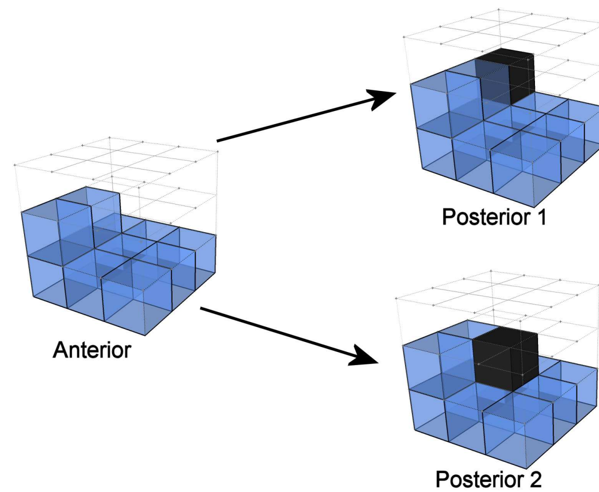

Figure 6.10: GA Experiment 1 – An illustration of the rule developed by the genetic algorithm.

density readings remain constant in this simple setup.

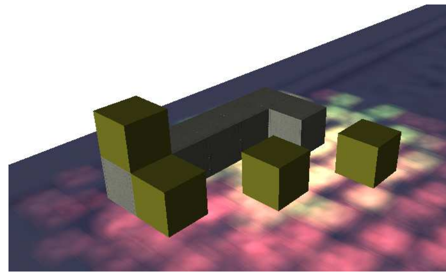

Figure 6.11: GA Experiment 1 – Four agents use the evolved rule to build an exact duplicate of the comparison structure.

This experiment shows that the genetic algorithm for SwarmArchitect is capable of producing a rule that can build a simple structure to our specifications. The resulting rule, when employed by SwarmArchitect agents, will produce the exact same structure as the one used by the fitness function to calculate the fitness value, as is shown in figure 6.11.

6.2.3 Experiment 2

Experiment 2 is designed to show how the genetic algorithm performs when we wish to evolve several rules which work together. In this experiment, the genetic algorithm must produce a set of rules which are able to build an open square. This will test the ability of the genetic algorithm to develop a set of rules that achieve a high fitness in a large and complex problem space.

Figure 6.8 shows the seed structure and the comparison structure for this experiment.

In this experiment, the world has the dimensions $12 \times 12 \times 12$ and we will use 8 agents who are allowed 200 iterations to do their work. Furthermore, the rule set which achieved the perfect fitness score in Experiment 1 is included in the initial population.

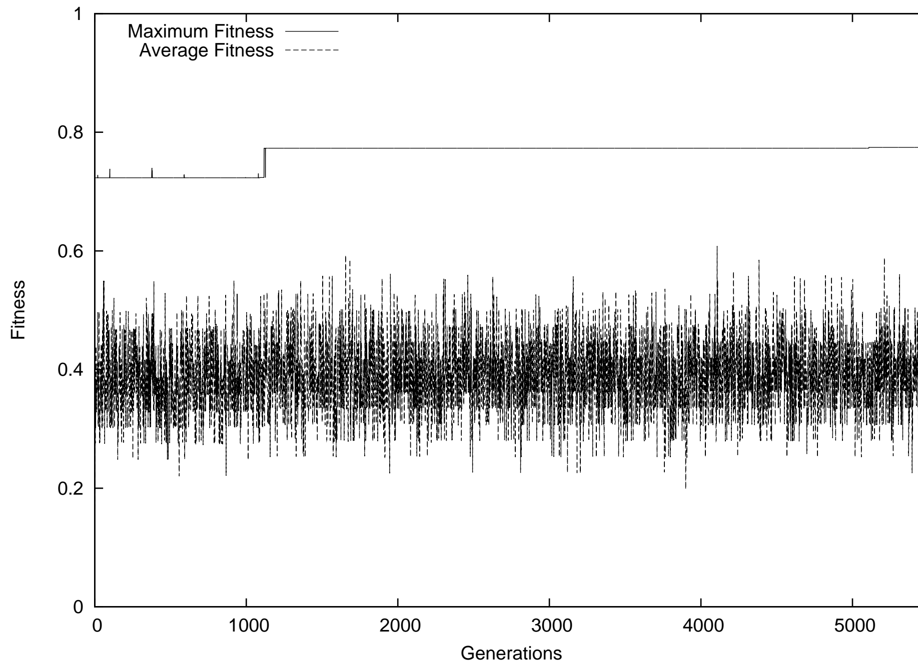

Figure 6.12: GA Experiment 2 – The graph shows the progression of the maximum and average fitness over time.

The experiment was terminated after about 3 hours 20 minutes at 6000 generations when it became apparent that the algorithm would not significantly improve the fitness in any reasonable time frame.

6.2.4 Experiment 3

Experiment 3 will test the performance of the genetic algorithm with a much larger and more complex structure for comparison. Figure 6.13 shows the seed structure and figure 6.14 shows the comparison structure for this experiment.

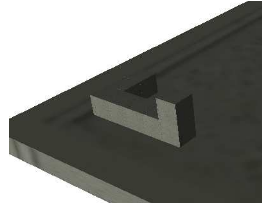

Figure 6.13: GA Experiment 3 – The seed structure for experiment 3.

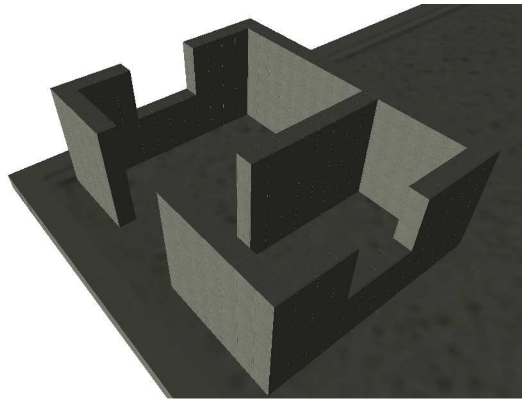

Figure 6.14: GA Experiment 3 – The target structure for experiment 3.

Since the size of both the world space and the target structure is so much greater than in the previous experiments, it becomes necessary to increase the number of agents and the number of iterations in order to ensure that there are enough resources to build such a large structure. This comes at the cost of slower progression through the generations.

The world dimensions are $25 \times 25 \times 25$, the number of agents is 12, and maximum iterations is set at 500.

The experiment was terminated after about 3 hours at 1500 generations when it became apparent that the algorithm would not significantly improve the fitness in any reasonable time frame.

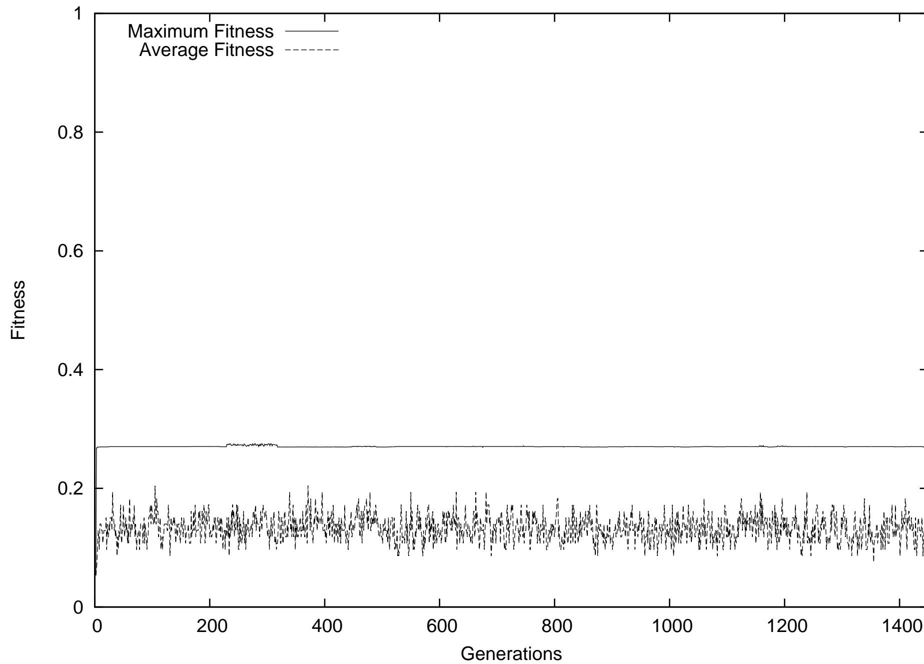

Figure 6.15: GA Experiment 3 – The graph shows the progression of the maximum and average fitness over time.

6.3 Summary

6.3.1 SwarmArchitect

The experiments conducted with the SwarmArchitect algorithm demonstrate the capability of the branching rules and the performance increasing effects of the pheromone system.

The experiments on branching rules demonstrate how an agent can make fairly complex decisions on where to place building blocks, without the need for very complex artificial intelligence. The encoding of the branching rules coupled with the constantly maintained density maps allow such decisions to be taken practically instantaneously as the result of strictly numerical computation.

The third experiment shows a dramatic improvement in the time it takes agents to find building sites and complete construction when the pheromone system is put to use. The results demonstrate the efficiency and adaptability of this simple communication and cooperation strategy based upon the

principles of stigmergy where information is shared through the environment.

6.3.2 Genetic Algorithm

In the first experiment, the genetic algorithm showed a good result, finding a perfect rule to fit the comparison structure. The other two experiments were unsuccessful in developing any useful rule sets in the limited time allowed.

As the graphs show, in all three experiments, noticeable improvements in fitness did not happen continually in small increments, but rather in larger leaps every so often. These results speak to the highly complex nature of the problem space. Small steps in the right direction are very difficult to identify and reward adequately, and as such, improvements must happen in greater leaps before they are able to affect the evolutionary process. This makes the genetic algorithm much more dependent on raw computing power to cover more of the search space in a shorter amount of time in order to find the larger improvements that are able to be picked up by the fitness function.

Moreover, the implementation described in this work is very basic. It does not make use of the large body of research describing methods of improving the performance of genetic algorithms with a less than ideal fitness function. In experiments 2 and 3, the genetic algorithm became stuck in local optima, from which it would have been difficult and time consuming to escape. Smoothing out the problem landscape by adjusting the fitness function would undoubtedly help prevent this situation, as would a more focused examination into more advanced selection and reproduction strategies to suit this particular problem.

The success of the first experiment, however, is an indication that with further efforts to reduce the noise and granularity of the fitness function, as well as general performance improvements, an approach based on image comparison could eventually produce more complex sets of build rules for Swarm-Architect.

Chapter 7

Conclusion

7.1 SwarmArchitect

The design and implementation of SwarmArchitect encompasses numerous achievements. First, recall that in the scientific process employed in swarm intelligence we progress from observation of natural behavior, to modelling said behavior and, finally, to practical application of the knowledge gained from the model. SwarmArchitect demonstrates a way of progressing collective building from the modelling stage into the application stage.

Moreover, the combination of short range direct perception and large scale indirect perception in the form of density maps is a novel approach to the problem of allowing agents to construct large scale architectural features. Implementations that rely solely on gradients of pheromone and template functions to guide large scale construction commonly produce architecture with an organic and natural appearance. The fact that the SwarmArchitect approach allows large scale features to be constructed without an organic appearance is owing to the combination of quantitative and qualitative stigmergy which both influence building decisions. We are able to qualitatively match a specific local configuration to specific building actions, and let the choice between them depend on the quantitative influence of global building densities.

Our use of density maps for large scale perception is highly efficient. When agents require large scale information upon which to base their decisions, they have instantaneous access to an always up-to-date aggregate data view. Keeping maps up-to-date is merely a matter of incrementing a counter when a building block is placed.

The use of branching rules influenced by density map values remains true to the ideals of swarm intelligence as agents decide upon a course of action by means of fast mathematical computation, rather than complex symbolic AI. Also, the presence of this slightly more complex decision making has no adverse affects on the task of searching the environment for triggering configurations as it only comes into play after a triggering configuration has been found.

7.2 Genetic Algorithm

Experiments showed disappointing results from the genetic algorithm. When attempting to evolve more complex rule sets, the algorithm would quickly become stuck in a local optimal and move very little after that.

Designing a completely algorithmic fitness function for this problem space is obviously very challenging. At best, one can hope to develop a general heuristic.

While using RMS image comparison to obtain an objective measure of the similarities of two three-dimensional structures is a valid approach, it clearly cannot stand alone as the sole measure of fitness in anything but the simplest cases. This objective similarity measure must be coupled with other indicators of fitness in order to reduce the granularity of the fitness function and better reward small steps in the right direction, even when these steps result in a short term drop in similarity between the structures being compared.

Of course, it is also possible that a more advanced evolution strategy could help mitigate the limitations of our fitness function, however the development of such was deemed outside the scope of this work.

7.3 Summary of Contributions

The major contributions of this work are contained within the SwarmArchitect algorithm. SwarmArchitect contributes with the following:

- A collective building algorithm incorporating methods from other swarm intelligence applications.
- A demonstration of a way to combine both quantitative and qualitative stigmergy in one collective building algorithm.

- A mechanism for providing instant large scale environmental information to simple agents with very small direct perception ranges.
- A powerful language in the form of branching rules, capable of expressing a variety of large scale features indicative of human-like architecture.

Minor contributions include the following:

- A method for comparing two three-dimensional structures based on RMS image comparison.
- A graphical test and visualisation environment allowing for experimentation and interaction with the running algorithm.

7.4 Future Work

The work in this report spans very wide, and therefore a number of interesting aspects could benefit from a much more in-depth treatment.

7.4.1 SwarmArchitect

The branching rule innovation of SwarmArchitect is a powerful feature which should be explored more thoroughly. This feature could benefit from further experimentation to uncover both strengths and weaknesses of the approach.

In this work we have assumed only 2 branches to every rule, however, the definition allows for an arbitrary number of possible actions and as such, it would no doubt be interesting to discover whether the benefits gained from adding more branches would outweigh the added complexity concerns.

Additionally, the current design does not allow agents to decide to take no action when they encounter a triggering configuration. This means that agents are only able to stop building if they, through building actions, eliminate all triggering configurations from the world. The impact of allowing agents to not perform any action is certainly worthy of further study.

7.4.2 Genetic Algorithm

The one aspect of this work which is in most need of additional work is the genetic algorithm. Although the method of comparing two structures based on their density maps provides us with an objective measure of the dissimilarity between them, the fitness function must take other objective features into account as well. A serious inquiry into which ways the fitness function could be improved would prove highly beneficial to the whole body of work within this report.

Furthermore, longer term experiments and general performance improvements to the genetic algorithm could also yield better results than the ones demonstrated here.

7.4.3 Test & Visualisation Environment

As a useful addition to further experimentation with SwarmArchitect, an extension to the Test & Visualisation Environment which would aid the researcher in hand crafting rules would prove very beneficial. In the current implementation, making rules for SwarmArchitect is an arduous task requiring lots of work with pen, paper and calculator in order to encode the various component parts of each rule. A graphical tool to quickly make up and inspect rules and inject them into a running simulation would greatly ease efforts to test and understand the interplay between several rules.

The current implementation of the Test & Visualisation Environment is very closely tied to the SwarmArchitect algorithm. Additional design and development could expand it into a general purpose architecture for collective building research. This could entail packaging the Test & Visualisation Environment as a library for inclusion in C++ programs and provide a standard interface to provide efficient access between the environment and the simulation.

Appendix A

Summary

Inspired primarily by insects, *swarm intelligence* is a method of constructing robust and adaptive decision making and optimisation systems of various sort by mimicing the behavior of naturally occurring swarm systems. Fundamental concepts in swarm intelligence include *self-organisation* and *stigmergy*. Stigmergy is the concept of agents communicating through their environment rather than directly between individuals. Swarm intelligence has been applied to a variety of different problem areas with good results, and in particular, the approach known as *Ant Colony Optimisation* has been the subject of extensive research.

Another field within swarm intelligence deals with collective construction of different kinds, inspired by nest building behavior in social insects. So far, research in this area has been focused mainly on simulating natural construction behaviors. This report details the development of a collective building algorithm which we dub SwarmArchitect, which is meant to explore the possibilities of applying the swarm intelligence approach to the construction of human-like architecture.

SwarmArchitect combines and adapts concepts and approaches from existing work within swarm intelligence and contributes some novel solutions. It serves as a demonstration of how the particular challenges in using collective building to achieve human-like structures may be overcome. Our solution makes use of both quantitative and qualitative stigmergy. A number of simple agents move about in a three-dimensional space, placing building blocks when they encounter a configuration that can be matched from their library of building rules. Agents' movement is influenced by two types of pheromone, one attractive type which is deposited along with building blocks and one repulsive type which is deposited whereever agents move.

A proposal for the use of a genetic algorithm is presented, along with a discussion of the challenges involved in applying this technique to SwarmArchitect. We propose the use of grey-scale image comparison algorithms in the implementation of the fitness function.

Experiments show how our proposed branching rule structure in combination with instantaneously and constantly updated aggregate views of global building densities can be used to make complex building decisions based on local and global stigmergic information. We also show the usefulness of our pheromone deposit strategy which dramatically increases efficiency of agents as a means of coordinating and distributing search and building efforts. Experiments made with the genetic algorithm show that the approach based on image comparison shows promise but requires much more focused development in order to be truely useful.

In the beginning stages of development it became necessary to create a graphical environment in order to view, control and interact with the SwarmArchitect algorithm. We describe the Test & Visualisation Environment for SwarmArchitect, a tool which aids development and experimentation on our algorithm by providing at-a-glance views of pertinent information and emerging patterns.

Bibliography

[1] E. Bonabeau, S. Guerin, D. Snyers, P. Kuntz, G. Theraulaz, and F. Cogne. Complex three-dimensional architectures grown by simple agents: An exploration with a genetic algorithm. *Evolutionary Computation*, 1998.

[2] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence - From Natural To Artificial Systems*. Oxford University Press, 1999.

[3] J. L. Deneubourg, S. Aron, S. Goss, and J. M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, 3:159–168, 1990.

[4] J.-L. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chretien. The dynamics of collective sorting: Robot-like ant and ant-like robot. In *In Proceedings of the First Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 356–365, 1991.

[5] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the travelling salesman problem. *Biosystems*, 43(2):73–81, July 1997.

[6] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[7] S. Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, 1993.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[9] Dan Ladley and Seth Bullock. Logistic constraints on 3d termite construction. In *ANTS Workshop*, pages 178–189, 2004.

[10] Guy Theraulaz and Eric Bonabeau. Coordination in dsitributed building. *Science*, 269:686–688, 1995.

[11] Guy Theraulaz and Eric Bonabeau. Modelling the collective building of complex architectures in social insects with lattice swarms. *Journal of Theoretical Biology*, 177:381–400, 1995.

[12] D. L. Wilson, A. J. Baddeley, and R. A. Owens. A new metric for grey-scale image comparison. *Intern. J. Computer Vision*, 24:5–17, 1997.