A Swarming Approach to Terrain Mapping

by

Gary Ruben

Dept. of Electrical & Computer Systems Engineering

Monash University

August 1998

Submitted in partial fulfillment of the requirements for the degree of
Master of Engineering Science (Coursework & minor thesis)

Author:	Gary Ruben
Supervisor:	Raymond Jarvis Professor, Director of Intelligent Robotics Research Centre, Monash University
Supplementary Supervisor:	R. Andrew Russell Associate Professor, Deputy Head of Department, Electrical & Computer Systems Engineering, Monash University

Abstract

Recently there has been an increase in interest amongst researchers into systems of multiple autonomous mobile agents or robots. Much of the work has focused on examination of maximising the efficiencies which can be achieved through the use of multiple agents.

For some tasks, the use of multiple agents may be justified purely in terms of advantage over single-robot systems. Examples of another category of tasks have been investigated by workers in the field. One of these is coordinated box pushing. In this category, the use of multiple agents is necessary to perform the task. This thesis approaches another member of this category for which the author believes multiple autonomous agents are a requirement; that of terrain mapping by a group of robots.

The aim of this thesis was to develop the algorithms and identify the requirements of each member of the robot surveying team. To achieve this, computer simulation was used. A governing philosophy was to keep each of the robots as simple and cheap as possible.

Declaration

I certify that this minor thesis contains no material which has been accepted for the award of any other degree or diploma in any University, Institution or College and that to the best of my knowledge and belief this thesis contains no material previously published or written by another person except where due reference is made in the text of the minor thesis.

Signed by the author . . . . . .

Contents

Figures

Algorithms

Equations

Acknowledgements

I would like to thank Ray Jarvis for his time and guidance in this work, especially for making time for me as my deadline was approaching.

I would also like to thank Julian Byrne for his pointers and code samples relating to resection.

I thank my mother, Inge Ruben for helping me during the proofing phase of this thesis and for continually correcting my language and grammar in my younger years. I think there is some evidence in here that she did a good job. I thank my father, Rudolf Ruben for always taking an interest in my work. I can now show him some evidence that there was some.

I thank Robin Morrison and David Jenkinson for allowing me some freedom to organise breaks from my employer in order to pursue this work.

1 Introduction

1.1 Multiple robot systems

There are two groups interested in the study of multiple autonomous agents; the Distributed Artificial Intelligence (DAI) researchers and the artificial life (Alife) researchers [Mat95]. The slant of these two groups differs slightly.

The DAI group is generally interested in examining the coordination between agents which may be internally complex. Practical examples might be the coordination of complex robots or research into *intelligent agents*, which are not physically embodied. In contrast, the Alife group is in general more interested in aspects of emergent behaviours of often large numbers of simple agents. A good introduction to the field is provided in [Ste95]. Alife models were originally devised to aid in the study of natural evolution in biological systems. Probably for practical reasons, most of the work in Alife tends to take the form of software simulations. Areas of interest usually include the study of cooperation between agents and comparison of their control methods eg. Completely autonomous, hierarchical or centralised.

Paraphrasing Taylor and Jefferson [Lan95] “one of the most fundamental and successful insights into the field of AL has been the development of a population modelling paradigm that represents a population procedurally, rather than in terms of differential equations whose solution determines the state of the population. The population is modelled as a set of coexecuting computer programs, one for each cell or one for each organism. We consider this feature, the representation of organisms by programs, to be the defining feature of ALife models”. In the context of Swarm Robotics, each cell or organism, representing one robot, is interchangeably called an *agent* or *unit*.

DAI can be further subdivided into the areas of Distributed Problem Solving and Multi-Agent Systems (MAS). It is the MAS subfield which is concerned with utilising a system of heterogeneous agents to achieve some goal. Creating physical MAS’s, capable of performing useful work, is a current area of interest of robotics researchers.

The work presented in this thesis could be categorised somewhere midway between the MAS and Alife classifications, as it deals with a smaller number of agents than are generally considered in Alife studied yet the Agents are relatively simple internally.

The issues surrounding interaction between individuals in groups of multiple robots fall under the title Swarm Robotics (SR). MAS can be said to encompass SR in that it deals with generic agents, whereas SR deals with physically embodied agents ie. robots.

One of the aims of SR research is to exploit the advantages of using a group of Robots to achieve a task over using a single complex robot to achieve the same task. In other cases, tasks exist which cannot be performed by a single robot, necessitating an SR approach. A robot swarm may carry several benefits:

- Redundancy - Failures of individuals lead to a gradual, not catastrophic, failure of the system. Loss of an individual member of a swarm is not disastrous, whereas loss of a subsystem of a complex robot usually results in catastrophic failure.
- Simplification - Individual robots may be simpler and cheaper.
- Time Efficiency - A number of robots may achieve a task in a shorter time than a single robot. Maximising the increase in efficiency is one of the main areas of investigation in this field.
- Resource Efficiency - Limited resources may be allocated amongst members of a heterogeneous robot team in order to maximise the efficiency of the team.

Note that in these fields, anthropomorphisation runs rife. The author apologises in advance for any terminology which maintains this trend.

1.1.1 Communication

For the swarm to have a useful task achieving behaviour, the members must communicate.

Directed communication is communication aimed at a particular receiver or receivers. *Indirect communication* is communication based on the observation of other agents' external states. Indirect communication can be by *signposting*, where robots broadcast their state by advertising it visually or in some localised way. Another method of indirect communication is by changing the local environment, so that when it is visited by other robots, the change is sensed. This is termed *stigmergy*, so-named by the French biologist P.P. Grassé during his investigations of termites [BeHo94], which leave chemical trails to aid nest building and foraging.

1.1.2 Cooperation

Cooperation is defined as an action by one agent which assists another agent in the achievement of its task. This involves communication of one of the types identified above.

Explicit cooperation is interaction which involves exchanging information between agents or performing actions to benefit another agent in achieving some task. That is, a type of altruistic behaviour. *Implicit cooperation* also involves an exchange of information or performance of a task, but in this case it is a necessary part of the agent's own behaviour

and is not explicitly intended to benefit another agent. Cases where such a benefit exists are examples of symbiosis.

The behaviours of agents in a robotic swarm must use cooperation to deal with interference which adversely impacts the swarm's task achieving efficiency.

Multi-agent systems are subject to two types of interference; *resource competition* and *goal competition*. Resource competition arises when agents share and compete for resources such as space, energy, information or objects. Goal competition arises when agents in a heterogeneous group compete to achieve mutually incompatible goals or subgoals.

Interference of both types must be reduced to decrease the adverse impact on the group's efficiency. In the behavioural approach, this is achieved by building or evolving social behaviour sets, leading to a reduction in global cost functions related to the achievement of the overall swarm's task. This is often at the expense of increases in cost functions at the scale of individual agents, although social rules can equally lead to reductions in these cost functions, translating to increases in efficiency of individual agents.

Multi-agent control strategies aim to exert control over individuals in the swarm to allow the swarm's task to be achieved. Moreover, they partially or fully define the task itself. The control strategy may be controlled by imbuing certain agents in a heterogeneous swarm with planning abilities and designing in communication hierarchies to support information about the task and agents' states reaching the planner agents. At the other extreme, the behaviour of the swarm may arise as a result of individuals' subtasks together producing an emergent global behaviour.

1.2 Swarm taxonomy

Dudek et. al. [DuJe93] propose a taxonomy for organising different swarm models. The distinguishing factors, along with their possible classifications:

- swarm size
  - Alone (1 robot),
  - Pair (2 robots),
  - limited group size with respect to task size,
  - large effectively infinite group size.

A swarm size of *alone* is a case where the authors, by their own admission, include a non-swarm system to complete the classification system[1].

[1]By so doing, Dudek et. al. create a good example of a Russellian Strange Loop by including a set which contains all members outside of itself.

- communication range — none,
  communication between near agents,
  infinite (able to communicate with any other agent).
- communication topology — broadcast,
  address,
  tree,
  graph,
  other.
- communication bandwidth — high (communication cost is free),
  motion (communication carries a cost related to the distance between communicating agents),
  low (communication cost is high).
- swarm reconfigurability — static,
  coordinated rearrangement (requires communication),
  dynamic rearrangement (agents can redistribute arbitrarily).
- unit processing ability — non-linear summation unit (eg. processing equivalent to a single artificial neuron),
  finite state automaton,
  push-down automaton,
  Turing machine equivalent (general computation).
- swarm composition — homogeneous,
  heterogeneous.

A swarm composition is still classified as heterogeneous if its member agents are physically similar if the programming of the agents differs.

1.3 The approach

For this thesis work the behaviours of members of a swarm of simple robots were studied through creation of a simulation environment.

The Borland Delphi Object PASCAL language was used to implement the simulator.

The author applied the Swarm Robotic approach to the task of terrain-surveying. This seems to be a challenging task as situated agents operate at the scale of terrain features whereas what is desired is to combine these into an overall map. This task is analogous to that undertaken by explorers and continental circumnavigators; that of mapping the land

features and coastlines of a land mass, where the explorer can only see a small portion of landscape or coastline at any one time.

A central assumption of this work is that the map cannot be built by remote sensing, ie. that each point in the map must be visited to obtain the feature data for that point. An example of where this might be necessary is mine sweeping. Another example might be ocean floor exploration by mining companies, say to obtain magnetic or gravitational field maps. In this case, accurate position information cannot be derived from satellite GPS signals and must be determined through more conventional surveying techniques.

The feature data comprising the terrain can be any measurable physical quantity. Some other examples are altitude, seismic and radioactivity readings. Any references to the term feature data will subsequently assume them to be single numerical values. This is for convenience only as, for example, it is possible to directly measure gravitational gradient which results in a 3-dimensional tensor. However, since a feature map based on tensor fields requires more than 2 dimensions for its representation, these were not considered.

In the simulator, agents were realised as Delphi Object PASCAL objects. Internal to the agents, behaviours were programmed as Finite State Machines (FSMs) whose states change in response to external localised stimuli. Under the described taxonomy, the unit processing ability of members of the swarm is *Turing machine equivalent (general computation)*. Thus, the agents are able to process the sensor data using traditional computational techniques.

2 Surveying with a group of robots

2.1 Surveying

2.1.1 History, Techniques and Technology

To contextualise this work, it is helpful to describe traditional Surveying techniques [Bri95].

Surveying is believed to have originated in ancient Egypt. There is evidence that its development must have occurred prior to the building of The Great Pyramid of Khufu c.2700 B.C.

Current-day surveying uses many technologies developed this century [BaRa84]. These include the theodolite and laser rangefinder. A theodolite is a tripod-mounted telescope through which the surveyor can sight to set targets and accurately measure angles in both horizontal and vertical planes. Modern theodolites can measure directly to an accuracy of about 1 arc second or $^{1}/_{3600}$ of a degree [PrUr89]. A laser or microwave rangefinder allows precise measurement of the distance to set targets.

The two main types of surveying are Geodetic and Plane. Plane surveying is concerned with a level of accuracy whereby the curvature of the Earth (or other substrate) may be ignored. When a higher degree of accuracy is required, Geodetic surveying techniques are used which account specifically for this curvature.

In general, a reference framework is established by accurately measuring the angles and length of sides of a network of triangles. Traditionally, the corners of these triangles have been placed on hilltops, each visible from at least two others. Smaller scale surveying work then builds on this accurate framework. Often, a lower level of accuracy is acceptable for this smaller scale. Establishment of the framework may use Geodetic surveying techniques and the smaller scale work make use of Plane techniques. The work in this thesis is limited to Plane surveying.

Electronic Data Measurement (EDM) using a laser operating at optical or infra-red wavelengths works by firing a beam at a set of precisely positioned corner mirrors and measuring the return trip time of flight of the beam. The time of flight is determined using interferometry techniques. This type of laser rangefinding is used at scales of tens to hundreds of meters. This is the scale at which the robot surveyors which are the subject of this thesis operate.

More recently, scanning laser rangefinders have become available which do not need to be aimed at mirrors and which can measure the range to multiple features in a single scan, which takes of the order of 100ms.

A microwave radio beam can be used over longer distances. This also has the advantage of being able to penetrate atmospheric disturbances. A disadvantage is that an active retransmitter is required at the target to transmit the return beam. Thus the time of flight to the target is half the distance between the total-round-trip time and the receive-to-retransmit delay in the retransmitter.

Although aerial or remote surveying gives an accurate map or survey, it does not cover all surveying situations. Ground surveying must still be used, for example, under a forest canopy where information about the shape of the ground is needed or if the features to be mapped cannot be identified from aerial images such as property boundaries or changes in soil and vegetation. Also, ground surveying is used to accurately determine elevation for use in planning roads, railways and drainage networks.

2.2 Behaviour model approach to multiple robot systems

Traditional approaches to the investigation of swarming systems would model the agents' behaviours as rule-based or finite state machine (FSM) computations. By examining the efficiency with which the swarming system achieves its task, the behaviour FSMs are redesigned by the researcher to improve efficiency. As this can be a fraught exercise, recently there has been a move toward allowing the behaviours or behaviour selection mechanisms of individuals to evolve [ShFu93][LuSp96][MiMa97]. This may be done by endowing them with genetic-algorithms whose outputs code for differing selections. The genetic code is then allowed to evolve according to some fitness measure [KoMe95]. ie. There has been a shift from a top-down to a bottom-up approach. This has been most evident within the Alife group; less-so amongst DAI researchers, revealing the differing interests of the groups.

However, the newer approach is often based on combining a subset of behaviours which have been identified using the traditional approach. Later work might then seek to evolve the behaviour selection criteria to maximise some aspect of task achievement [Mata94][ScMa96]. The work in this thesis is based on the traditional approach as the application of SR to accurate surveying has not been investigated before.

The behaviour model upon which this thesis is based is due to Maja Matarić [Mat94]. This work grew out of the work of Rodney Brooks [Bro86]. Brooks' approach to achieving useful, real-time behaviour by a robot operating in a complex world is to decompose task

solving into simple behaviours which are combined by a hierarchy of competing processing units.

Matarić built on this behavioural approach by applying it to the problem of interacting multiple robots. She proposed a basic set of robot behaviours, which may be combined to generate more complex behaviours. These behaviours form a minimal set, the combination of whose members allow a complete implementation of interaction and navigation in a plane [Mat95]. The basic behaviours are called *avoidance*, *following*, *aggregation*, *dispersion*, *homing* and *wandering*.

A behaviour is defined as an operator which guarantees a particular goal. The goals fall into two categories; *maintenance goals* or *attainment goals*. A maintenance goal is one which ensures that some time persistent task is being achieved or a dynamic equilibrium is being maintained. An attainment goal places the agent in a terminal state with respect to that behaviour; once achieved, the behaviour state enters a completed state.

An example of a maintenance goal is ‘maintain energy stores by collecting food’. An example of an attainment goal is ‘navigate to a goal’.

2.2.1 Basic behaviours

The basic behaviours, *avoidance*, *following*, *aggregation*, *dispersion*, *homing* and *wandering* may be described in formal notation. They are included here from [Mat94] (with slight corrections) in terms of vector positions p and scalar distances d, and distance thresholds δ_{avoid}, $\delta_{disperse}$ and $\delta_{aggregate}$. The descriptions are based on standard Cartesian geometry of a 2-dimensional plane.

Safe Wandering

$$\frac{dp_j}{dt} \neq 0 \quad and \quad \forall i \forall j \quad d_{i,j} > \delta_{avoid} \qquad (1)$$

where p_j is the jth agent’s position. This states that the agent must move continuously whilst maintaining a distance from all other agents greater than δ_{avoid}, the value of which is set to avoid collisions between agents.

Following

Following aims to reach and maintain a minimum angle θ between the position of the leading agent i and the following agent j.

$$
\begin{aligned}
& i = leader, \quad j = follower \\
& 0 \le \frac{dp_j}{dt} \cdot (p_i - p_j) \le \left\| \frac{dp_j}{dt} \right\| \| p_i - p_j \| \\
& \theta \to 0 \quad \Rightarrow \quad \cos\theta \to 1 \quad \Rightarrow \\
& 0 \le \frac{dp_j}{dt} \cdot (p_i - p_j) \le \left\| \frac{dp_j}{dt} \right\| \| p_i - p_j \|
\end{aligned} \quad (2)
$$

The product of the velocity and position difference is to be taken in the vector sense. By keeping this dot product positive, the *follower* agent attempts to decrease its distance to the *leader*. As the angle between velocity and distance vectors approaches 0, the upper bound on the dot product approaches the scalar shown.

Dispersion

Dispersion aims to reach and maintain a minimum distance between agents.

$$
\forall i \forall j \quad d_{i,j} > \delta_{disperse} \quad and \quad d_{disperse} > \delta_{avoid} \quad (3)
$$

which states that the distances between all agents are to kept greater than a threshold $\delta_{disperse}$ which is in turn greater than the threshold distance defined to avoid collision.

Aggregation

Aggregation is the complementary basic behaviour to dispersion. It aims to reach and maintain a maximum distance between agents.

$$
\forall i \forall j \quad d_{i,j} < \delta_{aggregate} \quad (4)
$$

Homing

The homing basic behaviour aims to decrease the distance between the agent and a goal location called "home".

$$
\forall j \quad \frac{dp_j}{dt} \cdot (p_j - p_{home}) < 0 \quad (5)
$$

By keeping this product (of the velocity and vector position difference between the agent and home position) negative, the agent always decreases its distance to home.

2.2.2 Combining basic behaviours

Basic behaviours must be combined in some way to form useful composite behaviours.

One example of a composite behaviour is *flocking*. The term defines a collective motion of agents toward a goal. It may be realised as a combination of *safe-wandering*, *aggregation* and *dispersion*. By combining the *flocking* behaviour with *following*, a slightly more complex composite behaviour called *herding* is realised.

The combination of basic behaviours to form more complex composite behaviours may be performed in two ways; directly or temporally. Direct combination of behaviours allows all basic behaviours to simultaneously contribute to the behaviour output. Direct combination may be realised by summing direction and velocity vectors, which are the outputs of the basic behaviours. The *flocking* and *herding* behaviours are formed through direct combination of their basic constituent behaviours.

Temporal combination of basic behaviours to form more complex behaviours is performed through sequencing by a finite state machine. Sensor data is used by the agent to define events which cause changes of state.

The composite behaviours possessed by the robot surveying team members in this investigation required only temporal combination of the *safe-wandering* and *homing* basic behaviours.

2.3 Issues with multiple robot systems

2.3.1 Cooperation through communication

The members of a robot swarm can use communication to improve the efficiency of some global task. Communication thus provides a method for trading off improvement in the global cost function against local cost functions [Mat97]. If an improvement of the global cost function is measured, a problem of credit assignment arises. Reward must be distributed amongst the agents whose contributions to the global cost function vary. If the credit assignment problem can be solved, this reward may then be used by individual agents to affect behaviour selection.

To solve the assignment of credit involves communication between agents. Aspects of the global task state can be sensed by individual agents and this information disseminated amongst the agents to build the global task cost function.

2.3.2 Specialisation amongst agents

A Swarm Robotic (SR) system may be classified as homogeneous or heterogeneous, based on whether the agents comprising the system are identical or specialised, both physically and with respect to their behaviour sets.

Homogeneity of agents has the following advantages:

- A given number of identical agents may be able to be manufactured more cheaply than a heterogeneous group.
- Standardisation in all aspects relating to the group, such as delivery and maintenance is simplified.
- Redundancy is built into a homogeneous group. If individuals in the group fail, since the agents all have a common goal, this will be achieved by the remaining agents.

Heterogeneity has the following advantages:

- Resources may be allocated to group members in a cost effective manner. Where agents' subsystems are expensive, it is not sensible to provide these same subsystems to all members of a group.
- Communication hierarchies may be established, often leading to a more natural system of swarm management.

2.4 Applying the behaviour model approach to surveying

2.4.1 Introduction

To date most research seems to have applied the multiple agent behavioural approach to tasks such as goal-seeking, puck collection etc. That is, tasks of little practical value. The reasons for choosing tasks such as these are to simplify the investigation as this is still a young field of investigation. Also, it is usually not the tasks themselves which are the desired outcome of studies so far, but the investigation of increasing system efficiency through careful behaviour selection by the investigator. Much of the work in this area of investigation deals with artificial behaviour selection, which is more concerned with techniques for optimising robot behaviour through learning.

It was with the intention of finding an area of investigation of some potential practical value that the terrain mapping task was selected.

Note that it may not be necessary for a map to be built at all for some tasks such as navigation [Lit94]. Cohen [Coh96] successfully shows a method for navigating toward goals

by a team of robots where no internal model for the environment need be generated. This information is instead related to the physical arrangement of the agents. That is, robots move to positions in the environment where they can act as markers by which a specialised robot navigator may find a route to the goal. However, the primary aim of the work presented in this thesis is to generate the map itself. It is the terrain map data which is the desired information of the system. This requires that an internal representation be produced somewhere.

2.4.2 A Multiple Agent Approach

Traditionally, to build a map, the cartographer must establish reference coordinates and then map the features relative to these reference points.

A lone surveyor who visits a remote site could place reference markers at strategic points around and throughout the site, then sight to these reference markers from various points throughout the site to obtain the survey data; a tedious task. Recognising the tedious nature, surveying is often carried out by a small party of people to expedite the work. In this project the author recognises that one could 'roboticise' the surveying equipment to carry out this task.

2.4.3 Description of the Proposed Scheme

It is possible for the programmer of the robot swarm members to identify several maintenance goals which must be achieved to ensure the success of the team. The programming of behaviours exhibited by agents which meet the maintenance goals is then performed, enabling the team to perform its task. Since the robot team in this thesis is performing a time persistent task, the maintenance goals become simultaneous subtasks of the team. These subtasks may then be assigned amongst different members of a heterogeneous swarm.

The scheme proposed to perform the task of terrain surveying is to tessellate the area to be surveyed. This physically constrains the problem to a finite region of a potentially infinite plane. By dividing the area in this way, subtasks can be identified for the swarm. It is assumed that no initial information is known about the area.

Subtasks

- Define the current area to be surveyed
- Move the group to the area
- Build a map of the defined area

The robot swarm is assumed to be completely autonomous. It does not require communication with any outside entities to perform its task. It is envisaged that the team would be placed into the area to be surveyed, left to carry out the task and then the map would be retrieved upon completion.

The initial strategy chosen by the author primarily addresses the requirement that it be able to achieve the task in a physically implementable manner. Robustness and efficiency were regarded as secondary aspects of the scheme. These are considered in later discussions.

Consider, the subtask of defining the area to be mapped.

In the proposed scheme, some agents are used as reference markers until the terrain surrounding them has been adequately mapped by other agents. This provides a means to achieve the first subtask, provided these reference marker agents can be sensed and used to constrain the survey area.

The technique investigated establishes a reference coordinate system whose position, orientation and scale are fixed by the triangular arrangement of three of the situated agents, henceforth called *Reference* agents. These agents are mobile but can fix their positions to establish the reference coordinate system whilst other mobile agents move around collecting feature data from the environment (*Roaming* agents). It was decided to tessellate the terrain map into triangles as this represents the simplest system.

Then, when a determination is made that the defined area has been adequately surveyed, the Reference agents move so as to a define a new area.

In order to compile an overall map based on the terrain map data from each defined area, a single coordinate system must be established. To establish a two dimensional Cartesian coordinate system, a minimum of two agents are required; one to provide a zero reference and the other to fix one of the axes. One reason for using three Reference agents is that this is the minimum number required to allow one of them to be moved without destroying the coordinate references. The reference system would be lost if two Reference agents were not stationary at any one time to establish the coordinate system.

When the Roaming agents have mapped the area marked out by the Reference agents adequately, one of them is signalled to move a new position. The moving Reference agent uses the coordinate system established by its two stationary counterparts to accurately position itself at the new reference position. Figure 1 shows the signalling sequence, which allows the Reference agents to traverse the terrain as a group.

To support this movement, communication based on simple protocols is required between Reference agents.

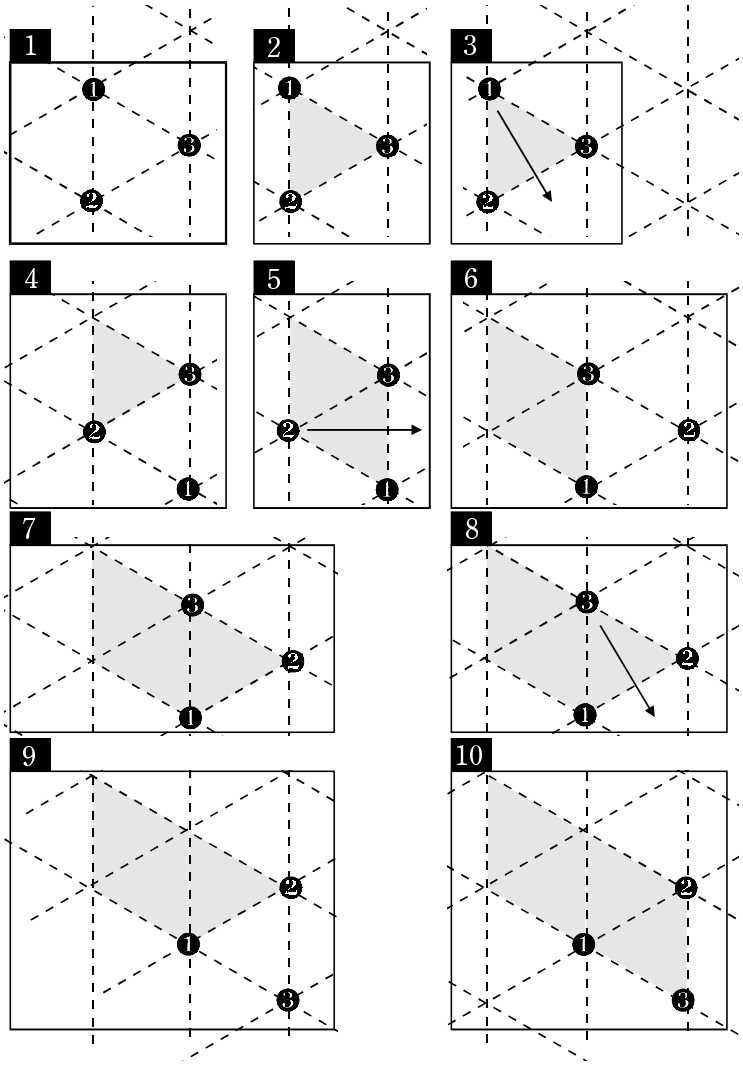


Figure 1. Conceptual tessellation of terrain with Reference agent motion.

1. Initial agent position. 2. Region survey completed. 3. Reference Agent #1 signalled to move. 4. Reference Agent #1 completed move. 5. Region survey completed, Reference Agent #2 signalled to move. 6. Reference Agent #2 completed move. 7. Region survey completed. 8. Reference Agent #3 signalled to move. 9. Reference Agent #3 completed move. 10. Region survey completed and ready to signal Agent #1 again.

To perform the second subtask of moving the group to the area, the Roaming agents must be programmed with a behaviour which moves them into the area of interest based on them sensing the Reference agents' positions.

Finally, to meet the subtask of building a map, the feature data collected must somehow be compiled into an overall map.

To avoid the problems of trying to marry together many disparate maps, one special agent is given the task of building the map. This agent, which is one of the Reference agents, is known as the *Mapping* agent. The remaining Reference agents are given the name *Beacon* agents. Thus there are two Beacon agents, one Mapping agent and a minimum of one Roaming agents.

Communication of the internal state of some agents to others provides a mechanism for behaviour selection of agents. The communication can be hierarchically organised, as was done in the surveying robot team. The Mapping agent, which is at the top of the hierarchy, manages the task through directed communication to other Beacon agents.

In order to build a map, each feature datum must be associated with a position. The Roaming agent obtains the feature and must relay its value back to the Mapping agent. What is not so obvious is how the position associated with the feature is established.

It is desirable to minimise the complexity of sensors on the Roaming agents, since this would allow the overall cost of Roaming agents to be minimised, allowing more of them to be used to minimise surveying time (assuming that the task of roaming and measuring features is the time consuming part of the exercise; not communication or map construction). It could be that the Roaming agent attempts to accurately establish its own position based on its measurements of Beacon agents' positions. One approach to this method might be to measure the angles to each Beacon agent using a vision system. An optical system which mapped a 360° view onto a 720 pixel stripe Charge Coupled Device (CCD) would, by Nyquist's theorem, give a 1° angular resolution. A typical flatbed scanner might contain a 4800 pixel stripe CCD (8" at 600dpi), which with a suitable optical system could give a 0.15° angular resolution. The Roaming agent measures the angle between each point-source Beacon, which illuminates a region of stripes of the CCD.

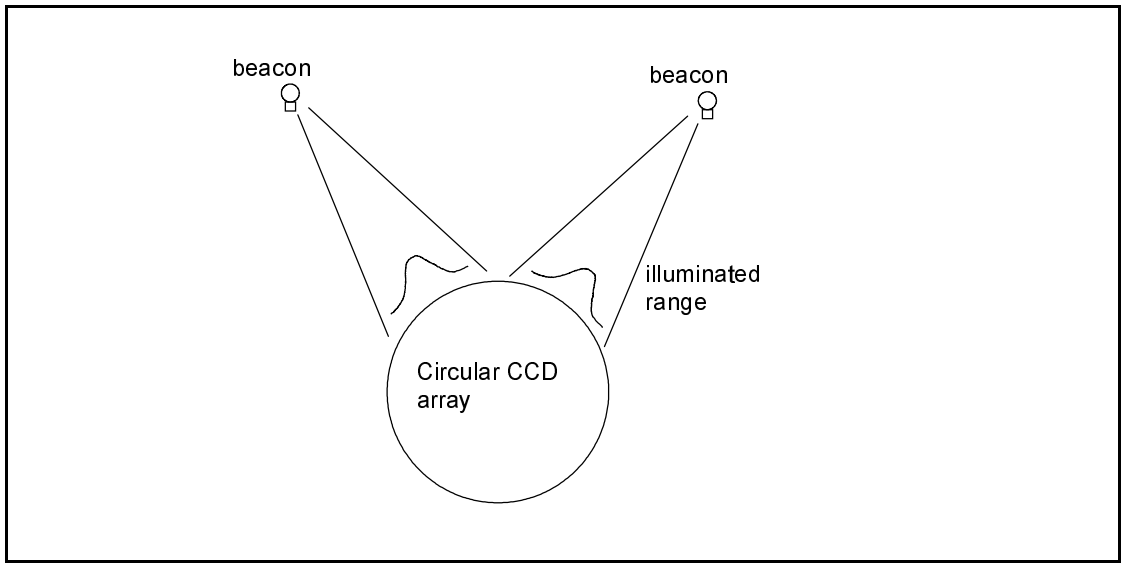


Figure 2. CCD angle sensor

The brightness variation over this region should allow the angular positions of the beacons to be established reasonably accurately. One advantage of this approach is that only the Reference agents need to emit energy to permit position data to be determined except when the Roaming agents transmit their feature data. Each pixel of the CCD array could have a cylindrical lens (where the axis of the cylinder is in the horizontal plane) so that vertical position variation of the Beacon and Mapping agents is catered for. Also, vertical collimators between adjacent pixels would ease in localisation by reducing the angle in the horizontal plane over which the pixel detects a Beacon.

The alternative approach where the Reference agents attempt to fix the position of Roaming agents should not be dismissed. To simplify this situation, a scheme is necessary whereby each Roaming agent in turn indicates its wish to have its position measured, perhaps by broadcasting its feature data in a serial data format from an omnidirectional infrared LED. Each reference agent reads this data stream and obtains an angular fix on the transmitter, along perhaps with an identification label transmitted as part of the data stream. This could aid assembly of the data by the Mapping agent. In this scheme, the two Beacon agents must relay their angular position data back to the Mapping agent, increasing the likelihood of errors and slowing down the collection of data.

The selected approach is to allow the Roaming agents to determine their own positions, with inherently increased errors due to cheaper sensors with lower angular resolution. They then transmit their feature and position data directly back to the Mapping agent using a low-cost radio system, similar to a radio ethernet. ie. any scheme which resolves conflicts where multiple Roaming agents transmit ‘over the top’ of each other’s transmissions. An

advantage of this technique is that systematic position errors should cancel each other out, as they will only be associated with a single Roaming agent. This is in contrast to the potentially graver problem of systematic errors from Reference agents applying to the positions of all Roaming agents.

In the case of a Roaming agent in the presence of two Beacon agents, b_1,b_2 and one Mapping agent, m, data, d, from an individual Roaming agent can be represented as the 4-tuple;

$$< f \measuredangle m, f \measuredangle b_1, f \measuredangle b_2, d >, \text{ where } f \text{ is the forward direction} = 0^o$$

or

$$< \alpha, \beta, \gamma, d >$$

The behaviour of the Roaming agents must rely purely on this data. Also, if the behaviours can be described in terms of the raw angle data, processing and behaviour implementation by the Roaming agent will be simplified.

In order to correctly form this data, all three Reference agents must be visible by the agent in question and the sensors and associated processing must behave ideally. If this is not the case, either the tuple will not be able to be formed or incorrect data will be produced. In the case where the tuple cannot be formed, this can be the trigger for a behaviour selection which could, say, seek to move the agent such that its vision is not impaired. In the other case, where incorrect data is produced, this cannot be sensed and the inherent fault-tolerance of the surveying system is relied upon.

The order of data is also important. It is therefore necessary for the Roaming agent to be able to identify which is the Mapping agent. To support this identification, it is proposed that the Mapping agent's beacon would be modulated somehow and that the Roaming agents would be sensitive to this modulation. It is not necessary for the Roaming agents to know the identity of the Beacon agents as they transmit the angle data ordered by increasing angle. Thus, the Mapping agent, which knows the positions and identity of the Beacon agents, can match the data to the correct Reference agent.

The tuple is relayed back to the Mapping agent to enable it to determine the transmitting agent's position and associated feature datum. The method used for this determination is called *resection*[2].

[2]See Appendix 1.

2.5 Agent descriptions

One simple way of modelling an autonomous agent is to subdivide it into blocks representing sensors, behaviours, effectors and physical traits. See Figure 3.

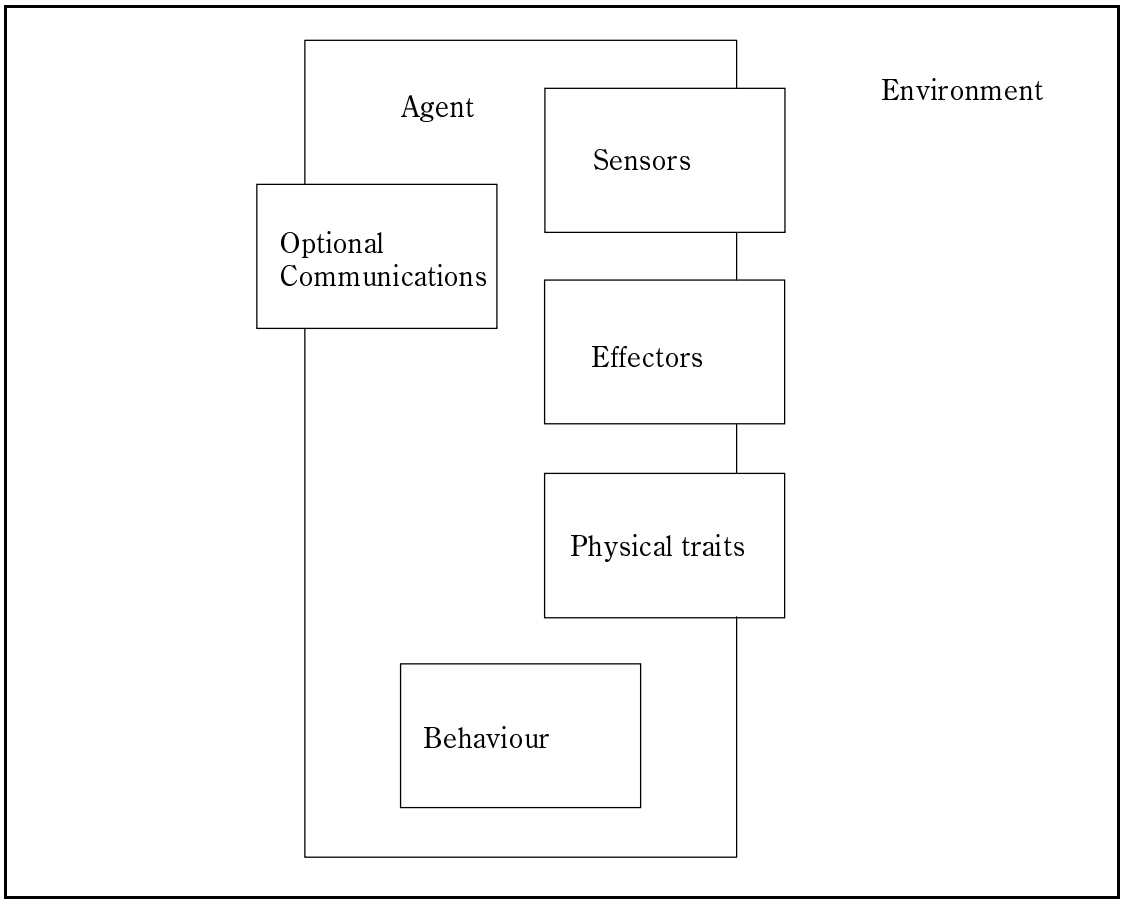


Figure 3. Agent model

The agents perform different roles in the survey task. To support this, they differ in each of these identified blocks, leading to the robot surveying team being classed as heterogeneous.

A communication block could be added as shown in the diagram, or this role can be subsumed by the sensor and effector blocks, the act of communication essentially being to induce an energy disturbance into the environment through an effector and for this disturbance to be sensed by some other agent.

For this simulation, no modelling of physical traits was performed. However, some assumptions were made to simplify the model. For example, it is assumed that agents have zero extent; quite an assumption.

2.5.1 Roaming agent description

Sensors

Roaming agents have two sensors. The first is the CCD angle sensor described above. The second is a sensor to gather the feature data at the current location.

If the model is extended to allow interaction between Roaming agents, as suggested in a later section, it will be necessary to add a sensor to measure the proximity of nearby agents, or at least to detect collisions with other agents using a bump sensor or whiskers.

Behaviours

Roaming agents implement two basic behaviours. The first is *safe-wandering*. The second is *homing*.

The Roaming agents behaviour allows them to achieve the maintenance goal of the collection of terrain feature data and its transmission to the Mapping agent. The basic behaviours on which the composite behaviour is based are both maintenance goal achieving behaviours.

By using the angle data collected of the beacon positions, it is possible for Roaming agents to determine the area they should be surveying. The beacons lie at the vertices of a convex geometric figure (in our case the figure is a triangle, however, if more beacon agents are available, another higher order figure would work). If the measured difference between two subsequent angles exceeds 180°, the Roaming agent has left the area currently being surveyed. If this determination is made, the *homing* behaviour is selected which drives the Roaming agent toward the centroid of the Reference agent (Beacon and Mapping agent) positions. Thus angle data is used to select between the *safe-wandering* and *homing* behaviours.

Effectors

Roaming agents have two effectors. The first is a drive mechanism which allows them to travel at constant speed in a forward direction. They can turn left and right with a minimum specified radius. The second effector is a transmitter to broadcast relay angle and feature data in a directed or addressed manner to the Mapping agent.

In a physical implementation, the transmitter could be either an omnidirectional or directional antenna, since the Roaming agent can detect the direction of the Mapping agent. This is not important to the current simulation.

2.5.2 Beacon agent description

Sensors

Beacon agents have three sensors. These must support navigation of the agent to a new survey reference position. They must also support accurate ranging to other reference agents to ensure that this reference position is established accurately.

The first sensor is the CCD angle sensor described above. It is possible to navigate to the approximate position of a new reference position using only the angle data returned from this sensor.

The second sensor is a laser rangefinder. This provides support for accurate ranging. To support laser rangefinders on reference agents of the type traditionally used by surveyors, corner mirrors must be attached to all reference agents. Normally, laser rangefinders must be accurately aimed to ensure that the corner mirror is struck and the return beam captured by the interferometer. If a scanning laser rangefinder is used instead, the corner mirrors are not required. However, the extra range information from a scanning rangefinder could cause a problem in that it must be correlated with Reference agent positions and the remainder thrown away.

As some angle information is available from the CCD sensor, this provides a starting point for some undefined control mechanism to aim the laser. The laser could scan that region until a return beam is detected or the beam could be spread to permit inaccurate aiming or some combination of these could be applied.

Finally, to support protocols for moving to a new position, Beacon agents must carry a communication transceiver. Commands from the Mapping agent to take up a new reference beacon position and indicate that the new position has been reached are implemented as part of these protocols. The receiver part of this can be classified as a sensor.

Behaviours

The behaviour of the Beacon agents does not fit neatly into the behavioural approach model, as stated. Firstly, a new basic behaviour, which could be called *waiting*, is required, since Beacons spend most of their time sitting doing nothing. This could be considered to be a special type of *homing*, where the goal is always the agent's position.

The other behaviour implemented by the Beacon agents is *homing*, to move the agent to its new reference position when the Mapping agent signals it. However, this is performed using a staged approach. The Beacon agent normally rests until receiving a command from the Mapping agent to establish a new reference position. It then uses the angle data from

the CCD angle sensor to place itself close to the new goal position. Finally, it uses range data to accurately place home in on the new reference position, before returning to a resting state.

The Reference agents behaviours (Mapping agent behaviour is the same as for the Beacon agent) achieve the maintenance goal of acting as position references for other agents. To form this composite behaviour, both basic maintenance and goal-achieving behaviours are combined temporally.

Effectors

Beacon agents have three effectors. Like Roaming agents, the first is a drive mechanism which allows them to travel at constant speed in a forward direction. They can also start and stop and turn left and right with a minimum specified radius.

The second effector is the beacon itself which provides an omnidirectional, localised, detectable energy source. In a physical implementation, this could be a cylindrical light source, modulated in some way to allow identification of the source as a Beacon agent by Roaming agents.

The transmitter part of the communication transceiver is the third effector.

2.5.3 Mapping agent description

A Mapping agent can be thought of as a Beacon agent with the extra responsibilities of receiving the survey data, building it into a map and commanding the Beacon agents to move to new reference positions.

Sensors

Mapping agents are similar to Beacon agents and therefore also have three sensors and use them in the same way described for Beacon agents.

Behaviours

The *homing* behaviour implemented by the Mapping agent is identical to that of the Beacon agents.

Effectors

The description for these is similar to that for the Beacon agent. However, the beacon itself is modulated differently to Beacon agents in some way to allow identification of the source

as a Mapping agent by Roaming agents. This allows the Roaming agents to correctly order their angle data.

Map Building and Surveying Task Management

The Mapping agent receives survey data and uses this to build the map using resection and statistical averaging of feature data received from coincident survey positions. The agent monitors the coverage of the area currently being surveyed. When this coverage, whose measure is based on a sample region of the area currently being surveyed, exceeds a threshold, the Mapping agent commands the Beacon agents in turn to move to new positions. The mapping task is halted until the Beacon agents respond that they have acquired their new position.

The robot team builds a map of a strip of the terrain. If it is desired to have a more generalised area surveyed, a higher level behaviour would have to be programmed on the Mapping agent. The generalised area would be conceptually overlaid with the tessellation grid and an algorithm to cover each tile area in turn established. Although this was not investigated, it could be implemented in the current simulator. The only difference would be that the Mapping agent would signal Reference agents in a different sequence to move the team to regions outside the strip.

2.6 Taxonomy of the surveying swarm

As described in section 1.2, robot swarm systems may be classified according to a taxonomy. It is useful to do this to categorise the research and relate it to other research in related areas.

The swarm size classification is *limited group size with respect to task size*.

The communication range is *infinite (able to communicate with any other agent)* although it could equally be *communication between near agents* since all agents constrain their proximity and are always within communication range of each other.

The communication topology is *address*. All Roaming and Beacon agents transmit their position data only to the Mapping agent. The Mapping agent signals the Beacon agents to move.

The communication bandwidth is *high (communication cost is free)*.

Swarm reconfigurability is a combination of *coordinated rearrangement (requires communication)* which refers to the Beacon and Mapping agents and *dynamic rearrangement (agents can redistribute arbitrarily)* which applies to the Roaming agents.

Unit processing ability is *Turing machine equivalent (general computation)*.

Finally, the swarm composition is *heterogeneous*.

2.7 Combining Behaviours

Where individual basic behaviours must be combined, they may be combined directly, by computing each behaviour and summing the resultant velocity vectors, which are the behaviour outputs. This method of combining behaviours is applicable to the achievement of an individual goal.

In order to perform higher level tasks, the behaviours, which may be basic or combined behaviours, must be combined temporally, for example by a finite state machine.

Many implementations by existing researchers have used layered behaviours, where behaviours are combined using the *Subsumption Architecture* [Bro86] method. This has not been done in this project; the simpler approach of non-layered or traditional finite state machines being used instead, since it was always intended to keep individual agents' behaviours simple. In fact, it was not necessary to combine basic behaviours in any way other than temporally for this project.

3 The Simulator

3.1 The problems of simulations

The use of simulations in robotics research is often criticised for the reason that simulation can never hope to account for all real world phenomena. Because of this, the legitimacy of simulation results may validly be questioned.

However, provided the scope of the simulation and its limitations are understood it may often be a valuable mode of investigation. The cost of implementing custom hardware when a robot research question arises can be high. Thus, simulation is often a valid first step toward the implementation of a working system.

The basic tenet of simulation is that a simulator provides a model which the programmer believes captures the essential aspects of the real system it is attempting to simulate. It must always be expected that early models will have failings and any results from work with a simulation should be tested on physical systems to validate the model and any conclusions which are based on it. Note that it has been shown in some cases that imperfections in real-world systems can lead to simpler handling of tasks by physically embodied robots than in a simulation environment which may introduce simulation artefacts. The problems and the way with which they have been dealt in this work are now described.

3.2 Simulator Issues

3.2.1 Discrete time problem

Simulations by researchers at Xerox Parc have shown that the discretisation of time, or time-stepped approach taken by simulation environments can affect the dynamics of the agents within the environment [HuGl96]. The example investigated is the Iterated Prisoner's Dilemma (IPD) problem. It was shown that using a stochastic method of dealing with time led to conclusions which directly challenged (invalidated) previous results.

With this in mind, it was decided for this project to randomise the time sequencing of simulated agents, but not to take a completely stochastic approach to time. In other words, the discretised unit of time remains a constant value. However, the scheduler which executes each agent does so in a random order.

For an analogous reason, the simulation environment uses a continuous space model based on floating point representations. The alternative is to use a quantised or tessellated model of space. Note that there is a distinction between the space itself being quantised and the

survey regions being tessellated. If the space being modelled were itself quantised, positional errors of the same order of scale as the agents operating in the environment would accumulate and may have unforeseen effects on simulation results.

Two aspects of the simulation acting as inputs and outputs of the surveying task do tessellate the simulation space at a scale comparable to the agents. These are the terrain map itself and the internal representation of the feature data map formed by the mapping agent. However, these represent models from the point of view of the agents and as such are not a part of the space in which they operate. Thus, they are not constrained by the requirement that they be continuous.

3.2.2 Sensor errors

In all simulations, regardless of the area of investigation, any modelling of real world phenomena must be modelled, implying simplification. In the specific area of autonomous agent research where an agent has an interface between the real world and an idealised computational block, in our case a generalised Turing machine, simulations often assume that some or all of the sensor information is perfect. Similarly, the interaction of effectors with the environment is simplified in models. It is these blocks which interface with the real world where particular attention must be paid to modelling to maximise the validity of any results.

3.3 Development tool

The simulator was developed using the Borland Delphi Object PASCAL language. This language was chosen for the following reasons:

- PASCAL has a well recognised syntax which eases the interpretation by readers of any coded algorithms.
- Generation of a graphical model environment and graphical user interface is made easy by the Delphi toolset.
- It is an implementation of a modern object-oriented language. The task of simulating many autonomous interacting agents lends itself naturally to an object-oriented approach.

Some Object PASCAL issues are briefly described here to aid the following discussion.

- Code modules or files are referred to as *units*. This is not to be confused with the term units used by the Alife community to refer to an agent.

- Object-oriented programming concepts are outside the scope of this work. Briefly, Object PASCAL implements a modern object-oriented model supporting polymorphism and inheritance with abstract classes.
  Objects, which are instances of classes, contain data not directly accessible by other objects. These data are referred to as *fields*.
  Access to the objects conceptually occurs by sending a message to the object which invokes one of the object's defined *methods*. A method is a defined message interface to the object. In Object PASCAL, methods may be PASCAL procedures or functions or another interface type called a property.
- Some naming conventions used by Borland, which have been adhered to, are that class type names are prefixed with the letter **T**. Private field names are prefixed with the letter **F**.

3.4 Object hierarchy

The following object classes were defined for the simulator:

- **TForm**
  This class is actually defined by the Delphi environment. It contains the properties and methods and other classes associated with the graphical window in which the simulator is displayed within the operating system environment.
- **TAgentManager**
  This is the notional 'world' within which the agents reside. It contains those properties and methods which are associated with maintaining the lists of agents, sequencing their execution etc.
- **TAgent**
  This is the agent class itself. According to the object-oriented technique of polymorphism, the behaviour methods may be overridden to create different types of derived agent classes. These derived agent classes represent agents with different behaviours. Polymorphism has been used to implement the different agent types whilst maintaining a standard message interface to other objects.
- **TRoamingAgent**
  The Roaming agent.
  This class is inherited from the **TAgent** class.

- **TBeaconAgent**

  The Beacon agent.

  This class is inherited from the **TAgent** class.

- **TMappingAgent**

  The Mapping agent.

  This class is inherited from the **TAgent** class.

- **Tvector**

  This is a math module implementing a vector Abstract Data Type for generalised use by the behaviour methods.

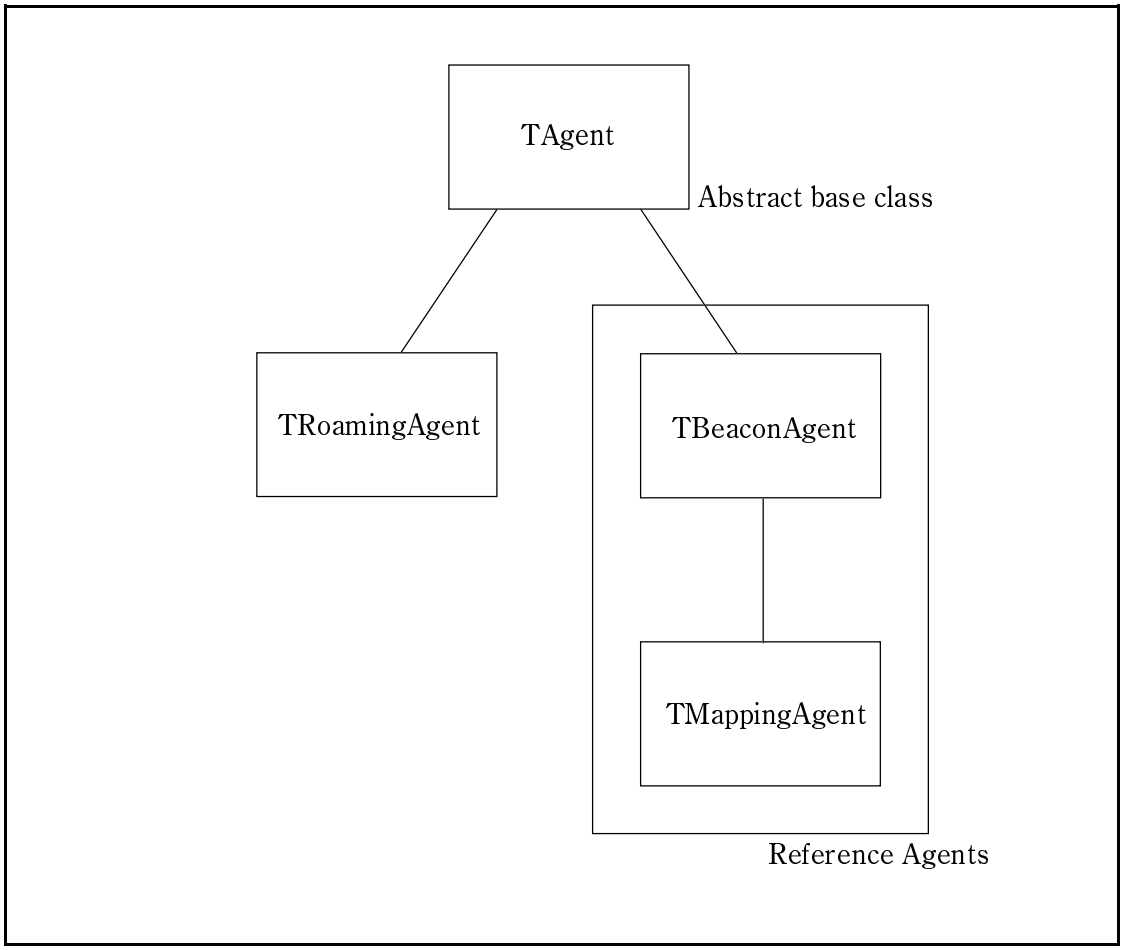


Figure 4. Agent object class hierarchy

3.5 TAgentManager class description

The **TAgentManager** class defines a manager object which is responsible for maintaining and controlling all agent objects in the simulation. There is only one instance of **TAgentManager** instanciated in the **Unit1** unit, called *agents*.

3.5.1 Fields

Fgeneration This field holds the iteration value of the simulation. It's value is displayed on the main window on the status bar. It is accessed by the **generation** accessor property.

agentList This field is a list of pointers to the agent objects. Agent objects are accessed by dereferencing a list item pointer. It uses the Delphi-provided **Tlist** class, which provides methods to control the list.

bmList This field is another list of pointers to the agent objects. However, this list only contains pointers to Reference agent objects. ie. Beacon or Mapping agent objects.

rList This field is another list of pointers to the agent objects. However, this list only contains pointers to Roaming agent objects.

3.5.2 Methods

draw This method invokes the **draw** methods of all agent objects being managed. That is all agent objects contained by the **agentList**.

sequenceAgents This method invokes the execute methods of all agent objects being managed in a random order. It is repeatedly called by the main application process.

txSignal This method is invoked by agent objects wishing to communicate using a simple defined protocol described in the **TMappingAgent** class description.

getRData This method extracts the position and local terrain feature datum measured by a Roaming agent and returns them to the caller, which is always the Mapping agent. Note that the position vector is returned directly so the simulated Mapping agent does not carry out a resection analysis to compute the position. To modify the simulator so that a resection analysis is done, a similar method would have to be added which returned the angle data from the Roaming agent.

getRQuantity, **getBMPos**, **getBMId** and **getBMQuantity**

These methods are all services invoked by agent objects to determine respectively the number of Roaming agents, the position of an indexed Reference agent, the Id field of Reference agents and the number of Reference agents.

An attempt to make the code independent of the number of Reference agents has been made in this unit, although the individual agents' behaviour code is very dependent on there being two Beacon agents and one Mapping agent.

3.6 TAgent class description

The **TAgent** class is an abstract base class - object instances can only be created from its derived classes. It provide fields and methods inherited by all derived classes.

Its most important method is **execute**. This method is invoked on each agent instance by the agents object, which is defined in the **AgentManager** unit, in order to execute the agent. **TRoaming** and **TBeacon** agent classes are directly descended from the **TAgent** class. The **TMapping** agent class is descended from the **TBeacon** agent class. The agent objects are instances of a polymorphically defined hierarchy. Because of this polymorphism, the agents may be executed, signalled and drawn by calling the one method name.

3.6.1 Fields

Fid This field holds a unique identifier which is allocated when the agent object is first created.

FlastPos, **Pos**, **Direction**

These fields contain vectors of the agent's position and direction information. This data is never directly accessed by the agent itself, except to generate other data which the agent would have direct access to, since in reality agents do not know this information. However, it makes sense to store these fields with the objects to which they apply.

Fstate This is a state variable used by the behaviour finite state machine.

3.6.2 Methods

behave This method implements the behaviour code as a general finite state machine. This is a *virtual* method. This means that there is no code

associated with its implementation in the **TAgent** abstract base class. Only derived classes provide code for this method.

move This method updates the position information, conceptually moving the agent in its forward direction by one distance unit.

execute This is the main method interface to the agent. It is invoked by the agents **TAgentManager** object on each agent object. This is a virtual method.

draw This method is invoked to get the agent object to draw itself into the defined window pane. This is a virtual method.

3.7 TRoamingAgent class description

The **TRoaming** agent class is descended from the **TAgent** class. Thus, it encapsulates all fields and methods of the **TAgent** class.

The **execute**, **behave** and **draw** methods are overridden and versions specific to this class type are provided. When the **agents** management object calls the **execute** method, the Roaming agent object just invokes its **behave** method and returns when complete.

3.7.1 Fields

Fangles This is a **TList** object, which is a Delphi-provided class. It is used to contain a list of angles from the point of view of the agent to the Reference agent positions. That is, it represents the sensor data which would be provided by an angle sensor.

3.7.2 Methods

senseBeacons This method is called by the agent's own **behave** method to generate the **Fangles** data.

behave This method provides the behaviour code as a general finite state machine.

execute This method simply calls the **behave** method for a **TRoamingAgent** agent.

draw This method is called to instruct the agent to draw itself.

3.7.3 Roaming Agent Behaviours

The **behave** method implements the Roaming agent behaviours.

The only behaviours implemented by the Roaming Agents are *safe-wandering* and *homing*, to confine the agents to the area inside the convex hull formed by the Beacon and Mapping

agent positions. The behaviour of the agent has been designed so that the *safe-wandering* and *homing* behaviours are never executed simultaneously. The *safe-wandering* behaviour simply moves the agent forward and turns it by a small, random amount.

The *homing* behaviour directs the agent toward the average measured angle of the three Reference agent positions. This is determined by generating and summing three unit direction vectors, which have the same angles as the measured sensor angle data.

```
state 1:
Safe Wandering:
move forward by d_forward
turn randomly
if (angle between 2 subsequent ordered angles > 180deg) then
  state 2

state 2:
Homing:
move forward by d_forward
if (angle between all sets of 2 subsequent ordered angles < 180deg) then
  state 1
turn to face the direction of the sum of unit direction vectors to all reference agents
```

Algorithm 1. Roaming agent behaviour algorithm finite state machine.

3.7.4 Roaming Agent Behaviour Selection

Behaviour selection is based on the angles which would be measured directly by a physically embodied agent. It is possible for the Roaming agent to determine whether it is inside or outside the convex hull formed by the Reference agent positions, based purely on angle measurements. If the angles to the Reference agents are ordered, a difference between subsequent angles of greater than 180° indicates to the agent that it inside the defined survey region. The result of this inside/outside measurement is used for behaviour selection.

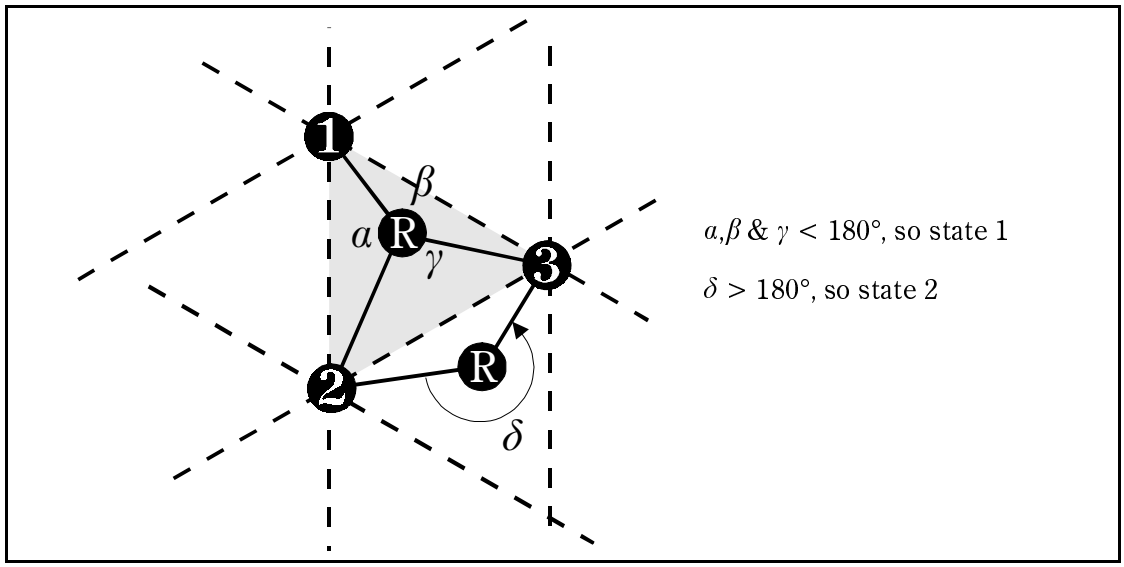


Figure 5. Roaming agent behaviour selection.

Safe-wandering is executed inside the region and *homing* is executed outside the region.

3.8 TBeaconAgent class description

The **TBeacon** agent class is descended from the **TAgent** class. Thus, it encapsulates all fields and methods of the **TAgent** class.

3.8.1 Fields

Fangles This is a **TList** object, which is a Delphi-provided class. It is used to contain a list of angles from the point of view of the agent to the Reference agent positions. That is, it represents the sensor data which would be provided by an angle sensor.

Fdistances This is another **TList** object. It is used to contain a list of distances from the point of view of the agent to the Reference agent positions. That is, it represents the sensor data which would be provided by a laser rangefinder sensor.

FsignalMove This is a flag which is set when the agent receives a **SIG_MOVE** signal.

Fgoal1Reached, Fgoal2Reached

These fields hold state information. They are used by the *homing* behaviour code to determine when the agent has reached its goal

3.8.2 Methods

senseBeacons This method is called by the agent's own **behave** method to generate the **Fangles** and **Fdistances** data.

behave, execute, draw

Although the code is unique to the **TBeacon** agent, the descriptions for these methods are the same as for the **TRoaming** agent.

signal This method implements the signal receiver of the **TBeacon** agent's communication transceiver.

3.8.3 Beacon Agent Behaviours and Behaviour Selection

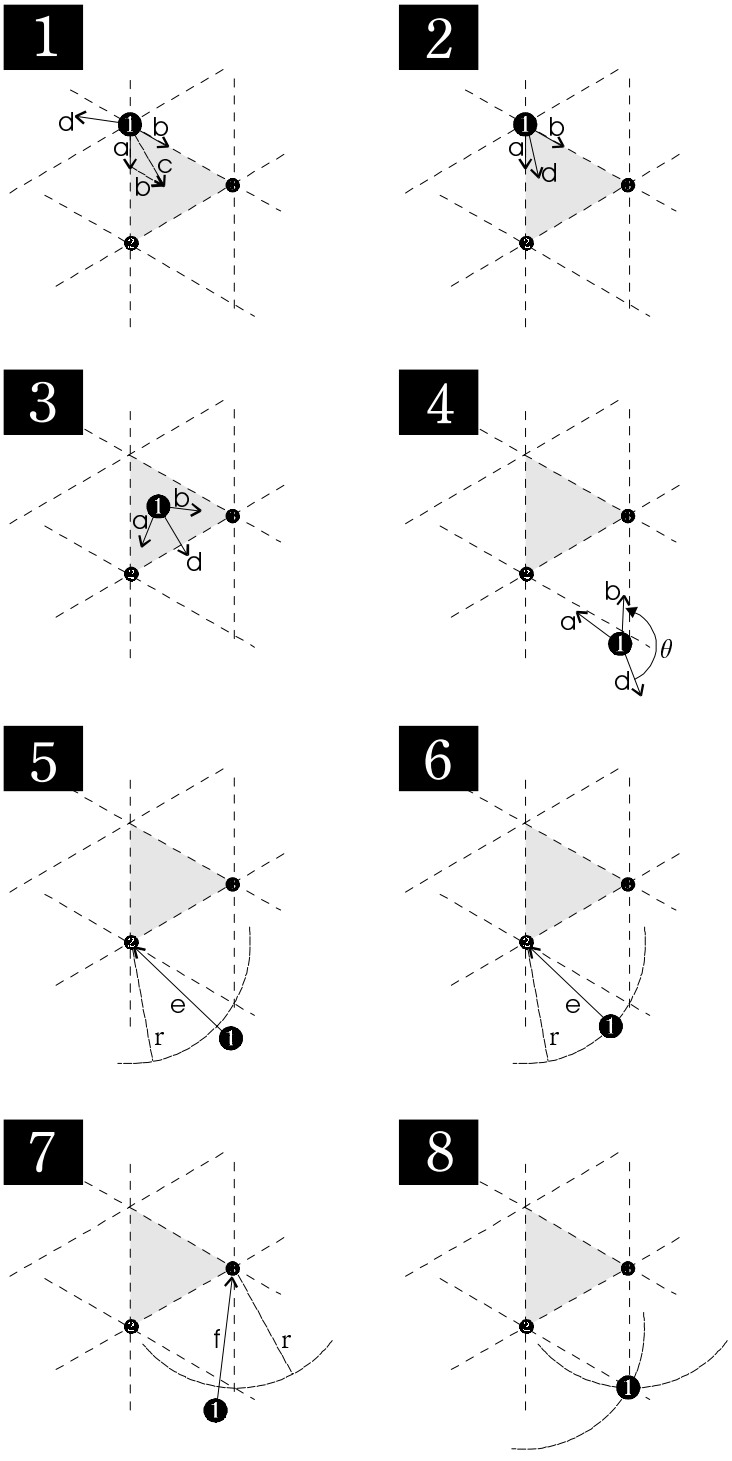
The Beacon agents implement two basic behaviours. The new basic behaviour, here called *waiting*, describes the act by the Beacon when it is sitting doing nothing. The other Beacon agent behaviour is *homing*, to move the agent to its new reference position when the Mapping agent signals it.

The *homing* behaviour has itself been implemented as a composite behaviour, as its realisation requires the temporal combination of several simpler behaviours, each using different sensors, by use of a finite state machine.

The Beacon agent sits in the *waiting* state until it receives a **SIG_MOVE** signal from the Mapping agent.

It then switches to the composite *homing* behaviour shown in Figure 6 and attempts to establish its new reference position by orienting itself so that it is pointing between the other two Reference agents. It is then positioned to move to the new goal. By moving forward whilst keeping the angles measured to each of the other Reference agents equal on both sides, the Beacon agent intersects the line between the other Reference agents and moves to the other side of this line. Given that the agent is keeping the angles on both sides approximately equal, when the angle exceeds 150°, the agent should be near to its goal. Theoretically, if the agent traced a perfectly straight path from a point starting at exactly equal distances from the other reference agents, this would land it right on the goal. As this will never occur in reality, the *homing* behaviour now uses distance information from the more accurate ranging data to place itself at exactly equal distances from the other reference agents. This is done by alternately ranging to each of the other two Reference agents in turn and moving to establish a defined distance from each. In this way, the agent homes in on its ultimate goal position.

Finally, when the goal is reached, a signal is sent to the Mapping agent to notify it.



1. The Beacon agent (B1) is facing in direction **d**. Vectors **a** & **b** point at Reference agents R2 & R3 respectively. B1 receives a signal to move to a new position and so turns toward the vector **c** = **a** + **b**.
2. B1 orients itself to point between **a** & **b**.
3. B1 moves, attempting to keep the angles **a**△**d** & **b**△**d** equal.
4. B1 reaches a point where **b**△**d**>150° and switches to homing by use of distance measurement.
5. B1 measures its distance to R2, |**e**| to be ≠r, the goal distance.
6. B1 moves along direction **e** until it reaches the goal distance, r. It then switches to measuring its distance to R3.
7. |**f**| is found to be ≠r, the goal distance.
8. After alternating measurements to R2 and R3 and moving, it reaches the goal position and returns to *waiting*.

Figure 6. Beacon agent composite homing behaviour implementation

```
State 1:
Waiting:
sit and wait for signal from TMappingAgent to move
if SIG_MOVE signal received then
  FsignalMove := FALSE
  state 2

State 2:
Homing 1:
move forward by d_forward
form 2 unit vectors pointing at other reference agents
sum the 2 vectors and turn to point that direction
if our direction vector is pointing between the 2 vectors then
  state 3

State 3:
Homing 2:
move forward by d_forward
turn to keep the angles to the other reference agents equal in magnitude
if the magnitude of one of the angles exceeds 150 degrees then
  Fgoal1Reached := FALSE
  Fgoal2Reached := FALSE
  state 4

State 4:
Homing 3:
move forward by d_forward
if distance to reference agent A is within goal distance range then
  if Fgoal2Reached then
    state 6
  else
    Fgoal1Reached := TRUE
    state 5
else
  Fgoal2Reached := FALSE
  turn toward goal distance

State 5:
Homing 4:
move forward by d_forward
if distance to reference agent B is within goal distance range then
  if Fgoal1Reached then
    Fstate 6
  else
    Fgoal2Reached := TRUE
    Fstate 4
else
  Fgoal1Reached := FALSE
  turn toward goal distance

State 6:
Send message:
send SIG_AT_GOAL
Fstate 1
```

Algorithm 2. Beacon agent behaviour algorithm finite state machine.

3.9 TMappingAgent class description

The **TMapping** agent class is descended from the **TAgent** class. Thus, it encapsulates all fields and methods of the **TAgent** class.

3.9.1 Fields

Fmap This is an object of type **TMap**. The **TMap** object contains an array of feature data values in the range 0 to 255, corresponding to a grey-scale level. It provides methods to add and retrieve data points and clear the map.

FcentroidX, FcentroidY

These fields hold the coordinates of the centroid of the current survey area. This is necessary to establish the sample region which is used to determine the survey ratio or region coverage.

F0mapX, F0mapY

These fields hold the coordinates in the World coordinate system corresponding to the zero coordinates of the internal map object.

FreferenceStopped

This is a flag which is used to indicate whether map data should be collected. It is **TRUE** when all reference agents are stationary and **FALSE** otherwise.

FmapState This field contains the state variable for the mapping controller finite state machine.

FnextRefAgent

This field contains a reference index to the next Reference agent to be signalled to move

3.9.2 Methods

execute, draw, signal

Although the code is unique to the **TMapping** agent, the descriptions for these methods are the same as for the **TBeacon** agent.

getMapRatio This method is provided to gain access to the current survey ratio or region coverage value of the sample region defined by the **Fcentroid** fields.

3.9.3 Reference agent control communication protocol

To control the movement of the Reference agents by the Mapping Agent, a simple protocol was devised. It contains two messages, **SIG_MOVE** and **SIG_AT_GOAL**. The **AgentManager** unit defines these messages and the **TAgentManager** class provides a method to control communication between agents.

When the Mapping agent wishes to signal a Roaming agent that is time to take up a new position, it sends a **SIG_MOVE** message with a parameter containing a Reference agent index value. It then stops collecting terrain feature data. The **TAgentManager** object sends the message on to the corresponding agent. Note that the indexed agent may be itself. This signals the Reference agent to move.

When the Reference agent reaches its destination, it sends a **SIG_AT_GOAL** message. The **TAgentManager** object sends the message on to the Mapping agent. When the Mapping agent receives the message, it resumes the collection of terrain feature data.

3.9.4 Mapping Agent Behaviours and Behaviour Selection

The **TMapping** agent class, being derived from the **TBeacon** agent class, encapsulates its behave method. In other words, since Beacon and Mapping agents are both types of Reference agents they behave identically.

3.9.5 Mapping Agent Mapping algorithm

The Mapping agent builds and maintains the terrain map and based on its internal state, controls movement of the Reference agents through the signalling protocol. The algorithm which controls all these functions is implemented as a simple finite state machine as part of the execute method, before the Mapping agent's behave method is called.

In addition, the mapping controller controls alignment of the finite internal map with the conceptually infinite terrain. It does this by calculating the centroid of the Reference agents' positions and aligning this with the centre of the internal map. Thus the internal map must always be large enough to completely cover the area of a circle passing through all three Reference agent positions.

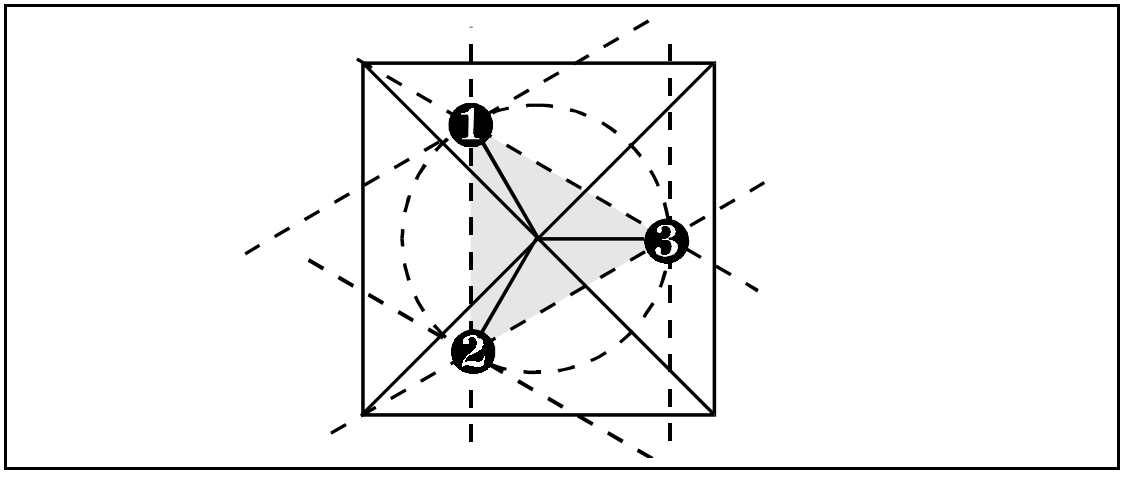


Figure 7. Internal map overlaying region being surveyed.

The Mapping agent signals the Reference agents when it determines that the current mapping ratio or region coverage value exceeds a threshold. To compute this value, a 10 × 10 region in the centre of the internal map is sampled. The value returns a ratio of the number of array elements in this region which have been written to since the map was last

cleared. For example, if 50 of the array elements in the 10 × 10 region have been written to, the value is 0.5. The **TMap** class provides a method to do this calculation.

When the Mapping agent overlays the internal map with the area being surveyed, this 10 × 10 region overlays the centroid of the Reference agent positions. Provided that the Roaming agent behaviour provides symmetric (not necessarily uniform) coverage of the region being mapped, this central region will be representative of the total region coverage.

```
FmapState 1:
no mapping is done in this state
if SIG_AT_GOAL signal is received then
  set up new map centroid coordinates
  set new 0 map coordinates to correspond with the current region being surveyed
  FmapState 2

FmapState 2:
add feature data from each roaming agent to the map
if current area has been adequately surveyed (mapRatio > threshold) then
  copy internal map to main map image
  clear the internal map ready for the next survey area
  send a SIG_MOVE message to the next reference agent
  update next reference agent index
  FmapState 1
```

Algorithm 3. Mapping agent mapping algorithm finite state machine.

3.10 User Interface

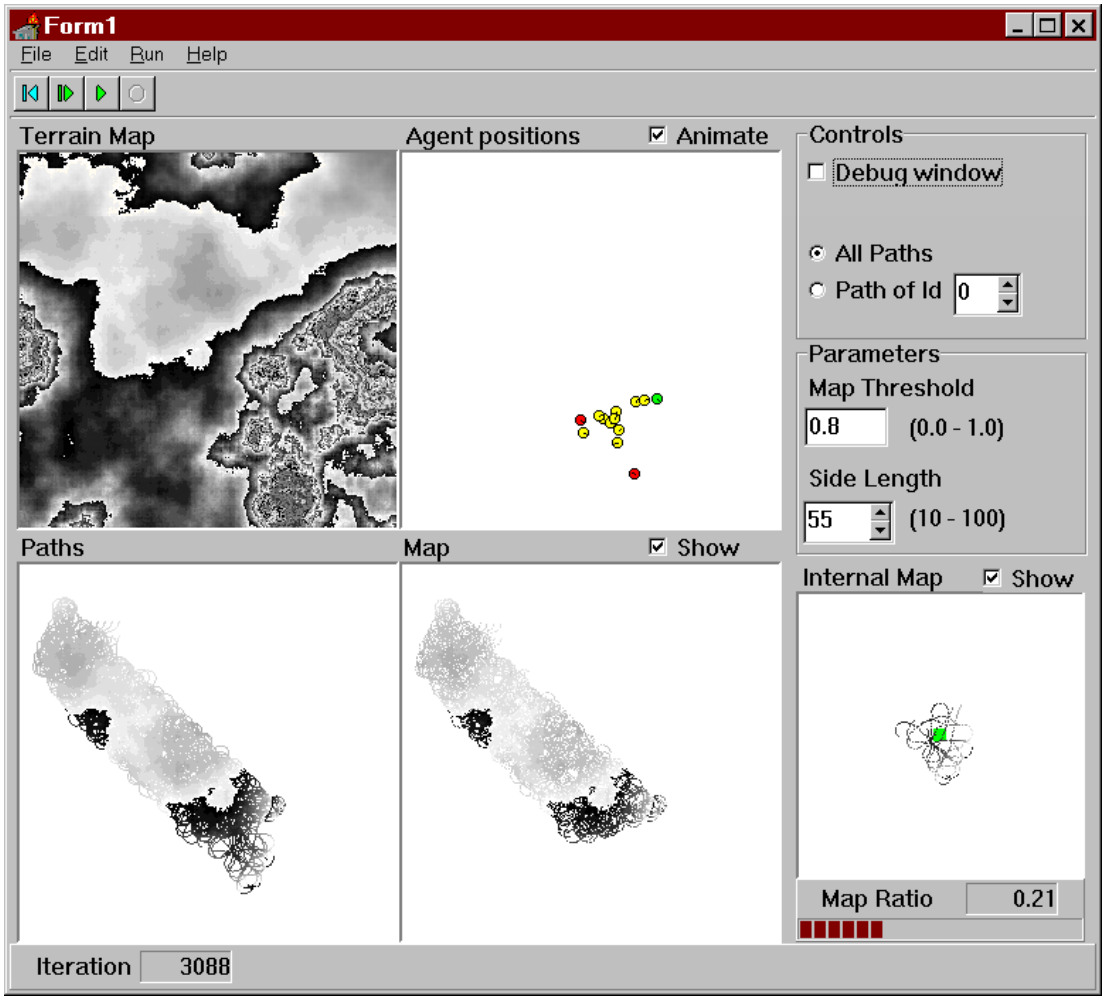


Figure 8. Simulator user interface window

The simulator has a standard window, icon, mouse, pop-up/pull-down menu (WIMP) based user interface.

The simulation environment is divided into 5 panes, a Controls group and a Parameters group. There is a control bar along the top and a status bar along the bottom.

The top left pane contains the terrain data. The raw terrain map data is provided by a square greyscale bitmap file, which can be created by any graphical paint package. This terrain map may be loaded from an external file using the mouse. The map shown is preloaded.

The top right pane shows the agent positions.

The bottom left pane shows the agent paths. The *All Paths* and *Path of Id* RadioButtons in the Controls group control what is displayed in the Paths pane. The default selection *All Paths* will cause the paths of all Roaming agents to be displayed modulated by the terrain map pixel value. If *Path of Id* is selected, the agent with the Id determined by the edit box will be displayed.

The bottom right *Map* pane shows the map data which has been accumulated by the mapping agent. Data from the internal map which is displayed in the *Internal Map* pane is copied to the *Map* pane when the survey of the current region has been completed.

The Parameters group contains the *Map Threshold* and *Side Length* settings. The *Map Threshold* setting controls when the Mapping agent signals reference agents to take up new positions in the terrain. A coverage value of 0.4 requires that 40% of the pixels covering the region of the internal map represented by the lime green square in the *Internal Map* pane have been traversed by a Roaming agent. The *Map Ratio* value below the *Internal Map* pane shows the current coverage value and the progress bar below it reaches full when the coverage matches the threshold.

The *Side Length* setting determines the length of the side of the equilateral triangles which are used to tessellate the survey area. That is, the distance a Reference agent uses to establish its position in the terrain relative to other Reference agents. Note that setting this to less than half the value of the current distance between two reference agents when the third moves will mean that the third agent can never establish a reference position.

The buttons on the control bar along the top control the simulation execution. With these, the simulation may be reset, single-stepped, run or stopped.

The status bar along the bottom displays the current simulation iteration value.

3.11 Modelling Simplifications

Amongst the many simplifications made by the simulator, the following are of particular note:

- Agents are point-like and are hence not subject to collisions.
- Agents are point-like and are hence not subject to obscuration. ie. the beacons and Mapping agents are always visible.
- There are no other obscuring features in the landscape being studied. Any differences in altitude between agents are accounted for by the optical sensing system. Assuming that the agents can always travel along the path desired, or if not, that the behaviour could be modified to ensure agents do not become stuck.

- Angle data sensed by Roaming agents is only used locally to implement selection between the *safe-wandering* and *homing* behaviours. It is not transmitted to the Mapping agent for it to use to determine the position of the Roaming agent using resection. Thus, no association of the Mapping Agent/Beacon is made with its sensed angle data. In reality, for the Mapping Agent to be able to calculate the Roaming agent's position, this association must be made, so that angles are associated with the correct beacons.

4 Simulator Results

4.1 Illustration of agent dynamics

To illustrate some aspects of the simulator, the following extracts showing agent dynamics are included. The sequence proceeds from left to right across the page and shows a robot survey team with a survey area side length of 100.

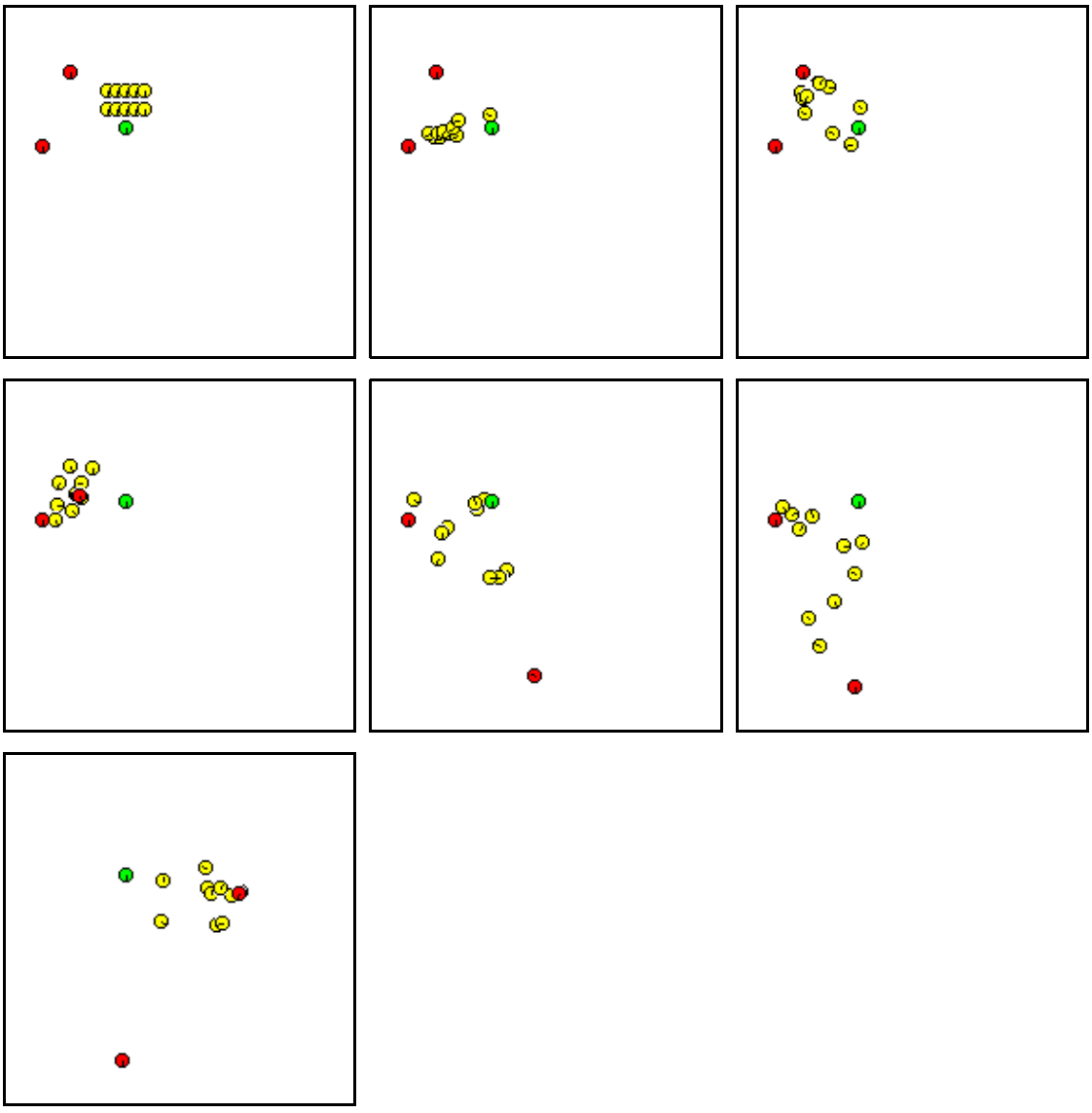


Figure 9. Agent motion series.

If the frames are numbered in sequence from 1 to 7, frame 1 shows the initial positions of all members of the robot survey team. The three agents in triangular configuration are the Reference agents. The two leftmost of these are the Beacon agents. The agent on the

bottom right is the Mapping agent. The ten remaining agents in military configuration are the Roaming agents.

Frame 2 shows the team just after the simulation has been started (Iteration 35). Note that the Roaming agents, which initially started in a tight formation have quickly diverged.

Frame 3 is just after mapping of the first survey region has been completed. The first Reference agent has been signalled but has not yet moved (Iteration 131).

Frame 4 shows the signalled Reference agent executing its *homing* behaviour (Iteration 160). The Mapping agent has stopped collecting feature data from the Roaming agents. Note that most of the Roaming agents, upon finding themselves outside the moving region defined by the Reference agents, are executing their *homing* behaviours and are already moving to the new region.

Frame 5 is the point at which the moving Reference agent has stopped using range data to establish its distance from the leftmost Beacon agent and has switched to ranging to the Mapping agent (Iteration 285).

Frame 6 shows the Reference agent in its *waiting* state, having reached its goal by alternately ranging to the other Reference agents (Iteration 392). The Reference agent positions visibly lie at the corners of an isosceles triangle. When it reached the goal, the agent sent a **SIG_AT_GOAL** signal to the Mapping agent. The Mapping agent then resumed accumulating data from the Roaming agents.

Frame 7 shows the second Reference agent moving to establish its new position, having received a signal from the Mapping agent.

4.2 Effect of angle sensor disturbance on the terrain map

Although the code is not included in the simulator, an investigation was done to examine the effect of errors in position determination by the Roaming agents. To model the errors expected to be present in the angle data tuples returned by a real robot, a small amount of random noise was simulated. The following figures illustrate the effects of adding different amounts of noise. This was done by modifying the **TAgentManager getRData** method, which normally returns the actual position vector of a Roaming agent, to randomly perturb the vector. This was intended to include the effects due to limited resolution of the angle measuring sensor. The map building technique averages multiple feature data values which coincide spatially in the internal map. To obtain the following terrain maps, a map ratio of 0.98 was entered. The area side length value was 55.

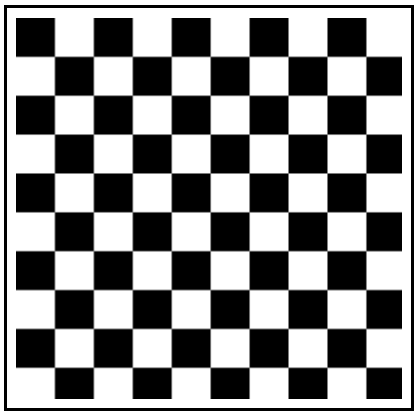


Figure 10. Terrain data used to illustrate the effect of position errors.

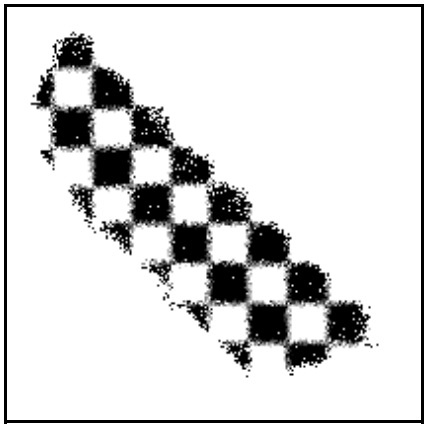


Figure 11. Blurring of the internal map with a position error range of 4.0

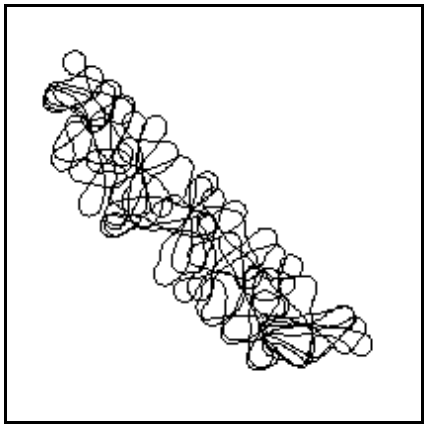


Figure 12. The path followed by a Roaming agent in forming the terrain map.

Figure 12 illustrates the path followed by a single Roaming agent in the normal execution of its behaviour set.

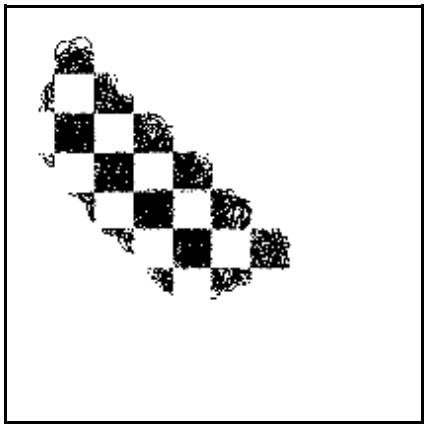


Figure 13. Blurring of the internal map with a position error range of 2.0

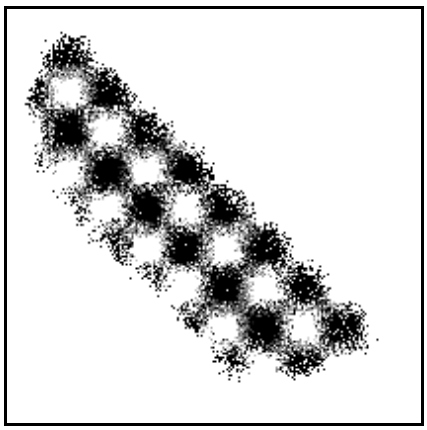


Figure 14. Blurring of the internal map with a position error range of 10.0

Although the images may not reproduce well here, it can be seen that the brightness values of the terrain data map are averaged with the nearby values. The effect is similar to convolving the terrain data with a two-dimensional low-pass Gaussian filter mask. The terrain map building technique successfully incorporates the disturbed position data to give a slightly degraded terrain picture, which maintains the integrity of features at scales above that of the error component. If the feature map is intended to show the presence or absence of specific elements of the terrain, instead of provide a continuous value map, it is possible to apply a threshold function to the map data. This has been done in Figure 15 for illustration purposes. The simulator does not perform this function.

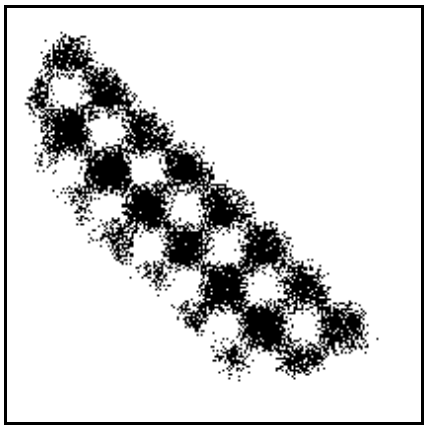


Figure 15. Thresholded internal map with a position error range of 10.0

4.3 Paths taken by agents with different roles

4.3.1 Roaming agent paths

The next series of frames illustrates the motion of a Roaming agent as it collects feature data.

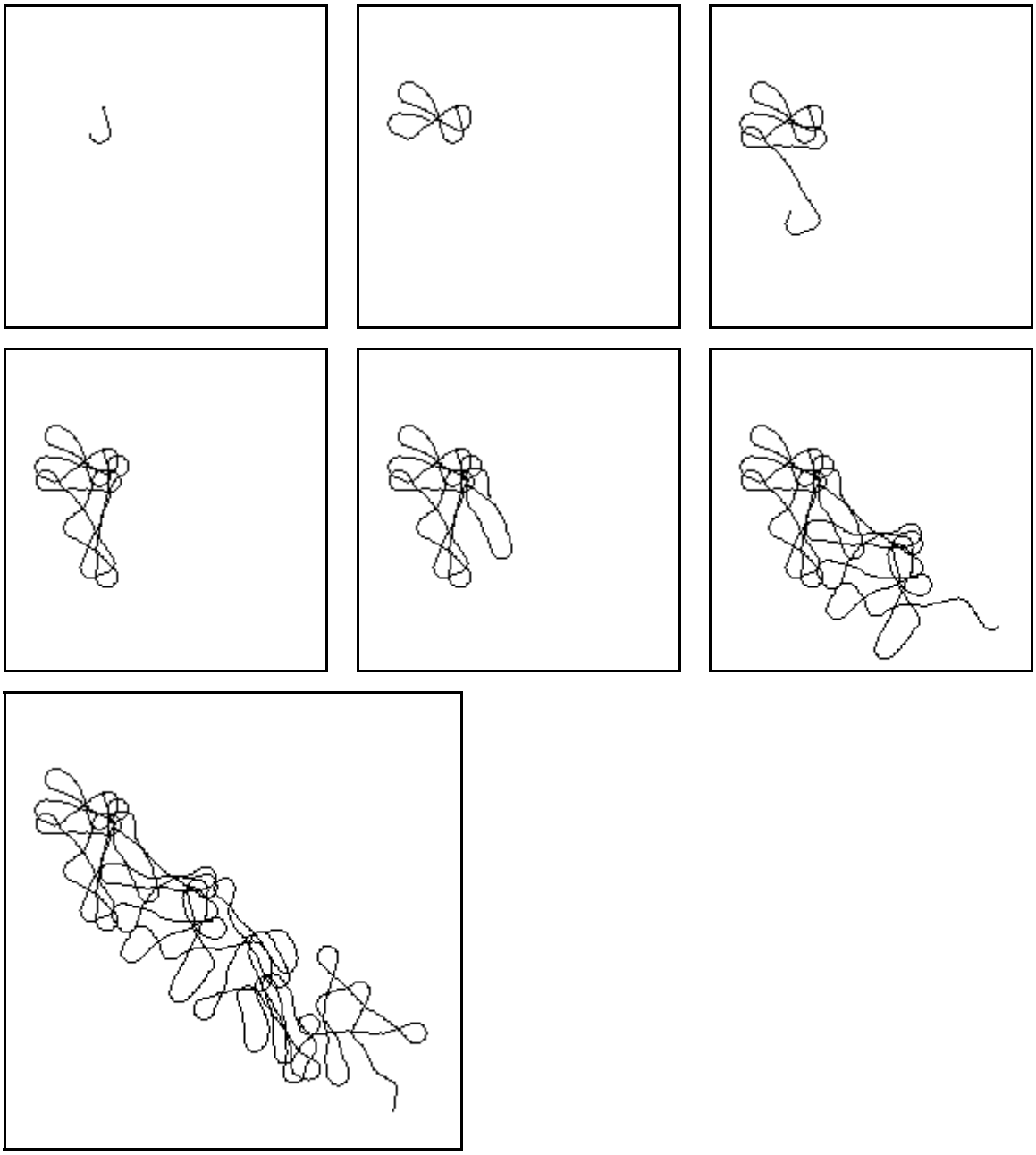


Figure 16. Roaming agent path evolution for a survey simulation run

The simulation parameters for this run were; Roaming agents = 10, side length = 70, map ratio threshold = 0.6.

If the frames are numbered in sequence from 1 to 7, frame 1 shows the path travelled by the Roaming agent soon after starting.

In frame 2, the survey of the first region has just been completed. In frame 3 the agent has moved into the new region in response to the Reference agent move. Frame 4 shows the situation when the survey of the second region has just been completed. The remaining frames show how the path continued to evolve throughout the simulation run.

To illustrate the effect of varying the side length, the following two simulations were run.

The following figures illustrate the path that a single Roaming agent follows for the two runs. The initial conditions for each run were identical except for the side length. The simulation parameters for this run were; Roaming agents = 10, side length = 100 then 55, map ratio threshold = 0.8.

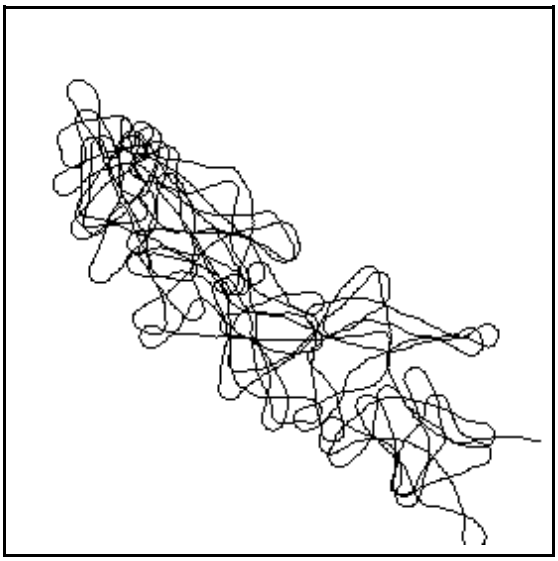


Figure 17. Roaming agent path with a side length of 100

Some features of the path in Figure 17 are identifiable. A concentration of tight curves along an axis indicates that this axis marked a region boundary. The density of the path overlap is reasonably uniform indicating that the composite behaviour which has been designed for the Roaming agent should lead to uniform coverage of the terrain, given that there is no agent interaction.

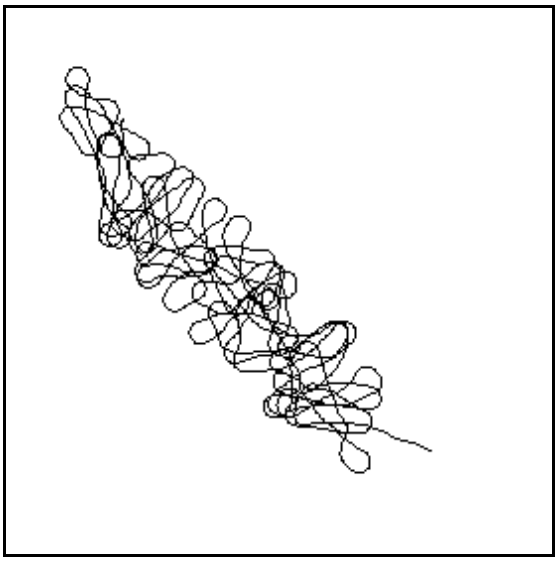


Figure 18. Roaming agent path with a side length of 55

It is notable from Figure 18 that a greater percentage of tight loops exist in the total path. This is due to the finite turning radius of the Roaming agent. The greater the side length, the less time is spent executing the *homing* behaviour as a ratio of total time.

4.3.2 Reference agent paths

Figure 19 shows the paths of the three Reference agents executing a simulation with a survey area side length of 100.

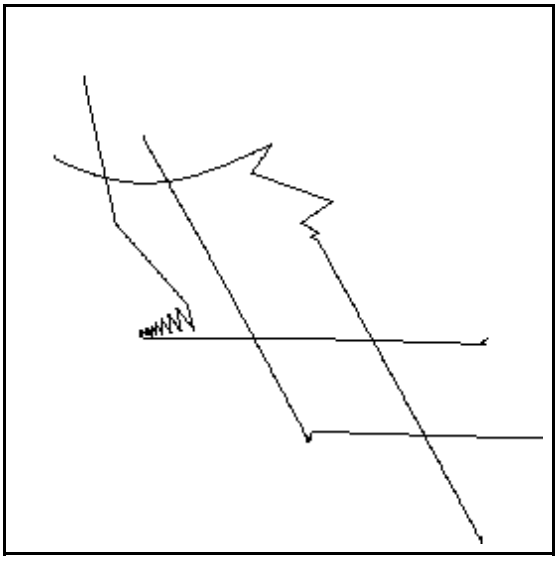


Figure 19. Reference agent paths with a side length of 100

The first Reference agent to move is the topmost in the environment. It can be seen that it travels along a relatively straight path until the measured angle to one of the other initial Reference agent positions is 150°. At this point, it takes a sharp left turn, switching to the range-based *homing* behaviour states. It travels in a straight line away from the other one of the other initial Reference agent positions until it reaches the goal distance. Finally, it zig-zags along, alternating ranging to the two Reference agents until the goal region is reached. At this point it stops.

The second Reference agent to move is the leftmost. It can be seen that, since it starts much closer to one Reference agent than the other, by keeping the angles to each equal, a curved path if followed, which should be a rough circular arc. This arc is followed by a short, straight section and then the zig-zagging behaviour of the first agent is repeated.

Now, all three agents are in equilateral-triangular arrangement, so the third Reference agents *homing* behaviour proceeds efficiently to place it at its goal with a minimum of range-based states needing to be executed.

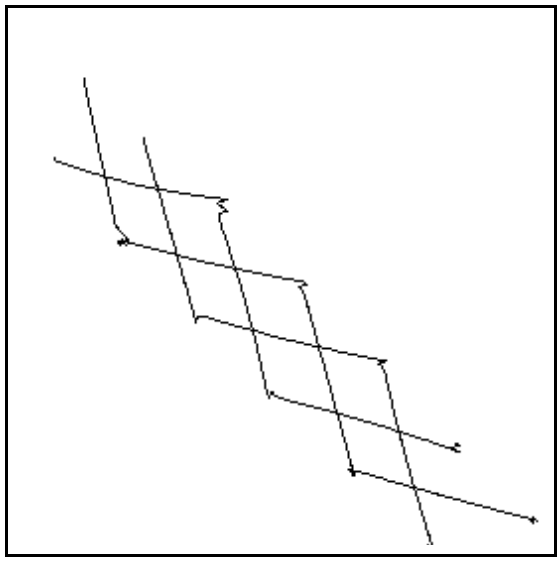


Figure 20. Reference agent paths with a side length of 55

The initial configuration of the Reference agents in Figure 20 was the same as that in Figure 19. In this case, since the agent spacing was of the same order as the defined side length, the Reference agents *homing* behaviours are efficient from the outset.

4.4 Analysis of simulation result repeatability

The first aspect of the simulation results to be analysed was the repeatability of results of simulation runs from a given set of starting conditions. The following graph plots three runs

with the same initial configuration, which is the same as that shown in Figure 21. The simulation parameters for these runs were; Roaming agents = 10, side length = 55, map ratio threshold = 1.0.

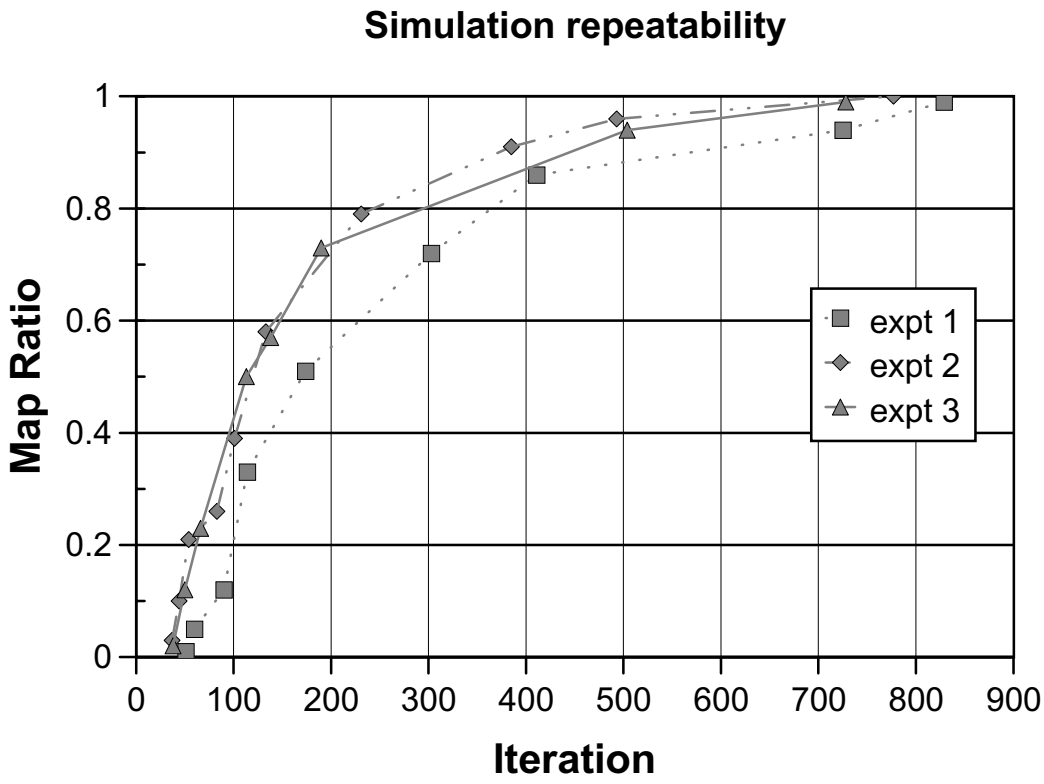


Figure 21. Repeatability of simulation results.

This chart demonstrates that the various experimental runs were highly repeatable. It is therefore reasonable to assume that variations in subsequent results are due to the controlled changes in the parameters in different simulation runs and not due to variation introduced by the simulator.

4.5 Survey area coverage times with a given team size

A comparison was made of the time taken for the robot survey team to cover survey areas of different sizes.

The following experiment was performed by allowing at least two Reference agents to move to new positions, ensuring an equilateral triangular arrangement. All experiments from now on were also allowed to establish these initial conditions before the experimental data were collected. The iteration axes were corrected for zero offset by subtracting the starting iteration value. Note that because of this subtraction, all results shown may suffer

from a slight x-axis offset error. In order to avoid this, an absolute 0 iteration point would have to be chosen, such as the point at which the first Roaming agent encroaches upon the mapping ratio sample region.

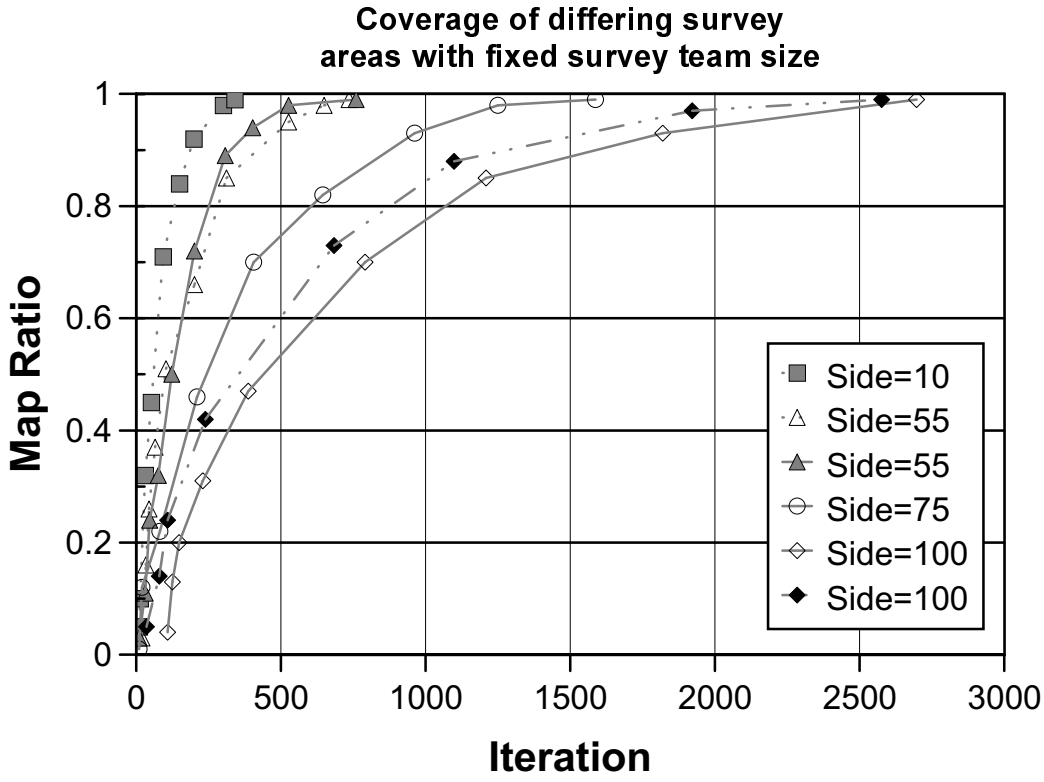


Figure 22. Area coverage vs time with 10 agents for varying areas.

A triangular area defined by sides of $100\times100\times100$ is $10^2=100$ times bigger than a $10\times10\times10$ area. An area of $100\times100\times100$ is $2^2=4$ times bigger than a $50\times50\times50$ one. An area of $100\times100\times100$ is $(100/55)^2=3.3$ times a $55\times55\times55$ area.

Given these factors, it might be expected that the time to survey an area could be found by multiplication of a standard time by this factor. The results show that this relationship seems to hold except when the area becomes small. The time taken to survey the $10\times10\times10$ area is greater than would be expected. The reason for this is that the Roaming agent model implements a minimum turning radius as shown in Figure 18, which means that, the smaller the area, the greater the percentage of time is spent executing the *homing* behaviour, extending the time.

4.6 Fixed area coverage times with a changing team size

The next experiment looked at the coverage versus time, but this time the region size was fixed and the number of Roaming robots was varied. It was expected that, since the agents do not interact, the time to achieve a given coverage would be inversely proportional to the time take by a single agent.

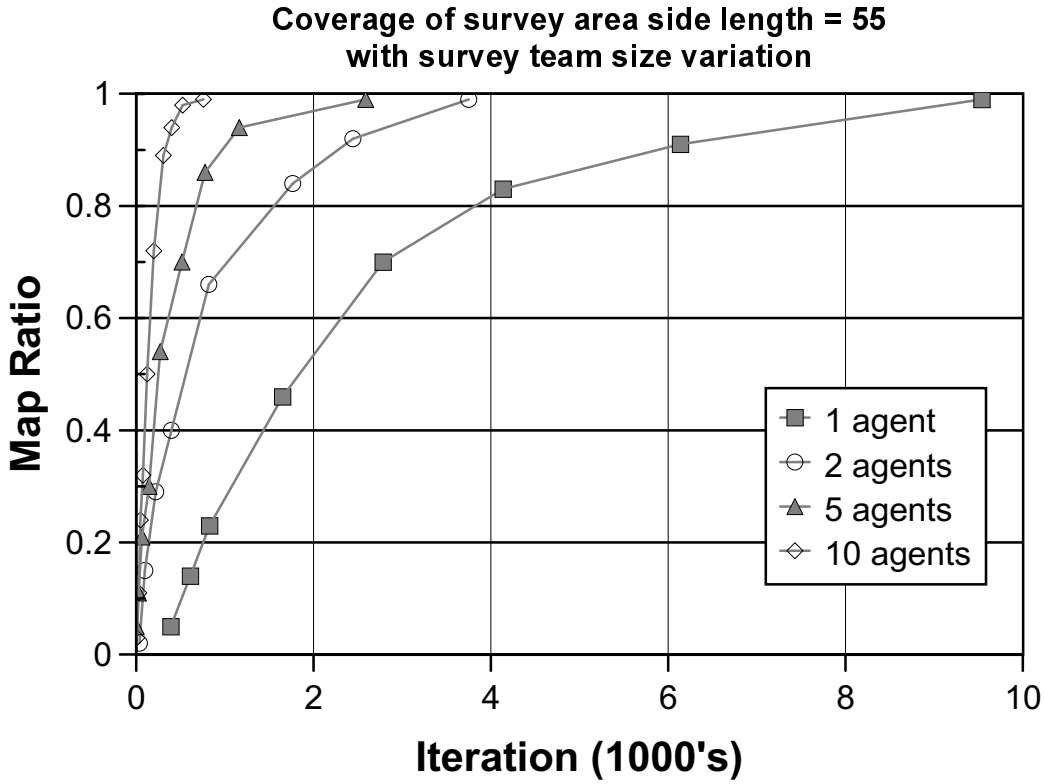


Figure 23. Area coverage vs time with variation in team size

The results show that there is a diminishing return in terms of the cost of extra agents added to the environment versus the improvement in survey time. That is, the addition of one agent to take the number from one to two gives a substantial improvement, more than doubling the survey rate. The subsequent addition of another three agents has a lesser effect on survey time and so on.

The results show that some inverse relationship with survey time seems to exist, although it does not seem to be in direct proportion. The same data has been plotted on a logarithmic x-axis, to clarify the relationship. This is done in Figure 24.

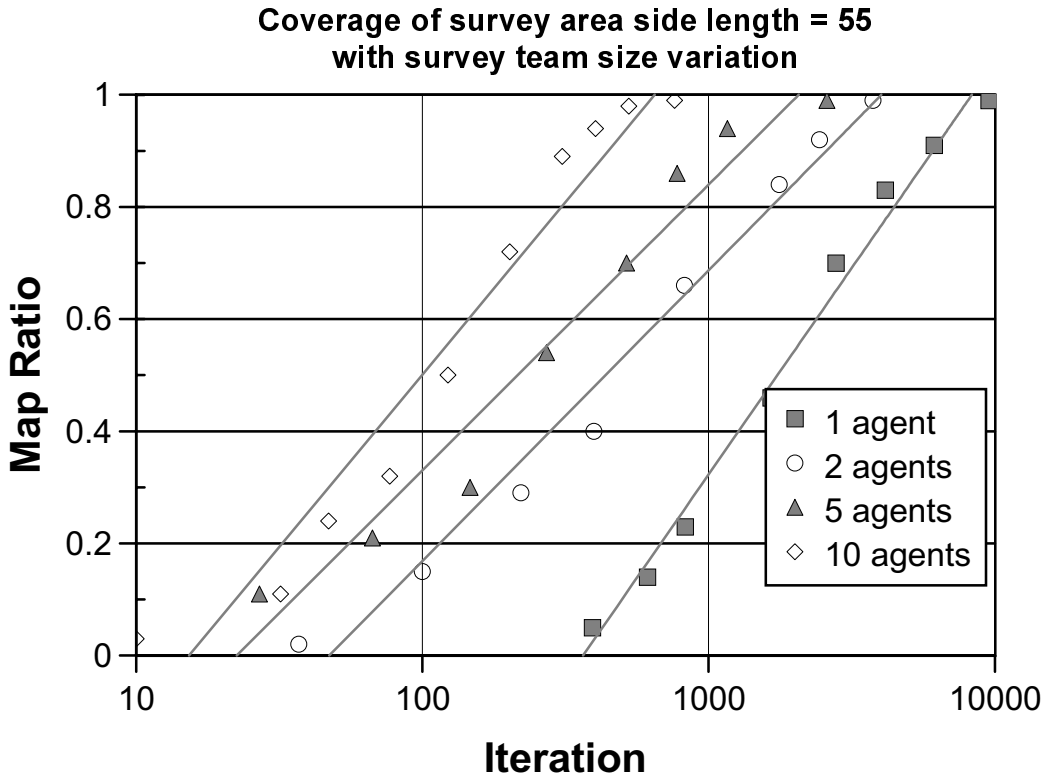


Figure 24. Area coverage vs time with varying team size (log plot).

The graph actually shows that the coverage is not exactly logarithmic over time, each curve exhibiting a slightly sigmoidal appearance with respect to the fitted curve. If the coverage was logarithmic and the time to achieve a given coverage was inversely proportional to the time take by a single agent, the fitted curves should have equal gradients and equally spaced intercepts. This seems to be approximately true, although more experimental results are probably required to demonstrate it.

It is expected that this relationship would change significantly if cooperation between Roaming agents was implemented by addition of the *dispersion* behaviour.

Results of this experiment repeated with a smaller survey area side length are included in Figure 25 and Figure 26.

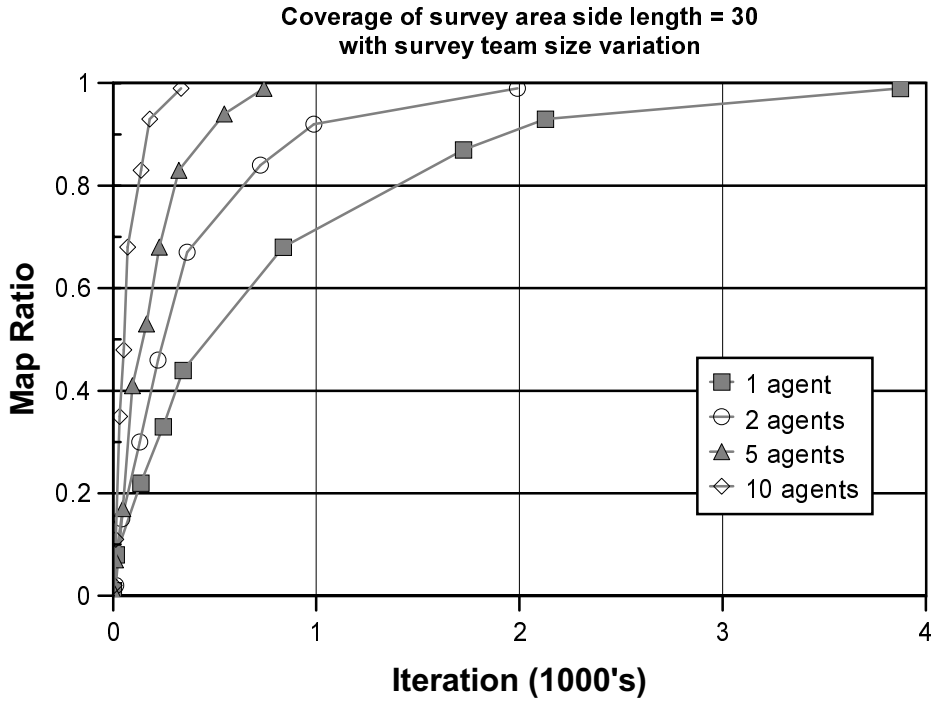


Figure 25. Area coverage vs time with varying team size. Side = 30

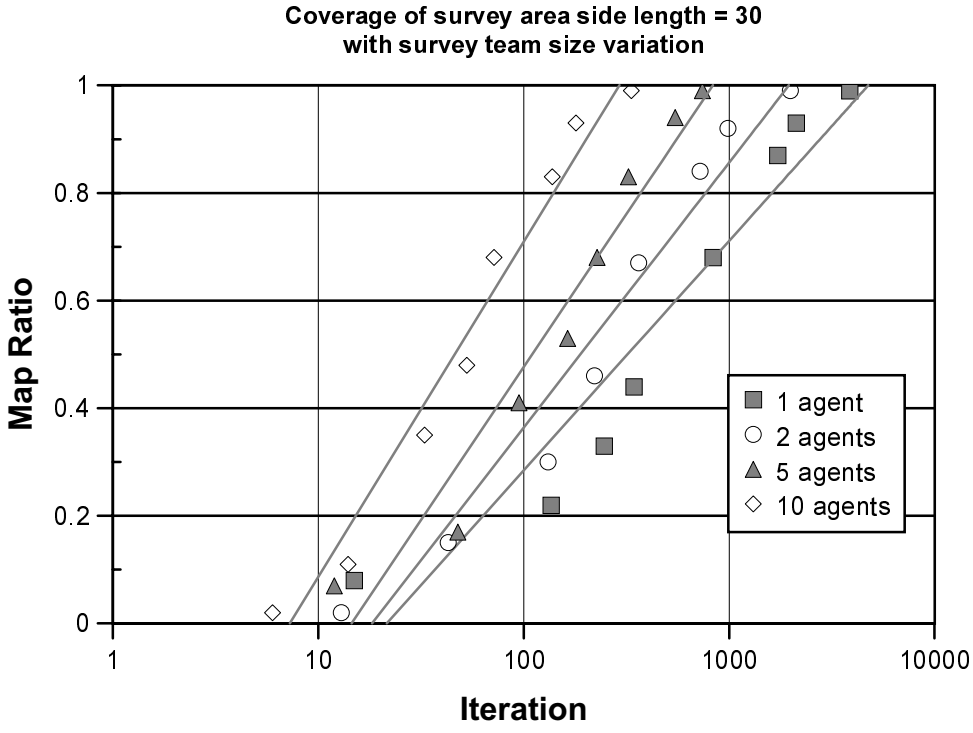


Figure 26. Coverage vs time varying team size (log plot). Side = 30

5 Further Work

This terrain mapping simulation may be extended to account for real-world conditions in the following areas:

- Modelling physical extent of agents.
- Obscuration of beacons.
- Addition of error models to sensors and effectors
- Accountability for additional data to reduce sensor error.

It may also be extended to examine improvements in task performance through:

- Addition of *dispersion* to Roaming agents' behaviour set.

5.1 Modelling physical extent of agents

Agents were modelled as points. Physical extent was ignored to speed and simplify the simulation. Including the modelling of extent in the agent model would identify problems which have thus far been ignored, such as obscuration of agents by each other.

A practical system would, by definition, require support for problems due to extent. This would involve additional sensors; the simplest being a bump sensor to detect direct contact with other agents. As discussed below, if an additional *dispersion* behaviour was included, requiring extra sensors, it may not be necessary to detect direct contact.

5.2 Obscuration of beacons

In the real world, beacons may be obscured for several reasons. These include

- Blind spots endemic to the sensor.
- Obscuration by obstacles in the environment, including other agents with physical extent.
- Obscuration by the environment itself.
- Distance limits of the sensor.

To account for obscuration by the environment itself, the Beacon agents could supplement their behaviour, say by searching for local terrain altitude maxima to mimic the action of surveyors who establish a reference grid by placing reference markers on hills. This would modify the perfect equilateral triangular regions and cause the terrain strip being surveyed

to diverge from a straight path. However, the Mapping agent could keep track of this divergence and correct for it by modifying its goal position during its own movement phase.

Increasing the number of Beacon agents to minimise obscuration would have the added benefit of providing redundancy. It is likely that any practical system would require support for additional Beacons.

Clearly, moving Beacons must always ensure that the Mapping beacon remains visible from their final reference-establishing position. In some cases, say in the case where a Beacon robot must travel through a valley to reach its new position, it may be possible that the Mapping robot cannot be kept in sight during the traverse to the new position. Special algorithms to support this could be developed.

Obscuration of beacons from Roaming agent positions results in incomplete angle tuples. Where redundant beacons exist in the system, an obscured Beacon will cause an angle tuple to Beacon matching problem. That is, some algorithm must be devised by the Mapping agent to try to match reduced sets of angle data with the anonymous beacons. One solution here is to remove the anonymity by increasing the Roaming agents' sensor complexity so that angle data is associated with an identified Beacon. This would account for the converse problem of detection of phantom Beacons.

5.3 Addition of error models to sensors and effectors

Real-world embodied agents or robots suffer from noise and resolution limitations in sensor data. This can manifest as small perturbations in the data or as gross errors resulting in spurious values. Although the behaviours were designed to account for small random errors, they may need extension to account for spurious data. Errors in the received signal messages can effectively be ignored if the communications system is built on an error-detecting or correcting protocol. However, errors from detection of extra Beacons or obscuration cannot be ignored.

The effectors, in our case drive mechanisms, will interact with the environment through slippage. This may become important if localised cooperative behaviours are added. For example, to develop robust collision avoidance algorithms may rely on inertia and slippage being modelled.

5.4 Accountability for additional data to reduce sensor error.

Increasing the number of Beacon agents to minimise obscuration also permits the extra data sent by the Roaming agents to be used to reduce errors. Although the distance measuring sensors are very accurate, the coordinate system can drift as tiny errors

accumulate in the sensor readings. In a practical system, which surveys a more generalised area than a single strip of terrain, it would be possible leave a Beacon agent at one position while the rest of the members of the robot team venture out to survey the area. As small tiles of the larger tessellated area are surveyed, the Mapping agent would bring the team back within distance measuring range of the still stationary Beacon. At this point, accumulated errors could be measured.

5.5 Addition of Dispersion to Roaming agents' behaviour set

Improvements to the efficiency of carrying out the mapping task may be gained by adding the basic *dispersion* behaviour.

Dispersion acts as a repulsive force between agents and should improve the time to map out a given area. As the surveying task currently benefits from multiple agents revisiting the same point by providing extra data to minimise raw data errors statistically, application of the *dispersion* behaviour to this task should be careful not to negate this aspect by careful selection of the $\delta_{disperse}$ distance threshold value.

The reason for not including *dispersion* as part of the initial investigation was mainly that time constraints limited the scope of the study. Also, additional sensors are required to support its implementation and these would have to be built into the model. A physical robot which implements *dispersion* requires sensory information about the positions of agents in its immediate vicinity.

The dispersion algorithm could make use stigmergy to improve the efficiency of covering the region being mapped by marking traversed areas in some way. However, given that the likely application areas where this approach is aimed include diverse environments, modification of the local environment may be difficult or inappropriate.

If *dispersion* is added to the basic behaviour set of Roaming agents, it will have to be combined directly with the temporally-combined *safe-wandering* and *homing* behaviours currently implemented. This can be done in two ways. Either the direct combination of *safe-wandering* and *dispersion* behaviours can be temporally combined with the *homing* behaviour or the temporal combination of *safe-wandering* and *homing* behaviours can be directly combined with the *dispersion* behaviour.

That is, in graph notation

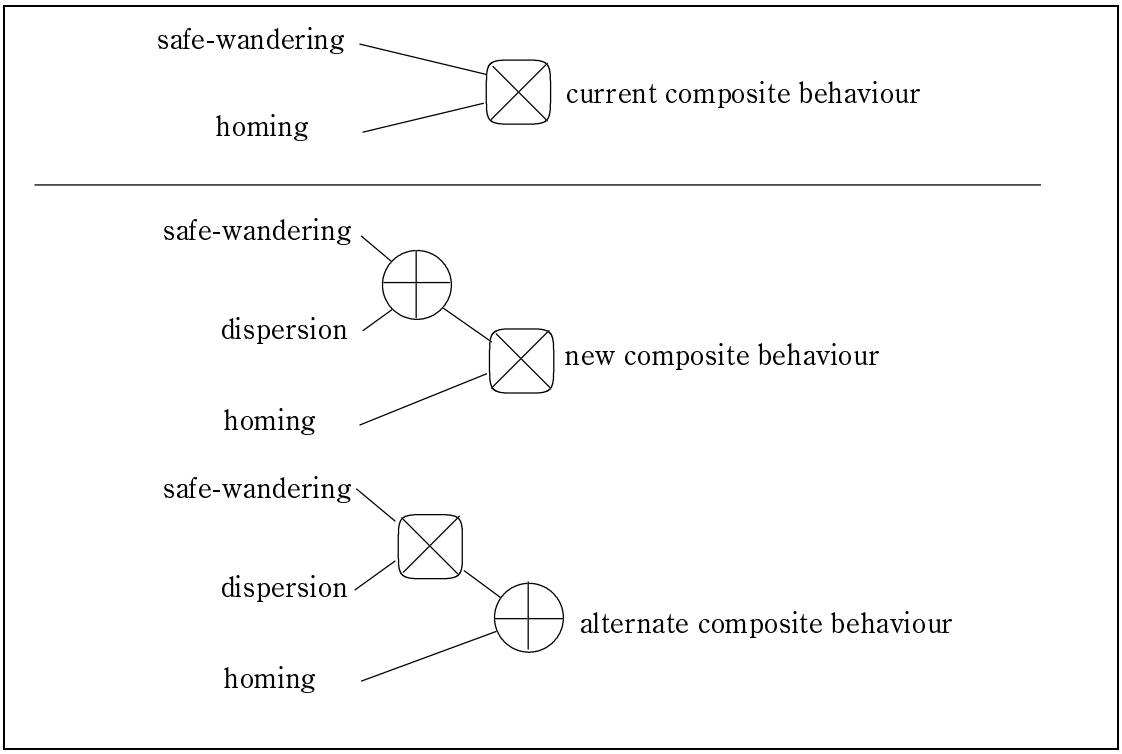


Figure 27. Combination options for composite Roaming behaviours

It is probable that both methods of combination would result in a similar quantitative behaviour.

An alternative to this might be for the Mapping agent to calculate the *dispersion* vectors for all agents and communicate them back to each agent. This places an additional computational burden on the Mapping agent and requires that a reverse communication channel be established.

6 Summary

A swarm robotic approach was successfully applied to the task of terrain mapping in a simulation environment.

To achieve this, the necessary subtasks were identified. These were

- Define the current area to be surveyed
- Move the group to the area
- Build a map of the defined area

The behavioural approach to swarm control was applied. In order to simultaneously achieve the subtasks, a heterogeneous robot team was assembled. Three types of member agents carried out different aspects of the task. The agent types were Roaming, Beacon and Mapping robots. These performed the following tasks. The Roaming agents performed retrieval of feature data from the terrain. The Beacon agents, along with the Mapping agent, established a coordinate reference system for the task. They also allowed relative position information to be sensed by Roaming agents for association with sensed terrain feature data. The Mapping agent assembled the data passed back to it by the Roaming agents into a coherent terrain map.

Coordination of the team members was achieved through the assignment of appropriate behaviours to different agent types. High level management and coordination of the robot team was assigned to the Mapping agent, as it collated the information on which management decisions could be based.

The object models which simulated the agents were described in detail. Behaviour algorithms for each agent type were described.

Finally, the ability of the team to achieve its designed task was assessed.

The cost of adding extra Roaming agents to the environment was found to have a diminishing return with respect to the improvement in time taken to survey the area. However, the extra data redundancy afforded by these agents and its incorporation into the final terrain map may justify their addition.

There are many areas in which further work could enhance the surveying approach. The inclusion of short-range sensing by Roaming agents would allow the basic *dispersion* behaviour to be added to its behaviour set. This is expected to lead to an improvement in survey time.

The incorporation of physical extent to the agent model would allow a number of interesting aspects to be investigated. Sources of real-world errors such as obscuration of beacons by agents could then be included in the simulation. Additionally, obscuration of Reference agents by features in the terrain could be included in this. The associated problem of navigation around obstacles inhabiting the survey region could also be investigated.

Appendices

Appendix 1. Resection

The angles measured from a survey site to a set of at least three targets of known position may be used to determine one's own position. This technique is known as resection [BaRa84]. There are two methods, the direct method and Tienstra's method.

Direct method

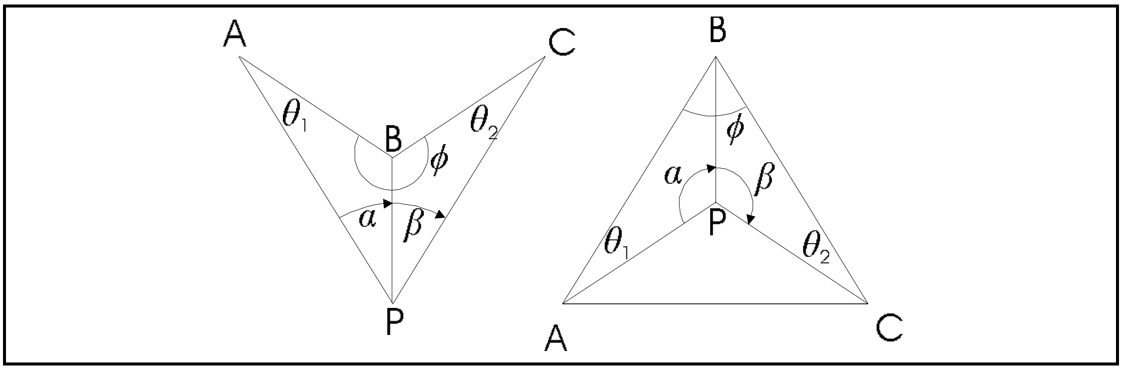


Figure 28. Labelling of vertices and angles for the direct method.

Given a labelling of the vertices and angles of a surveyor at point P, with known measured values for the coordinates A, B and C and angles α and β,resection analysis by the direct method proceeds as follows:

By the sine rule,

$$\frac{BP}{\sin\theta_1} = \frac{AB}{\sin\alpha} \quad \Rightarrow \quad BP = AB\frac{\sin\theta_1}{\sin\alpha}$$

and

$$\frac{BP}{\sin\theta_2} = \frac{BC}{\sin\beta} \quad \Rightarrow \quad BP = BC\frac{\sin\theta_2}{\sin\beta}$$

Hence

$$AB\frac{\sin\theta_1}{\sin\alpha} = BC\frac{\sin\theta_2}{\sin\beta} \quad \ldots\ldots \quad (6)$$

Also,

$$\theta_1 + \theta_2 = 360° - \phi - a - \beta \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (7)$$

Since a and β are known from measurement and AB, BC and ϕ may be calculated from the coordinates of A, B and C, we are left with two equations in the two unknowns θ_1 and θ_2. After solving for θ_1 and θ_2, all remaining angles and lengths may be deduced.

Tienstra's method

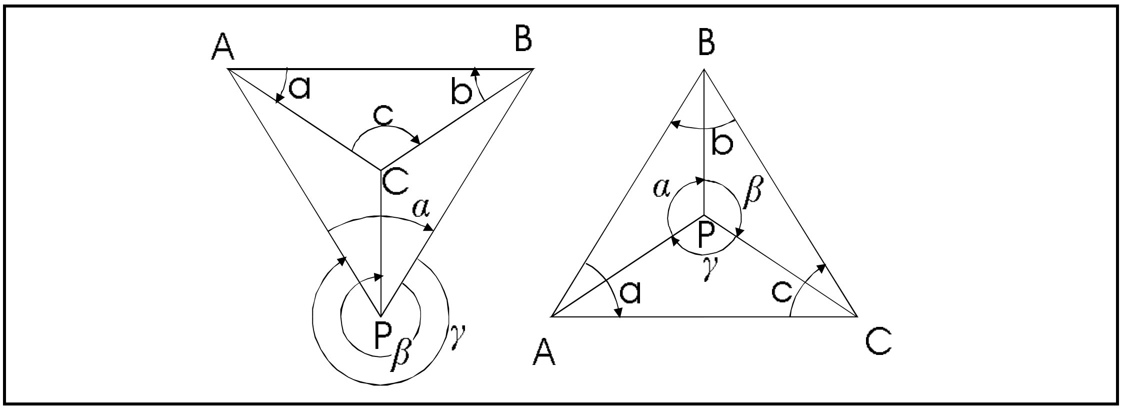


Figure 29. Angle and vertex labelling for Tienstra's method.

A much simpler resection method is attributed to Tienstra. The labelling above and the fact that all angles are measured in a clockwise direction are important to this method.

$P(E_P,N_P)$ may be found given the coordinates of $A(E_A,N_A)$, $B(E_B,N_B)$ and $C(E_C,N_C)$ and the measured angles a, β and γ.

First the three factors K_1, K_2 and K_3 are calculated from:

$$K_1(\cot a - \cot \beta) = 1$$

$$K_2(\cot b - \cot \gamma) = 1$$

$$K_3(\cot c - \cot a) = 1$$

Calculation of a, b and c involves some trigonometric calculation.

Then the coordinates of P are calculated using:

$$E_P = \frac{K_1 E_A + K_2 E_B + K_3 E_C}{K_1 + K_2 + K_3}$$

$$N_P = \frac{K_1 N_A + K_2 N_B + K_3 N_C}{K_1 + K_2 + K_3}$$

Appendix 2. Simulator Listings

Listing Agent.pas

```
{
Terrain Mapping Simulator
Agent File

File:         agent.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This unit implements all Agent classes (object definitions).
              The base abstract class is called the Agent class.
              Its most important method is "execute". This method is invoked
              on each agent instance by the agents object, which is defined
              in the AgentManager unit, in order to execute the agent.

              Roaming and Beacon agent classes are directly descended from the
              Agent class. The Mapping agent class is descended from the
              Beacon agent class.

              The agent objects are instances of a polymorphically defined
              heirarchy. Because of this polymorphism, the agents may be
              executed, signalled and drawn by calling the one method name.

              A fuller description of the heirarchy and methods may be found
              in the thesis.
}

unit Agent;

{*************************************************************************}
interface

uses
  Vector, Map, Graphics, SysUtils, DebugUnit, Classes, Dialogs, AgentManager;

const
  AGENT_RAD = 4;
  AGENT_DIAM = AGENT_RAD + AGENT_RAD;
  { Limits in degrees Roaming agents may turn in each time step.
    Reference agents are not limited in their turning radius to aid their
    homing behaviours and allow them to accurately land at their goal.
    Keeping the side just > 1.0 means a reference agent seeking a new goal
    will land in the goal since its velocity is 1.0 }
  TURN_LIMIT = 10;
  REFERENCE_GOAL_SIZE = 0.6;  // half side length of reference agent goal region

type
  TAgent = class
  private
    Fid: Integer;           // A place for the Agents' Identification
    FlastPos: TVector;      // Last Position vector used by drawing routine
    Fstate: Integer;        // State variable
    Fsender: TAgentManager; // Used to store reference to the Sending object
                            // which will always be the agents object
    procedure behave; virtual; abstract; // How the Agent type behaves.
  public
    // Move the following fields to the private section later
    Pos: TVector;           // Current Position vector
    Direction: TVector;     // Unit direction vector

    constructor Create; virtual;
    destructor Destroy; override;
    procedure move;         // Moves the agent from its current position
                            // according to the direction vector
    procedure execute(Sender: TAgentManager); virtual; abstract;
                            // Called to execute the Agent.
    procedure draw(canvas: TCanvas); virtual; abstract;
    procedure reportDebug; virtual;
    function distanceTo(Sender: TAgent): Single;
    property id: Integer read Fid write Fid;  // Fid accessor property
  end;
```

```
TBeaconAgent = class(TAgent)
private
  Fangles: TList;
    { n-tuple of beacon/mapping agent angles anti-clockwise w.r.t. forward
      direction, where n is the number of other beacon and mapping agents. ie.
      all except for ourselves. A TList is a linked list object defined in
      Delphi's Classes unit. It can contain any type of object since it uses
      generic pointers to refer to objects in the list. In our case, we place
      simple variables of type Single in the list in order to make use of the
      list's Sort method and dynamic Add and Remove methods.
      The elements of our TList are referred to by the expression
      Single(Fangles.Items[m]^), where m is 0..n-1 }
  Fdistances: TList;
    { n-tuple of beacon/mapping agent distances, where n is the number of
      other beacon and mapping agents. ie. all except for ourselves.
      A TList is a linked list object defined in
      Delphi's Classes unit. It can contain any type of object since it uses
      generic pointers to refer to objects in the list. In our case, we place
      simple variables of type Single in the list in order to make use of the
      list's Sort method and dynamic Add and Remove methods.
      The elements of our TList are referred to by the expression
      Single(Fdistances.Items[m]^), where m is 0..n-1 }
  FsignalMove: Boolean;
    { this flag is set when the signal method is called with the correct
      signal type by the TMappingAgent to inform
      the Beacon Agent that it is time to take up a new position }
  Fgoal1Reached, Fgoal2Reached: Boolean;
    { these fields are used by the homing behaviour code to determine
      when the agent has reached its goal }
  procedure senseBeacons;
    { Procedure to sense angles w.r.t. forward direction. ie. this procedure
      updates the Fangles tuple with the current sensor data }
public
  constructor Create; override;
  destructor Destroy; override;
  procedure execute(Sender: TAgentManager); override;
                                      // Called to execute the Agent.
  procedure behave; override;         // How the Agent behaves.
  procedure signal(msg: sigType); dynamic; // called to signal to this agent
  procedure draw(canvas: TCanvas); override;
end;


TMappingAgent = class(TBeaconAgent)
private
  Fmap: TMap;                         // Map of current survey area
  FcentroidX, FcentroidY: Single;     // Centroid of current survey area
  FOmapX, FOmapY: Single;             // O coords of current survey area
  FreferenceStopped: Boolean;
  FmapState: Integer;                 // State variable
  FnextRefAgent: Integer;
        // Refers to next reference agent to signal via a signalMove message
  procedure updateCentroid;
public
  constructor Create; override;
  destructor Destroy; override;
  procedure execute(Sender: TAgentManager); override;
  procedure draw(canvas: TCanvas); override;
  procedure signal(msg: sigType); override; // called to signal to this agent
  function getMapRatio: Single;
end;
```

```
  TRoamingAgent = class(TAgent)
  private
    Fangles: TList;
      { n-tuple of beacon/mapping agent angles anti-clockwise w.r.t. forward
        direction, where n is the number of beacon and mapping agents. A TList
        is a linked list object defined in Delphi's Classes unit. It can
        contain any type of object since it uses generic pointers to refer to
        objects in the list. In our case, we place simple variables of type
        Single in the list in order to make use of the list's Sort method and
        dynamic Add and Remove methods.
        The elements of our TList are referred to by the expression
        Single(Fangles.Items[m]^), where m is 0..n-1 }
    procedure senseBeacons;
      { Procedure to sense angles w.r.t. forward direction. ie. this procedure
        updates the Fangles tuple with the current sensor data }
  public
    constructor Create; override;
    destructor Destroy; override;
    procedure execute(Sender: TAgentManager); override;
                                          // Called to execute the Agent.
    procedure behave; override;           // How the Agent behaves.
    procedure draw(canvas: TCanvas); override;
    procedure reportDebug; override;
  end;

  TanglePtr = ^Single;
  TdistPtr = ^Single;

{*****************************************************************************}
implementation

uses
  Unit1;

const
  D_AVOID = AGENT_DIAM + 5;
  D_FOLLOW = AGENT_DIAM + 15;
  PI = 3.14159265359;
  DELTA_THETA = PI/4;
  DELTA_PHI = PI/8;
  VELOCITY = 1;
  D_FORWARD = VELOCITY;

var
  idVal: Integer = 0;     // This class variable assigns id's to Agents
  numAgents: Integer = 0; // The no. of agent objects that currently exist

{*****************************************************************************
* TAgent methods
*****************************************************************************}

{************************************
* Constructor for TAgent
************************************}
constructor TAgent.Create;
begin
  inherited Create;

  { Allocate the vector object fields }
  Pos := TVector.Create;
  FlastPos := TVector.Create;
  Fstate := 1;        // initial internal state for all agent types
  Direction := TVector.Create;
  id := idVal;
  idVal := idVal + 1;

  numAgents := numAgents + 1;
end;
```

```
{*************************************
* Destructor for TAgent
*************************************}
destructor TAgent.Destroy;
begin
  { Free the vector objects }
  Pos.Free;
  FlastPos.Free;
  Direction.Free;

  numAgents := numAgents - 1;
  if numAgents = 0 then
    idVal := 0;   // reset id's for newly allocated agents

  inherited Destroy;
end;



{*************************************
* Debug routine for TAgent
*************************************}
procedure TAgent.reportDebug;
var
  s: String;
begin
  s := Format('%3d', [Fid]) + ':  ';
  s := s + Format('%6.2f', [Pos.x]) + '  ';
  s := s + Format('%6.2f', [Pos.y]) + '  ';
  s := s + Format('%6.2f', [FlastPos.x]) + '  ';
  s := s + Format('%6.2f', [FlastPos.y]) + '  ';
  s := s + Format('%5.2f', [Direction.x]) + '  ';
  s := s + Format('%5.2f', [Direction.y]) + '  ';

  with DebugForm.Image1.Canvas do
    begin
    if (Self.Fid and 1) = 1 then
      Font.Color := clGray        //odd
    else
      Font.Color := clBlack;      //even
    TextOut(1,Self.Fid*14,s);
    end;
end;



{*************************************
* TAgent.move
* This method moves the agent from its current
* position according to the direction vector
*************************************}
procedure TAgent.move;
var
  tempVec: TVector;
begin
  try
    tempVec := TVector.Create;
    tempVec.copy(Pos);
    tempVec.add(Direction);
    Pos.copy(tempVec);
  finally
    tempVec.Free;
  end;
end;
```

```
{*************************************
* TAgent.distanceTo
* Returns the distance from the sender agent to the current agent
*************************************}
function TAgent.distanceTo(sender: TAgent): Single;
var
  tempVec: TVector;
begin
  try
    tempVec := TVector.Create;
    tempVec.copy(Self.Pos);
    tempVec.sub(sender.Pos);
    Result := tempVec.abs;
  finally
    tempVec.Free;
  end;
end;



{*************************************
* sortFunction
* A pointer to this function is passed to the TList Sort method of
* TRoamingAgents to sort the Singles their lists contain.
*************************************}
function sortFunction(Item1, Item2: Pointer): Integer;
begin
  if Single(Item1^) > Single(Item2^) then
    Result := 1
  else if Single(Item1^) = Single(Item2^) then
    Result := 0
  else
    Result := -1;
end;



{*****************************************************************************
* TBeaconAgent methods
*****************************************************************************}

{*************************************
* Constructor for TBeaconAgent
*************************************}
constructor TBeaconAgent.Create;
begin
  inherited Create;
  Fangles := TList.Create;
  Fdistances := TList.Create;
  FsignalMove := FALSE;
end;



{*************************************
* Destructor for TBeaconAgent
*************************************}
destructor TBeaconAgent.Destroy;
var
  i: Integer;
  tempAngle: TanglePtr;
  tempDist: TdistPtr;
begin
  while Fangles.Count > 0 do
    begin
    tempAngle := Fangles.Items[0];
    Fangles.Delete(0);
    Dispose(tempAngle);
    end;

  while Fdistances.Count > 0 do
    begin
    tempDist := Fdistances.Items[0];
    Fdistances.Delete(0);
    Dispose(tempDist);
    end;

  Fangles.Free;
  Fdistances.Free;
  inherited Destroy;
end;
```

```
{*************************************
* TBeaconAgent.senseBeacons
* Procedure to sense angles w.r.t. forward direction. ie. this procedure
* updates the Fangles tuple with the current sensor data
* This procedure will have to change to incorporate missed data if beacons
* are able to become obscured or sensor errors cause problems. Also sensor
* errors will be introduced here. Also, if >3 beacons are used to reduce
* errors, this will force this procedure to change. Also, in a real system, the
* angle to the mapping agent must be identified in the data transmitted from the
* roaming agent to the mapping agent, so that the roaming agent's poition may
* be calculated. This is not done here as it is not a requirement for the
* simulator to operate, since the position calculation is not done.
* To sense angles, the procedure uses absolute position information. A
* physically embodied TBeaconAgent would not have access to this position data.
* It is used here to generate the angle data which would be directly accessible.
*************************************}
procedure TBeaconAgent.senseBeacons;
var
  i: Integer;
  tempAngle: TanglePtr;
  tempDist: TdistPtr;
  tempVector: TVector;
begin
  { First clean up (remove) the old angle list contents }
  while Fangles.Count > 0 do
    begin
    tempAngle := Fangles.Items[0];
    Fangles.Delete(0);
    Dispose(tempAngle);
    end;

  { Next clean up (remove) the old distances list contents }
  while Fdistances.Count > 0 do
    begin
    tempDist := Fdistances.Items[0];
    Fdistances.Delete(0);
    Dispose(tempDist);
    end;

  try
    begin
    tempVector := TVector.Create;

    for i := 0 to Fsender.getBMQuantity - 1 do
      begin
        if Fsender.getBMId(i) <> Self.id then
          try
            { Create new Singles to hold an angle and a distance respectively }
            tempAngle := New(TanglePtr);
            tempDist := New(TdistPtr);
            { Fsender contains the AgentManager object; Get the position of the
              i'th Beacon or Mapping agent }
            tempVector := Fsender.getBMPos(i, tempVector);
            { The vector from our position which points at the i'th agent is
              VECTOR := BM_POS_VECTOR - OUR_POS_VECTOR }
            tempVector.sub(Self.Pos);
            { Get the angle between our direction and the i'th B or M agent }
            tempAngle^ := Direction.signedAngleDeg(tempVector);
            { Add the angle to the list of angles }
            Fangles.Add(tempAngle);
            { Get the distance from our position to the i'th B or M agent }
            tempDist^ := tempVector.abs;
            { Add the distance to the list of distances }
            Fdistances.Add(tempDist);
          except
            Dispose(tempAngle);
            Dispose(tempDist);
          end;
      end;
    end;
  finally
    tempVector.Free;
  end;
end;
```

```
{*************************************
* TBeaconAgent.signal
* This procedure implements the signal interface to the beacon agent
* from the agent manager
*************************************}
procedure TBeaconAgent.signal(msg: sigType);
begin
  case msg of
    SIG_MOVE:         // This is the only meaningful message type for a Beacon
      begin
      FsignalMove := TRUE;
      end;
  end;
end;



{*************************************
* TBeaconAgent.behave
* Behaviour method. This procedure selects a behaviour and executes
* it for a TBeaconAgent.
* Behaviour is the assessment of sensor information and selection of the action
* to be taken based on it.
*************************************}
procedure TBeaconAgent.behave;
var
  angle1, angle2: Single;
  dist1, dist2: Single;
  tempVector, sumVector: TVector;
begin
  case Fstate of
    1:
      { stationary and waiting for signal from TMappingAgent to move }
      begin
      if FsignalMove then
        begin
        FsignalMove := FALSE;
        Fstate := 2;
        end;
      end; {case Fstate = 1}

    2:
      { move to bisect the line between the other agents }
      begin
      { move forward by d_forward }
      Self.move;
      Self.senseBeacons;  // Accumulate data on Beacon/Mapping agent positions

      angle1 := Single(Fangles.Items[0]^);
      angle2 := Single(Fangles.Items[1]^);

      { create direction vectors, sum them and move toward that direction }
      try
        tempVector := TVector.Create;
        sumVector := TVector.Create;

        sumVector.Copy(Self.Direction);

        { create 1st direction vector }
        tempVector.Copy(Self.Direction);
        tempVector.rotateDeg(angle1);
        sumVector.add(tempVector);

        { create 2nd direction vector }
        tempVector.Copy(Self.Direction);
        tempVector.rotateDeg(angle2);
        sumVector.add(tempVector);

        angle1 := Self.Direction.signedAngleDeg(sumVector);
        if angle1 > 180 then
          angle1 := angle1 - 360;
      finally
        tempVector.Free;
        sumVector.Free;
      end;

      { no need to truncate angles because turning radius of reference agents
        is unlimited }
      Self.Direction.rotateDeg(angle1);     // turn

      if abs(angle1) < 10 then
        {our direction vector must now be pointing between those pointing at
         the 2 beacon agents}
        Fstate := 3;
```

```
  end; {case Fstate = 2}

3:
  { move to new point by keeping the angles to the other agents equal
    until the magnitude of one of the angles exceeds 140 degrees}
  begin
  { move forward by d_forward }
  Self.move;
  Self.senseBeacons;  // Accumulate data on Beacon/Mapping agent positions

  { The angles are unsorted, so make no assumptions about their order.
    If angle1 is to the left of the forward direction, angle 2 will be to
    the right and vice versa }
  angle1 := Single(Fangles.Items[0]^);
  angle2 := Single(Fangles.Items[1]^);

  { Turn toward the direction whose angle is larger.
    Remember rotateDeg method parameter is in clockwise direction, so
    we need to negate the argument. Sum of angles will always be close
    to 360degrees. Turn this into a small +ve or -ve angle to rotate by }
  Self.Direction.rotateDeg(-(angle1 + angle2 - 360) / 2);

  if (angle1 > 150) and (angle2 > 150) then
    { agent must be near its goal now. Go to final position seeking states }
    begin
    Fgoal1Reached := FALSE;
    Fgoal2Reached := FALSE;
    Fstate := 4;
    end;
  end; {case Fstate = 3}

4:
  { agent must be near its goal now. Seek the goal based on distances }
  begin
  { move forward by d_forward }
  Self.move;
  { Accumulate angle & distance data on Beacon/Mapping agent positions }
  Self.senseBeacons;

  dist1 := Single(Fdistances.Items[0]^);
  angle1 := Single(Fangles.Items[0]^);

  if (dist1 < Form1.side + REFERENCE_GOAL_SIZE) and
     (dist1 > Form1.side - REFERENCE_GOAL_SIZE) then
    { At goal distance, swap to ranging to other beacon }
    begin
    if Fgoal2Reached then
      { at goal position }
      Fstate := 6
    else
      begin
      Fgoal1Reached := TRUE;
      Fstate := 5;
      end;
    end
  else
    begin
    Fgoal2Reached := FALSE;
    if dist1 > Form1.side then
      { we are too far away, so turn toward beacon }
      begin
      { convert 'unsigned' to 'signed' angle }
      if angle1 > 180 then
        angle1 := angle1 - 360;
      end
    else
      { we are too close, so turn away from beacon }
      begin
      angle1 := angle1 - 180;
      end;

    Self.Direction.rotateDeg(-angle1);       // turn
    end;
  end; {case Fstate = 4}

5:
  { agent must be near its goal now. Seek the goal based on distances }
  begin
  { move forward by d_forward }
  Self.move;
  { Accumulate angle & distance data on Beacon/Mapping agent positions }
  Self.senseBeacons;

  dist2 := Single(Fdistances.Items[1]^);
  angle1 := Single(Fangles.Items[1]^);
```

```
      if (dist2 < Form1.side + REFERENCE_GOAL_SIZE) and
         (dist2 > Form1.side - REFERENCE_GOAL_SIZE) then
        { At goal distance, swap to ranging to other beacon }
        begin
        if Fgoal1Reached then
          { at goal position }
          Fstate := 6
        else
          begin
          Fgoal2Reached := TRUE;
          Fstate := 4;
          end;
        end
      else
        begin
        Fgoal1Reached := FALSE;
        if dist2 > Form1.side then
          { we are too far away, so turn toward beacon }
          begin
          { convert 'unsigned' to 'signed' angle }
          if angle1 > 180 then
            angle1 := angle1 - 360;
          end
        else
          { we are too close, so turn away from beacon }
          begin
          angle1 := angle1 - 180;
          end;

        Self.Direction.rotateDeg(-angle1);       // turn
        end;
      end; {case Fstate = 5}

    6:
      { this is a transient state which sends the message that the agent
        has reached its goal }
      begin
      Fsender.txSignal(SIG_AT_GOAL, 0);
      Fstate := 1;
      end; {case Fstate = 6}
    end; {case Fstate}
end;



{************************************
* TBeaconAgent.execute
* This method performs all actions required to execute the agent in question
************************************}
procedure TBeaconAgent.execute(Sender: TAgentManager);
begin
  Fsender := Sender;

  { Calculate update-position vector. This is where the action happens.
    ie. the Behaviour is executed. }
  Self.behave;
end;
```

```
{************************************
* TBeaconAgent.draw
* draw method for TBeaconAgent
************************************}
procedure TBeaconAgent.draw(canvas: TCanvas);
begin
  if Form1.Animate.Checked then
    begin
    { Erase old }
    canvas.Brush.Color := clWhite;
    canvas.Pen.Color := clWhite;
    canvas.Ellipse(Trunc(Self.FlastPos.x-AGENT_RAD),
                   Trunc(Self.FlastPos.y-AGENT_RAD),
                   Trunc(Self.FlastPos.x+AGENT_RAD),
                   Trunc(Self.FlastPos.y+AGENT_RAD));
    Self.FlastPos.copy(Self.Pos);   // Update Last Position to current position

    { Draw new }
    canvas.Brush.Color := clRed;
    canvas.Pen.Color := clBlack;
    canvas.Ellipse(Trunc(Self.Pos.x-AGENT_RAD),
                   Trunc(Self.Pos.y-AGENT_RAD),
                   Trunc(Self.Pos.x+AGENT_RAD),
                   Trunc(Self.Pos.y+AGENT_RAD));
    canvas.MoveTo(Trunc(Self.Pos.x), Trunc(Self.Pos.y));
    canvas.LineTo(Trunc(Self.Pos.x+(Self.Direction.x*(AGENT_RAD-0.5))),
                  Trunc(Self.Pos.y+(Self.Direction.y*(AGENT_RAD-0.5))));
    end;

  { show our path if specifed }
  if Form1.rb1Path.Checked and (Id = Form1.SpinEdit1.Value) then
    Form1.OutputImage.Canvas.Pixels[Trunc(Self.Pos.x),
                                    Trunc(Self.Pos.y)] := clBlack;
end;



{*****************************************************************************
* TMappingAgent methods
*****************************************************************************}

{************************************
* Constructor for TMappingAgent
************************************}
constructor TMappingAgent.Create;
begin
  inherited Create;

  Fmap := TMap.Create;
  { Setting FreferenceStopped true will force the map coords to be set up by
    the mapping agent }
  FreferenceStopped := TRUE;
  FmapState := 1;                     // Init State variable
  FnextRefAgent := 0;
end;



{************************************
* Destructor for TMappingAgent
************************************}
destructor TMappingAgent.Destroy;
begin
  { Free the map objects }
  Fmap.Free;

  inherited Destroy;
end;
```

```
{************************************
* TMappingAgent.execute
* This method performs all actions required to execute the agent in question
************************************}
procedure TMappingAgent.execute(Sender: TAgentManager);
var
  i, j, col: Integer;
  tempVector: TVector;
  data, ratio, ratioThresh: Single;
begin
  Fsender := Sender;

  case FmapState of
    1:
      begin
      { Check for a signal being received by the mapping agent from a
        reference agent to indicate that all reference agents have stopped }
      if FreferenceStopped then
        begin
        FreferenceStopped := FALSE;
        { Set new map centroid coordinates }
        updateCentroid;
        { Set new 0 map coordinates to correspond with the current region
          being surveyed }
        F0mapX := FCentroidX - MAP_SIDE / 2;
        F0mapY := FCentroidY - MAP_SIDE / 2;
        FmapState := 2;
        end;
      end;

    2:
      begin
      { Add feature data from each agent to the map. This is easier than
        implementing message queues and having Roaming agents transmit their
        data to the mapping agent }
      { all reference agents are stationary, so it is OK to retrieve Roaming
        agent data }
      try
        tempVector := TVector.Create;
        for i := 0 to Fsender.getRQuantity - 1 do
          begin
          { Fsender contains the AgentManager object; Get the position of the
            i'th Beacon or Mapping agent }
          Fsender.getRData(i, tempVector, data);
          Fmap[trunc(tempVector.x - F0mapX),
               trunc(tempVector.y - F0mapY)] := data;
          end;
      finally
        tempVector.Free;
      end;

      { Test whether current area has been adequately surveyed. If so, perform
        actions to move to a new region }
      ratio := Fmap.mapRatio(Trunc(FcentroidX - F0mapX),
                             Trunc(FcentroidY - F0mapY));
      ratioThresh := StrToFloat(Form1.ebMapThresh.Text);
      Form1.pnMapRatio.Caption := Format('%3.2f', [ratio]);
      Form1.pbMapProgress.Position := trunc(100 * ratio / ratioThresh);

      if ratio > ratioThresh then
        begin
        { copy internal map to main map here }
        if Form1.cbShowMainMap.Checked then
          begin
          for i := 0 to MAP_SIDE - 1  do
            for j := 0 to MAP_SIDE - 1 do
              begin
              col := trunc(Fmap[i, j]);
              if col >= 0 then
                { build grey colour }
                Form1.tiMainMapImage.Canvas.Pixels[Trunc(F0mapX) + i,
                                                   Trunc(F0mapY) + j] :=
                                          col or (col shl 8) or (col shl 16);
              end;
          end;

        { clear the internal map ready for the next survey area }
        Fmap.clear;

        { Send a message to the next reference agent for it to establish a new
          reference position }
        Fsender.txSignal(SIG_MOVE, FnextRefAgent);
        FnextRefAgent := (FnextRefAgent + 1) mod Fsender.getBMQuantity;
        FmapState := 1;     // Change state to wait for new position signal
        end;
```

```
      end;

    end; { case }

  { mapping agent's own behaviour is executed here }
  Self.behave;

end;



{**************************************
* TMappingAgent.getMapRatio
* Returns the current map ratio so it can be displayed by the environment
**************************************}
function TMappingAgent.getMapRatio: Single;
begin
  Result := Fmap.mapRatio(Trunc(FcentroidX - FOmapX),
                          Trunc(FcentroidY - FOmapY));
end;



{**************************************
* TMappingAgent.updateCentroid
* This method updates the coordinates of the centroid of the 3 reference
* agents
**************************************}
procedure TMappingAgent.updateCentroid;
var
  i: Integer;
  xSum, ySum: Single;
  tempVector: TVector;
begin
  xSum := 0.0;
  ySum := 0.0;
  try
    tempVector := TVector.Create;

    for i := 0 to Fsender.getBMQuantity - 1 do
      begin
      { Fsender contains the AgentManager object; Get the position of the
        i'th Beacon or Mapping agent }
      tempVector := Fsender.getBMPos(i, tempVector);
      xSum := xSum + tempVector.x;
      ySum := ySum + tempVector.y;
      end;
    { Calculate average X coord }
    FcentroidX := xSum / 3.0;
    { Calculate average Y coord }
    FcentroidY := ySum / 3.0;
  finally
    tempVector.Free;
  end;
end;



{**************************************
* TMappingAgent.signal
* This procedure implements the signal interface to the mapping agent
* from the agent manager
**************************************}
procedure TMappingAgent.signal(msg: sigType);
begin
  case msg of
    SIG_MOVE:
      begin
      FsignalMove := TRUE;
      end;
    SIG_AT_GOAL:
      begin
      FreferenceStopped := TRUE;
      end;
  end;
end;
```

```
{*************************************
* TMappingAgent.draw
*************************************}
procedure TMappingAgent.draw(canvas: TCanvas);
begin
  if Form1.Animate.Checked then
    begin
    { Erase old }
    canvas.Brush.Color := clWhite;
    canvas.Pen.Color := clWhite;
    canvas.Ellipse(Trunc(Self.FlastPos.x-AGENT_RAD),
                   Trunc(Self.FlastPos.y-AGENT_RAD),
                   Trunc(Self.FlastPos.x+AGENT_RAD),
                   Trunc(Self.FlastPos.y+AGENT_RAD));
    Self.FlastPos.copy(Self.Pos);   // Update Last Position to current position

    { Draw new }
    canvas.Brush.Color := clLime;
    canvas.Pen.Color := clBlack;
    canvas.Ellipse(Trunc(Self.Pos.x-AGENT_RAD),
                   Trunc(Self.Pos.y-AGENT_RAD),
                   Trunc(Self.Pos.x+AGENT_RAD),
                   Trunc(Self.Pos.y+AGENT_RAD));
    canvas.MoveTo(Trunc(Self.Pos.x), Trunc(Self.Pos.y));
    canvas.LineTo(Trunc(Self.Pos.x+(Self.Direction.x*(AGENT_RAD-0.5))),
                  Trunc(Self.Pos.y+(Self.Direction.y*(AGENT_RAD-0.5))));
    end;

  { show our path if specifed }
  if Form1.rb1Path.Checked and (Id = Form1.SpinEdit1.Value) then
    Form1.OutputImage.Canvas.Pixels[Trunc(Self.Pos.x),
                                    Trunc(Self.Pos.y)] := clBlack;

end;



{*****************************************************************************
* TRoamingAgent methods
*****************************************************************************}

{*************************************
* Constructor for TRoamingAgent
*************************************}
constructor TRoamingAgent.Create;
begin
  inherited Create;
  Fangles := TList.Create;
end;



{*************************************
* Destructor for TRoamingAgent
*************************************}
destructor TRoamingAgent.Destroy;
var
  i: Integer;
  tempAngle: TanglePtr;
begin
  while Fangles.Count > 0 do
    begin
    tempAngle := Fangles.Items[0];
    Fangles.Delete(0);
    Dispose(tempAngle);
    end;
  Fangles.Free;
  inherited Destroy;
end;
```

```
{*************************************
* TRoamingAgent.reportDebug
* Debug routine for TRoamingAgent
*************************************}
procedure TRoamingAgent.reportDebug;
var
  s: String;
begin
  inherited reportDebug;

  if Fangles.Count = 3 then
    begin
    s :=     Format('%d', [Fstate]) + ' ';
    s := s + Format('%3.0f', [Single(Fangles.Items[0]^)]) + ' ';
    s := s + Format('%3.0f', [Single(Fangles.Items[1]^)]) + ' ';
    s := s + Format('%3.0f', [Single(Fangles.Items[2]^)]) + ' ';
    end;

  with DebugForm.Image1.Canvas do
    begin
    if Self.Fid and 1 = 1 then
      Font.Color := clOlive       // odd
    else
      Font.Color := clGreen;      // even
    TextOut(375,Self.Fid*14,s);
    end;
end;



{*************************************
* TRoamingAgent.senseBeacons
* Procedure to sense angles w.r.t. forward direction. ie. this procedure
* updates the Fangles tuple with the current sensor data
* This procedure will have to change to incorporate missed data if beacons
* are able to become obscured or sensor errors cause problems. Also sensor
* errors will be introduced here. Also, if >3 beacons are used to reduce
* errors, this will force this procedure to change.
* To sense angles, the procedure uses absolute position information. A
* physically embodied TRoamingAgent would not have access to this data. It is
* used here to simulate the angle data which would be directly accessible.
* sortFunction is used by senseBeacons to sort the beacon angles.
*************************************}
procedure TRoamingAgent.senseBeacons;
var
  i: Integer;
  tempAngle: TanglePtr;
  tempVector: TVector;
begin
  { First clean up (remove) the old angle list contents }
  while Fangles.Count > 0 do
    begin
    tempAngle := Fangles.Items[0];
    Fangles.Delete(0);
    Dispose(tempAngle);
    end;

  try
    begin
    tempVector := TVector.Create;

    for i := 0 to Fsender.getBMQuantity - 1 do
      begin
        try
          { Create a new Single to hold an angle }
          tempAngle := New(TanglePtr);
          { Fsender contains the AgentManager object; Get the position of the
            i'th Beacon or Mapping agent }
          tempVector := Fsender.getBMPos(i, tempVector);
          { Now the vector from our position which points at the i'th agent is
            VECTOR := BM_POS_VECTOR - OUR_POS_VECTOR }
          tempVector.sub(Self.Pos);
          { Now Get the angle between our direction and the i'th B or M agent }
          tempAngle^ := Direction.signedAngleDeg(tempVector);
          { Add the angle to the list of angles }
          Fangles.Add(tempAngle);
        except
          Dispose(tempAngle);
        end;
      end;
    end;
  finally
    tempVector.Free;
  end;
```

```
  { Sort list of angles }
  Fangles.Sort(sortFunction);
end;



{*************************************
TRoamingAgent.behave
Behaviour method. This procedure selects a behaviour and executes
it for a TRoamingAgent.
That is, this method implements the temporally combined behaviour through the
use of a finite state machine.
Behaviour is the assessment of sensor information and selection of the action
to be taken based on it.
A TRoamingAgent moves randomly in a region bounded by the triangle whose
vertices are the positions of TMappingAgent and the TBeaconAgent's.
A TRoamingAgent's sensors allow it to identify the mapping agent and measure
the angles to the beacon agents.
To determine that it is inside the triangle formed by the convex hull of the
mapping and beacon agents, no angle between two subsequent ordered angular
position readings, measured by the sensors can exceed 180deg. The first pass
implementation of the behaviour fsm thus has two states;

state 1:
Safe Wandering:
move forward by d_forward
turn randomly
if (angle between 2 subsequent ordered angles > 180deg) then
  turn to face direction bisecting the two angles
  state 2

state 2:
Move inside fence:
move forward by d_forward
if (angle between all sets of 2 subsequent ordered angles < 180deg) then
  state 1
*************************************}
procedure TRoamingAgent.behave;
var
  angle1, angle2, angle3: Single;
  tempVector, sumVector: TVector;
begin
  case Fstate of
    1:          // inside fence
      begin
      { move forward by d_forward }
      Self.move;

      { turn randomly (up to +/-TURN_LIMIT degrees) }
      Self.Direction.rotateDeg(-TURN_LIMIT);
      Self.Direction.rotateDeg(Random(2*TURN_LIMIT));

      { if (angle between 2 subsequent ordered angles > 180deg) then
          state 2 }
      Self.senseBeacons;  // Accumulate data on Beacon/Mapping agent positions

      angle1 := Single(Fangles.Items[0]^);
      angle2 := Single(Fangles.Items[1]^);
      angle3 := Single(Fangles.Items[2]^);

      if (angle2 - angle1 > 180) or
         (angle3 - angle2 > 180) or
         (angle1 - angle3 + 360 > 180) then
        Fstate := 2;

      end;

    2:          // outside fence
      begin
      { move forward by d_forward }
      Self.move;
      Self.senseBeacons;  // Accumulate data on Beacon/Mapping agent positions

      { if (angle between all sets of 2 subsequent ordered angles < 180deg) then
          state 1 }
      angle1 := Single(Fangles.Items[0]^);
      angle2 := Single(Fangles.Items[1]^);
      angle3 := Single(Fangles.Items[2]^);

      if (angle2 - angle1 <= 180) and
         (angle3 - angle2 <= 180) and
         (angle1 - angle3 + 360 <= 180) then
        Fstate := 1;
```

```
{$DEFINE ROAMING_BEHAVIOUR_2}
{$IFDEF ROAMING_BEHAVIOUR_1}
      { convert sensed angles to +/-180 range }
      if angle1 > 180 then
        angle1 := angle1 - 360;
      if angle2 > 180 then
        angle2 := angle2 - 360;
      if angle3 > 180 then
        angle3 := angle3 - 360;

      { take the average as the desired direction }
      angle1 := -(angle1 + angle2 + angle3) / 3;
{$ELSE IFDEF ROAMING_BEHAVIOUR_2}
      { create direction vectors, sum them and move toward that direction }
      try
        tempVector := TVector.Create;
        sumVector := TVector.Create;

        sumVector.Copy(Self.Direction);

        { create 1st direction vector }
        tempVector.Copy(Self.Direction);
        tempVector.rotateDeg(angle1);
        sumVector.add(tempVector);

        { create 2nd direction vector }
        tempVector.Copy(Self.Direction);
        tempVector.rotateDeg(angle2);
        sumVector.add(tempVector);

        { create 3rd direction vector }
        tempVector.Copy(Self.Direction);
        tempVector.rotateDeg(angle3);
        sumVector.add(tempVector);

        angle1 := Self.Direction.signedAngleDeg(sumVector);
        if angle1 > 180 then
          angle1 := angle1 - 360;
      finally
        tempVector.Free;
        sumVector.Free;
      end;

{$ENDIF}
      { truncate angles because turning radius of agent is limited }
      if angle1 > TURN_LIMIT then
        angle1 := TURN_LIMIT
      else if angle1 < -TURN_LIMIT then
        angle1 := -TURN_LIMIT;
      Self.Direction.rotateDeg(angle1);
      end;
    end;  {case Fstate}

end;  {TRoamingAgent.behave}



{**************************************
* TRoamingAgent.execute
* This method performs all actions required to execute the agent in question
**************************************}
procedure TRoamingAgent.execute(Sender: TAgentManager);
var
  DeltaX, DeltaY: Single;
begin
  Fsender := Sender;

  { Calculate update-position vector. This is where the action happens.
    ie. the Behaviour is executed. }
  Self.behave;
end;
```

```
{************************************
* TRoamingAgent.draw
************************************}
procedure TRoamingAgent.draw(canvas: TCanvas);
begin
  if Form1.rbAllPaths.Checked then
    Form1.OutputImage.Canvas.Pixels[Trunc(Self.Pos.x),
                                    Trunc(Self.Pos.y)] :=
                Form1.TerrainMap.Canvas.Pixels[Trunc(Self.Pos.x),
                                               Trunc(Self.Pos.y)]
  else
    { show only the path of the specifed agent }
    if Id = Form1.SpinEdit1.Value then
      Form1.OutputImage.Canvas.Pixels[Trunc(Self.Pos.x),
                                      Trunc(Self.Pos.y)] := clBlack;

  if Form1.Animate.Checked then
    begin
    { Erase old }
    canvas.Brush.Color := clWhite;
    canvas.Pen.Color := clWhite;
    canvas.Ellipse(Trunc(Self.FlastPos.x-AGENT_RAD),
                   Trunc(Self.FlastPos.y-AGENT_RAD),
                   Trunc(Self.FlastPos.x+AGENT_RAD),
                   Trunc(Self.FlastPos.y+AGENT_RAD));
    Self.FlastPos.copy(Self.Pos);   // Update Last Position to current position

    { Draw new }
    canvas.Brush.Color := clYellow;
    canvas.Pen.Color := clBlack;
    canvas.Ellipse(Trunc(Self.Pos.x-AGENT_RAD),
                   Trunc(Self.Pos.y-AGENT_RAD),
                   Trunc(Self.Pos.x+AGENT_RAD),
                   Trunc(Self.Pos.y+AGENT_RAD));
    canvas.MoveTo(Trunc(Self.Pos.x), Trunc(Self.Pos.y));
    canvas.LineTo(Trunc(Self.Pos.x+(Self.Direction.x*(AGENT_RAD-0.5))),
                  Trunc(Self.Pos.y+(Self.Direction.y*(AGENT_RAD-0.5))));
    end;
end;

end.
```

Listing AgentManager.pas

```
{
Terrain Mapping Simulator
AgentManager File

File:         agentmanager.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This unit implements the AgentManager class (object definition).
              This defines a manager object which is responsible for
              maintaining and controlling all agent objects in the simulation.
              There is only one instance of this class instanciated in the
              Unit1 unit.
}

unit AgentManager;

{*****************************************************************************}
interface
uses Classes, SysUtils, Unit1, Dialogs, Vector;

const
  CULL_AGENT_DISTANCES_RADIUS = 20.0;
  MAXAGENTS = 80;

type
  { signals which may be sent between agents }
  sigType = (SIG_MOVE, SIG_AT_GOAL);

  TAgentManager = class
  private
    Fgeneration: Integer;
    agentList: TList;     //list containing all agents
    bmList: TList;        //list containing just beacon and mapping agents
    rList: TList;         //list containing just roaming agents
  public
    constructor Create;
    destructor Destroy; override;
    procedure debugAgents;
    procedure draw;
    procedure sequenceAgents;
    procedure txSignal(msg: sigType; parm: Integer);
    procedure getRData(index: Integer; var vector: TVector; var data: Single);
    function getRQuantity: Integer;
    function getBMPos(index: Integer; vector: TVector): TVector;
    function getBMId(index: Integer): Integer;
    function getBMQuantity: Integer;
    property generation: Integer read Fgeneration write Fgeneration;
      { Fgeneration accessor property }
  end;


{*****************************************************************************}
implementation

uses Agent;

{************************************
* Constructor for TAgentManager
************************************}
constructor TAgentManager.Create;
var
  agent: TAgent;
  txtFile: TextFile;    // Handle for initialisation file
  ch: Char;
  temp:Single;
begin
  {Allocate the agent list}
  inherited Create;

  { Allocate the TList which will hold the agents }
  agentList := TList.Create;
  bmList := TList.Create;
  rList := TList.Create;

  AssignFile(txtFile, 'init.dat');
  Reset(txtFile);
  { m=mapping b=beacon r=roaming }
  while not Eof(txtFile) do
    begin
    Read(txtFile, ch);
    case ch of
      ';':      // comment line - skip to next line
```

```
        begin
        Readln(txtFile);
        Continue;
        end;
      Chr(10), Chr(13):  // blank line - skip to next line (ignore)
        Continue;
      'm':      // mapping agent
        begin
        agent := TMappingAgent.Create;
        bmList.Add(agent);
        end;
      'b':      // beacon agent
        begin
        agent := TBeaconAgent.Create;
        bmList.Add(agent);
        end;
      'r':      // roaming agent
        begin
        agent := TRoamingAgent.Create;
        rList.Add(agent);
        end
      else
        ShowMessage(Concat('Invalid character ', ch));
      end;

    { Initialise agent }
    with agent do
      begin
      { Position }
      Read(txtFile, temp);
      Pos.x := temp;
      Read(txtFile, temp);
      Pos.y := temp;
      { Direction }
      Read(txtFile, temp);
      Direction.x := temp;
      Read(txtFile, temp);
      Direction.y := temp;
      Direction.unify;        // Ensure direction vector is a unit vector
      end;

    { Add the agent to the list }
    agentList.Add(agent);
    if agentList.Count > MAXAGENTS then
      MessageDlg('MAXAGENTS in AgentManager exceeded',
        mtError, [mbOk], 0);

    end; {while}

  CloseFile(txtFile);
  Form1.SpinEdit1.MaxValue := agentList.Count - 1;
  Fgeneration := 0;
end;



{************************************
* Destructor for TAgentManager
************************************}
destructor TAgentManager.Destroy;
var
  i: Integer;
begin
  { Free the agent objects }
  for i := 0 to agentList.Count - 1 do
    TAgent(agentList.Items[i]).Free;

  { Deallocate the TLists which held the agents }
  agentList.Free;
  { No need to Free the abjects of the bmList or rList lists since they were
    freed above as members of the agentList }
  bmList.Free;
  rList.Free;

  inherited Destroy;
end;
```

```
{*************************************
* TAgentManager.debugAgents
* Call debug routines of all agents under our control
*************************************}
procedure TAgentManager.debugAgents;
var
  i: Integer;
begin
  for i := 0 to agentList.Count - 1 do
    TAgent(AgentList.Items[i]).reportDebug;
end;



{*************************************
* TAgentManager.txSignal
* This method controls signalling of messages between agents
* msg contains the message to be sent
* parm is an optional parameter
*************************************}
procedure TAgentManager.txSignal(msg: sigType; parm: Integer);
var
  i: Integer;
begin
  case msg of
    SIG_MOVE:
      { this message is always txed from the mapping to a reference agent }
      begin
      { reference agent index is the optional message parameter }
      i := parm;
      end;

    SIG_AT_GOAL:
      { this message is always txed from a reference to the mapping agent }
      begin
      { get mapping agent id }
      for i := 0 to bmList.Count - 1 do
        if TAgent(bmList.Items[i]) is TMappingAgent then
          break;
      end;
    end; { case }

  { send the message to the target agent }
  TBeaconAgent(bmList.Items[i]).signal(msg);
end;



{*************************************
* TAgentManager.draw
* Call draw routines of all agents under our control
*************************************}
procedure TAgentManager.draw;
var
  i: Integer;
begin
  for i := 0 to agentList.Count - 1 do
    TAgent(AgentList.Items[i]).draw(Form1.AgentImage.Canvas);  // Draw Agent
end;



{*************************************
* TAgentManager.getBMPos
* Get Beacon or Mapping Agent Position vector
* index - The index of an Agent pointed to by the bmList.
* Note: List index is 0 based.
* Updates vector with the position of the Agent referred to by index.
* If the bmList index is exceeded, the TVector returned contains a -ve x value.
*************************************}
function TAgentManager.getBMPos(index: Integer; vector: TVector): TVector;
begin
  if index < bmList.Count then
    vector.copy(TAgent(bmList.Items[index]).Pos)
  else
    begin
    vector.x := -1.0;
    vector.y := -1.0;
    MessageDlg('bmList out of range error in TAgentManager getBMPos method.',
      mtError, [mbOk], 0);
    end;
  Result := vector;
end;
```

```
{*************************************
* TAgentManager.getRData
* Get Roaming Agent Position vector and associated feature data
* index - The index of an Agent pointed to by the agentList.
* Note: List index is 0 based.
* Updates vector with the position of the Agent referred to by index.
* If the rList index is exceeded, the TVector returned contains a -ve x value.
*************************************}
procedure TAgentManager.getRData(index: Integer; var vector: TVector;
                                 var data: Single);
var
  agentPtr: TAgent;
begin
  if index < rList.Count then
    begin
    agentPtr := TAgent(rList.Items[index]);
    vector.copy(agentPtr.Pos);
    { data is grey value of the terrain map, which is represented by the
      lower 8 bits of the retrieved colour value }
    data := Form1.TerrainImage.Canvas.Pixels[Trunc(agentPtr.Pos.x),
                                             Trunc(agentPtr.Pos.y)] and $0FF;
    end
  else
    begin
    vector.x := -1.0;
    vector.y := -1.0;
    data := -1.0;
    MessageDlg('rList out of range error in TAgentManager getRData method.',
      mtError, [mbOk], 0);
    end;
end;



{*************************************
* TAgentManager.getBMId
* Return the id property of the indexed Beacon or Mapping agent
*************************************}
function TAgentManager.getBMId(index: Integer): Integer;
begin
  Result := TAgent(bmList.Items[index]).id;
end;



{*************************************
* TAgentManager.getBMQuantity
* Get total number of Beacon and Mapping agents in bmList
*************************************}
function TAgentManager.getBMQuantity: Integer;
begin
  Result := bmList.Count;
end;



{*************************************
* TAgentManager.getRQuantity
* Get total number of Roaming agents in rList
*************************************}
function TAgentManager.getRQuantity: Integer;
begin
  Result := rList.Count;
end;
```

```
{************************************
* TAgentManager.sequenceAgents
* This procedure sequences the execution of all agents in a random order by
* the following method:
* create a list of agents
* choose a random agent called A
* execute agent A
* remove A from the list
* repeat until all agents have been executed
************************************}
procedure TAgentManager.sequenceAgents;
var
  tempAgentList: array[0..MAXAGENTS-1] of Boolean;
  i, j: Integer;
  dx, dy, Length: Single;
  safe: Boolean;
  tempVec: TVector;

begin
  for i := 0 to agentList.Count - 1 do
    tempAgentList[i] := True;
  Self.debugAgents;

  for i := 0 to agentList.Count - 1 do
    begin
    { choose a random agent called j }
    j := Random(agentList.Count);
    while not tempAgentList[j] do
      j := (j + 1) mod agentList.Count;

    { execute agent j }
    TAgent(AgentList.Items[j]).execute(Self);
    { remove j from the list }
    tempAgentList[j] := False;
    end;

  { Draw Agents' current positions }
  for i := 0 to agentList.Count - 1 do
    TAgent(AgentList.Items[i]).draw(Form1.AgentImage.Canvas);

  Inc(Fgeneration);
  Form1.CountPanel.Caption := IntToStr(Fgeneration);
end;

end.
```

Listing DebugUnit.pas

```
{
Terrain Mapping Simulator
DebugUnit File

File:         debugunit.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This is the "debug form" or unit file associated with the debug
              form of the Terrain Mapping Simulator.
              The TDebugForm object is defined here along with the event
              handlers for the form controls.
}

unit DebugUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TDebugForm = class(TForm)
    Label1: TLabel;
    DebugPanel: TPanel;
    Image1: TImage;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
end;

var
  DebugForm: TDebugForm;

implementation

{$R *.DFM}

uses
  Unit1;

procedure TDebugForm.FormCreate(Sender: TObject);
begin
  Image1.Canvas.Font.Name := 'Courier New';
end;

procedure TDebugForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Form1.cbDebug.checked := FALSE;
end;

end.
```

Listing Map.pas

```
{
Terrain Mapping Simulator
Map File

File:         map.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This unit implements a map class.
              The associated map object is contained by the Mapping agent,
              which is defined in the Agent unit.
}

unit Map;

{*****************************************************************************}
interface

uses
  Unit1, Dialogs, SysUtils, Windows;

const
  MAP_SIDE = 200;

type
  TMap = class
  private
    { The map is kept in this structure }
    FmapArray: array [0..MAP_SIDE-1, 0..MAP_SIDE-1] of Single;
    FnArray: array [0..MAP_SIDE-1, 0..MAP_SIDE-1] of Integer;
    function getFeature(x, y: Integer): Single;
    procedure addPoint(x, y: Integer; newVal: Single);
  public
    constructor Create; virtual;
    procedure clear;
    function mapRatio(x, y: Integer): Single;
    property feature[x, y: Integer]: Single read getFeature
                                            write addPoint; default;
  end;

{*****************************************************************************}
implementation

const
  SAMPLE_SIDE = 10;

{*****************************************************************************
* TMap methods
*****************************************************************************}

{************************************
* Constructor for TMap
************************************}
constructor TMap.Create;
var
  i, j: Integer;
begin
  inherited Create;

  { Initialise all map contents }
  for i := 0 to MAP_SIDE-1 do
    for j := 0 to MAP_SIDE-1 do
      begin
      FmapArray[i, j] := 0.0;
      FnArray[i, j] := 0;
      end;
  with Form1.tiMapImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, MAP_SIDE-1, MAP_SIDE-1);
    Brush.Color := clLime;
    Pen.Color := clLime;
    Rectangle((MAP_SIDE - SAMPLE_SIDE) div 2,
              (MAP_SIDE - SAMPLE_SIDE) div 2,
              (MAP_SIDE + SAMPLE_SIDE) div 2-1,
              (MAP_SIDE + SAMPLE_SIDE) div 2-1);
    end;
end;
```

```
{*************************************
* mapRatio function for TMap
* This function returns a value in the range 0.0 .. 1.0 which
* represents the ratio of array elements within a 10x10 range
* which have been written to since the creation of the array.
* The 10x10 range is centred about the parameter values x, y.
* If the array bounds are exceeded, an exception is raised.
*************************************}
function TMap.mapRatio(x, y: Integer): Single;
var
  i, j, count: Integer;
begin
  count := 0;
  try
    for i := x-(SAMPLE_SIDE div 2) to x+(SAMPLE_SIDE div 2-1) do
      for j := y-(SAMPLE_SIDE div 2) to y+(SAMPLE_SIDE div 2-1) do
        if FnArray[i, j] > 0 then
          Inc(count);
    Result := count / 100.0;
  except
    on ERangeError do
      begin
      Result := 0.0;
      ShowMessage('Array bounds exceeded in TMap mapRatio method');
      end;
  end;
end;



{*************************************
* clear method for TMap
* This method clears all feature data from the map
*************************************}
procedure TMap.clear;
var
  i, j: Integer;
begin
  { Initialise all map contents }
  for i := 0 to MAP_SIDE-1 do
    for j := 0 to MAP_SIDE-1 do
      begin
      FmapArray[i, j] := 0.0;
      FnArray[i, j] := 0;
      end;
  if Form1.cbShowMap.checked then
    with Form1.tiMapImage.Canvas do
      begin
      Brush.Color := clWhite;
      Pen.Color := clWhite;
      Rectangle(0, 0, MAP_SIDE-1, MAP_SIDE-1);
      Brush.Color := clLime;
      Pen.Color := clLime;
      Rectangle((MAP_SIDE - SAMPLE_SIDE) div 2,
                (MAP_SIDE - SAMPLE_SIDE) div 2,
                (MAP_SIDE + SAMPLE_SIDE) div 2-1,
                (MAP_SIDE + SAMPLE_SIDE) div 2-1);
      end;
end;



{*************************************
* getFeature accessor method for TMap
* This method allows a feature to be read from the map feature data set
*************************************}
function TMap.getFeature(x, y: Integer): Single;
begin
  if FnArray[x, y] > 0 then
    Result := FmapArray[x, y]
  else
    Result := -1.0;
end;
```

```
{************************************
* addPoint method for TMap
* This method allows a feature to be added to the map feature data set
************************************}
procedure TMap.addPoint(x, y: Integer; newVal: Single);
var
  n: Integer;
  col: Integer;
begin
  Inc(FnArray[x, y]);
  n := FnArray[x, y];
  FmapArray[x, y] := FmapArray[x, y] * (n-1)/n + newVal/n;

  if Form1.cbShowMap.checked then
    begin
    { build grey colour }
    col := trunc(FmapArray[x, y]);
    Form1.tiMapImage.Canvas.Pixels[x, y] := col or
                                            (col shl 8) or
                                            (col shl 16);
    end;
end;

end.
```

Listing Unit1.pas

```
{
Terrain Mapping Simulator
Unit1 File

File:         unit1.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This is the "main form" or unit file associated with the main
              form of the Terrain Mapping Simulator.
              The TForm1 object is defined here along with the event handlers
              for the form controls.
}

unit Unit1;

{$R+} {Range checking}
{$S+} {Stack checking}

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Buttons, Vector, Menus, Spin,
  Clipbrd, ComCtrls;

const
  MAX_X = 250;
  MAX_Y = 250;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    TerrainImage: TImage;
    Controls: TGroupBox;
    StatusPanel: TPanel;
    CountPanel: TPanel;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    N1: TMenuItem;
    PrintSetup1: TMenuItem;
    Print1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    HowtoUseHelp1: TMenuItem;
    SearchforHelpOn1: TMenuItem;
    Contents1: TMenuItem;
    Run1: TMenuItem;
    mnRestart: TMenuItem;
    mnSingleStep: TMenuItem;
    mnRun: TMenuItem;
    mnStop: TMenuItem;
    Edit1: TMenuItem;
    Copy1: TMenuItem;
    PrintDialog1: TPrintDialog;
    PrinterSetupDialog1: TPrinterSetupDialog;
    Bevel1: TBevel;
    ToolPanel: TPanel;
    Bevel2: TBevel;
    btRestart: TSpeedButton;
    btStop: TSpeedButton;
    btRun: TSpeedButton;
    btSingleStep: TSpeedButton;
    OpenDialog1: TOpenDialog;
    OutputPanel: TPanel;
    OutputImage: TImage;
    AgentPosPanel: TPanel;
    AgentImage: TImage;
    lbIteration: TLabel;
    gbParm: TGroupBox;
    lbMapThresh: TLabel;
    Label1: TLabel;
    Label2: TLabel;
    lbSLLim: TLabel;
    ebMapThresh: TEdit;
    PopupMenu1: TPopupMenu;
    Copy2: TMenuItem;
    cbDebug: TCheckBox;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    pnInternMap: TPanel;
```

```
    tiMapImage: TImage;
    seSide: TSpinEdit;
    Panel2: TPanel;
    lbMapRatio: TLabel;
    pnMapRatio: TPanel;
    pbMapProgress: TProgressBar;
    rbAllPaths: TRadioButton;
    rb1Path: TRadioButton;
    SpinEdit1: TSpinEdit;
    cbShowMap: TCheckBox;
    Animate: TCheckBox;
    Label7: TLabel;
    pnMap: TPanel;
    tiMainMapImage: TImage;
    cbShowMainMap: TCheckBox;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure StepClick(Sender: TObject);
    procedure RunClick(Sender: TObject);
    procedure StopClick(Sender: TObject);
    procedure RestartClick(Sender: TObject);
    procedure PrintSetup1Click(Sender: TObject);
    procedure Print1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure TerrainImageDblClick(Sender: TObject);
    procedure seSideChange(Sender: TObject);
    procedure Copy2Click(Sender: TObject);
    procedure cbDebugClick(Sender: TObject);
  private
    Play: Boolean;
    backgroundFileName: String;
    { Private declarations }
  public
    { Public declarations }
    terrainMap: TBitMap;
    side: Single;       // length of side of triangle
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses
  Agent, AgentManager, DebugUnit;

var
  agents: TAgentManager;

{******************************************************************************}
{
TForm methods
}
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  terrainMap := TBitMap.Create;
  backgroundFileName := 'landscape1.bmp';
  terrainMap.LoadFromFile(backgroundFileName);

  agents := TAgentManager.Create; // Create an instance of the agent manager
  side := 55;
  with AgentImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
  with OutputImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
  with tiMainMapImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
end;
```

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  i: Integer;
begin
  terrainMap.Free;
  agents.Free;
end;



procedure TForm1.StepClick(Sender: TObject);
begin
  agents.sequenceAgents;
end;



procedure TForm1.RunClick(Sender: TObject);
var
  i: Integer;
begin
  btRestart.Enabled     := False;
  btSingleStep.Enabled  := False;
  btStop.Enabled        := True;
  btRun.Enabled         := False;

  mnRestart.Enabled     := btRestart.Enabled;
  mnSingleStep.Enabled  := btSingleStep.Enabled;
  mnStop.Enabled        := btStop.Enabled;
  mnRun.Enabled         := btRun.Enabled;

  Play := True;
  while Play do
    begin
    agents.sequenceAgents;
    Application.ProcessMessages;
    end;
end;



procedure TForm1.StopClick(Sender: TObject);
begin
  btRestart.Enabled     := True;
  btSingleStep.Enabled  := True;
  btRun.Enabled         := True;
  btStop.Enabled        := False;

  mnRestart.Enabled     := btRestart.Enabled;
  mnSingleStep.Enabled  := btSingleStep.Enabled;
  mnStop.Enabled        := btStop.Enabled;
  mnRun.Enabled         := btRun.Enabled;

  Play := False;
end;
```

```
procedure TForm1.RestartClick(Sender: TObject);
var
  i: Integer;
begin
  with AgentImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
  with OutputImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
  with tiMainMapImage.Canvas do
    begin
    Brush.Color := clWhite;
    Pen.Color := clWhite;
    Rectangle(0, 0, Width, Height);
    end;
  agents.Free;
  agents := TAgentManager.Create;
  agents.draw;
end;



procedure TForm1.PrintSetup1Click(Sender: TObject);
begin
  PrinterSetupDialog1.Execute;
end;



procedure TForm1.Print1Click(Sender: TObject);
var
  Index: Integer;
  PrinterString: String;
  f: TextFile;
begin
  if PrintDialog1.Execute then
    begin
//    AssignPrn(f);
//    Rewrite(f);
//
//    {print any text here eg.}
//
//    PrinterString := 'Track ';
//    Writeln(f, PrinterString);
//
//    CloseFile(f);
    end;
end;



procedure TForm1.Exit1Click(Sender: TObject);
begin
  Application.Terminate;
end;



procedure TForm1.TerrainImageDblClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    begin
    with terrainMap do
      begin
      backgroundFileName := OpenDialog1.FileName;
      LoadFromFile(backgroundFileName);
      Parent := nil;
      end;

    TerrainImage.Picture.Bitmap.Canvas.Draw(0, 0, terrainMap);
    end;
end;
```

```
procedure TForm1.seSideChange(Sender: TObject);
begin
  side := TSpinEdit(Sender).Value;
end;



procedure TForm1.Copy2Click(Sender: TObject);
begin
  Clipboard.Assign(TImage(PopupMenu1.PopupComponent).Picture.Bitmap);
end;



procedure TForm1.cbDebugClick(Sender: TObject);
begin
  DebugForm.Visible := cbDebug.Checked;
end;

end.
```

Listing Vector.pas

```
{
Terrain Mapping Simulator
Vector File

File:         vector.pas
Author:       Gary Ruben
Date:         Aug 1998
Description:  This unit implements a 2D-Vector Abstract Data Type.
              Accessor methods (properties) allow access to the x and y
              coordinate Fields.
              There are methods allowing conjugation, addition, rotation, dot
              product etc.
              Note that if two vectors are added (or any other operation on 2
              vectors which produces a resultant) the fields of the Vector
              object performing the operation will be updated with the result.
              Therefore, to perform A := B + C if one wishes to keep both B
              and C,
                - First create A through an explicit TVector.Create
                - Next copy B to A using the copy method
                - Add C to A using the add method
                - Finally, remember to destroy A through an explicit
                  TVector.Destroy when finished with A

              Note that the accessor properties x and y are not in a
              published section since this component will not be used
              in the Object Inspector.
}

unit Vector;

(*****************************************************************************)
interface

type
  TVector = class
  private
    Fx: Single;                         // x component of vector
    Fy: Single;                         // y component of vector
  public
    procedure neg;                      // A -> -A
    procedure conj;                     // A -> A*
    procedure unify;                    // A -> A/|A|
    procedure mult(scalar: Single);     // A -> scalar*A
    procedure divide(scalar: Single);   // A -> A/scalar
    procedure add(vec: TVector);        // A -> A+vec
    procedure sub(vec: TVector);        // A -> A-vec
    procedure copy(vec: TVector);       // A -> vec
    procedure rotateDeg(theta: Single); // A -> A rotated through theta degree
    procedure rotateRad(phi: Single);   // A -> A rotated through phi radian
    function dot(vec: TVector): Single; // A . vec
    function angleDeg(vec: TVector): Single; // angle between A & vec degree
    function angleRad(vec: TVector): Single; // angle between A & vec radian
    function signedAngleDeg(vec: TVector): Single;
      // signed angle between A & vec degree - clockwise is +ve
    function signedAngleRad(vec: TVector): Single;
      // signed angle between A & vec radian - clockwise is +ve
    function abs: Single;               // |A|
    property x: Single read Fx write Fx; // Fx accessor property
    property y: Single read Fy write Fy; // Fy accessor property
  end;


(*****************************************************************************)
implementation

uses Dialogs, SysUtils, Math;
{$DEFINE MATHUNITAVALIABLE}

const
  PI = 3.14159265359;


{************************************
* A -> -A
* Negates the vector object
* Tested OK
************************************}
procedure TVector.neg;
begin
  Fx := -Fx;
  Fy := -Fy;
end;
```

```
{**************************************
* A -> A*
* Conjugates the vector object
* Tested OK
**************************************}
procedure TVector.conj;
begin
  Fy := -Fy;
end;



{**************************************
* A -> A/|A|
* Turns the vector into a unit vector
* Tested OK
**************************************}
procedure TVector.unify;
var
  abs: Single;
begin
  try
    abs := Sqrt(Sqr(Fx) + Sqr(Fy));
    Fx := Fx / abs;
    Fy := Fy / abs;
  except
    on EDivByZero do
      MessageDlg('Divide by 0 error in TVector class divide unify.', mtError,
        [mbOk], 0);
  end;
end;



{**************************************
* A -> scalar*A
* Multiplies the vector by scalar
* Tested OK
**************************************}
procedure TVector.mult(scalar: Single);
begin
  Fx := Fx * scalar;
  Fy := Fy * scalar;
end;



{**************************************
* A -> scalar/A
* Divides the vector by scalar
* Tested OK
**************************************}
procedure TVector.divide(scalar: Single);
begin
  try
    Fx := Fx / scalar;
    Fy := Fy / scalar;
  except
    on EDivByZero do
      MessageDlg('Divide by 0 error in TVector class divide method.', mtError,
        [mbOk], 0);
  end;
end;



{**************************************
* A -> A+vec
* Adds vec to the vector object
* Tested OK
**************************************}
procedure TVector.add(vec: TVector);
begin
  Fx := Fx + vec.Fx;
  Fy := Fy + vec.Fy;
end;
```

```
{**************************************
* A -> A-vec
* Subtracts vec from the vector object
* Tested OK
**************************************}
procedure TVector.sub(vec: TVector);
begin
  Fx := Fx - vec.Fx;
  Fy := Fy - vec.Fy;
end;



{**************************************
* A -> vec
* Copies the fields of vec to the vector object
* Tested OK
**************************************}
procedure TVector.copy(vec: TVector);
begin
  Fx := vec.Fx;
  Fy := vec.Fy;
end;



{**************************************
* A -> A rotated anti-clockwise through theta degree
* Rotates the vector object by theta degree
* Tested OK
**************************************}
procedure TVector.rotateDeg(theta: Single);
var
  tempFx: Single;
  tempCos, tempSin: Extended;
begin
  tempFx := Fx;
{$IFDEF MATHUNITAVAILABLE}
  SinCos(theta * PI / 180.0, tempSin, tempCos);
    // Faster than calling Sin & Cos separately
{$ELSE}
  tempCos := Cos(theta * PI / 180.0);
  tempSin := Sin(theta * PI / 180.0);
{$ENDIF}
  Fx := Fx * tempCos - Fy * tempSin;
  Fy := Fy * tempCos + tempFx * tempSin;
end;



{**************************************
* A -> A rotated anti-clockwise through phi radian
* Rotates the vector object by phi radian
* Tested OK
**************************************}
procedure TVector.rotateRad(phi: Single);
var
  tempFx: Single;
  tempCos, tempSin: Extended;
begin
  tempFx := Fx;
{$IFDEF MATHUNITAVAILABLE}
  SinCos(phi, tempSin, tempCos);    //Faster than calling Sin & Cos separately
{$ELSE}
  tempCos := Cos(phi);
  tempSin := Sin(phi);
{$ENDIF}
  Fx := Fx * tempCos - Fy * tempSin;
  Fy := Fy * tempCos + tempFx * tempSin;
end;
```

```
{*************************************
* A . vec
* Returns dot or inner product of the vector object and vec
* Tested OK
*************************************}
function TVector.dot(vec: TVector): Single;
begin
  Result := Fx * vec.Fx + Fy * vec.Fy;
end;



{*************************************
* unsigned angle between A & vec degree
* Returns the angle between the vector object and vec
* by calculating theta = arccos((A . vec) / (|A|.|vec|))
* Tested OK by hammering this with vectors whose components
* were randomly generated using (random-0.5)*3*10e15 and
* (random-0.5) as two separate tests.
*************************************}
function TVector.angleDeg(vec: TVector): Single;
var
  param: Extended;
begin
  try
    param := self.dot(vec) / (self.abs * vec.abs);
    if param > 1.0 then
      {Floating Point errors will cause ArcCos range exception.
       Correct this before it happens}
      param := 1.0
    else if param < -1.0 then
      param := -1.0;
    Result := ArcCos(param) / PI * 180.0;
  except
    {handle case of magnitude of either vector = 0}
    on EDivByZero do
      Result := 0;
  end;
end;



{*************************************
* unsigned angle between A & vec radian
* Returns the angle between the vector object and vec
* Tested OK by hammering this with vectors whose components
* were randomly generated using (random-0.5)*3*10e15 and
* (random-0.5) as two separate tests.
*************************************}
function TVector.angleRad(vec: TVector): Single;
var
  param: Extended;
begin
  try
    param := self.dot(vec) / (self.abs * vec.abs);
    if param > 1.0 then
      {Floating Point errors will cause ArcCos range exception.
       Correct this before it happens}
      param := 1.0
    else if param < -1.0 then
      param := -1.0;
    Result := ArcCos(param);
  except
    {handle case of magnitude of either vector = 0}
    on EDivByZero do
      Result := 0;
  end;

end;
```

```
{*************************************
* signed clockwise angle between A & vec degree
* Returns the angle between the vector object and vec
* ie. returns the angle from self which must be rotated through in a clockwise
* direction to obtain a vector coincident with vec.
* The range of the return value is thus 0 -> 360 degree
* Tested OK
*************************************}
function TVector.signedAngleDeg(vec: TVector): Single;
var
  temp1, temp2: Extended;
begin
  try
    temp1 := ArcTan(self.y/self.x);
    {correct ArcTan for quadrant}
    if self.x < 0.0 then
      temp1 := temp1 + PI
    else if self.y < 0.0 then
      temp1 := temp1 + 2 * PI;
  except
    if self.y >= 0.0 then
      temp1 := PI/2
    else
      temp1 := -PI/2;
  end;


  try
    temp2 := ArcTan(vec.y/vec.x);
    {correct ArcTan for quadrant}
    if vec.x < 0.0 then
      temp2 := temp2 + PI
    else if vec.y < 0.0 then
      temp2 := temp2 + 2 * PI;
  except
    if vec.y >= 0.0 then
      temp2 := PI/2
    else
      temp2 := -PI/2;
  end;

  Result := (temp1 - temp2) / PI * 180.0;
  if Result < 0.0 then
    Result := Result + 360;
end;



{*************************************
* signed clockwise angle between A & vec radian
* Returns the angle between the vector object and vec
* ie. returns the angle from self which must be rotated through in a clockwise
* direction to obtain a vector coincident with vec.
* The range of the return value is thus 0 -> 2*PI radian
* Tested OK
*************************************}
function TVector.signedAngleRad(vec: TVector): Single;
var
  temp1, temp2: Extended;
begin
  try
    temp1 := ArcTan(self.y/self.x);
    {correct ArcTan for quadrant}
    if self.x < 0.0 then
      temp1 := temp1 + PI
    else if self.y < 0.0 then
      temp1 := temp1 + 2 * PI;
  except
    if self.y >= 0.0 then
      temp1 := PI/2
    else
      temp1 := -PI/2;
  end;


  try
    temp2 := ArcTan(vec.y/vec.x);
    {correct ArcTan for quadrant}
    if vec.x < 0.0 then
      temp2 := temp2 + PI
    else if vec.y < 0.0 then
      temp2 := temp2 + 2 * PI;
  except
    if vec.y >= 0.0 then
```

```
      temp2 := PI/2
    else
      temp2 := -PI/2;
  end;

  Result := temp1 - temp2;
  if Result < 0.0 then
    Result := Result + 2 * PI;
end;



{*************************************
* |A|
* Returns magnitude of the vector object
* Tested OK
*************************************}
function TVector.abs: Single;
begin
  Result := Sqrt(Sqr(Fx) + Sqr(Fy));
end;



(*****************************************************************************)
{
This test code attached to a button was used to test the above functions.
The input values come from 4 edit boxes vecx,vecy,vecbx,vecby.
The result values are sent to 2 edit boxes resx,resy.

procedure TForm1.CalcClick(Sender: TObject);
var
  vec, vecb, res: TVector;
begin
  // Construct required vectors
  vec := TVector.Create;
  vecb := TVector.Create;
  res := TVector.Create;
  try
    // read source vector components
    vec.x := StrToFloat(vecx.Text);
    vec.y := StrToFloat(vecy.Text);
    vecb.x := StrToFloat(vecbx.Text);
    vecb.y := StrToFloat(vecby.Text);

    // now perform some vector math
    res.copy(vec);
    res.sub(vecb);                    //Change this line to test methods

    // display results of vector operation
    resx.Text := FloatToStr(res.x);
    resy.Text := FloatToStr(res.y);
  finally
    // Destroy all allocated vectors
    vec.Free;
    vecb.Free;
    res.Free;
  end;
end;
}

end.
```

Listing Swarm.dpr

```
{
Terrain Mapping Simulator
Delphi Project File

File:         swarm.dpr
Author:       Gary Ruben
Date:         Aug 1998
Description:  This is the "main" or project file of the Terrain Mapping
              Simulator.
              The project has been developed using Borland (Inprise) Delphi 2
              on a PC-80486-DX2-66 with 20M RAM.
              It has not been tested on any other systems.
              It is best run at 1024x768 pixels with a full colour graphics
              card.
              It compiles to a single executable file which at last check
              was 348k in size.

              Support files required for execution are an agent initialisation
              file and a terrain map bitmap (.BMP) file which may be generated
              with any PC graphics package. The agent initialisation file
              contains a list of agent's with initial position and direction
              vector data. It must be named init.dat
              The terrain map bitmap file is typically a square greyscale file.
              The swarm.exe program initally contains a sample .BMP file.

              No on-line help has been written yet, so this will have to
              suffice:

              The simulation environment is divided into 5 panes, a Controls
              group and a Parameters group. There is a control bar along the
              top and a status bar along the bottom.

              The top left pane contains the terrain data. This may be changed
              by double-clicking on the pane with the left mouse button.

              The top right pane shows the agent positions. This may be
              disabled by unchecking the associated "Animate" checkbox to speed
              up the simulation.

              The bottom left right pane shows the agent paths. The "All Paths"
              and "Path of Id" RadioButtons in the Controls group control what
              is displayed in the Paths pane.
              The default selection "All Paths" will cause the paths of all
              Roaming agents to be displayed modulated by the terrain map pixel
              value.
              If "Path of Id" is selected, the agent with the Id determined by
              the edit box will be displayed.

              The bottom right "Map" pane shows the map data which has been
              accumulated by the mapping agent. Data from the internal map
              which is displayed in the "Internal Map" pane is copied to the
              "Map" pane when the survey of the current region has been
              completed.
              The "Map" and "Internal Map" pane drawing may be disabled by
              unchecking the associated "Animate" checkbox to speed
              up the simulation.

              The bitmaps contained in any pane may be copied to the Windows
              clipboard by right clicking on the pane and selecting Copy from
              the popup menu.

              In addition to allowing selection of the agent paths displayed,
              the Controls group contains a Debug window checkbox, which if
              checked, will show agent position, direction and state
              information. This is unlikely to be of use to anyone running the
              program.

              The Parameters group contains the "Map Threshold" and
              "Side Length" settings.
              The "Map Threshold" setting controls when the Mapping agent
              signals reference agents to take up new positions in the
              terrain. A coverage value of 0.4 requires that 40% of the pixels
              covering the region of the internal map represented by the lime
              green square in the "Internal Map" pane have been traversed by a
              Roaming agent. The "Map Ratio" value below the "Internal Map"
              pane shows the current coverage value and the progress bar below
              it reaches full when the coverage matches the threshold.
              The "Side Length" setting determines the length of the side of
              the equilateral triangles which are used to tesselate the survey
              area. That is, the distance a reference agent uses to establish
              its position in the terrain relative to othe reference agents.
              Note that setting this to less than half the value of the current
```

```
              distance between 2 reference agents when the 3rd moves will mean
              that the 3rd agent can never establish a reference position.
              Instead, a Pavlov's dog behaviour results where the agent
              continually runs back and forth between the other reference
              agents.

              The buttons on the control bar along the top control the
              simulation execution and mirror the "Run" menu controls.
              With these, the simulation may be reset, single-stepped, run or
              stopped.
              The status bar along the bottom displays the current simulation
              iteration value.

              For any copyright issues associated with the use of this code,
              contact the Electrical and Computer Systems Engineering
              department at Monash University.
}

program Swarm;

uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1},
  Agent in 'Agent.pas',
  Vector in 'Vector.pas',
  AgentManager in 'AgentManager.pas',
  Map in 'Map.pas',
  DebugUnit in 'DebugUnit.pas' {DebugForm};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TDebugForm, DebugForm);
  Application.Run;
end.
```

Listing of example Init.dat

```
;m=mapping b=beacon r=roaming
;format  [type] [x posn] [y posn] [x dirn] [y dirn]
;        [m|b|r] (0..MAX_X) (0..MAX_Y) (-1.0..1.0) (-1.0..1.0)
; eg. m 10 15 1 1


r 50 50 0 1
r 55 50 0 1
r 60 50 0 1
r 65 50 0 1
r 70 50 0 1

b 30 30 0 1
b 15 70 0 1
m 60 60 0 1
```

Bibliography

[BaRa84] Bannister, A. & Raymond, S., "Surveying Fifth edition", Longman Scientific & Technical, 1984.

[Bri95] Britannica CD 2.0, Encyclopædia Britannica, 1995

[Bro86] Brooks, R., "A Robust Layered Control System for a Mobile Robot", *IEEE Journal of Robotics and Automation, Vol.RA-2 No.1*, 1986

[Bro89] "A Robot that Walks; Emergent Behaviours from a Carefully Evolved Network", *Neural Computation*, pp.253-262, 1989.

[BeHo94] Beckers, R., Holland, O.E. & Deneubourg, J.L., "From Local Actions to Global Tasks: Stigmergy and Collective Robotics", *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, MIT Press, pp181-189, 1994.

[Cal95] Calvert, C., "Delphi Unleashed", Sams Publishing, 1995.

[Coh96] Cohen, W., "Adaptive Mapping and Navigation by Teams of Simple Robots",*Robotics and Autonomous Systems Vol.18, No.4*, pp.411-434, 1996.

[CoSt95] Cornell, G. & Strain, T., "Delphi Nuts & Bolts for Experienced Programmers", Osborne McGraw-Hill, 1995.

[DeGo89] Deneubourg, J.L. & Goss, S., "Collective Patterns and Decision Making", *Ethology, Ecology and Evolution I*, pp.295-311, 1989.

[DuJe93] Dudek, G., Jenkin, M., Milios, E. & Wilkes., D., "A Taxonomy for Swarm Robots", *Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, Yokohama, Japan*, pp.441-447, July 26-30, 1993.

[GoMa97] Goldberg, D., Matarić, M., "Interference as a Tool for Designing and Evaluating Multi-Robot Controllers", *Proceedings, AAAI-97, Providence, Rhode Island, July 27-31*, pp.637-642, 1997

[HuGl96] Huberman, B.A., Glance, N.S., "Evolutionary Games and Computer Simulations", downloaded from Xerox internet site, to appear in *Proceedings National Academy of Sciences (USA)*, 1996.

[HuMe94] Husbands, C., Meyer, Wilson eds., “From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behaviour.”, MIT Press, 1994

[KoMe95] Kodjabachian, J., Meyer, J., “Evolution and development of control architectures in animats”, *Robotics and Autonomus Systems 16*, pp.161-182, 1995.

[Lan95] Langton, C. ed, “Artificial Life: an overview”, MIT Press, 1995.

[Lev92] Levy, S., “Artificial Life: The Quest for a New Creation”, Pantheon Books, 1992.

[Lit94] Littman, M.,“Memoryless policies: theoretical limitations and practical results”, *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behaviour, Husbands, C., Meyer, Wilson eds.*, MIT Press, pp.238-245, 1994

[LuSp96] Luke, S., Spector, L., “Evolving Teamwork and Coordination with Genetic Programming”, downloaded from internet site, to appear in *Proceedings Genetic Programming 96 (GP96), Stanford*, July 1996.

[MaNi95] Matarić, M., Nilsson, M., Simsarian, K.T., “Cooperative Multi-Robot Box-Pushing”, *Proceedings IROS-95, Pittsburgh, PA*, 1995.

[Mat94] Matarić, M., “Interaction and Intelligent Behaviour”, MIT PhD thesis, 1994, downloaded from MIT internet site.

[Mata94] Matarić, M.,“Learning to Behave Socially”, *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behaviour, Husbands, C., Meyer, Wilson eds.*, MIT Press, pp.453-462, 1994.

[Mat95] Matarić, M., “Issues and Approaches in the Design of Collective Autonomous Agents”, *Robotics and Autonomous Systems 16*, pp.321-331, 1995.

[Mat97] Matarić, M., “Using Communication to Reduce Locality in Multi-Robot Learning”, *Proceedings AAAI-97, Providence, Rhose Island, July 27-31*, pp.643-648, 1997.

[McF94] McFarland, D., “Towards Robot Cooperation”, *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behaviour, Husbands, C., Meyer, Wilson eds.*, MIT Press, pp.440-444, 1994

[MiMa97] Michaud, F., Matarić, M., "Behaviour Evaluation and Learning From an Internal Point of View", *Proceedings FLAIRS-97, Daytona, Florida*, May 1997.

[Mue96] Mueller, J.P., "Peter Norton's Guide to Delphi 2", Sams Publishing, 1996.

[PaTe96] Pacheco, X. & Teixeira, S., "Delphi 2 Developer's Guide", 1996, Borland Press.

[Pfe95] Pfeifer, R., "Cognition - Perspectives from autonomous agents", *Robotics and Autonomous Systems 15*, pp.47-70, 1995.

[PrUr89] Price, W.F. & Uren, J., "Laser Surveying", VonNostrand Reinhold, 1989.

[ScMa96] Schneider-Fontán, M., Matarić, M., "A Study of Territoriality: The Role of Critical Mass in Multi-Robot Adaptive Task Division", *From Animals to Animats 4, Fourth Conference on Simulation of Adaptive Behaviour (SAB-96)*, pp.553-561, 1996

[ShFu93] Shibata, T., Fukuda, T., "Coordinative Behaviour in Evolutionary Multi-Agent-Robot System", *Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, Yokohama, Japan*, pp.448-453, July 26-30, 1993.

[Ste95] Steels, L., "The Artificial Life Roots of Artificial Intelligence", *Artificial Life - An Overview, ed. Langton, C.*, MIT Press, pp.75-110, 1995

[SuSu90] Sugihara, K. & Suzuki, I., "Distributed Motion Coordination of Multiple Mobile Robot", *IEEE International Symposium on Intelligent Control*, Philadelphia, PA, pp. 138-143, September 1990.

[Uns93] Ünsal, C., "Self-organization in Large Populations of Mobile Robots", M.S. Thesis -Virginia Polytechnic Institute and State University, 1993, downloaded from internet site.

[Wan94] "On Sign-board Based Inter-Robot Communication in Distributed Robotic Systems", *IEEE International Conference on Robotics and Automation, Vol.2*, pp.1045-1050, 1994.

[XiVe94] Xia, F., Velastin, S.A., Davies, A.C., A Parallel Simulation of Multiple Mobile Robots Using the DORIS Design Method, *IEEE International Conference on Robotics and Automation Vol. 3*, pp.2482-2487, 1994.