

CONTENTS – VOLUME 1

Glossary	4
Symbology	7
Chapter 1	Introduction
1.1	The Notion of Emergence 10
1.2	Project Objectives 11
Chapter 2	Simulator System Design
2.1	Introduction 16
2.2	System Requirements 17
2.2.1	Introduction 17
2.2.2	System Model 22
2.2.3	Non-Functional Requirements 23
2.2.4	Requirements Specification 24
2.3	System Design 29
2.3.1	Introduction 29
2.3.2	Testing Strategy & Development Method 30
2.3.3	The Core System 33
2.3.4	Environment 35
2.3.5	Intentions 38
2.3.6	Agents 40
2.3.7	Arbitrator 47
2.3.8	Administrator 49
2.3.9	Prototyping Discoveries 53
2.3.10	Graphics 53

2.4 Life-Cycle Modifications and Issues

2.4.1	Introduction	55
2.4.2	Creating a Graphical Interface	55
2.4.3	Adding a Map Editor	60
2.4.4	Reducing Environment Complexity	62
2.4.5	Opportunities for Additional Experiments	62
2.4.6	Improving the Path-Searching Algorithm	62
2.4.7	Arbitration Classes	63

Chapter 3 Experimentation

3.1	Experiment 1 - The Mining Experiment	66
3.2	Experiment 2 - The Hunting Experiment	78
3.3	Experiment 3 - The Chain-Mining Experiment	92
3.4	Experiment 4 - The Crowd Dynamics Experiment	124

Chapter 4 System Applications & Limitations

4.1	Introduction	139
4.2	Applications of the System	139
4.3	Limitations of the System	140

Chapter 5 Project Management

5.1	Changes to the Plan	148
5.2	The Project Overall	150

CONTENTS – VOLUME 2

Appendix

Technical Notes	152
System Code	192
Acknowledgements	356
Bibliography	357

Glossary

A* Search	A famous and powerful method of computing the best route from A to B, bearing in mind anything that may lie along the route from A to B.
Agent	In this context, a piece of software that attempts to model an imaginary pseudo-organism as a distinct individual within the simulated environment.
Antecedent / Consequent	In this context, antecedent means “if” and consequent means “then”. This is a way of describing a rule that has a condition, which if satisfied, yields an effect.
ASCII	This stands for the American Standard Code for Information Interchange. It allows data to be recognised via a certain code for each character symbol, so that such data can cross hardware and software boundaries, yet still maintain its meaning.
Base Class	A concept where it can be determined that a collection of objects exist, such that for every object in the collection, each one contains a definition where all such objects have such a definition in common. The definition is always fundamental, for instance, the class bird is the base class of a parrot.
Behavioural Definition	A kind of rulebook inside an agent that determines how it behaves, according to things that are currently resident in its memory. It is therefore a subjective behaviour as there is the possibility that the agent’s memory might be incorrect. This is more correctly known as a “Production System.”
Black-Box Testing	This is where you test a function by comparing the actual outputs with the expected outputs. You do not consider the internal functionality of the function being tested.
Boolean Variable	This is a program variable that either has a true or false value.
Cartesian Coordinate	A pair of numbers that allow a location to be found in a two-dimensional grid, denoted as (column number, row number). For our purposes, the origin is the top-left.
Comparison Operator “<”	This is the default comparison method of two objects of the same type within a class. We are able to define its behaviour ourselves. However, if we wish to perform many types of comparison, we either have to state new operators from a specific list or define Comparison Predicate Classes.
Comparison Predicate	We often find ourselves having to compare objects of the

Classes	same type in various ways. Comparison predicates state the nature of such a comparison.
Console Application	Visual C++ Version 5 is a C++ development system that is naturally designed to create Windows 95/98™ applications. However, it is possible to create what are known as “console applications” that do not take full advantage of the Windows 95/98™ operating system and are thus simpler to produce.
Constructors	These are class functions that say what to do when a class instance is created (see Base Class).
CPU	Stands for Central Processing Unit. This is the processor in a computer.
Direct Addressing	This allows us to rapidly fetch items from a contiguous list. We need only specify an item’s location in the list to obtain it, rather than having to search the list until we find it.
Dynamically Bound	This is a process in C++ where the class object function being referred to is either present in the derived class or base class. The system will execute the base class variant of the function unless a derived variant exists.
Effectors	This is the means by which an agent affects its environment.
Emergent Properties	This is a function, behaviour or structure that manifests itself within a mechanism due to the interactions of its constituent members.
Gradient Field	A source that emits detectable tokens that diminish in strength the further away you are from the source. This is analogous to a scent that positively increases in strength as you get closer to its source.
GUI	This stands for Graphical User Interface.
Heuristic	A heuristic is like a guiding method that tells you how close you are to your target solution.
Inheritance	An object-oriented programming concept where objects may inherit the attributes of another object, because they are an extension of the object being inherited.
Injective	This is a function where each element of one domain is mapped to only one corresponding element in another domain.
Iterative Phased Development	Starting with a basic system structure and then adding more functionality within each iterative step. The effect is that the system becomes gradually more sophisticated, thus

providing some degree of protection against major design flaws. It is particularly useful when it is not possible to envisage the entire system right from the start of the design process.

Java	This is another more truly object-oriented language that allows much faster application development and simplifies most of the areas that C++ tends to complicate. However, Java's execution speed is very slow in comparison.
Life-Cycle	This is a bit of software design jargon that just means the time-span of the development phase.
Manhattan Distance	This is a form of Heuristic that calculates a value, representative of the distance between a current location and a goal location.
Mapping	This is a bit of mathematical jargon that simply means: if we plug in values at one end of the function, we always reach a predictable value at the output end of the function.
Multi-Agent System	A system that contains collections of agents with the intention to model the result of their interactive behaviours and impact on their shared environment.
Object-Orientation	A programming method that defines an abstract entity called an object that possesses data and functions within itself that are private to the object. The services of the object may only be accessed via public functions defined in the object, thus the data within the object is encapsulated and protected from any unintentional alteration. The object can be thought of as having behaviours relevant to its type, so it improves system coherence.
Operators	These are class methods that allow some form of effect to be carried out between two objects of the same class.
Percepts	This is the means by which an agent receives information about its environment.
Pointers	In C++, these are methods that allow you to refer to an object, rather than copy it and handle it directly.
Production System	A collection of rules that dictate an agent's reaction to information stored in its memory
Reflex Agent	This means that the agent has a triggered response to information in its memory. The response activates on the existence of items in memory and does not occur through any deliberation or planning.

Sound Behavioural Definition	See Behavioural Definition. This determines the agent's reaction to events in the grid environment that cause it to generate a sound.
Standard Template Library (STL)	This is a standard collection of methods included in C++ that provide low-level services such as lists, queues and vectors.
Substitutability	In Inheritance, all instances of derived classes must be capable of being substituted back for the Base Class. You should not weaken the base definition, only add to it.
Vector	A direction within a plane that has a straight line trajectory.
Vector (STL)	A Vector is a contiguous block of items that may be addressed very rapidly due to their contiguous organisation in the computer's memory.
Vector Boundary Checking	A Vector will always have a specific total number of values that it holds. If we use Direct Addressing on a Vector, the Vector will not inform us if we try to query an element that is outside the boundaries of the current Vector contents.
Virtual Function	This is an aspect of Inheritance in C++. A virtual function may be overridden in a derived class to restate the functionality of the inherited function.
White-Box Testing	This is where you design a test in such a manner that it interrogates the internal functionality of the function being tested. (See Black-Box Testing for testing outputs).

Symbology

Appendix Information

Generally, we have kept all technical information away from the report body and have situated it in the Appendix within Volume 2. Throughout this document, we have used a special notation to remind the reader of the existence of an appendix item and the page at which it may be found, whilst they are reading the associated text. Whenever you view the following graphic and information enclosed within the box, you shall know that an appendix item relevant to the current topic exists at the reference and page specified:

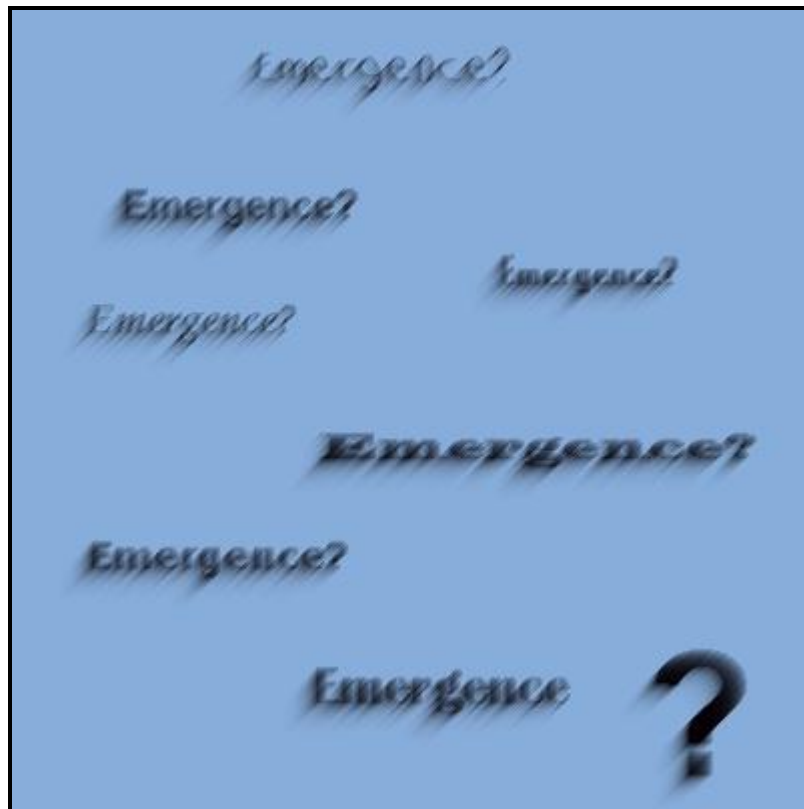


Code Examples

Any example code in this document is illustrated in italicised text unless it occurs in the Appendix.

CHAPTER 1

INTRODUCTION



1.1 The Notion of Emergence

Emergence is a difficult concept to define. If we allow ourselves to simply believe that emergence should be a surprise, or an unexpected phenomenon, then there is a degree of fallibility in our viewpoint. We could reason that an experienced individual might not be surprised at all, whereas an inexperienced individual might insist that they had no idea that the property would emerge. Emergence is perhaps better defined as a sum of parts. If we examine each individual part, all we perceive are its attributes and functions. But if we stand back and observe the whole, we begin to see patterns of interactions between the parts that we had not explicitly designed ourselves at that level of complexity. (Humphreys, 1997) offers six possible criteria for emergent phenomena using examples from macroscopic physics:

1. Novelty - a previously uninstantiated property comes to have an instance
2. Qualitative Difference - from the properties from which they emerge
3. Resolution - the property could not be possessed at a lower level
4. Laws - different laws apply to the emergent feature than the laws from which it emerges
5. Interactions - between their constituent members
6. Holistic - properties of the entire system rather than its local constituents

Although the author uses physics to illustrate the ideas, we can apply this work to our own realm of Artificial Intelligence. He mentions in this paper that the first criterion is not interesting because we are not seeking a new value of an existing property. This appears to fit with our initial argument that surprises are possible fallible viewpoints. He also states that not all criteria need to be satisfied in order for an emergent feature to exist.

There are other viewpoints on emergence. Rather than naively make a decisive claim that one is better than another, we shall define emergence for our purposes only to mean, **“a sum of parts, that exist in a system, due to the interactions of its constituent members.”**

Generally, in the field of Artificial Intelligence, three major categories of emergent phenomena may be identified:

1. Emergent Behaviour

Agents performing actions that had not been explicitly designed

2. Emergent Functionality

A function has arisen out of the phenomena

3. Emergent Structure

A pattern has arisen out of the phenomena

We illustrate examples of all of these categories in Chapter 3.

1.2 Project Objectives

Our main inspiration for this project was from the work of Mataric (1995). In this paper, the author describes a method of using base behaviours in mobile robots that facilitated emergent phenomena, such as flocking. We provide more discussion on this paper in Chapter 3.

There was a dual-pronged focus to this project:

1. Design and build a multi-agent simulation system

2. Experiment with the system to illustrate emergent phenomena

Essentially, we aimed to design and implement a general process from which homogeneous or heterogeneous groups of agents could be created, without having to make extensive alterations to the simulation backbone. This has been achieved, and we have leapt beyond our original objectives by creating a Windows 95/98™ interface. The interface has allowed us to perform experiments using different classes of agent and view their collective behaviours with the luxury of real-time or playback animated graphics.

We have utilised quite a vast array of programming topics that have been taught in this degree period. In some cases, we have extended beyond the scope of taught material in order to build upon the fundamental concepts and put them into practice.

Chapter 2 deals with the system design, starting at the lowest levels of abstraction, then gradually building up into a collection of coherent functional components that satisfy the system needs. We present the original design plan, edited for inclusion in this report, then we discuss the alterations and additions of features.

Chapter 3 charts a series of experiments using the simulation system. Several scenarios, analogous to real-world situations, have been created with varying emergent phenomena. The results of each experiment are illustrated and put into context.

We recommend that you familiarise yourself with the simulator's features before reading Chapter 3. These may be found in appendices T.5 and T.6.



T.5 - The Animate Main Interface - Page 173



T.6 - The Animate Map Editor - Page 179

Chapter 4 discusses the applications of emergent phenomena and limitations of the simulator with a suggestion as to how emergent phenomena may be used to solve problems in industry.

Chapter 5 reviews the project management plan that enabled this study to be completed. Initially, we take a broad look at how the objectives were originally structured and then we discuss the impact of adding new tasks, plus any associated difficulties encountered.

The **Appendix** in Volume 2 contains technical information regarding important features of the simulator and the simulator system code.

An **Acknowledgements** section has also been provided to credit those individuals who have played a beneficial role in this study.

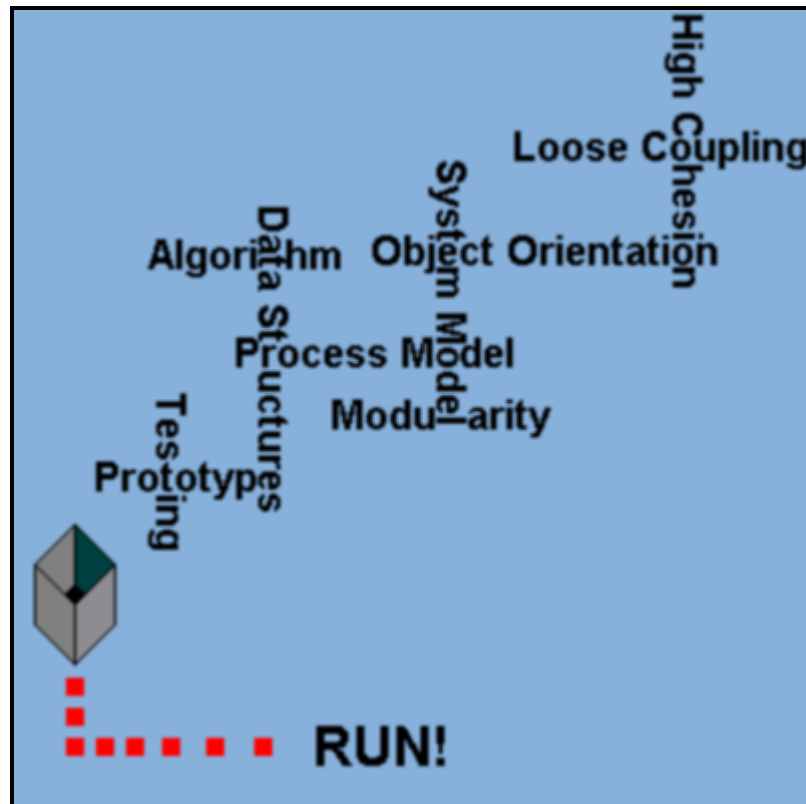
We were required to produce this report for a variety of readers. Some of those readers are not experts in the field of Artificial Intelligence but experts in other fields. Additionally, we are required to use a simplistic style and avoid the use of technical jargon.

This does not pose a problem in the case of the design sections of this report under Chapter 2. We may refer to design elements casually and metaphorically because they are things that do not yet exist and we have not used formal specification techniques. However, using metaphors when discussing software agents is clearly non-scientific and ambiguous. For this matter, Appendix T.1 provides a much more scientific discussion about the agents and the simulator system. Additionally, other appendices are quoted throughout this report that tackle interesting issues in a more formal manner.

CHAPTER 2

SIMULATOR SYSTEM

DESIGN



2.1 Introduction

This chapter is based on a design document that was completed during an earlier stage in this project. We have preserved the contents of the original document and edited its form for inclusion in this report. However, where the appendices in the original document referred to incomplete functions for the individual classes, we have substituted the final system code. There are some references to design matters that have not been implemented in the final system, or they have been altered. We have highlighted the most significant of these changes in section 2.4.

This introduction sets out the reasons for the software need. Section 2.2 illustrates the problem domain from a highly abstract viewpoint, allowing concepts to be absorbed before delving any deeper into the more complex issues. Section 2.3 finally hits the target with much more explicit design detail.

The Software Need

The user of this system shall be conducting experiments on the emergent properties of multi-agent systems. The designer need not concern themselves with the issues of emergent phenomena. The system that the user requires is a simulator that is capable of simulating an environment within which collections of agents deliberate and affect their surroundings. The user intends to add and redefine code within the finished system, so the system will be designed in such a manner that allows changes to be made with ease. Therefore, the components of the system must be as generic and coherent as possible, so

that dependencies between modules are not so specific that they prevent a change in system behaviour from being carried out swiftly.

What are Agents?

Before going any further, it is essential to gain an initial understanding of what the term “agent,” represents to the user. The agents that the user requires are artificial software entities that are capable of perceiving their environment, updating their memory as a result of perception and then performing actions on the environment. Actions will be a result of considering items in memory via a behavioural definition. This behavioural definition is like a rulebook that states what to do in whatever circumstance may arise that is of significance to the agent.

2.2 System Requirements

2.2.1 Introduction

This section provides an introductory guide into the problem domain. It will not be possible to set out every function and attribute that is required of the system in advance, as its use is for research experimentation.

First, we define what agents will all have in common. Then we discuss the agents' environment. Finally, we consider how we might manage a collection of agents in an

environment and simulate the passage of time and interactions of all the agents and objects within the environment.

Agents

All agents should possess a base set of fundamental abilities. Therefore, a base class of agent should represent the recipe from which all agents may be derived. The designer of the system need only consider the functionality of the base class agent, but must bear in mind that a heterogeneous collection of agents, derived from the base class, may be needed in any system run.

Let's introduce ourselves to the concept of a base class agent by considering the characteristics of the chap illustrated in the following cartoon:

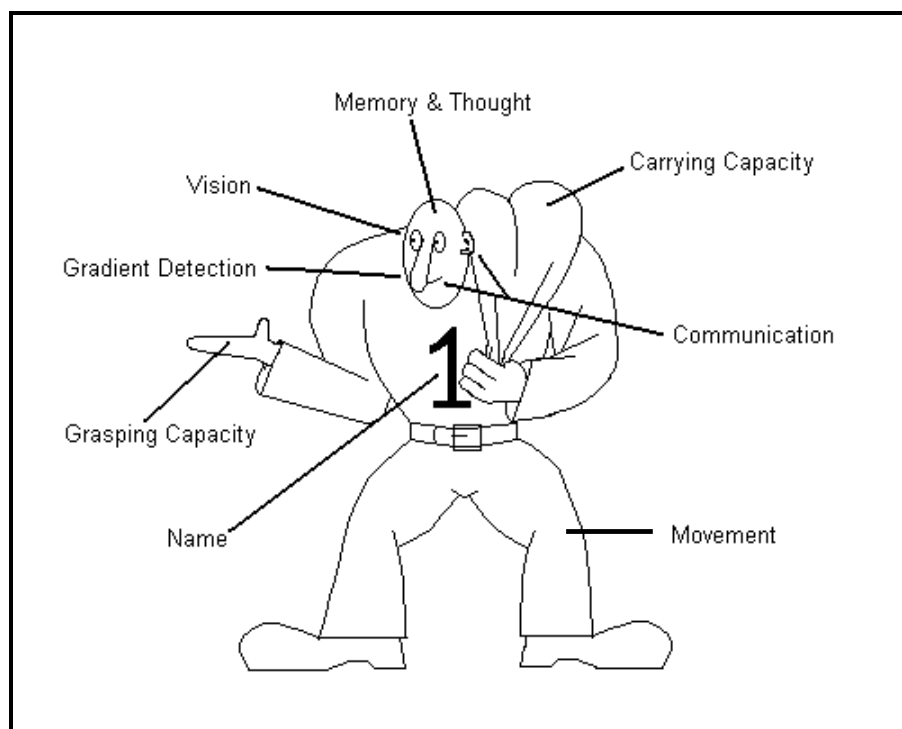


Fig 2.1 A Conceptualisation of the Base Class Agent

We all possess an intuitive feeling about what these categories represent, but we must simplify and restrict their meaning to fit the problem domain that interests us:

- **Memory and Thought**

Things that have been observed and goals being pursued.

- **Vision**

Receiving information from the environment, relative to the agent's location

- **Communication**

Making and perceiving sounds, relative to the agent's location

- **Gradient Detection**

Perceiving smells from the environment

- **Carrying capacity**

The maximum load that an agent can bear, in a rucksack

- **Grasping Capacity**

The maximum load that an agent can pick up at once in its hands in any turn

- **Movement**

Moving to an adjacent location in the environment, that is a space

- **Name**

An identification, making the agent an individual so that we may monitor it

These are the essential characteristics of each agent. Every agent created by the user will have these characteristics, some may be enhanced, whilst others ignored. However, all of them will be substitutable for the base class agent.

The Agents' Environment

The user requires a grid-based environment, analogous to a chessboard. The grid will have two dimensions, so there will be no concept of above or below. All grids should possess the following characteristics:

- Define only one possible object at any location
- Define locations as cartesian coordinate pairs
- Be able to define some objects as having a quantity
- Define all obstacles and other agents in the environment as impenetrable to an agent
- All grids will have a width that is equal to their height

The first item appears contradictory to the third, but the intention is to be able to specify an object as either taking up the entire space of the location, or having a quantity of itself that takes up the entire space of the location. For instance, an agent would take up an entire space, but a pile of sand would take up an entire space and have a quantity of grains. A pile of sand, having only one grain, would still take up an entire space. Simplicity is the underlying theme and we should not allow our understanding of our own environments to affect the design of the environment that the user has in mind. However, we should try to make the environment as believable as possible.

The following illustration adds some visual clarification to the concept of grids and also illustrates the possible directions that an agent may move to.

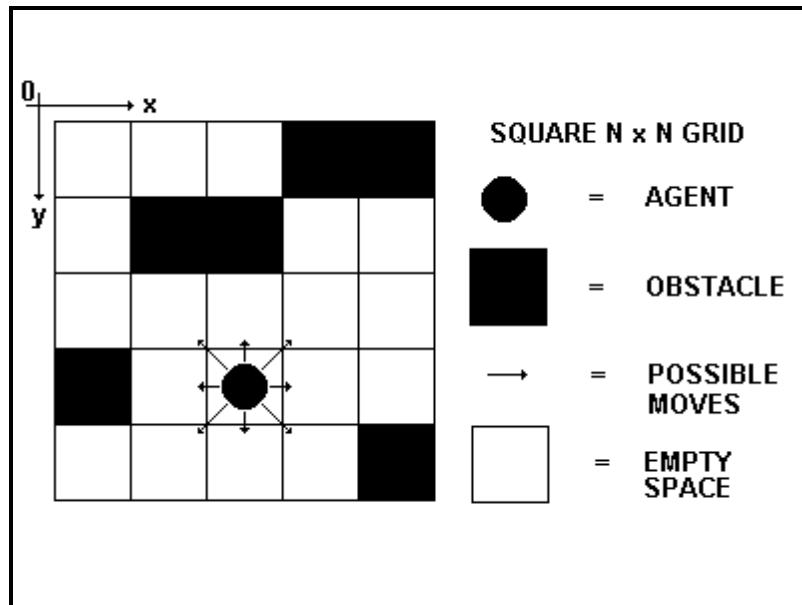


Fig 2.2 A Simple Two-Dimensional Grid

Managing the Simulation

It is impossible to adequately simulate a concurrent environment where anything may happen when it chooses to, on conventional computers. Therefore, the user desires a turn-based control of each moment in simulation time that allows everything to “say” what it would like to happen. This should be achieved by an administrator invoking every agent’s behavioural definition and then considering any conflicts of interest.

The most typical conflict is where an agent wishes to perform the same action at the same location as another agent. Imagine yourself walking down a street and encountering another person who wishes to step into the location that you wish to step into. Usually, a swift non-verbal gesture or a noble decline resolves the conflict of interests. You pass each other by and go about your business as if nothing had happened. Sometimes, you even predict the trajectory of all objects in your path and effortlessly weave a route through the

problem. It would require a considerably sophisticated system to allow agents to communicate a mutually acceptable solution or predict trajectories for such a conflict as efficiently as humans. Attention to such matters is an unnecessary distraction away from the whole point of the research topic. Therefore, we require an “unseen entity” that chooses an agent from the conflict as a winner and stops the loser from doing the same action. For this matter, an Arbitrator is desired to resolve such conflicts.

The Administrator will also be responsible for extracting data from the agents and their environment, setting up the initial state of the agents and the environment, and managing each turn until the set number of turns has been reached.

2.2.2 System Model

The following data-flow diagram provides an abstract, high-level viewpoint of the overall system. At this stage, we are only interested in how the various components fit together in order to obtain an initial picture of the system as a whole.

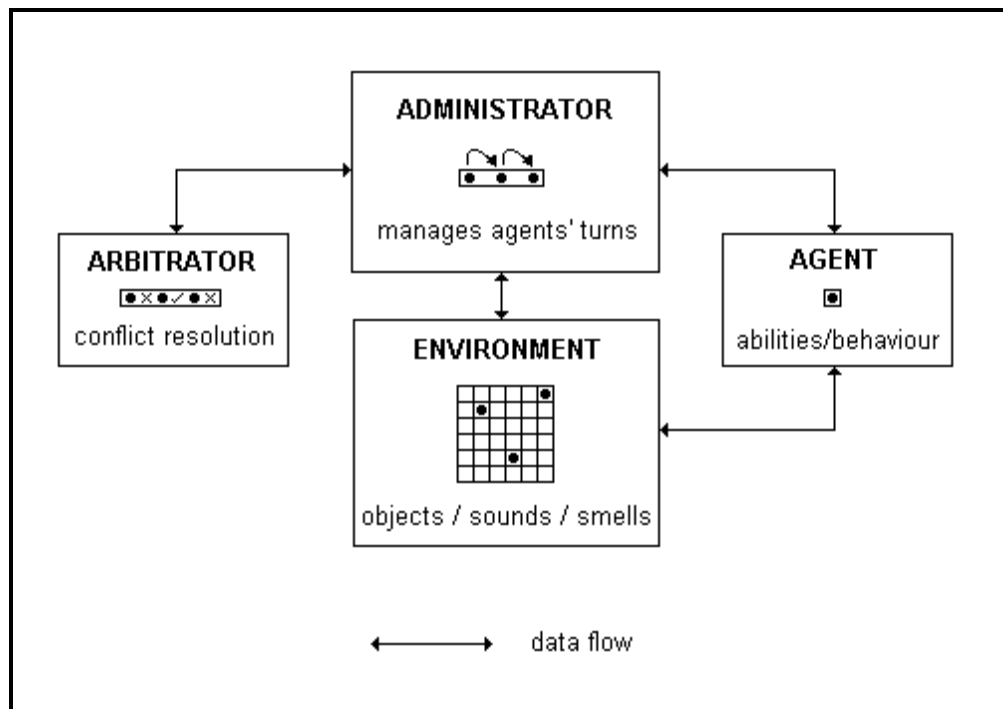


Fig 2.3 Data-Flow Model of the System

This system design has been prototyped in order to gain an understanding of its modular coherence and coupling relationships. Prototyping has identified the design as being most efficient in obtaining a satisfactory solution to the user's requirements.

2.2.3 Non-Functional Requirements

- **Agents**

The maximum number of agents should be infinite and thus within the bounds of the memory capacity of the operating system. The minimum should be one. It is not expected that agent numbers will ever be more than 30.

- **Artificial Environment**

Many accesses will be made to the environment, so it should perform any computation as swiftly as possible.

- **System**

The system should be implemented under Visual C++ Version 5 as a console application, using object-orientation and inheritance. There are no user interface requirements. There are no graphical requirements. All data input from files and output to files should be in the form of ASCII text.

2.2.4 Requirements Specification

In this section, we state more clearly and explicitly what it is that we are setting out to achieve. First, we shall consider the Agent, its composition and abilities. Then we shall discuss the functional attributes of the Environment. Our last topic will be the functional attributes of simulation management involving the Administrator and Arbitrator.

1. Agents

First, we consider the notion of the behavioural definition and then we shall add more detail to the abilities defined earlier in this chapter.

- **The Behavioural Definition**

The behavioural definition is now more precisely defined as follows:

The Administrator shall have the facility to invoke a single function within an agent that contains an ordering of behavioural functions, that represent the agent's behavioural definition.

- **Memory and Thought**

- a memory to hold objects perceived in the environment that persist until altered
- to be able to plan the shortest path to objects via memory
- to be able to wander aimlessly
- to be able to query memory for the existence of objects, either adjacent to the agent or somewhere else in the environment / memory
- to be able to pick up objects that are not other agents or obstacles within the allowance of the grasping and carrying capacity
- to be able to drop objects on a location that has enough capacity for the object
- to be able to forget all objects in memory, or all objects of a certain kind
- to be able to carry out a state that represents agent death
- to be able to allow a targeted location and object to persist in memory until a new target is set
- to be able to explore areas of the environment that have not yet been reached and memorised

- **Vision**

- a 360 degree perceptive range that acquires the environment into memory. The range should be definable by the user
- agents will be able to look over all objects and other agents in the environment
- **Communication**
 - to be able to perceive and make sounds
- **Gradient Detection**
 - to be able to perceive smells and follow them to their source. Only one smell should ever be active in the environment
- **Carrying capacity**
 - to define a load that an agent can carry. The agent's hands will still be free when the agent's carrying capacity is full. The analogy is of a backpack
- **Grasping Capacity**
 - to define a maximum load that an agent can pick up at once in any turn and place in its backpack
- **Movement**
 - to be able to move to one adjacent square in any turn in any direction
- **Name**
 - a number that identifies the agent and is unique

2. The Environment

The environment should possess the following attributes and functions:

- **Dimensions**

- Minimum size should be 10 by 10 squares
- Maximum size should be 40 by 40 squares

- **Functions**

- Represent locations as cartesian coordinate pairs
- Define obstacles as impenetrable to agents and other objects
- Set and get the contents of any location in the grid
- Initialise the environment
- Load an environment description from an ASCII text file
- Output an environment description to an ASCII text file
- Metamorphose an existing environment into a new dimension
- Define only one object at each location unless it is a collection of the same object and is not an agent or an obstacle

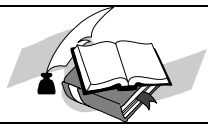
The environment must possess a method that allows new objects to be defined very rapidly. The objects that are required by the user are as follows:

- Uninhabited space
- Obstacles
- Agents
- Piles of minerals

The concept for these object definitions should transcend the entire software system.

3. The Arbitrator

We propose a process that arbitrates agents by their arrival time and then gives them a turn each, dependent on whether they still want to execute the same action at the same location. Details of how this process may be achieved occur in Appendix T.3.



T.3 - The Arbitration Process - Page 161

4. The Administrator

The Administrator orchestrates the entire simulation process but should not be responsible for making alterations to the environment on the basis of agents' actions. The Administrator should also be the point from which simulation settings may be set. These settings would be items such as the number of turns in the simulation, the number of simulations, how many agents to create and of what type.

The following functionality should be available within the Administrator definition:

- Create a collection of agents
- Initialise agents
- Read in an environment description from an ASCII text file before the simulation is running
- For every turn, allow every agent to deliberate using its behavioural definition

- Communicate with the Arbitrator to arbitrate the agents' intended actions
- Set the result of arbitration
- Allow every agent to execute its arbitrated action
- Allow the user to define a predicate that stops the simulation when a certain state has been achieved
- Provide for the output of every agent's state to an ASCII file
- Provide for the output of the state of the environment to an ASCII text file

It should be made possible within the Administrator to allow the user to easily define new functions that either query or alter the state of agents at any stage in the turn cycle, without fear of disrupting the integrity of the system. As the user will be researching methods to yield certain emergent behaviours, inevitable new rules and environmental processes will have to be introduced to obtain the desired results. It is simply not possible to deliver a system that does everything the user wants. Keeping the system behaviour generic and simple will allow the user to rapidly alter its processes, without having to make wide-scale alterations and deeply consider any knock-on effects.

2.3 System Design

2.3.1 Introduction

In this section, we consider the requirements set out in the previous sections in this chapter and propose a more technical solution to the problem. We shall produce some programming code examples, but mostly, we shall illustrate some general algorithmic activities and data structures that are most suitable for the topic under discussion. Any references to actual sample code are illustrated in italicised text throughout the remainder of this chapter.

Before taking any steps to formulate a design, we should consider what practices and disciplines we should adhere to, in order to maintain a consistent and methodical approach. Additionally, we should formulate a testing strategy that will complement the chosen design and try to anticipate the kinds of errors that tend to spawn from such a design. A programmer should be able to pick up this document and code a system without having to refer back to us for advice. We should aim defensively, to arm this person with the most appropriate methods so that we may consider this document to be a good design guide.

2.3.2 Testing Strategy & Development Method

The entire problem seems rather hazy at present, but we can make a start by considering the overall problem from a very wide and practical perspective. The intended system is for a multi-agent simulator that will not provide a finished product, but a simulation method that may be adapted to suit varying experimental problems. Therefore, the behaviour of the system cannot be decided right from the start, but we do know for certain what major components will occur in this design. Recall the system model that was introduced in section 2.2.2:

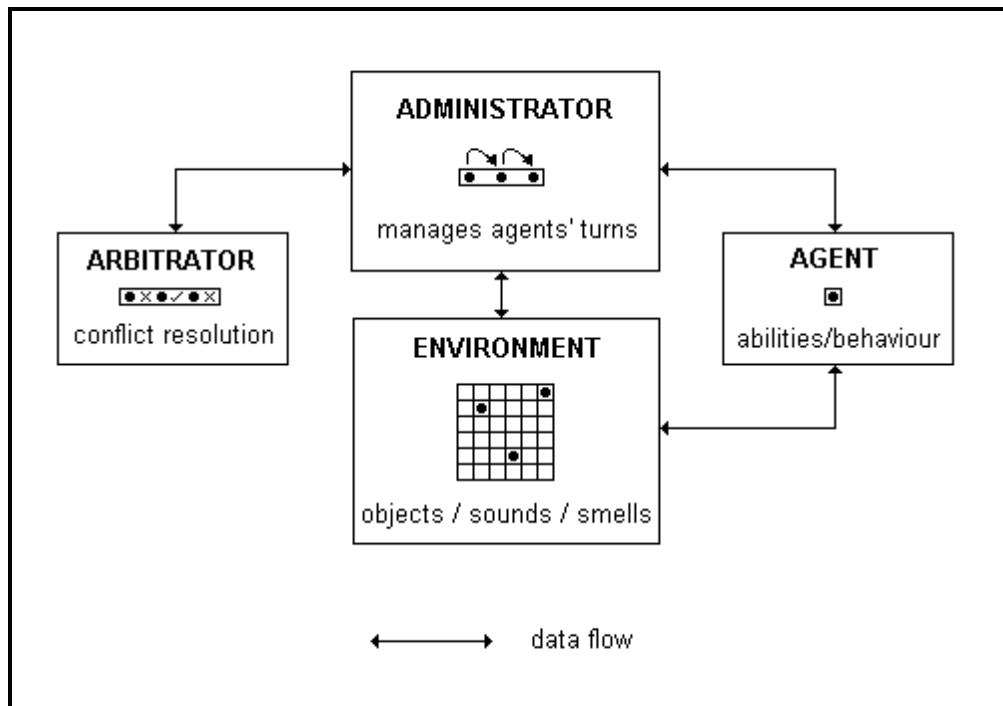


Fig 2.3 Data-Flow Model of the System

We can regard these components as the core of the system and use a code development technique known as Iterative Phased Development, based on the concepts of Pfleeger (1998). This method involves building a very basic core system with minimal functionality and then slowly building up the complexity and testing the resultant new systems within an iterative process. The advantages are that we will have had the opportunity to prototype certain methods along the way and thus get a realistic feel for how our solution is developing. We can start off with simple agents that do nothing, arbitrators that don't arbitrate and gradually add more functionality, observing how the system is growing and make any unexpected adjustments before it is too late.

Testing during the life-cycle should occur in several forms and should be incorporated into the development process. Three major forms of testing are proposed:

1. Black-Box Testing by the developer

2. Code Inspections by a colleague

3. Regression Testing when a new iteration milestone is achieved

Let's take a look at what these methods should involve:

1. Black-Box Testing by the developer

The developer should write test drivers designed to test the boundaries of the system code and reconcile expected outputs with observed outputs. This should occur shortly after the completion of each module and during the development of complex functions.

2. Code Inspections by a colleague

Peer review is a powerful means of trapping errors. I have selected a colleague to criticise my work and suggest corrections where he feels errors or inadequacies may be occurring.

3. Regression Testing when a new iteration milestone is achieved

Phased development tends to generate errors where an addition to the system causes a deviation in the original behaviour of a method developed earlier on in the iteration cycle. Each time that a fully working system is achieved, tests should be carried out that ensure we haven't corrupted the behaviour of methods that we had finished in any earlier iteration.

Some White-Box testing should be used for functions that are very complicated, but this will be very much up to the developer and is not expected to be a major part of the testing and development process.

2.3.3 The Core System

We have already identified four core components that constitute the system, illustrated in Fig. 2.3:

1. Environment

2. Agent

3. Arbitrator

4. Administrator

We should aim to develop minimal cases of each of these components, get the system running and then enter into our iterative phased development process. There is one last feature that we must consider before we examine the design of each component. That is, how do we feed the Arbitrator with details of the actions that each agent wishes to perform?

The requirements state that the Arbitrator must be generic and have little dependence on the data being arbitrated. So let us imagine a new object called an “Intent”. This should be a class that contains data structures representing the action and location at which an agent wishes to perform a task. We can use numbers to represent actions and allow these symbolic atoms to transcend the entire system. We can also enumerate objects in the same way. For instance:

```
enum { SPACE, OBSTACLE, AGENT, CARCASS, MINERAL, GRAB, DROP, MOVE,  
        UNKNOWN, NOTHING, SHOUT, SNIFF, DIE };
```

Such enumerated types attach a unique number to each name. Whenever we wish to refer to an object or action, we can simply refer to its name and not concern ourselves with the underlying numerical representation. This allows us to focus more clearly on the problems being solved. For this purpose, we propose the development of an “Intent” class that is discussed in section 2.3.5.

Many of the problem issues require a method of maintaining a collection of objects. An Environment is a collection of coordinates and objects, an Administrator is a collection of agents and an Arbitrator is a collection of agents’ intended actions. For this reason, it is expected that much use will be made of the Standard Template Library (STL) Vector type, due to its direct addressing features and ability to hold collections of many types of objects. Also, the STL has many algorithmic features for sorting collections and finding certain values within collections. If we make use of these functions, we save ourselves the task of having to develop our own algorithms to do the work. We should not be seeking to “re-invent the wheel”.

We have now completed an initial picture of the Core System components and these are illustrated in the following graphic:

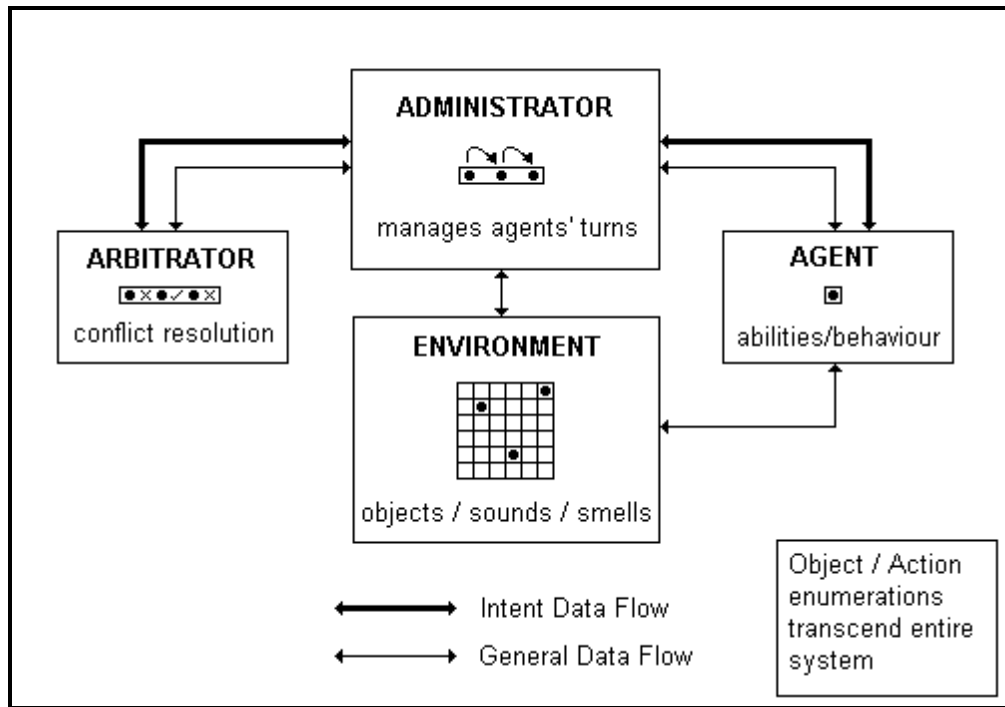
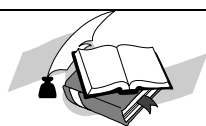


Fig 2.4 The Core System Revised

2.3.4 Environment

The requirements stated that a grid of square dimension, accessed via cartesian coordinate pairs was necessary. Effectively, we need to create a list of all possible locations and provide services that allow locations to be queried and set. We shall suggest using an STL Vector container and make use of the simple coordinate mapping illustrated in Appendix T.2. A vector controls a contiguous area of memory, thus enabling very rapid access to elements. The element mapping function in the grid implementation will produce an effective address offset that can be used to directly address items in the vector.



The requirements also stated that only one object should be located at any coordinate but some objects may also be a quantity of themselves. Additionally, we need to consider sounds and smells within the environment. Rather than complicate the form of data structure that will be present at each coordinate, we shall use four grids to represent four different levels of detail. This form of re-use for each definition will allow us to make changes much more easily.

The grids should be arranged as follows:

- | | |
|-----------------------|--|
| 1. Object Grid | - defines a type of object |
| 2. Value Grid | - defines the quantity of an object |
| 3. Sound Grid | - source of a sound emitted |
| 4. Smell Grid | - source of a smell emitted |

In order to simplify sounds and smells, we shall record merely the location of a source, rather than a gradient of tokens that swell out from the source. Although we define sounds and smells as a form of gradient field in our requirements, it would be too complex a task to manage a dynamic quantity of sounds and smells occurring in the environment. Each one would need a separate grid and special methods would have to be implemented to allow agents to detect all of the currently active sources. We have decided that agents will just search the sound and smell grids for a particular type and use a path-planning algorithm to guide them to the point of emission. Therefore, we may have more than one sound or smell active in the environment at any time, defined by a marker at the appropriate source location. It's important to keep this system as simple as possible, so we deviate slightly from our requirements, in the light of having now considered its implications.

The environment will require functions that check the validity of object quantities against types of objects, getting and putting functions to set and retrieve values at coordinates and we shall need to implement a constraint that only allows grid dimensions within the range 10 to 40. It would also be beneficial if we provide a coordinate validation service that allows a cartesian pair to be verified as being within the range of possible locations before an access is made to the grid itself.

The Grid class, of which there should be 4 instances, will possess the following data structures:

```
//      *** TYPE DEFINITIONS & GLOBALS ***

typedef vector<short> SHORTVECTOR;           // a Vector of "shorts"
typedef SHORTVECTOR :: iterator SHORTVECTOR_IT; // an iterator
const short MAX_DIMENSION = 40              // maximum dimension
const short MIN_DIMENSION = 10;            // minimum dimension

//      *** PRIVATE DATA ***

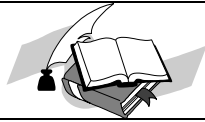
SHORTVECTOR g_list;                        // the Vector list of locations
short g_SIZE;                             // the dimension of the square grid
bool g_state;                             // an error state variable
```

We may use "g_state" as an error device that becomes active should an invalid coordinate pair be submitted to the class. This will prevent us from using erroneous values from a vector that does not implement Vector Boundary Checking and also produce feedback for algorithms outside the class that may be generating invalid coordinates. The class

Environment will wrap all four grids together and execute the appropriate grid function for the applicable context. The functionality of the Environment class will not be illustrated here, as it is mostly a collection of grids, or a “wrapper” class. All the developer need do is attach the right grid to the right context and yield the illusion of there only being one grid. Object types and their quantities may only be fully established once the system is in development. Code for the Grid class may be found in Appendix C.1. Code for the Environment class may be found in Appendix C.2.



C.1 - The Grid Class - Page 192



C.2 - The Environment Class - Page 207

2.3.5 Intentions

The Arbitrator needs to sort and compare a list of agents’ intended actions. Therefore, in order to allow us to compare a collection of cartesian pairs representing the location that an agent wishes to perform an action, we require a functional mapping from a pair to a unique number for every possible pair combination. Appendix T.2 details the means by which this mapping may be achieved under the context of two-dimensional grids, but we shall assume that we have this mapping method and focus on the internal characteristics of the Intent class.



The Intent class should possess the following data structures:

```
//      *** GLOBALS ***

const short MAX_PRIORITY = 3000;    // maximum [i_priority] an intent may have
const short MAX_KEY = (MAX_DIMENSION-1) + ((MAX_DIMENSION-1)
                                     * MAX_DIMENSION); // the maximum [i_key] value

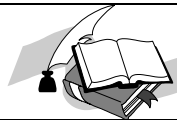
//      MAX_DIMENSION refers to the global data within a Grid class.


//      *** PRIVATE DATA ***

short i_name;           // agent's name
short i_action;         // agent's intended action
short i_priority;       // priority of that action
short i_x;              // x-coordinate where action is to take place
short i_y;              // y-coordinate where action is to take place
short i_key;            // unique number based on i_x, i_y and i_const
short i_const;          // max value that i_x and i_y may have
```

The data elements that the Arbitrator will be “interested” in are the name, action, priority and key. All of these items allow the list of agents’ intentions to be sorted in various ways to assist in the arbitration process. We may also provide some special sorting predicates in the form of classes as illustrated by Stroustrup (1997), so that we are not restricted to the definition of the comparison operator “<” in the Intent class. Additionally, the class will require getting and putting functions.

The implementation of this class occurs in Appendix C.3, together with definitions of sorting predicate classes.



C.3 - The Intent Class - Page 220

We now have a means of an agent expressing its intention to perform an action. Additionally, we have a means of capturing an agent’s state shortly after the agent has executed its behavioural definition and we can write back these states to the agents after arbitration.

2.3.6 Agents

The requirements stated that we needed a base class of agent from which all agents may be derived. The requirements also specified a fundamental collection of abilities that every base class agent should possess. Let’s remind ourselves of the major categories and their functions:

- **Memory and Thought**

- a memory to hold objects perceived in the environment that persist until altered
- to be able to plan the shortest path to objects via memory
- to be able to wander aimlessly
- to be able to query memory for the existence of objects, either adjacent to the agent or somewhere else in the environment / memory
- to be able to pick up objects that are not other agents or obstacles within the allowance of the grasping and carrying capacity
- to be able to drop objects on a location that has enough capacity for the object
- to be able to forget all objects in memory, or all objects of a certain kind
- to be able to carry out a state that represents agent death
- to be able to allow a targeted location and object to persist in memory until a new target is set
- to be able to explore areas of the environment that have not yet been reached and memorised

- **Vision**

- a 360 degree perceptive range that acquires the environment into memory. The range should be definable by the user
- agents will be able to look over all objects and other agents in the environment

- **Communication**

- to be able to perceive and make sounds

- **Gradient Detection**

- to be able to perceive smells and follow them to their source. Only one smell should ever be active in the environment
- **Carrying capacity**
 - to define a load that an agent can carry. The agent's hands will still be free when the agent's carrying capacity is full, the analogy is of a backpack
- **Grasping Capacity**
 - to define a maximum load that an agent can pick up at once in any turn and place in its backpack
- **Movement**
 - to be able to move to one adjacent square in any turn in any direction
- **Name**
 - a number that identifies the agent and is unique

We shall tackle each category in the order that they occur and introduce data structures and algorithmic concepts, gradually heading towards a total definition for this class.

- **Memory and Thought**

For recording the things that agents perceive, we can re-use the Grid class that was specified in section 2.3.4. In fact, we shall almost be making an exact copy of the Environment class, including its dimensions, as it conveniently gives us a definition for locations, objects, object quantities, sounds and smells. Memory may then be queried by matching an enumerated type of the kind defined in section 2.3.3, with the types resident in

the agent's memory. An agent can be made to forget things by simply erasing all instances of a chosen type from the agent's memory. We should also use a special enumerated type that represents an unknown location in an agent's memory.

In order to allow the agents to plan paths from their current coordinates to places that interest them in the environment, we need to take advantage of a heuristically guided searching technique known as "A* Search" as described in Russell & Norvig (1995). The most effective way of implementing this within the agent is to give the agent a special search memory that is a variant of the Grid class and matches the same dimensions as the environment. Appendix T.4 illustrates in detail the method that was finally developed, including the reasons why, but as this is a considerably technical issue, we shall assume that we have defined searching and continue with the rest of the implementation.



T.4 - The Path Searching Algorithm - Page 164

In order to allow agents to remember a location and object in memory that is currently being pursued, we should use variables that hold details of the object type and location. We should also use a boolean variable that tells us at any time whether the agent is targeting a location, so we don't cause the agent to forget target locations that were deduced in a previous turn.

In order to give agents a wandering behaviour, we need to develop an algorithm that counts all locations in memory that can be reached. Then we should select one of those locations at random and cause the agent to target that location as a place to head towards. Unknown locations would then gradually become uncovered.

To allow agents to pick up, carry and drop objects we should define variables that state the grasping capacity and carrying capacity of the agent. Whether a location has enough capacity for an object being dropped will simply be a matter of querying the agent's memory about the location. Picking up and dropping objects may be achieved by defining action enumerated types under the definition in section 2.3.3. As there will only be one type of object that has a quantity, we need not concern ourselves with different weights and measures that would typically refer to a situation when there are many different types of objects with quantities.

Agent death may be represented by changing the agents state from that of being alive, to that of being dead. We can make the agent switch its type to that of a carcass and define, through a function, the necessary tasks that should be carried out in order to prevent the agent from moving or interacting with the environment. However, we should not delete the agent from the environment, as we may wish to define a special activity that occurs when agents die. For instance, if an agent dies, it may rot and the agent could be made responsible for rotting, eventually switching its type to that of a free location once it has rotted away completely.

- **Vision**

We need an algorithm that creates a rectangle over the agent's current location representing its 360 degree visual range. This is a simplification, as it is clearly not exactly 360 degrees limited by a circular range. However, the simplification is acceptable.

- **Communication**

We need to make it possible for agents to access the Environment's sound grid and place a sound at the agent's current location. Additionally, agents must be able to detect the location of sounds in the Environment.

- **Gradient Detection**

We have decided to ignore gradients for smells and simply allow the agent to detect any smells present in the environment and plot its own course to the source of the smell.

- **Carrying Capacity**

We may just simply apply a variable here that denotes the maximum load capacity.

- **Grabbing Capacity**

This will be similar to Carrying Capacity, but have a smaller maximum value.

- **Movement**

This will be achieved by the agent removing its type and location from the Environment and its memory and then recording its new location and type in the Environment and its memory. We should include a check that ensures the agent has not been allowed to move to an already inhabited location, in case a failure occurs in arbitration or elsewhere.

- **Name**

We propose a unique number, held in a variable that no other agent of the same or any other type may share.

It is now possible to state the fundamental data structures that an agent will require:

```
//      *** PROTECTED DATA ***

grid ag_m;           // object memory
grid ag_s;           // search memory
grid ag_snd;         // sound memory
grid ag_sml          // smell memory
short ag_x;          // current x-coordinate
short ag_y;          // current y-coordinate
short ag_target_x;   // a target x-coordinate
short ag_target_y;   // a target y-coordinate
environment *ag_env; // pointer to agent's environment
short ag_name;       // name
short ag_v_range;    // vision range
intent ag_intent;    // intended action
```

<i>short ag_type;</i>	<i>// type of agent</i>
<i>short ag_orig_type;</i>	<i>// copy of type</i>
<i>short ag_ruck;</i>	<i>// carrying capacity</i>
<i>short ag_scoop;</i>	<i>// grasping capacity</i>
<i>bool ag_alive;</i>	<i>// whether agent is alive</i>
<i>bool ag_target;</i>	<i>// whether agent is targeting something</i>
<i>short ag_target_obj;</i>	<i>// object targeted</i>

The reasoning behind maintaining a copy of the agent's type is in case the agent changes its type during runtime or the scenario needs to be reset. In order to reset the agent, we need to know what its original type was at the start. We can easily access the functionality of the Environment by giving the agent a pointer to it. There are a lot of grids, but such simplicity will allow us to make changes to the system without having to consider too many side-effects. The code for the agent class may be found in Appendix C.4.



C.4 - The Agent Class - Page 231

2.3.7 Arbitrator

The requirements stated that the Arbitrator should be capable of arbitrating agents' intended actions without any dependance on the symbolic meaning of the actions. We have achieved this through the development of the Intent class, so now we need to concentrate on the matter of arbitration itself. Appendix T.3 contains a sample scenario that illustrates

how arbitration may be achieved and a suitable algorithm should be drawn from it. However, we must consider the other functions and data structures that will reside within the Arbitrator class.



T.3 - The Arbitration Process - Page 161

The arbitrator requires a copy of the last batch of agents' intents that were arbitrated in a previous turn so that it may deduce whether an agent wishes to perform the same action again in the following turn and set the agents' priorities accordingly.

```
//      *** TYPE DEFINITIONS ***
```

```
typedef vector<intent> INTENT_LIST;           // list of [intent]'s
```

```
typedef INTENT_LIST :: iterator INTENT_LIST_IT; // iterator
```

```
//      *** PRIVATE DATA ***
```

```
INTENT_LIST a_intents;           // copy of agents' last intents
```

```
short a_fail_value;             // value to load into intents that lose in arbitration
```

Because the Arbitrator is generic and should know nothing about the rest of the system, except for intents, we must supply it with the specification for what happens when an agent loses in arbitration. This is achieved with the private data item “a_fail_value.” Using the enumerated types first discussed under section 2.3.3, the type for this private data item would be “NOTHING”. This is because agents have to stop and wait if they are prevented

from executing an action through arbitration. As long as we specify the result of doing nothing in the agent's action execution function, we will always have an effective means of preventing the agent from carrying out its action through the arbitration of its intent and writing back of its intent through the Administrator. The code for the Arbitrator class may be found in Appendix C.10.



C.10 - The Arbitrator Class - Page 314

2.3.8 Administrator

We have finally arrived at the mechanism that should be responsible for orchestrating the entire process. The requirements specified that the Administrator should operate a turn-based process for every agent, set-up the initial environment and initial state of the agents. It should also allow the extraction of data from the environment and the agents.

We still want to aim towards producing a system that allows the user to make adjustments in a relatively easy manner, so we shall not provide one function that initiates the entire simulation process. If we keep each major stage as a separate function, it is easier for the insertion of special rules and data gathering techniques. Ignoring any initialisations for now, we can view the simulation loop as follows:

- 1. Update the environment with any smells**
- 2. Execute every agent's sound behavioural definition**

- 3. Execute every agent's main behavioural definition**
- 4. Collect all of the agents' intents**
- 5. Arbitrate the agents' intents**
- 6. Copy the arbitrated intents back to the corresponding agents**

So why do we execute a sound behavioural definition as well as the main behavioural definition? Smells will be controlled by the Administrator, so they can be loaded before each turn, but the agents themselves generate sounds. In order to allow agents to make sounds, receive smells and consider another action within a turn, we have to treat all of these events as different processes within the turn. This allows sound and smell to effectively be carried out at the same time as doing something else. This is because we feel that making a sound and detecting a smell is a simple matter and should not cause the agent to stop everything else that it is doing in order to take note. Remember that an agent may only execute one action in any turn due to the mechanics of arbitration. By having rounds within turns, we get around the problem of only one action per turn and we generate the concurrent illusion of agents making sounds and sniffing out smells whilst they move or pick something up. If the user finds this to be unsatisfactory, the execution of the sound behavioural definition may be easily removed. Additionally, if an agent's main behavioural definition has rules that execute on the basis of sounds or smells being detected, we are able to load the agent's memory before the rule is fired so the agents get an opportunity to act. If agents just emitted sounds and received smells during the invocation of their main behavioural definition, we would encounter timing problems that would result in only the agents whose order in the list of agents is greater than the creator of the sound, detecting the sound.

We also demonstrate a method to the user for incorporating new environmental or behavioural rules within a turn. By establishing a method, we save the user the trouble of having to ponder a way around the problem themselves.

We shall not provide an extensive set of data gathering methods, as these will depend entirely on the type of experiment that the user wishes to perform. We have already provided some data gathering functions in the classes discussed earlier and these could easily be invoked during a turn within the Administrator.

Let's take a look at the Administrator class data structures:

```
//      *** TYPE DEFINITIONS ***
```

```
typedef vector<agent*> AGENTS;           // a list of homogeneous / heterogeneous
                                         // agents
```

```
typedef AGENTS :: iterator AGENTS_IT;   // an iterator
```

```
//      *** PRIVATE DATA ***
```

```
AGENTS ad_agents;                       // a collection of pointers to agents
```

```
environment ad_env;                     // the environment
```

```
INTENT_LIST ad_intents;                 // agents' intentions
```

```
arbitrator ad_arb;                      // the arbitrator
```

```
bool ad_end_condition;                  // to be able to indicate that something has happened
```

Notice that the collection of agents is maintained using a vector container of pointers to the base class. We feel this is an efficient way to achieve a heterogeneous collection of objects derived from a base class within an STL Vector container based on our findings from Stroustrup (1997). Although most of the time the collection of agents will be the same, we want to make it possible for different classes of agents to be handled by the Administrator without any changes to the system process. Therefore, if all agents are derived from the base class and define their own unique behaviours through virtual function overrides, the context of which agent is being referred to will be dynamically bound for us. The concept of dynamic binding is discussed in Deitel & Deitel (1998). As long as we keep to the rules of substitutability for inheritance in C++, we can create any agent and still use the same administration method. This is a very powerful feature of an object-oriented language and will serve us well in this particular software problem.

The Administrator is also the keeper of the environment, although it is the agents themselves that update the environment as a result of their actions. The Administrator also needs a list of intents so that it may collect each intent from every agent, hand them to the Arbitrator and then deliver them back to the agents after arbitration.

The item “ad_end_condition” is a means of recording that something special has occurred within the simulation that the user can set, once a user-defined predicate has been satisfied. The boolean variable is given a truth value somewhere within the Administrator’s functionality, if the predicate condition is true. Then, the user need only check the truth of this variable during execution to see if their predicate has been fired. The code for the Administrator class may be found in Appendix C.11.



2.3.9 Prototyping Discoveries

Much of the inspiration for this design has been obtained by prototyping the various aspects and activities of the system. This overall method has shown great promise and should provide a general means of simulating a multi-agent environment with little additional coding overhead in terms of adjusting the system's behaviour. Prototyping code has not been supplied, as its implementation does not precisely match the requirements of this system. It would have caused confusion to the reader if such details had been included.

2.3.10 Graphics

The reader of this document will almost certainly be pondering how the user might obtain a visual reference as to whether the agents are behaving in the anticipated manner. Although a graphics system did not form an aspect of the user's requirements, prototyping has produced an experimental Java program that has the following functionality:

- Agents are animated using a data script from a run of the multi-agent simulator using functions to output the coordinates and state of an agent at every turn
- The Animator has the ability to play a sequence as a movie of frames and adjust the playback delay

- The Animator may be paused at a specific frame to inspect an interesting event
- The Animator may be played frame-by-frame either forwards or in reverse
- The Animator may be used to select a specific frame for inspection

A screen-shot of this Animator follows. Little else will be said about this animation program, but its existence will quench any doubts that the reader may have about the user's ability to visually appreciate the operation of a multi-agent simulator.

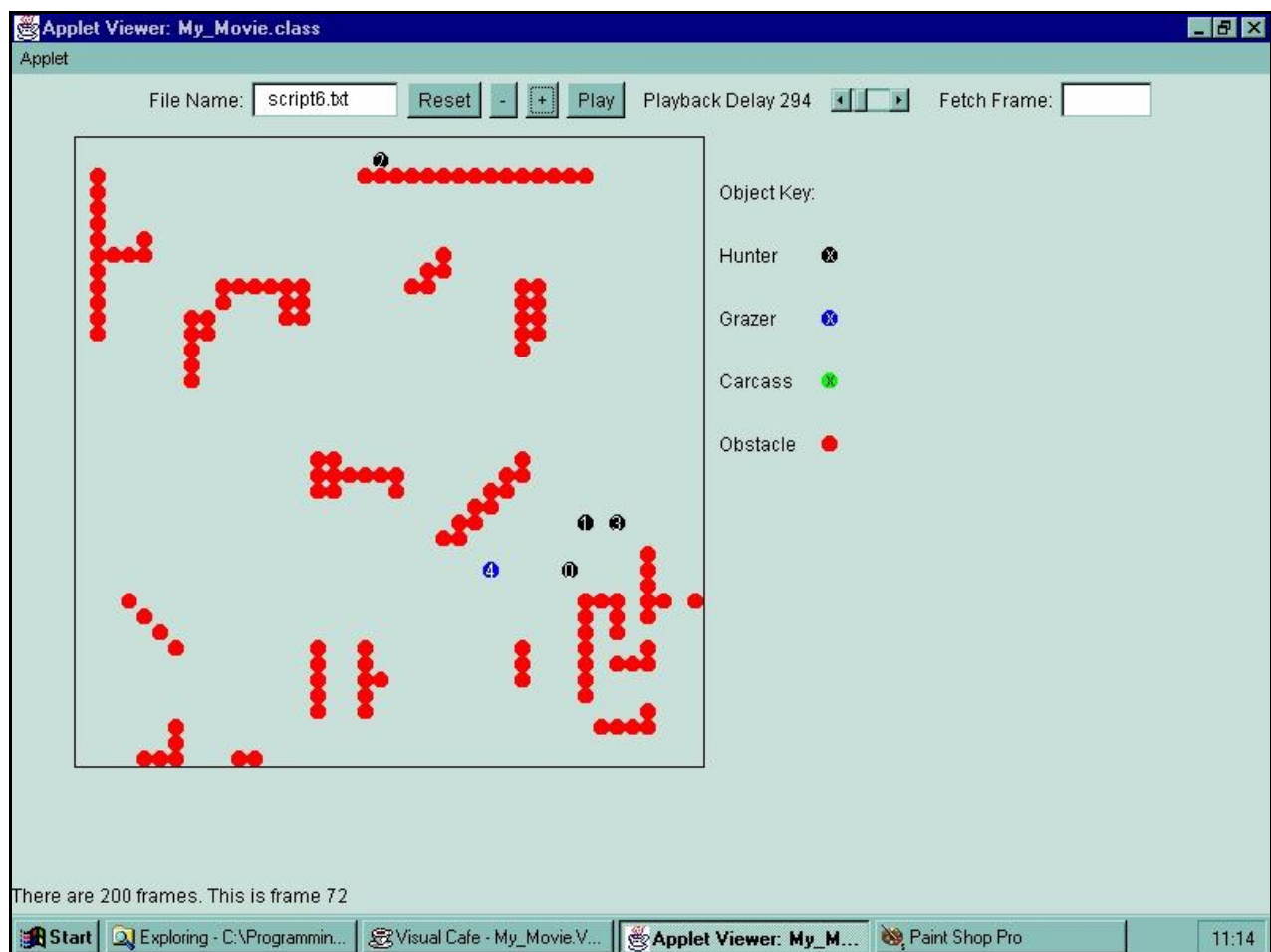


Fig 2.5 A Prototype Java Applet Animation Program

2.4 Life-Cycle Modifications and Issues

2.4.1 Introduction

In the remainder of this chapter, we discuss the alterations that were made to the system design during the software life-cycle. If you have already read some of the appendices, you will be aware of some of these changes. Many of these changes had quite a profound impact of the forms of experimentation that could be carried out. Indeed, much of what is discussed in the following sections plays a significant role in most of our experiments detailed in Chapter 3.

2.4.2 Creating a Graphical Interface

We commenced a course on Windows programming during the Spring Term and were surprised at how rapidly a graphical user interface (GUI) could be developed. We had already created a fully working simulator system and decided to overlay an interface to aid experimentation. A user interface and graphics were not aspects of the requirements, so we had to take a considered gamble over whether we could complete such a task and make it perform adequately for our needs.

A Windows 95/98™ GUI was created with the name “Animate.” It has some important features that provided all of the graphics that occur in our experiments. We have selected out the most salient aspects of the interface and discuss each one in this and the following sections. Special data structures were required in order to process scenario information. The details of these may be found in Appendix C.12.



C.12 - The Special Data Structures - Page 352

The following screen-shot is of the left-hand side of Animate’s main interface:

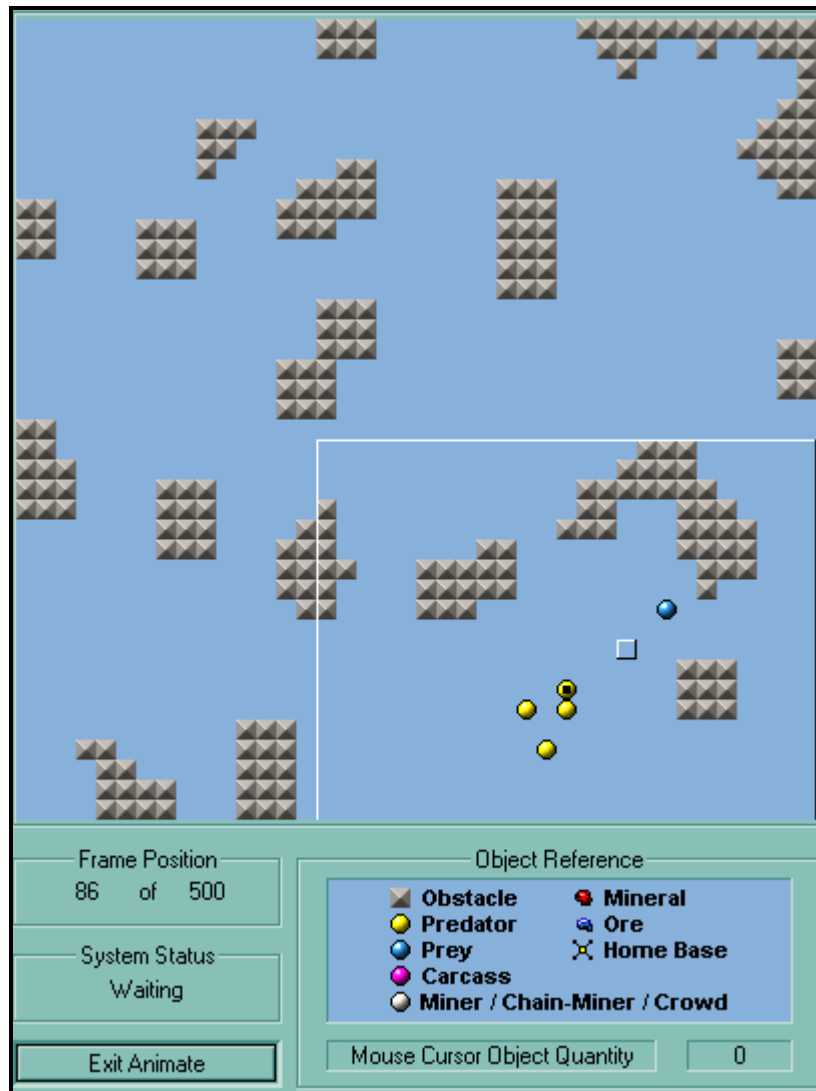


Fig 2.6 The Animate Interface - Left Section

The top half of this image contains the Animation Window where the environment is animated in real-time or during a playback session. Immediately below it lies a key to enable objects in the Animation Window to be swiftly identified. The “Mouse Cursor Object Quantity” allows the user to position the mouse cursor over an agent or object in the Animation Window during a simulation or playback session and obtain the quantity associated with that agent or object.

The next screen-shot is of the right-hand side of Animate’s main interface:

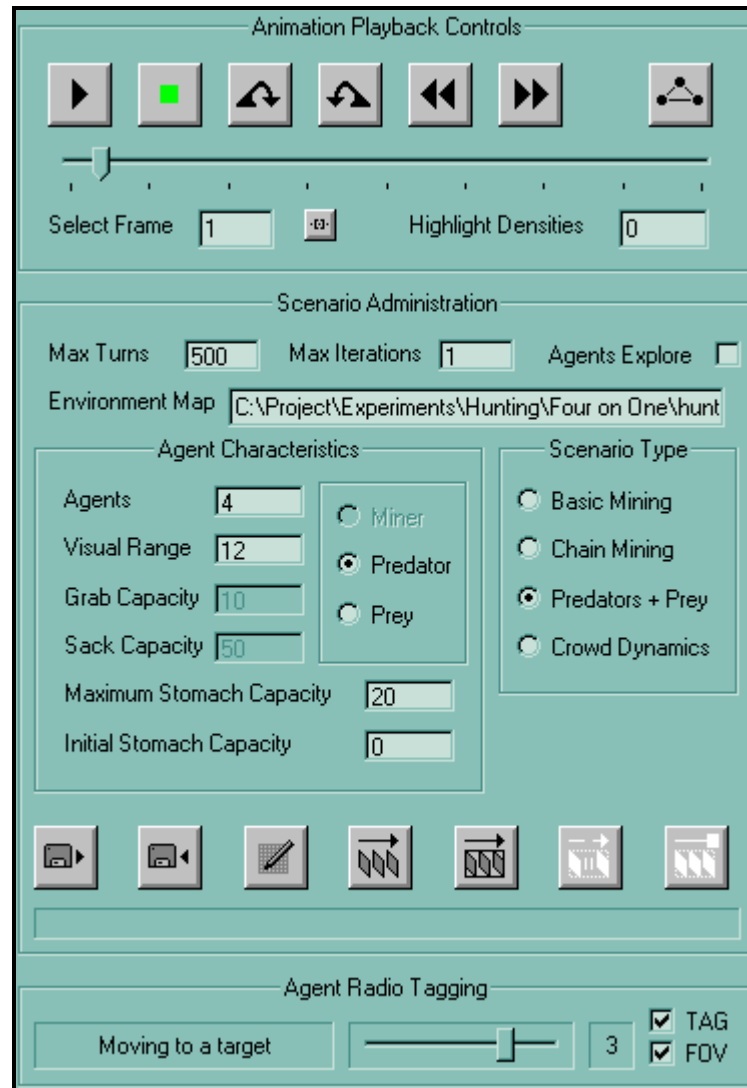



Fig 2.7 The Animate Interface - Right Section

The most notable features are the Playback Controls and Agent Radio Tagging. The controls under these sections feature heavily in our experiment results.

Playback Controls

We are able to select any frame within a scenario and jump to it by selecting the button .

We may then select the  button which is a toggle that runs either of two functions:

1. Trails Algorithm

2. Path Densities Algorithm

The Trails Algorithm plots a line in the Animation Window for every agent, that represents the path in the environment that each agent has taken from the first frame in the scenario to the frame specified by the “Select Frame” edit box. The Path Densities Algorithm counts all instances in an environment where any agent was located from the start frame to the frame specified and draws a grey-scale density image in the Animation Window. Additionally, densities equal to or over the amount specified in the “Highlight Densities” edit box are drawn in red to select out densities over a certain threshold.

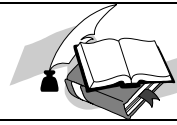
Both of these algorithms have proven very useful in illustrating the results of our experiments.

Agent Radio Tagging

By selecting the check box “TAG” and by dragging the adjacent slider bar until a specific agent name is selected, we are able to track a particular agent during animation. Data regarding the agent’s current action is displayed and a small black box is drawn over the agent in the Animation Window. Any location targeted by the agent is enclosed within a small three-dimensional rectangle. If “FOV” is selected, the simulator draws a three-dimensional rectangle centred over the tagged agent in the Animation Window to represent its field of view. This allows the user to monitor the behaviour of a particular agent and

observe the actions that resulted in a phenomenon occurring in light of its perceptive range.

More detailed discussion on the Animate interface may be found in Appendix T.5.



T.5 - The Animate Main Interface - Page 173

2.4.3 Adding a Map Editor

Creating environment maps manually was a tedious process so we added another feature to the Animate interface. We created a map editing facility that is pictured in the following screen-shot:

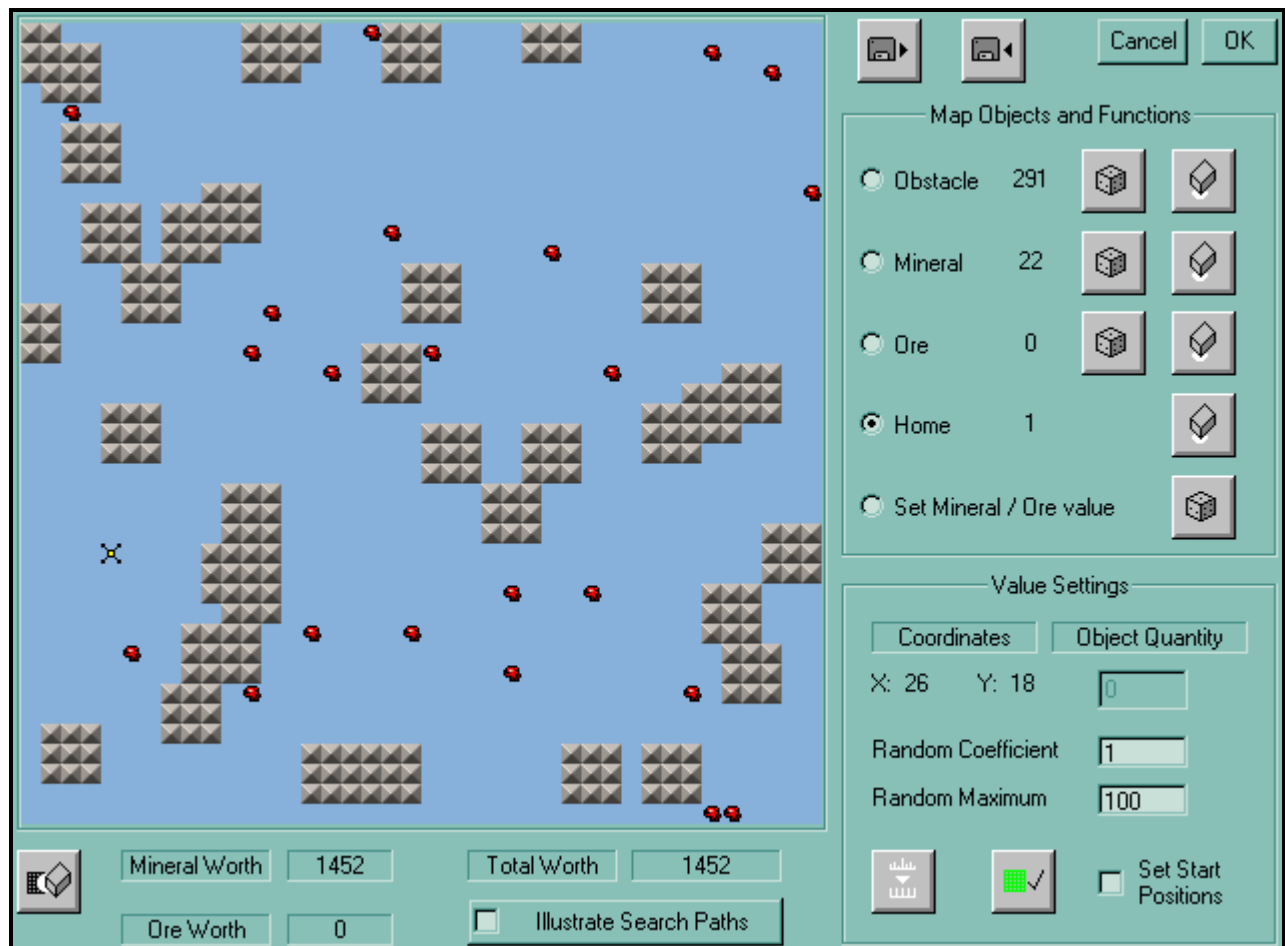


Fig 2.8 The Animate Map Editor

This editor allowed us to create environments rapidly and provided data on the quantities of objects that was useful for keeping track of experiment parameters. It also has some rather special features connected with path searching that are illustrated in Appendix T.4. More details on the Map Editor may be found in Appendix T.6.



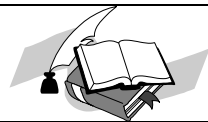
T.4 - The Path-Searching Algorithm - Page 164



T.6 - The Animate Map Editor - Page 179

2.4.4 Reducing Environment Complexity

Having created a GUI, better forms of experimentation became possible. Sounds and smells were eradicated from the original design and all environments were thus simplified. The visual power of real-time animated graphics had allowed us to perform more traditional and convincing experiments on emergent phenomena that more closely resembled the work of Mataric (1995). Therefore, an environment only contains the type of object and any quantity associated with it. This greatly simplified our original ambitions and removed a redundancy from the Environment class. The code for the Environment class may be found in Appendix C.2.



C.2 - The Environment Class - Page 207

2.4.5 Opportunities for Additional Experiments

We originally planned to only model a community of mining agents, but ensured that we could manage a heterogeneous collection of agents, should we be able to conduct further experiments. The creation of the user interface and its features allowed a wider spectrum of experimentation to be carried out. Therefore, we designed four categories of experiment that are discussed in detail in Chapter 3.

2.4.6 Improving the Path-Searching Algorithm

The original specification that we developed for searching in our original design document was flawed. We had struck the right chord but the note was not clear. Our definition of setting neighbour nodes was incorrect. Additionally, it was not necessary to use a heuristic such as Manhattan Distance. We spent considerable effort in getting path searching minimal and correct. We feel that our time was well invested and have arrived at a powerful solution for our specific needs. A detailed description of the Path-Searching Algorithm occurs in Appendix T.4.



T.4 - The Path Searching Algorithm - Page 164

2.4.7 Arbitration Classes

Our original design for arbitration was to match an agent's action and location, called an Intent, against other agents' Intents and handle any conflicts. This turned out to be too simple and we decided to develop two classes of action that all agents' actions would fit into. This resulted in the creation of a "C_INTERACT" and "C_INHABIT" class. Details about these changes can be found in Appendix T.3. Code for the Arbitrator class can be found in Appendix C.10.

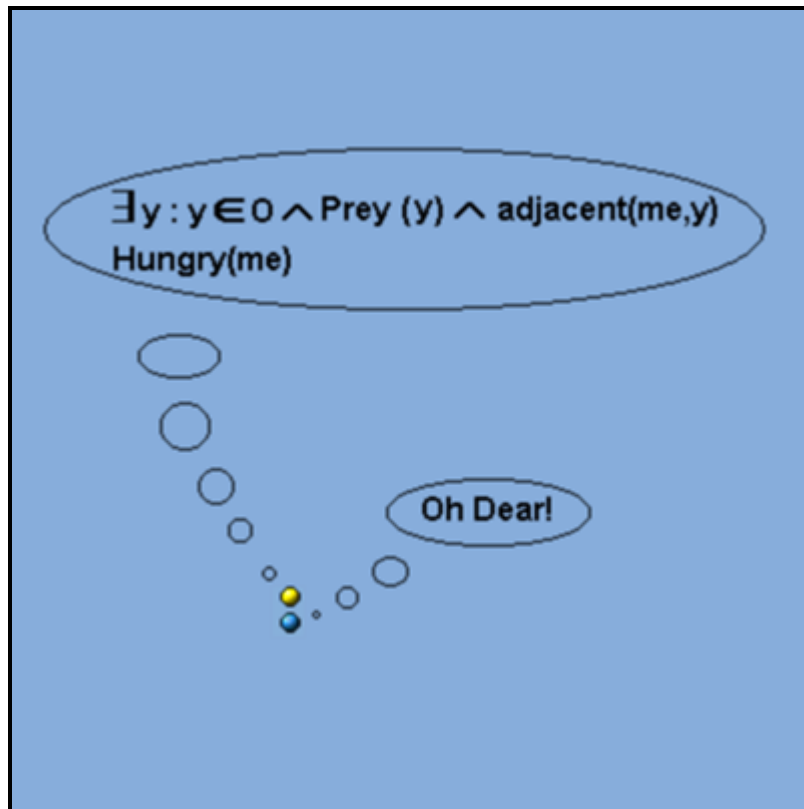


T.3 - The Arbitration Process - Page 161



CHAPTER 3

EXPERIMENTATION



3.1 Experiment 1 - The Mining Experiment

3.1.1 Introduction

In this experiment, we created a homogenous collection of mining agents derived from the base class agent. The purpose of these agents was to seek out mineral resources, mine the minerals and deposit them at a home base.

We utilised the behaviours of the base class agent and added a few more that would represent the acts of mining and depositing minerals. The following set of production rules were applied to each Miner agent:

```
IF backpack not empty AND adjacent to a home base
    THEN empty backpack at that home base
    FINISH
ENDIF
```

```
IF backpack not full AND adjacent to a mineral
    THEN mine that mineral
    FINISH
ENDIF
```

```
IF not targeting a mineral AND backpack not full AND knows of minerals
    IF can reach the closest mineral possible
        THEN seek that mineral
        FINISH
    ENDIF
ENDIF
```

```
IF not targeting a home base AND backpack full
    IF can reach the closest home base possible
```

```
        THEN seek that home base
        FINISH
    ENDIF
ENDIF
```

```
IF not targeting a home base AND backpack not empty AND do not know of minerals
    IF can reach the closest home base possible
        THEN seek that home base
        FINISH
    ENDIF
ENDIF
```

```
IF targeting anything
    IF can reach that target
        THEN seek that target
        FINISH
    ENDIF
ENDIF
```

```
IF unable to wander anywhere
    THEN do nothing
ELSE IF can seek the wander location
    FINISH
ELSE
    do nothing
ENDIF
```

```
FINISH
```

Code details for the Miner class may be found in Appendix C.5.



C.5 - The Miner Class - Page 258

There are four aspects of these mining agents that we are able to adjust:

1. Set their visual range between 2 and 20 cell units
2. Make them explore their environment or possess a complete knowledge of it at the beginning of each simulation
3. Set their grabbing capacity to a value between 1 and 1000 mineral units
4. Set their backpack capacity to a value between 1 and 1000 mineral units

3.1.2 Investigations

It was decided at first to experiment with varying permutations of the adjustable settings and visually inspect the results. If any form of emergent activity developed, the parameters would be noted and a set of iterative experiments planned to give some assurance that the activity was not purely by accident alone.

The most notable activity that emerged was when the agents possessed a large backpack capacity and a proportionately small grabbing capacity. The mineral values also had to be significantly high so that they would cause the agents to spend some time mining them. The agents tended to aggregate at mineral locations and form small groups. The groups would either stay together or split into new groups as the simulation progressed. It appeared visually, as if the agents were spontaneously creating “gangs” that would concentrate on particular mineral locations in a fairly efficient manner. Once the mineral had been exhausted, the “gang” would either move to the next mineral, split into “sub-gangs” to mine elsewhere, or head for the home base to empty their backpacks. We also noticed that making the agents explore their environment only tended to delay the inevitable formation of groups under these parameters.

3.1.3 Experiment Plan

We decided to formally run 50 scenarios and inspect each one for signs of the same grouping activity. Each experiment used the same environment and parameters. The parameters were set as follows:

- **900 turns**
- **Random system generated obstacles and mineral resources created 59 mineral resources ranging from 1 to 98 in value. This produced a total mineral worth of 3001, an average of 51 per mineral. This environment was used in every scenario**
- **1 home base situated by the experimenter remaining constant for every scenario**
- **10 agents placed randomly by the system for each scenario**
- **Visual range 6**
- **Grabbing capacity 1 mineral unit**
- **Backpack capacity 200 mineral units**
- **Exploring off**

The following screen-shot displays the environment used in this experiment:

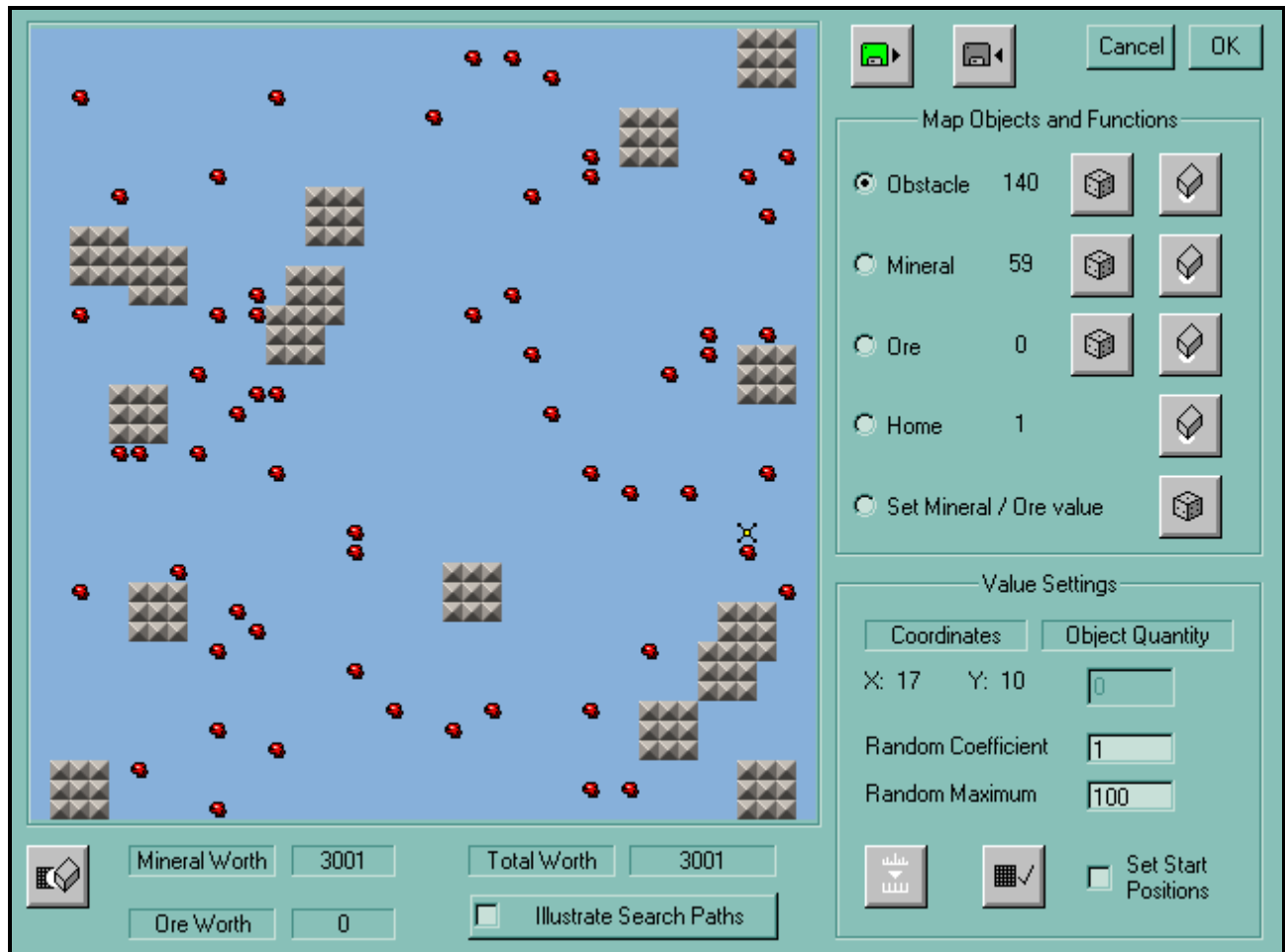


Fig 3.1 The Mining Experiment Environment

3.1.4 Experiment Results

Every scenario exhibited grouping behaviours. The groupings were by no means perfect but were clear enough to catch one's attention. At least a group of 4 agents manifested itself along with varying numbers of smaller or larger groups in each run. We have selected a particular scenario from the experiment set and shall illustrate its features. Later, we provide a few more examples from the experiment set and then discuss the results.

The initial state of this scenario, scenario 46, follows:

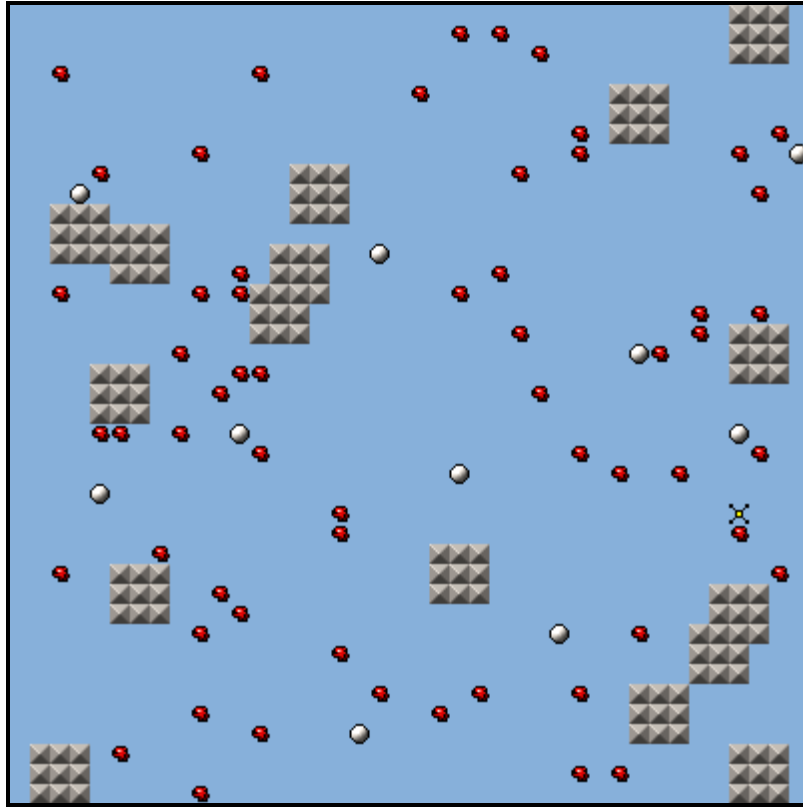


Fig 3.2 The Mining Experiment - Scenario 46, Frame 1

The first significant indications of group formation occurred at frame 273. This was mostly due to the fact that some agents had deposited the contents of their backpacks at the home base inside a relatively short period. Therefore, the home base had acted as a

congregation point from which new “excursions” could begin. As each agent is a clone, they will all pick the closest relative mineral to mine, hence the grouping begins:

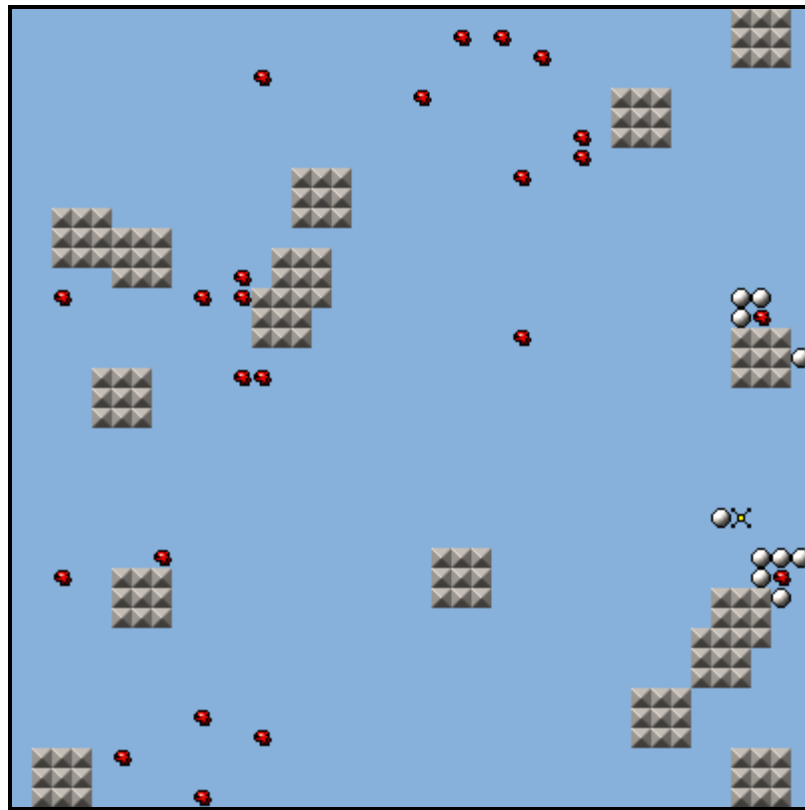


Fig 3.3 The Mining Experiment - Scenario 46, Frame 273

Two groups were clearly established by frame 369:

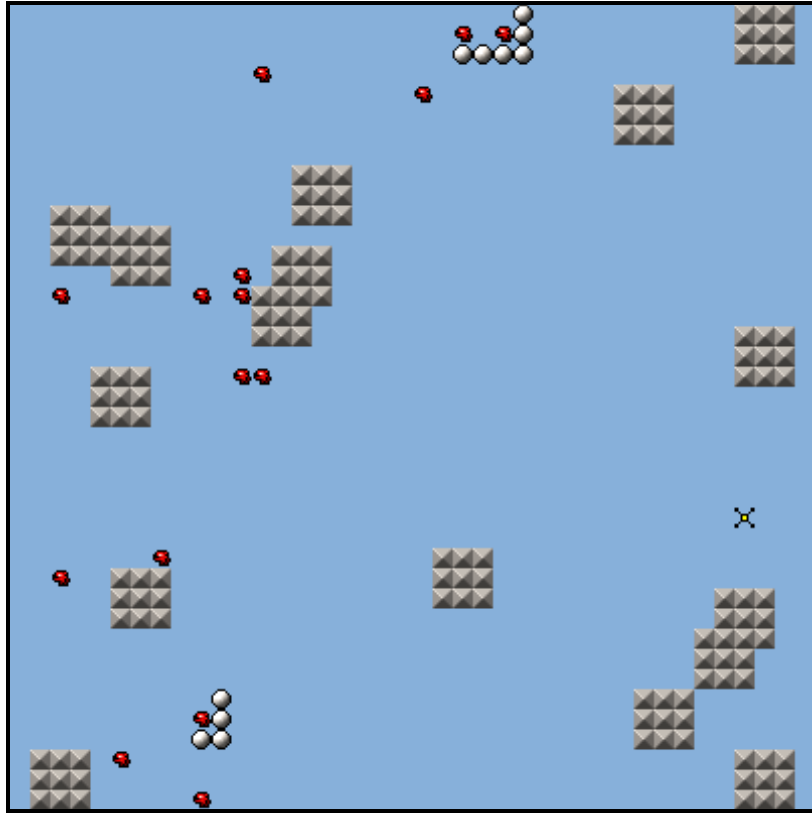


Fig 3.4 The Mining Experiment - Scenario 46, Frame 369

All minerals were collected by frame 693, so we ran the Path Density Algorithm and yielded the following concentrations:

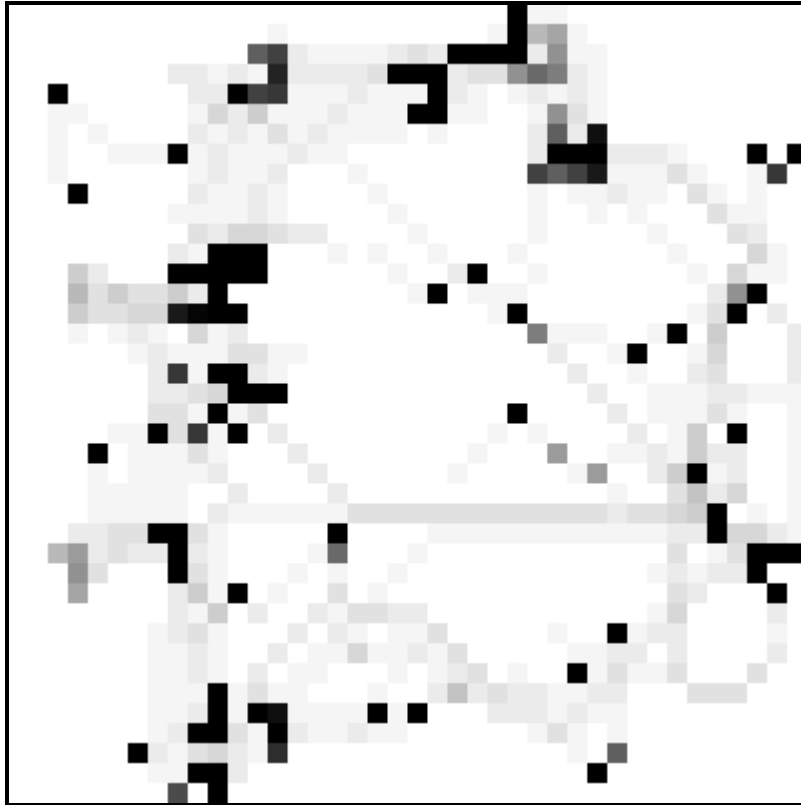


Fig 3.5 The Mining Experiment – Scenario 46, Frame 693, Path Densities

The varying degrees of shading correspond with the amount of times an agent has been positioned at that location. Darker shading indicates more time spent at that location. The dark angled shapes are mostly indicative of agent groups mining the same mineral resource. It is also interesting to see some of the agents' emergent path structures in the lighter shade. We deal with emergent path structures in Section 3.4.

The next screen-shot is a collection of four examples from the experiment set, illustrating different group formation states. The labels indicate the experiment number, scenario number and frame number respectively:

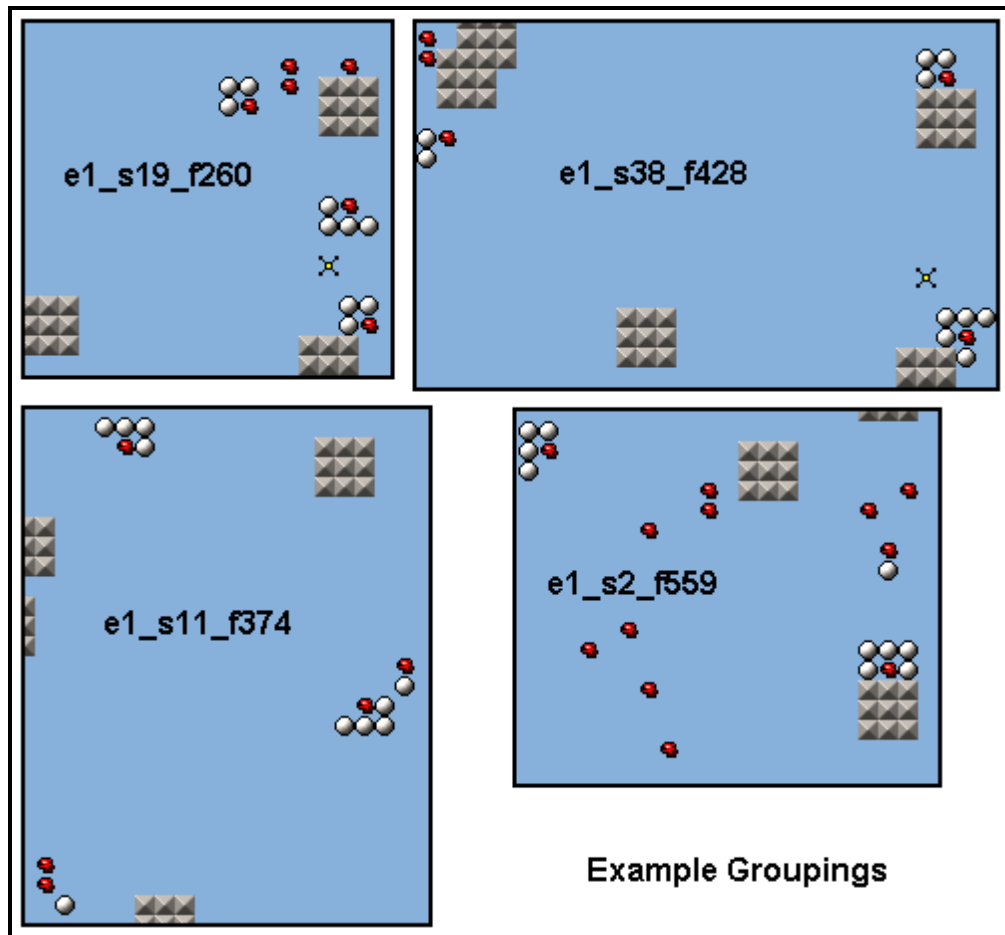


Fig 3.6 The Mining Experiment - Example Groupings

We were interested in establishing the lifetime of these groups, so we used the tagging method described in Section 2.4.2 to track particular agents throughout the simulations and observe their movements and actions.

It was apparent that group lifetimes were highly dependent on the proximity and distribution of resources. If many nearby resources existed, then the chances of the group splitting were high. However, if the closest resource was of some distance away from any other resources, the entire group would move to it, unless one of the members' backpacks was full.

We have selected agent number 7 from scenario 46 as our illustrative example. We begin at the moment agent 7 joins its first group at frame 269:

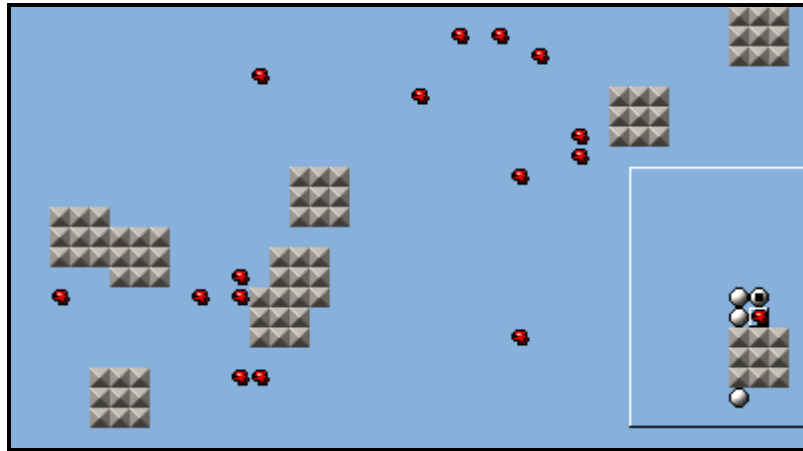


Fig 3.7 The Mining Experiment - Scenario 46, Agent 7, Frame 269

By the time frame 275 is reached, the mineral has been exhausted and the agents head off towards the top left. Agent 7 initially believes that a resource exists to the north and targets it. A small three-dimensional box indicates the target location:

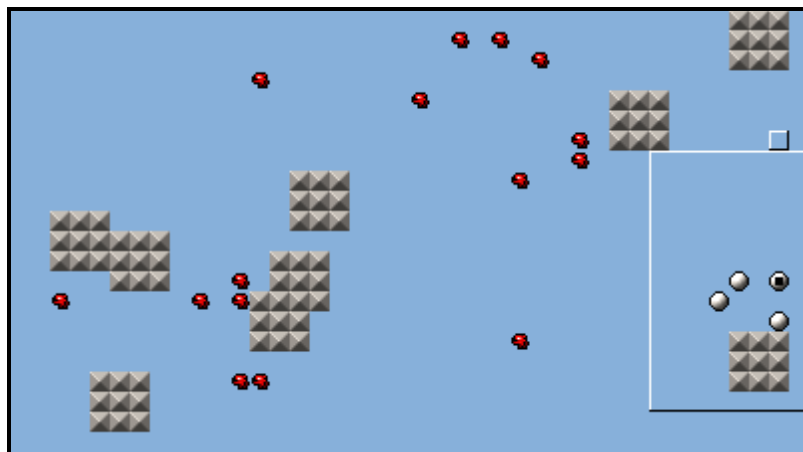


Fig 3.8 The Mining Experiment - Scenario 46, Agent 7, Frame 275

However, once Agent 7's targeted location comes under its visual field, it realises that the target no longer exists and targets the same mineral that the rest of the group are targeting at frame 277. Also notice that another agent is joining the group at this time:

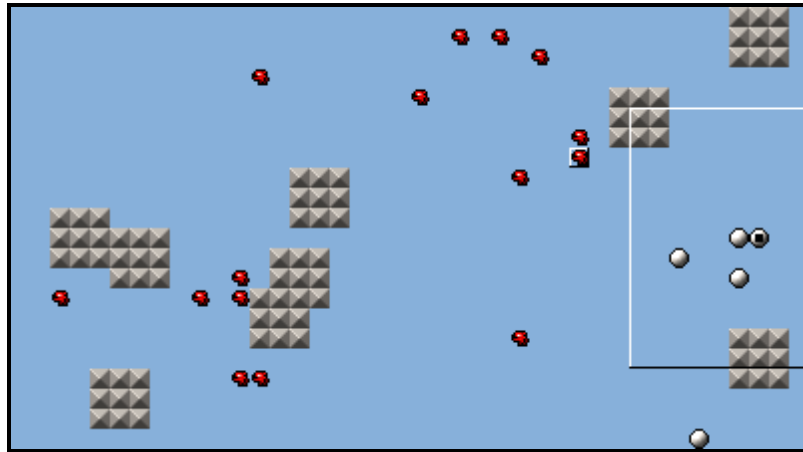


Fig 3.9 The Mining Experiment - Scenario 46, Agent 7, Frame 277

The group moves to the resource and begins mining. The agent pictured to the extreme south in frame 277 joins the group, by the time frame 348 is reached.

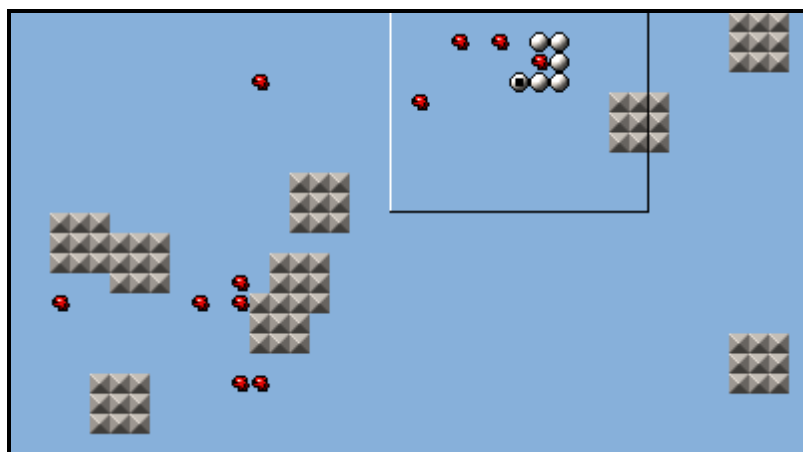


Fig 3.10 The Mining Experiment - Scenario 46, Agent 7, Frame 348

3.1.5 Conclusions

Clearly, there exists an emergent grouping behaviour that depends upon certain environmental features and agent characteristics. The groups are non-intentional, have life-times that depend mostly on mineral proximity's and would not exist at all were it not for the fact that the agents take a measure of time to mine a mineral and fill their backpacks. Also, the agents are clones and thus all behave in the same way. If they inhabit the same area, it is most likely that they will arrive at the same decisions.

What makes this so interesting is that there are no plans, no master control agent and no communication. However, the agents appear to behave in a collective way, mining minerals in groups and maintaining rough group membership integrity. Once a group deposits its load at the home base, it will mine the mineral that is closest. So the home base acts as a congregation point from which new "missions" may begin. Additionally, it is interesting to observe the way in which environments have fashioned the behaviour of a community of agents within this simulation system.

3.2 Experiment 2 - The Hunting Experiment

3.2.1 Introduction

In this experiment, we created a heterogeneous collection of Predator and Prey agents derived from the base class. The purpose of the Predator agents was to seek Prey agents, kill them, and feed from the resulting carcasses. The Prey agents had the task of staying alive by leaping 2 spaces to a safer position whenever a Predator agent was detected.

We utilised the behaviours of the base class agent and added a few others that would represent the acts of feeding from carcasses and avoiding predators. The following set of production rules were applied to each Prey agent:

```
IF knows of predator
    IF can avoid predator
        FINISH
    ENDIF
ENDIF
```

```
IF targeting anything
    IF can reach that target
        THEN seek that target
        FINISH
    ENDIF
ENDIF
```

```
IF unable to wander anywhere
    THEN do nothing
ELSE IF can not seek the wander location
    THEN do nothing
ENDIF
```

```
FINISH
```

The following set of production rules were applied to each Predator agent:

```
IF hungry AND adjacent to a carcass
    THEN feed from that carcass
    FINISH
ENDIF
```

```
IF hungry AND adjacent to a prey
    THEN pounce on that prey
    FINISH
ENDIF
```

```
IF hungry AND knows of carcass
    IF can reach the closest carcass possible
        THEN seek that carcass
        FINISH
    ENDIF
ENDIF
```

```
IF hungry AND knows of prey
    IF can reach the closest prey possible
        THEN seek that prey
        FINISH
    ENDIF
ENDIF
```

```
IF targeting anything
    IF can reach that target
        THEN seek that target
        FINISH
    ENDIF
ENDIF
```

```
IF unable to wander anywhere
    THEN do nothing
ELSE IF can not seek the wander location
    THEN do nothing
ENDIF
```

```
FINISH
```

These are very simple production systems that define “self-oriented” procedures for seeking out prey and avoiding predators. Although Predator agents feed, they do not possess energy levels that rise and fall according to their activities and level of ingestion. Therefore, if a Predator agent’s stomach is full, it no longer desires to hunt Prey agents and will just wander. Prey agents do not have energy levels and do not feed.

Code for the Predator class may be found in Appendix C.6. Code for the Prey class may be found in Appendix C.7.



C.6 - The Predator Class - Page 271



C.7 - The Prey Class - Page 280

We were aiming to keep this experiment as simple as possible. Had we introduced too many variables that might fit our intuitions about this kind of scenario, our data would require complex analysis in order to cater for wide-ranging influential factors.

There are three aspects of the Prey agents that we are able to adjust:

1. Set their visual range between 2 and 20 cell units
2. Make them explore their environment or possess a complete knowledge of it at the beginning of each simulation
3. Set their food value to any value between 0 and 32700

There are six aspects of the Predator agents that we are able to adjust:

1. Set their visual range between 2 and 20 cell units
2. Make them explore their environment or possess a complete knowledge of it at the beginning of each simulation
3. Set their initial stomach contents to a value between 0 and maximum stomach contents
4. Set their maximum stomach contents to a value between 10 and 500
5. Set their feeding rate to a value between 1 and 32700 carcass units
6. Set their Prey capturing efficiency to a value between 1 and 100%

3.2.2 Investigations

It was decided at first to experiment with varying permutations of these adjustable aspects and visually inspect the results. If any form of emergent activity developed, the parameters would then be noted and a set of iterative experiments planned to give some assurance that the activity was not purely by accident alone.

The most notable activity that emerged was a form of pack-hunting or flocking behaviour by the Predator agents when pursuing Prey agents. We also noticed that with large amounts of predators, Prey agents were often captured due to being surrounded by Predator agents. The flocking behaviour, in particular, resembled the kind reported by Mataric (1995). In this paper, mobile robots were given base behaviours that could be composed together in order to produce higher-level behaviours. A certain “recipe” of base behaviours caused the robots to flock within their experimental domain. Essentially, we have achieved the same result by making the Predator agents chase Prey agents. Therefore, a Prey agent has the function of steering the flock of Predator agents, whilst being pursued. We also noticed that sometimes the flock moved in a way that is similar to the osmotropotactic behaviour of ants observed by Mataric (1995), in that they all moved along unidirectional lanes.

3.2.3 Experiment Plan

Having noticed that flocking was not always responsible for Prey agent capture, we decided to plan a series of experiments that would evaluate the impact of flocking on the amount of

Predator agents fed, using a variety of predator numbers. We were interested in whether the emergent behaviour of flocking also had an emergent functionality in terms of food gained for the predator community. We decided to formally run a set of experiments, each having 50 scenarios and inspect the results. Each experiment used the same environment and parameters, except for variation in predator numbers. The parameters were set as follows:

- **Four experiments:**

- 1. 1 Predator on 1 Prey**
- 2. 2 Predators on 1 Prey**
- 3. 4 Predators on 1 Prey**
- 4. 8 Predators on 1 Prey**

- **500 turns for each scenario in each experiment**
- **An environment map generated by the experimenter**
- **All agents placed randomly by the system for each scenario**
- **Predator visual range 12**
- **Prey visual range 4**
- **Prey food value 200 carcass units**
- **Predator feeding rate 10 carcass units per turn**
- **Predator initial stomach contents 0**
- **Predator maximum stomach contents 20**
- **Predator capturing efficiency set to 40%**
- **Exploring off**

There will always be enough carcass units for every Predator agent should a Prey agent be captured quickly enough. Also, we have made the Prey agent rather “slippery”, by setting the predators’ capture efficiency to 40%, so that capture is not always inevitable from simply being close enough. Notice that the Predator agents have a superior visual range compared with Prey agents.

The following screen-shot illustrates the environment used for each experiment:

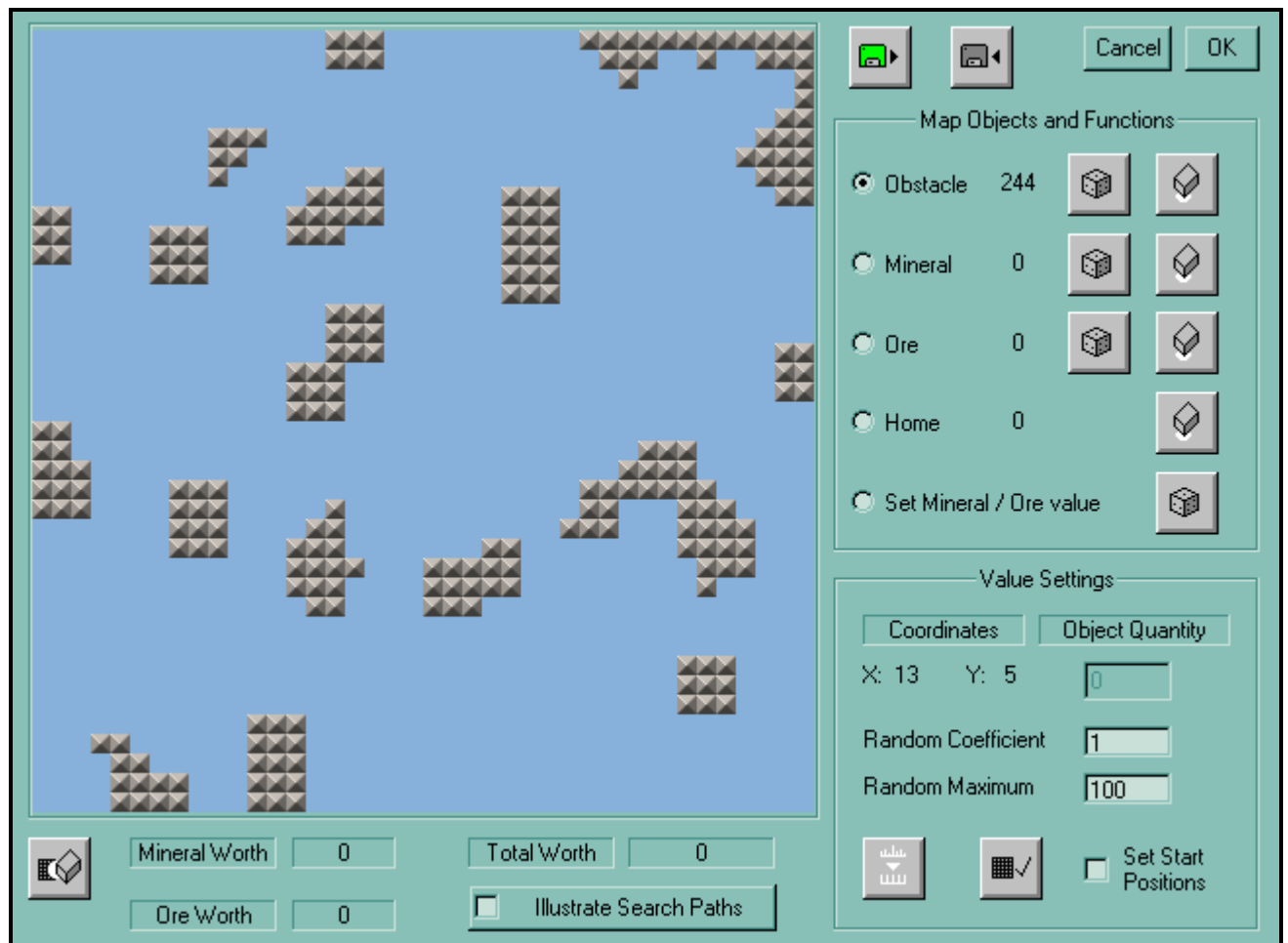


Fig 3.11 The Hunting Experiment Environment

3.2.4 Experiment Results

We divided the results into classes of events that led to a Prey agent being captured:

Flocking (flock)	Prey agent was “pressured” by a flock shortly before capture
Flocking and Surrounding (flock+)	Prey agent was being pursued by a flock, but also other Predator agents or flocks were involved in facilitating capture of the Prey agent
Surrounding (surround)	No flocking activity resulted in the Prey agent’s capture. The Prey agent was surrounded by individual Predator agents
Individual (individual)	Only one Predator agent was responsible for the capture of the Prey agent

For each of the four experiments, we measured the turn at which a kill was achieved and the total sum of the entire Predator agents’ stomach contents. Each scenario was then visually inspected and categorised under the above classes.

The following graph illustrates the impact of flocking and surrounding for all four experiments:

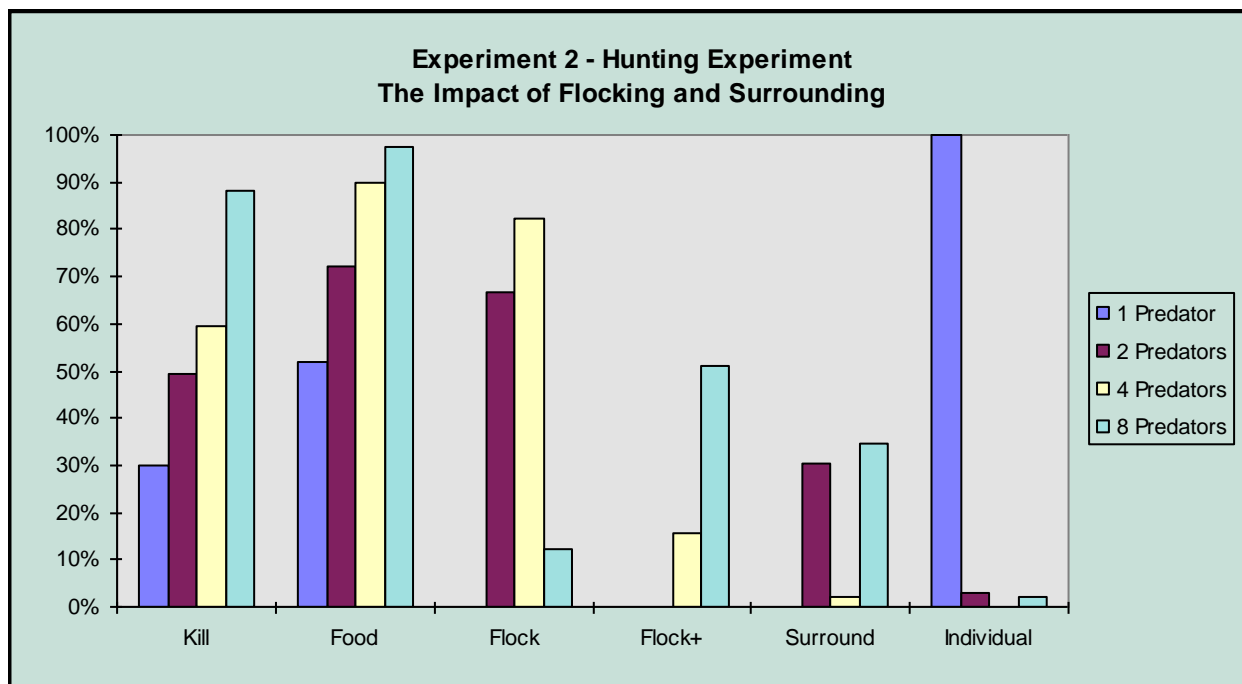
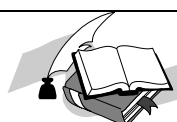


Fig 3.12 The Hunting Experiment Chart of Results

There is a correspondence between flocking and the increase in food obtained when the pack of Predator agents is of a certain size. However, large communities of Predator agents benefit mostly by combinations of flocking and surrounding. Under our model, we believe that there is an emergent functionality of flocking in terms of the food benefits, according to certain Predator agent pack sizes against one Prey agent.

Data tables and a description of how the data in the graph was processed occurs in Appendix T.7.



T.7 - The Hunting Experiment Data - Page 183

Of course, the emergent functionality is partly attributable to the emergent flocking behaviour, so we examined every scenario for illustrative examples of this phenomenon. We used the Trails Algorithm described in Section 2.4.2 to display the paths that every agent had taken in some of the scenarios. A Prey agent leaves a thicker line than a

Predator agent using the Trails Algorithm. The following screen-shot is from experiment 2 and illustrates two Predator agents flocking behind a Prey agent, leading to the eventual capture of the Prey agent at the top-right of the picture:

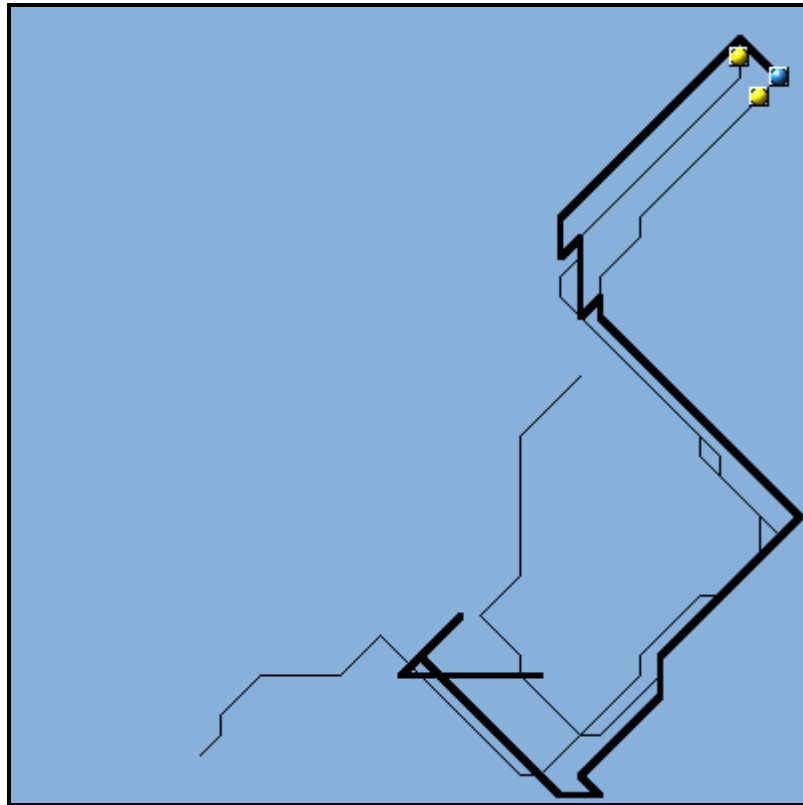


Fig 3.13 The Hunting Experiment - Experiment 2, Scenario 44, Agent Trails

The next screen-shot is from Experiment 4 and displays a much longer flocking period before the Prey agent is captured:

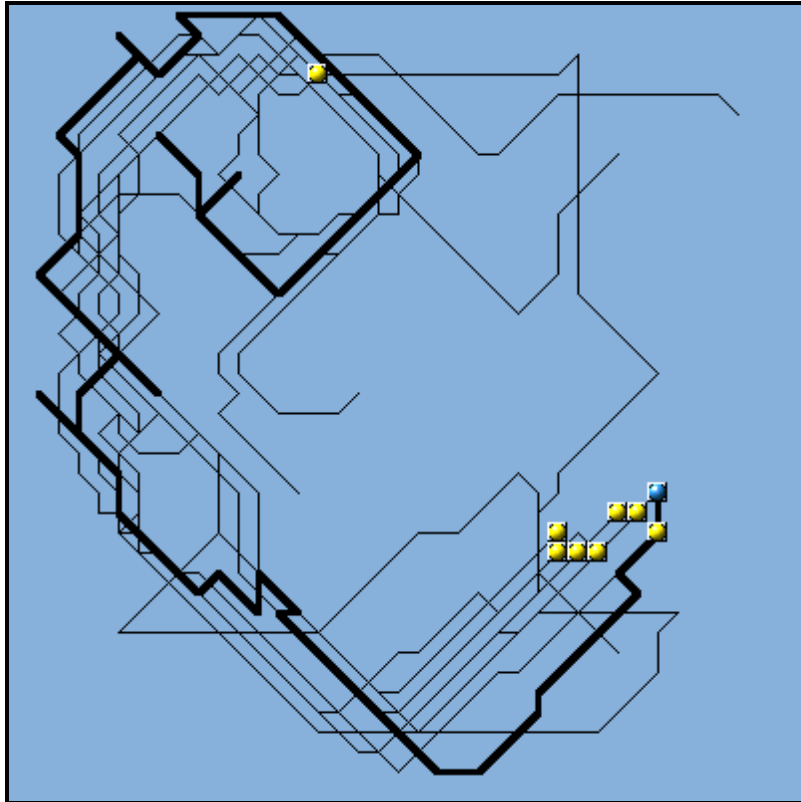


Fig 3.14 The Hunting Experiment - Experiment 4, Scenario 39, Agent Trails

There were also some rather interesting surrounding examples, one of which is illustrated below:

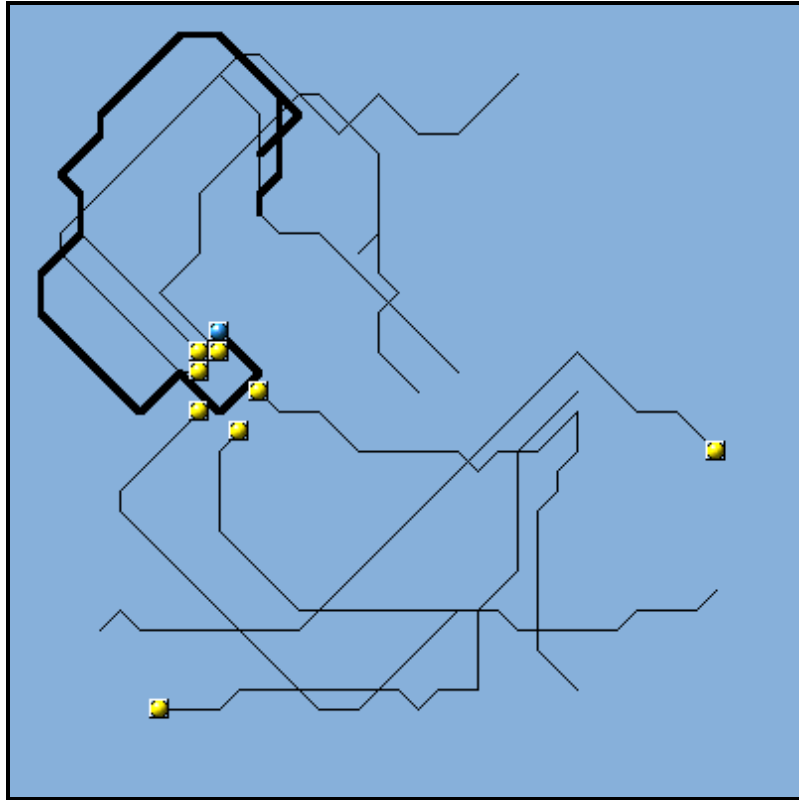


Fig 3.15 The Hunting Experiment - Experiment 4, Scenario 24, Agent Trails

In the above example, the Prey agent was being pursued by a flock of three Predator agents and was pushed into a closing net of three other Predator agents to the South. Such a devious and complex capture would normally be awarded a fair degree of “intelligence”, but we have generated it here using a composite of simple rules with no planning or communication. Incidentally, the Prey agent almost escaped from the net but was captured by the Predator agent immediately below it in this picture.

In order to illustrate the agents’ visual characteristics responsible for flocking more clearly, we have provided the following screenshot from Experiment 4, Scenario 9, which shows the extent of a Predator agent’s visual field. In this example, Predator agent 0 has been tagged and is targeting a location that the Prey agent had previously inhabited before leaping away from the flock. The framed inset also displays the Prey agent’s visual field for comparison:

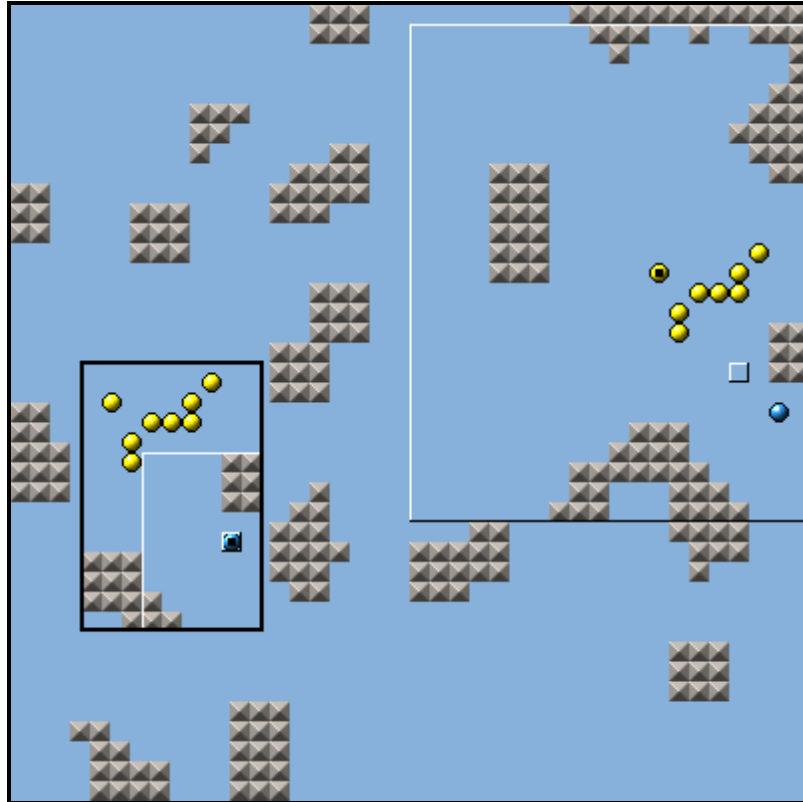


Fig 3.16 The Hunting Experiment - Experiment 4, Scenario 9, Agent Fields of View

3.2.5 Conclusions

We have encountered some emergent behaviours in this experiment that also have emergent functionality. The behaviours of these agents are not only determined by their inherent attributes but also by the components and dynamics of their environment. The flocking behaviour in particular was very clear.

Essentially, we feel that we have demonstrated how emergent phenomena may develop within a system and how possible benefits may occur when agents within artificial communities have identical responses to certain stimuli.

3.3 Experiment 3 – The Chain-Mining Experiment

3.3.1 Introduction

In this experiment, we developed the following hypothesis:

“A community of mining agents could form “transport chains” for transporting ore from the source to the destination through emergent behaviour.”

Designing for emergent behaviour may sound rather paradoxical, but our interest was generally fuelled by our observations of the real world. Humans form chains to transport cargo. For instance, from a truck to a storehouse. The people distribute themselves so as to use up the manpower in such a way that loads may be passed from person-to-person along a chain. We intuitively attribute quite a high degree of cognitive capability to organise such tasks. Indeed, the initial set-up of the chain requires knowledge of how many persons are available, the terrain difficulties, the distance to be travelled, the weight of the loads and who is to be positioned at each step along the chain. Additionally, either a supervisor is required to delegate the tasks or the group has to organise themselves. In all, we deem such an activity to be efficient at transporting loads from source to destination, given the circumstances that exist. Our question is, can we imitate such a behaviour, using very simple rules, so that the overall effect is an emergent phenomenon? Our motivation for attempting such an experiment is the notion that there may exist many behaviours in the real world that are inevitable. What we mean is that the dynamic attributes of the environment and basic attributes and goals of the agents that interact with the environment

shall dictate patterns of behaviour by default, irrespective of any planning by communication of concepts or global awareness of the task.

We planned a very simple experiment and did not make any attempt to model humans. We decided to adapt our Mining agents described in Experiment 1 and pondered the attributes that they might require in order to spontaneously form transport chains. Our modifications are by no means optimal, but we feel that they have evidenced a form of transport chains that is fairly convincing.

We made the following alterations:

- Miners now tire when they are carrying minerals. We gave them an endurance level that depleted by one for every turn that involved moving, whilst carrying a load of ore. If their endurance reached zero, they became exhausted. All other actions resulted in their endurance being incremented by one. Once their endurance was restored to its maximum, they were then recovered.
- If they become exhausted, they must drop their cargo and rest until recovered.
- Miners prefer to mine minerals, rather than carry them. This is to model the concept that carrying ore burns more energy. However, if they chance upon a pile of ore, they will pick it up and transport the load towards the home base if they are in a recovered state.
- Miners always drop loads of ore on top of other piles of ore. If no pile exists adjacent to them, they drop the load in the direction that they were heading. If there is not a space for this heading, they randomly select an adjacent space to deposit the load.
- Miners can pick up ore up to the value of their maximum backpack capacity. This is to reflect the notion that no effort is involved in picking something up, by comparison with mining it.

Of course, these alterations are highly simplified, but simplicity is the thread that we wish to follow. We are interested in how few rules are required in order to simulate transport behaviour within our simulation system and whether it can arise without direct co-operation, communication or planning.

We felt that these changes adequately simulated the difficulties of transporting heavy loads inside the framework of our system, where such loads caused tiredness. We also felt that by giving the Miner agents a preference to mine rather than carry ore, a form of energy conservation was being administered, which seems rather natural. They only pick up ore that they chance upon if they are recovered. Thus, we have defined a hierarchy of responses dependant on the Miner agents' current physical condition and opportunities that present themselves in the environment.

The following set of production rules were applied to each chain-mining agent:

```
IF tired  
  IF backpack not empty  
    IF adjacent to a home base  
      THEN empty backpack at that home base  
      FINISH  
    ENDIF  
    IF adjacent to ore  
      THEN empty backpack at that ore  
      FINISH  
    ENDIF
```

```
        IF facing a space OR a random space is adjacent
            THEN empty backpack at that space
            FINISH
        ENDIF
    do nothing
    erase any targeting
    FINISH
    ENDIF
do nothing
erase any targeting
FINISH
ENDIF
```

```
IF not recovered AND backpack is empty
    THEN do nothing
    FINISH
ENDIF
```

```
IF backpack not empty AND adjacent to a home base
    THEN empty backpack at that home base
    FINISH
ENDIF
```

```
IF backpack not full AND adjacent to a mineral
    THEN mine that mineral
    FINISH
ENDIF
```

```
IF not targeting mineral AND backpack not full AND knows of mineral
    IF can reach the closest mineral possible
        THEN seek that mineral
        FINISH
    ENDIF
ENDIF
```

```
IF not targeting a home base AND backpack is full
    IF can reach the closest home base possible
        THEN seek that home base
        FINISH
    ENDIF
ENDIF
```

```
IF not targeting a home base AND backpack not empty AND not knows of mineral
    IF can reach the closest home base possible
        THEN seek that home base
        FINISH
    ENDIF
ENDIF
```

```
IF backpack not full AND recovered AND adjacent to ore
    THEN pick up that ore
    FINISH
ENDIF
```

```
IF recovered AND not targeting ore AND backpack not full AND knows of ore AND not
knows of mineral
    IF can reach the closest ore possible
        THEN seek that ore
        FINISH
    ENDIF
ENDIF
```

```
IF targeting anything
    IF can reach that target
        THEN seek that target
        FINISH
    ENDIF
ENDIF
```

```
IF unable to wander anywhere
    THEN do nothing
ELSE IF can seek the wander location
    FINISH
ELSE
    do nothing
ENDIF
```

```
FINISH
```

The code for the chain-mining agents may be found in Appendix C.8.



C.8 - The Chain-Miner Class - Page 288

There are six aspects of these chain-mining agents that we are able to adjust:

1. Set the visual range between 2 and 20 cell units

2. Make the agents explore their environment or possess a complete knowledge of it at the beginning of each simulation
3. Set their grabbing capacity to a value between 1 and 1000 mineral units
4. Set their backpack capacity to a value between 1 and 1000 mineral units
5. Set their maximum energy level to a value between 1 and 32700
6. Set their energy recovery per turn to a value between 1 and 32700

3.3.2 Investigations

It was decided at first to experiment with varying permutations of these adjustable aspects and visually inspect the results. If any form of emergent activity developed, the parameters were then noted and a set of iterative experiments were planned to give some assurance that the activity was not purely by accident alone.

Our initial investigations showed that fairly convincing chaining structures developed most of the time, but they depended on the following major factors:

- The return route from the home base to the mineral lay along the same path
- There were sufficient agents to distribute over the route
- The mineral value was high, to provide a distribution of ore along the route and backpacks could be filled quite quickly
- Only a few minerals existed - best results came from one

- The energy levels of the agents were sufficiently high enough to allow them to travel a fair distance before they became exhausted and dropped the ore

It is difficult to be specific about numbers and quantities here, because the dependancies are highly intricate. There are many influencing factors that determine the existence and quality of the chain. However, when these factors are favourable, chain-mining will occur.

3.3.3 Experiment Plan

We created four experiments that would each illustrate varying circumstances and environments. No data was gathered, except for the scenario scripts. The results were visually inspected and analysed. Making agents explore the map only tended to delay the eventual chain-mining result, so we allowed all of our agents a complete knowledge of their environment from the start of each scenario.

Experiment 3.1

This experiment illustrated the best conditions that we have been able to identify in our investigations. The factors necessary are as follows:

- A clear route from the mineral to the home base along a horizontal or vertical path
- No obstacles
- A high mineral value

- Enough agents to fill the route based on their maximum energy values
- Ore is mined fairly quickly

The scenario parameters used were as follows:

- **800 turns, 50 scenarios**
- **Hand crafted map containing one mineral with a worth of 3000 units**
- **1 home base situated by the experimenter remaining constant for every scenario**
- **10 agents placed randomly by the system for each scenario**
- **Maximum energy level 12**
- **Visual range 6**
- **Grabbing capacity 10 mineral units**
- **Sack capacity 50 mineral units**
- **Exploring off**

Experiment 3.2

In this experiment we created a meandering route through the environment with some constricted passages. The aim was to observe the chaining behaviour over a long distance and non-direct route.

The scenario parameters used were as follows:

- **1200 turns, 50 scenarios**
- **Hand crafted map containing one mineral with a worth of 3000 units**
- **1 home base situated by the experimenter remaining constant for every scenario**
- **14 agents placed randomly by the system for each scenario**
- **Maximum energy level 12**
- **Visual range 6**
- **Grabbing capacity 10 mineral units**
- **Sack capacity 50 mineral units**
- **Exploring off**

Experiment 3.3

In this experiment, we observed the effect of having obstacles that obstruct the route from the mineral to the home base. Additionally, we see how having different path alternatives from the mineral to the home base affect the overall chain-mining effect.

The scenario parameters used were as follows:

- **1000 turns, 50 scenarios**
- **Randomly generated map remaining constant for every scenario**
- **1 home base situated by the experimenter remaining constant for every scenario**
- **12 agents placed randomly by the system for each scenario**
- **Maximum energy level 12**
- **Visual range 6**

- **Grabbing capacity 10 mineral units**
- **Sack capacity 50 mineral units**
- **Exploring off**

Experiment 3.4

In this experiment we only generated one scenario. The aim was to illustrate how the agents' group behaviour adapted to the situation of mineral resources away from the home base.

The scenario parameters used were as follows:

- **1444 turns**
- **Hand crafted map containing 4 mineral resources each with a value of 3000 mineral units**
- **1 home base situated by the experimenter**
- **8 agents placed by the experimenter adjacent to the home base**
- **Maximum energy level 12**
- **Visual range 6**

- **Grabbing capacity 10 mineral units**
- **Sack capacity 50 mineral units**
- **Exploring off**

3.3.4 Experiment Results

Experiment 3.1

The following screen-shot is of the environment used in every iteration of this experiment:

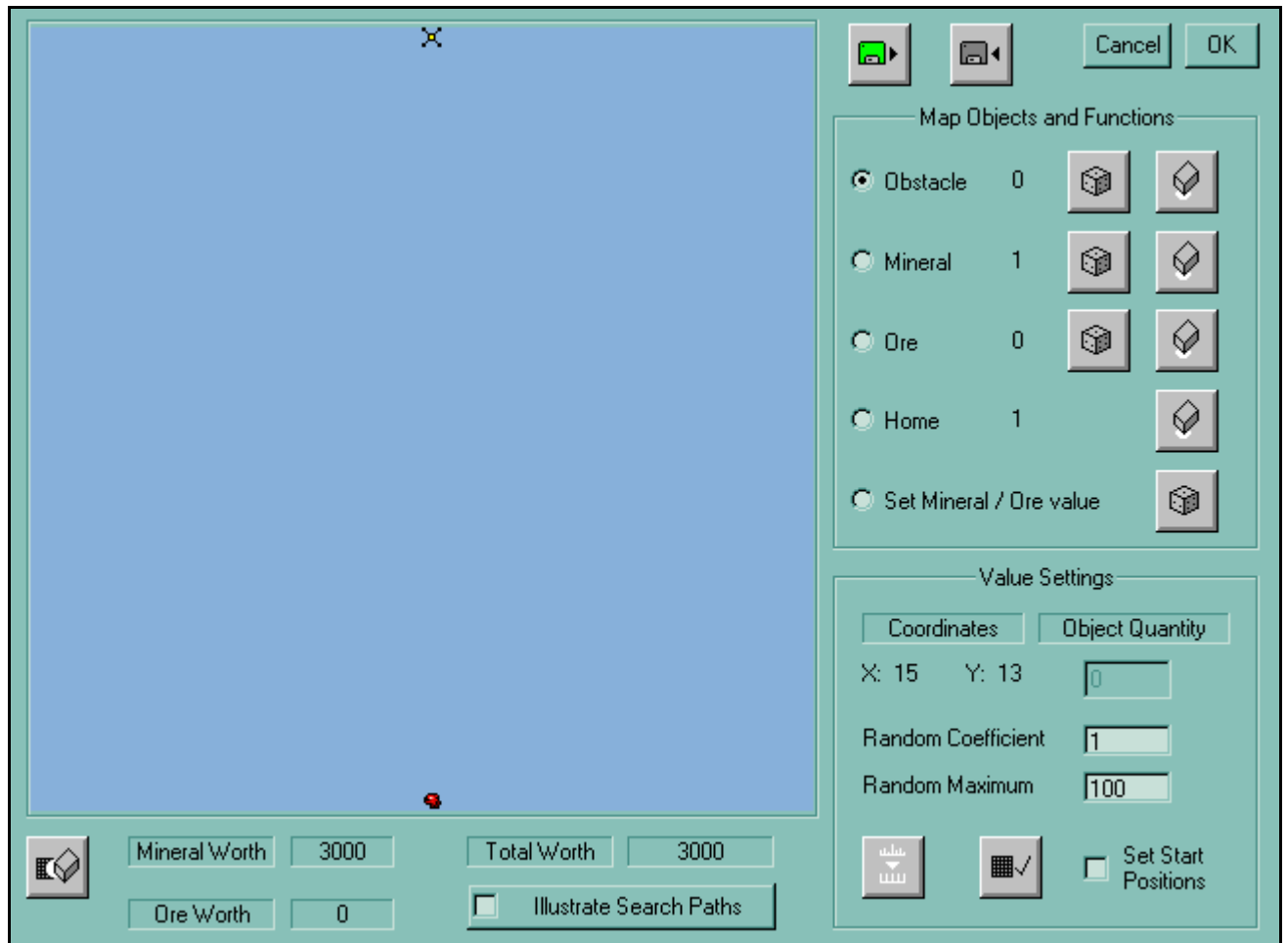


Fig 3.17 The Chain-Mining Experiment – Experiment 3.1 Environment

We studied the 50 scenarios in detail and found that all of them exhibited emergent chain-mining phenomena. We have selected scenario 50 as our illustrative example.

The initial state of the environment for scenario 50 is illustrated in the following screen-shot at frame 1:

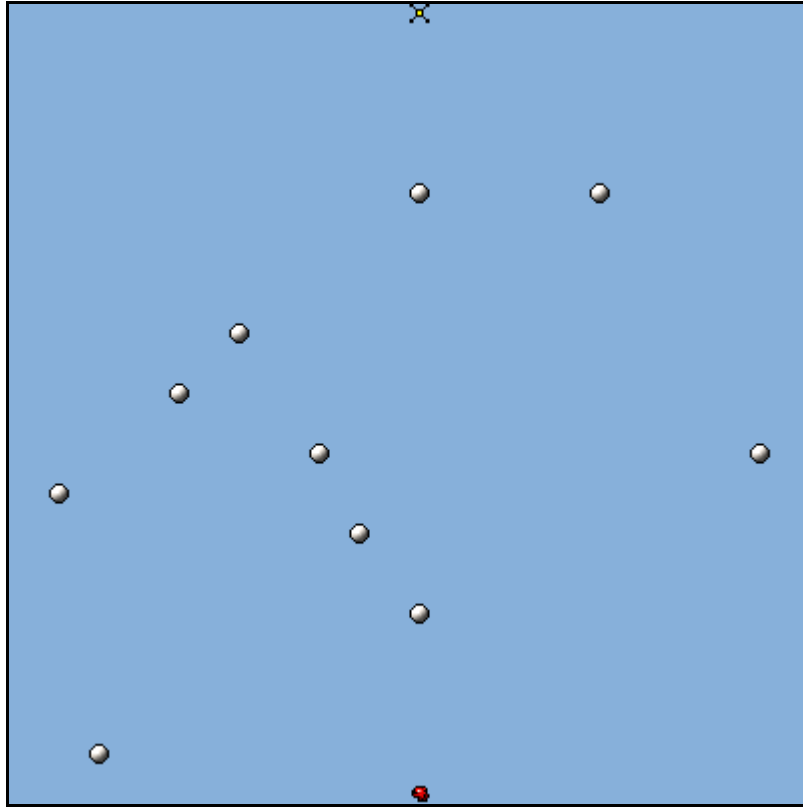


Fig 3.18 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 1

The first grouping of deposited ore was established by frame 49:

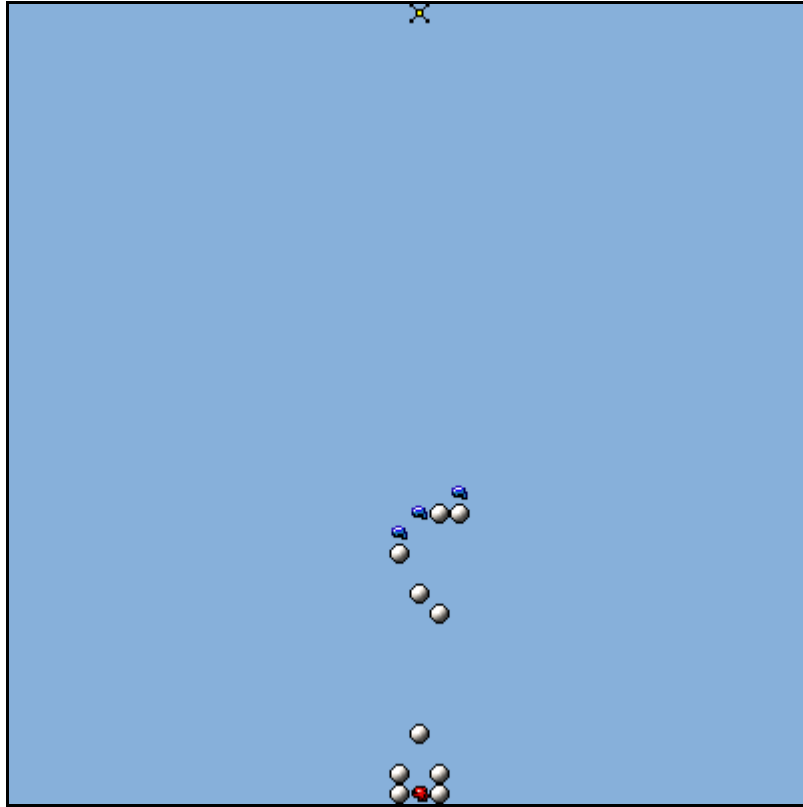


Fig 3.19 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 49

At frame 87, a second grouping of ore had developed:

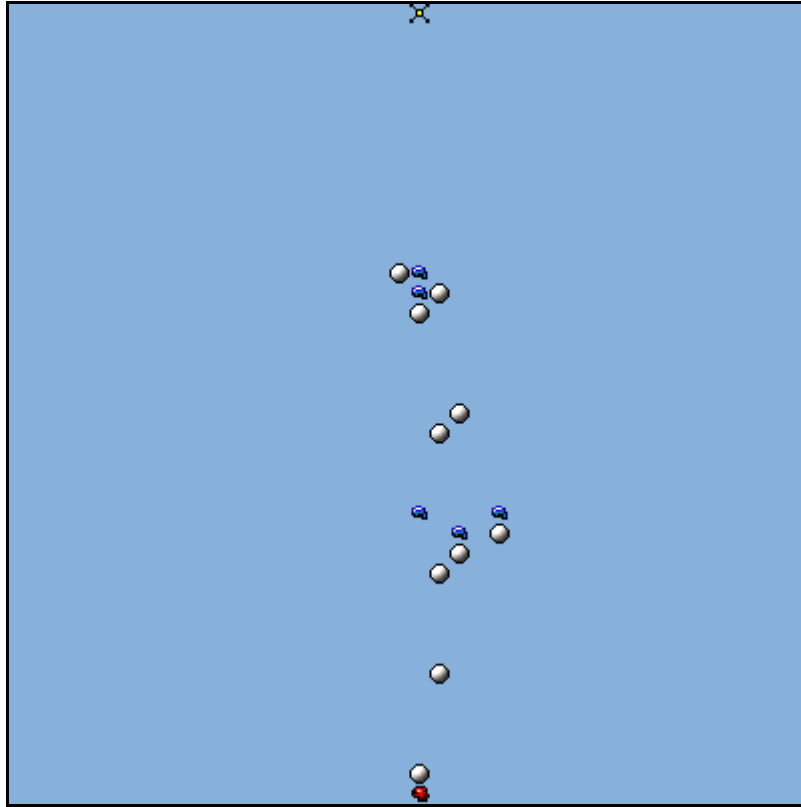


Fig 3.20 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 87

At frame 270, the chain had developed fully and groups of agents had become temporarily trapped between the piles of ore. This lead to them collecting and depositing the ore within their “transport region” in the chain. Each “transport region” being separated by a “link”:

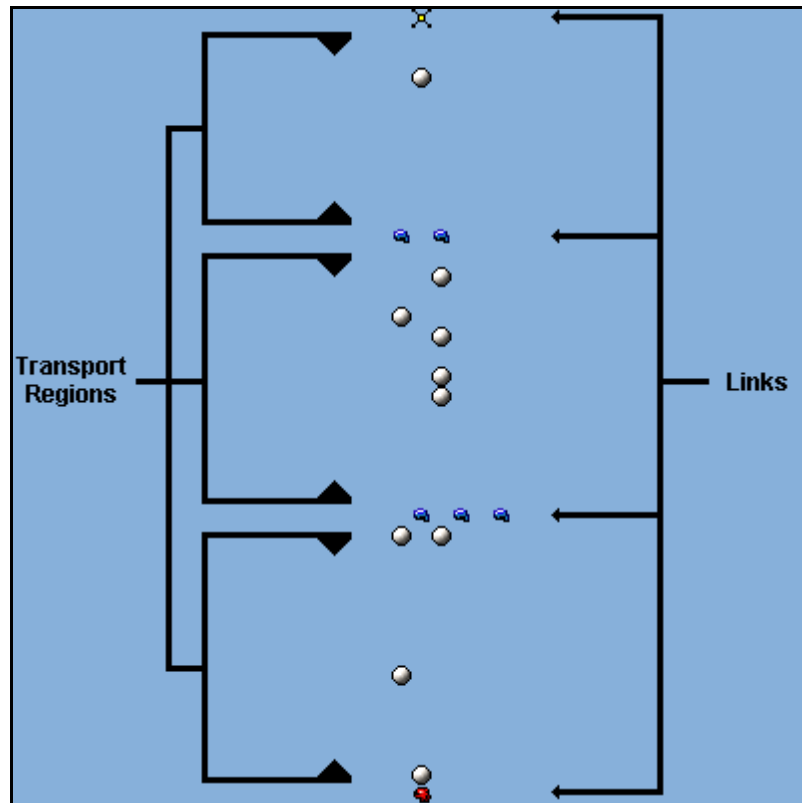


Fig 3.21 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 270

The “links” in this chain were unstable. If a sufficient supply of ore was not present to block the passage of agents in the “link”, they would descend down the chain and a new distribution of agents would become established. In this next screen-shot, frame 607, the piles of ore nearest the home base have become depleted, allowing agents above and below to trickle through to the next “link”:

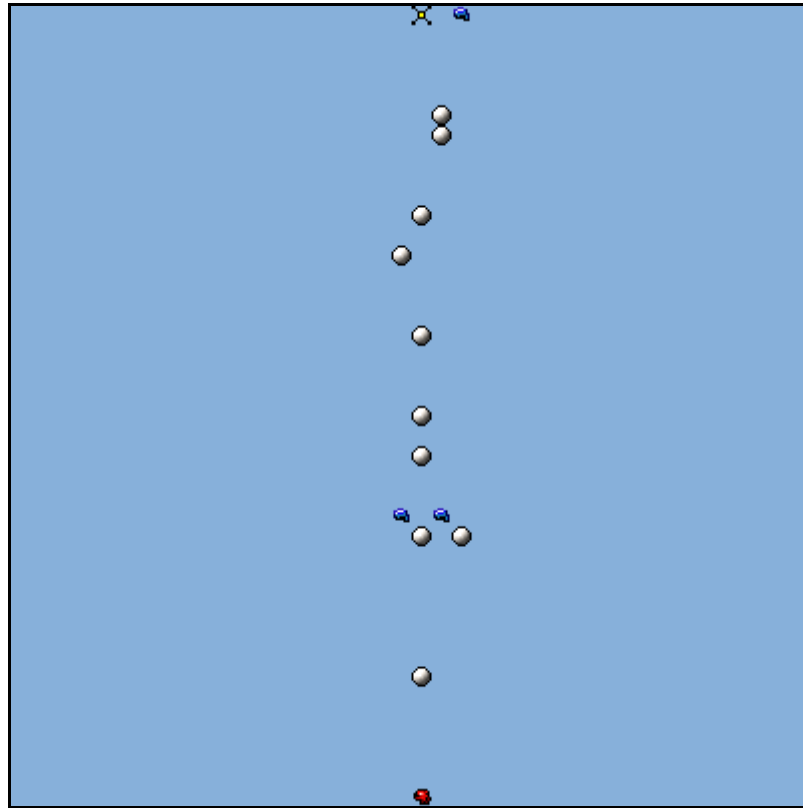


Fig 3.22 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 607

We had noticed that in particular, throughout this experiment, the chains were “self-adjusting” according to the presence of piles of ore. If sufficient supply of ore existed at one of the “links”, the agents between these “links” were mostly confined to passing ore between them. If a pile became depleted, agents would pass through to the next “link”.

Finally, we have illustrated the overall activity of these agents in this experiment by using our Path Density Algorithm to show the tracks that the agents have left in their environment. We set a threshold of 1000 that would highlight all densities that are greater or equal to this value in red. This screen-shot is at frame 724, when all of the ore had been deposited at the home base. Notice that the “hotspots” indicated in red correspond with the “links” in the chain that we had illustrated in Fig. 3.21:

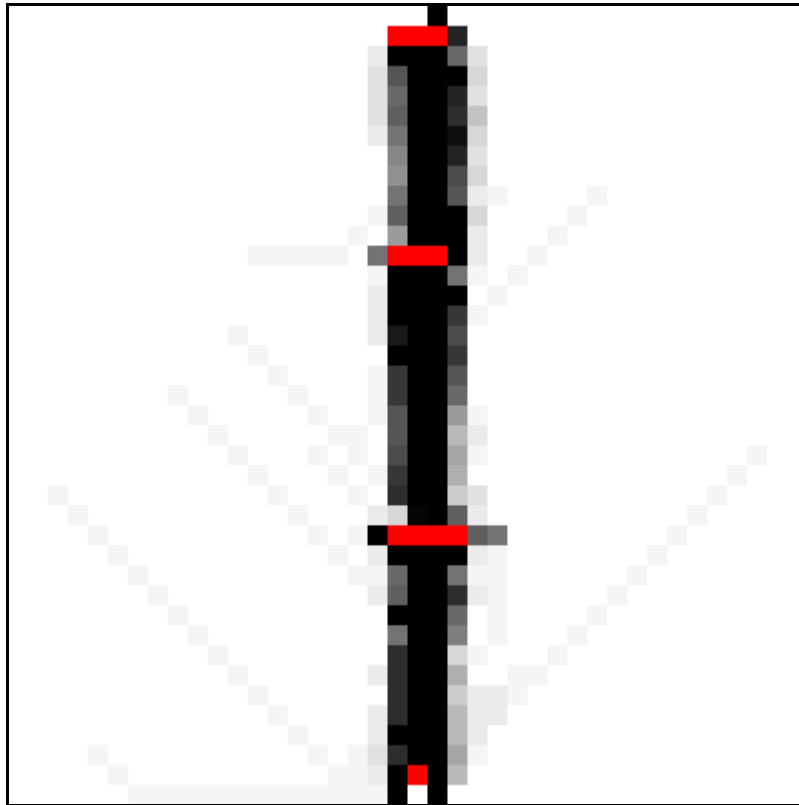


Fig 3.23 The Chain-Mining Experiment – Experiment 3.1, Scenario 50, Frame 724,
Path Densities

Experiment 3.2

The following screen-shot is of the environment used in every iteration of this experiment:

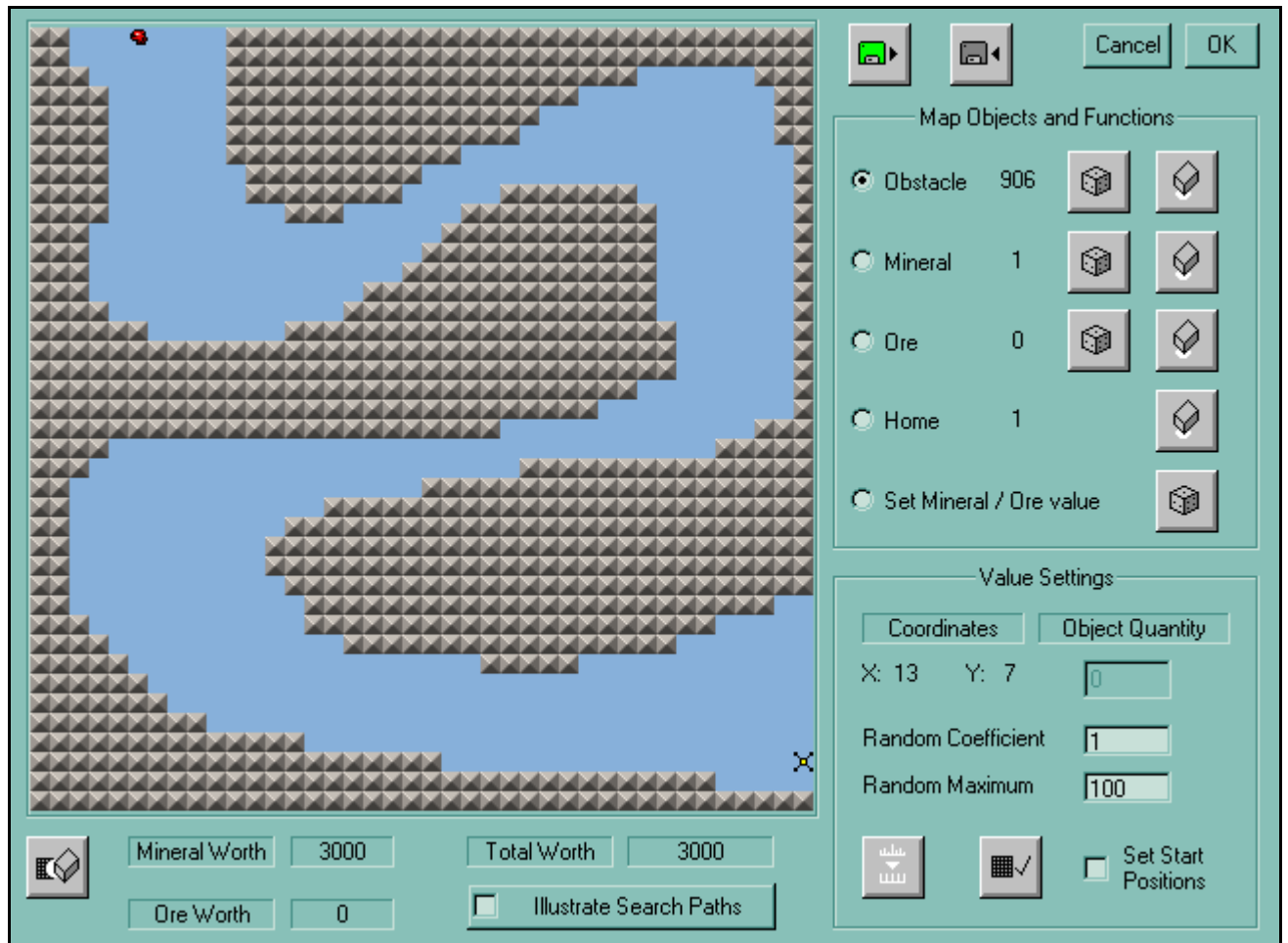


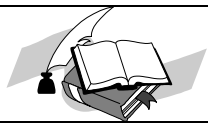
Fig 3.24 The Chain-Mining Experiment – Experiment 3.2 Environment

Of the 50 scenarios that we ran for this experiment, a familiar set of characteristics emerged:

- The “links” in the chains were not totally clear and moved positions
- Often the “links” collapsed
- Alternative routes back towards the mineral existed near the home base, resulting in a poor “transport region” at this location

The most fundamental reason for this result is the alternative routes. The form of search algorithm that we have developed for enumerating the minimum cost paths does not

necessarily plot the same route each time. The following screen-shots were taken from the Map Editing tool in our simulation program. We have used a special feature in the simulator to enumerate all possible paths from a certain point to a goal location using a form of “colour temperature” to indicate the distance. The colours range from white to black, through green and blue. Additionally, we are able to plot the minimum cost path between two certain points as a green line. A detailed description of the Path-Searching Algorithm and this feature may be found in Appendix T.4.



T.4 - The Path-Searching Algorithm - Page 164

This first screen-shot displays all of the routes, together with the minimum cost path from the home base to the mineral:

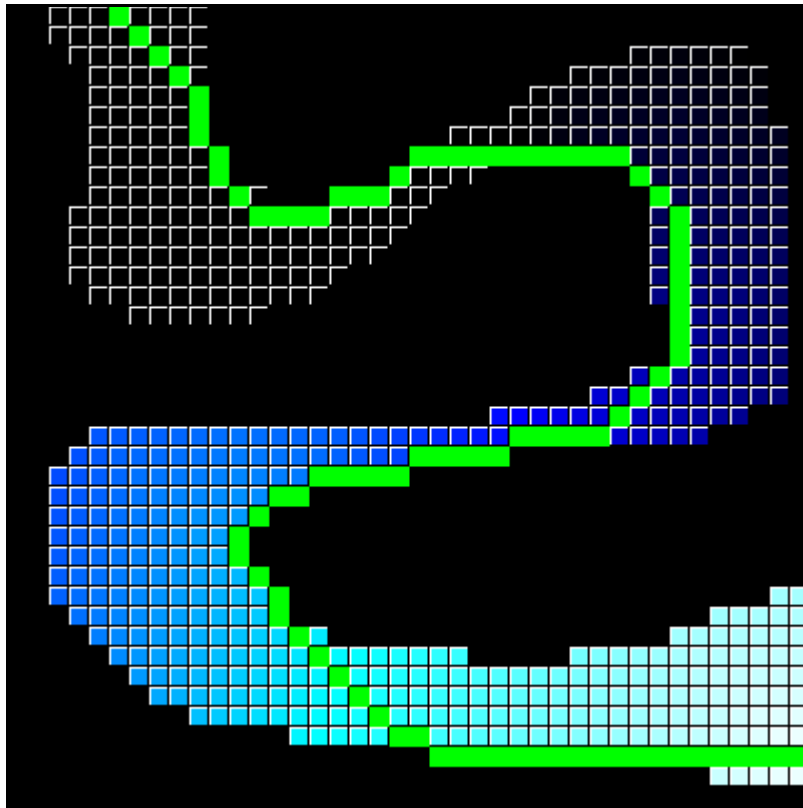


Fig 3.25 The Chain-Mining Experiment – Experiment 3.2, Paths from the Home Base to the Mineral

The next screen shot illustrates the minimum cost path from the mineral to the home base:

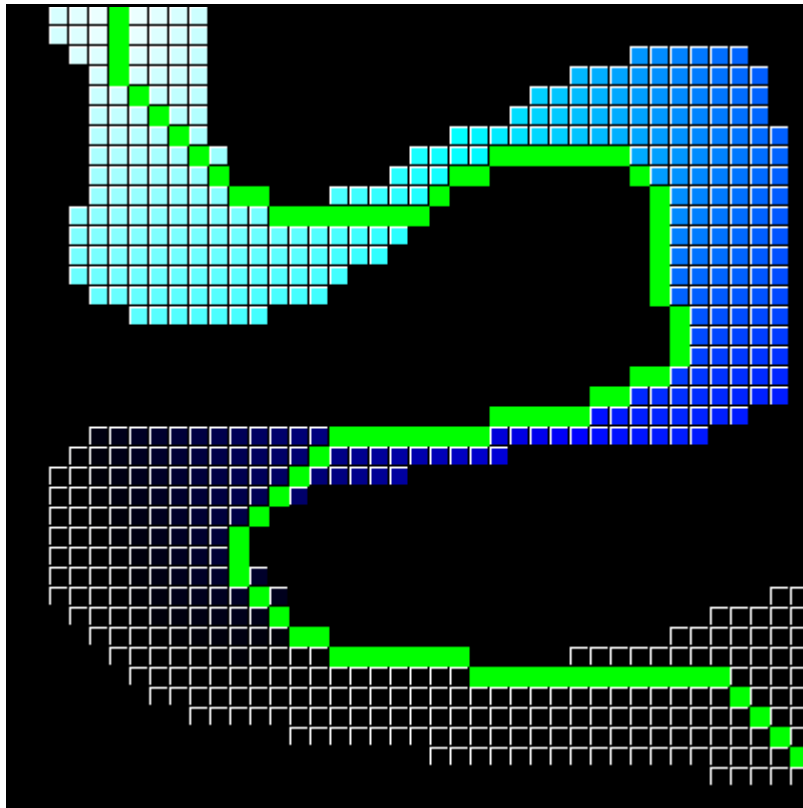


Fig 3.26 The Chain-Mining Experiment – Experiment 3.2, Paths from the Mineral to the Home Base

Notice the variation in the paths, particularly at the bottom-right of each screen-shot. Both paths are optimal, but due to the characteristics of our grid environment representation and the search algorithm, we get different path plots for the same route.

During the design stages of the simulator, this phenomena was noticed but was not considered to be a problem, as long as minimal paths were being generated quickly. Now we admit that this method has deteriorated our ability to demonstrate a certain aspect of our work.

Obviously, if route alternatives exist, the agents will not always collide along the same path. In effect, the chain is disrupted by agents passing by ore piles and failing to pick them up because they are not close enough. However, a crude chaining phenomenon is present in

these scenarios and we have illustrated this fact with the following screen-shot from scenario 32 at frame 262:

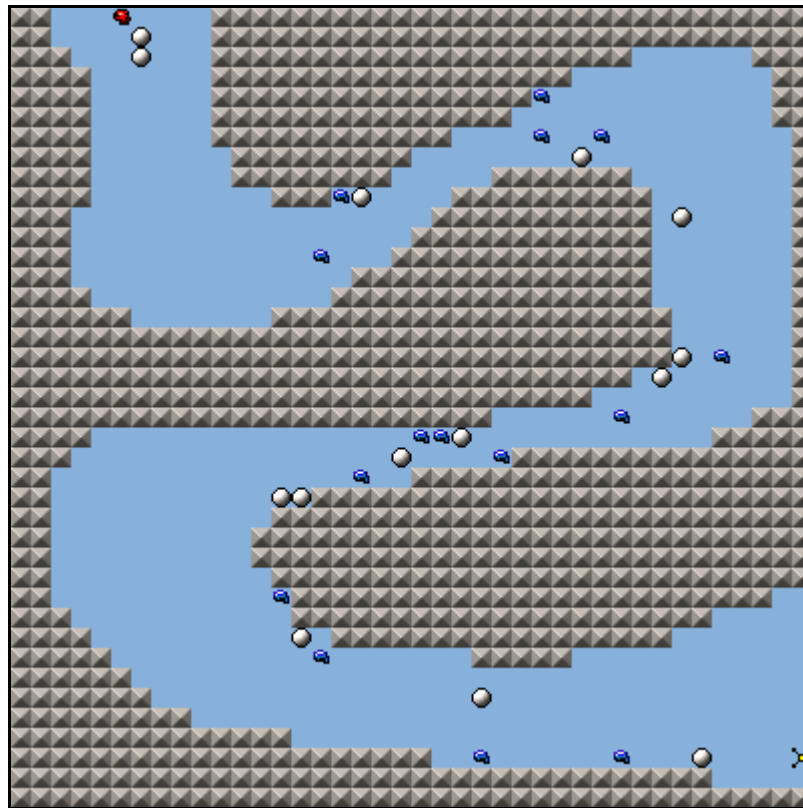


Fig 3.27 The Chain-Mining Experiment – Experiment 3.2, Scenario 32, Frame 262

Using the Path Density Algorithm, we have also illustrated the “hotspots” in red corresponding to areas where the density equalled or exceeded 1300 at frame 1200. Although some “links” are evident, there are areas where the “links” vary in position, particularly around the bottom-left bend in this illustration:

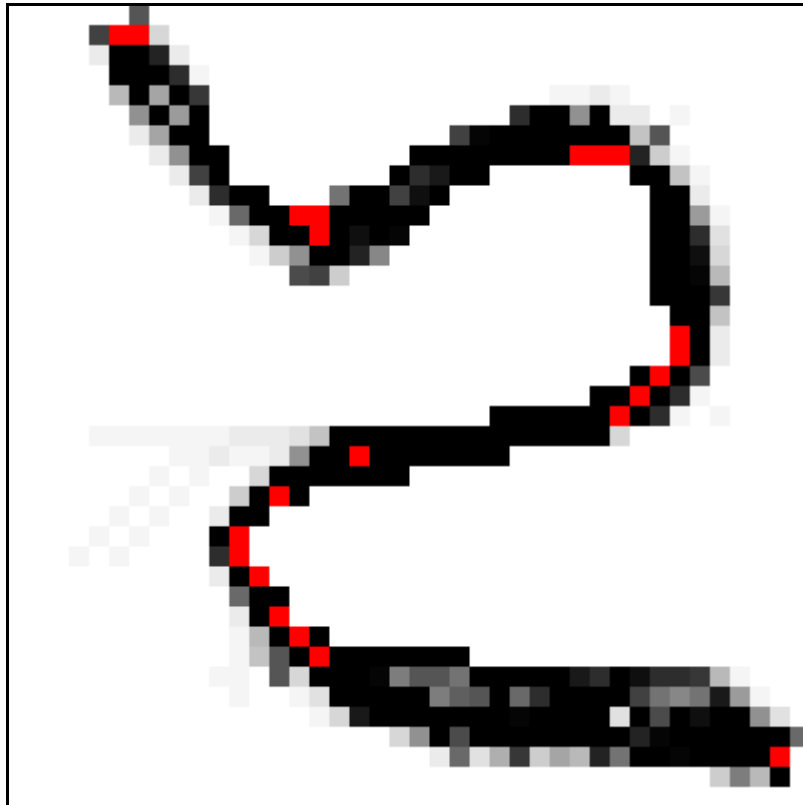


Fig 3.28 The Chain-Mining Experiment – Experiment 3.2, Scenario 32, Frame 1200,
Path Densities

Experiment 3.3

The following screen-shot is of the environment used in every iteration of this experiment:

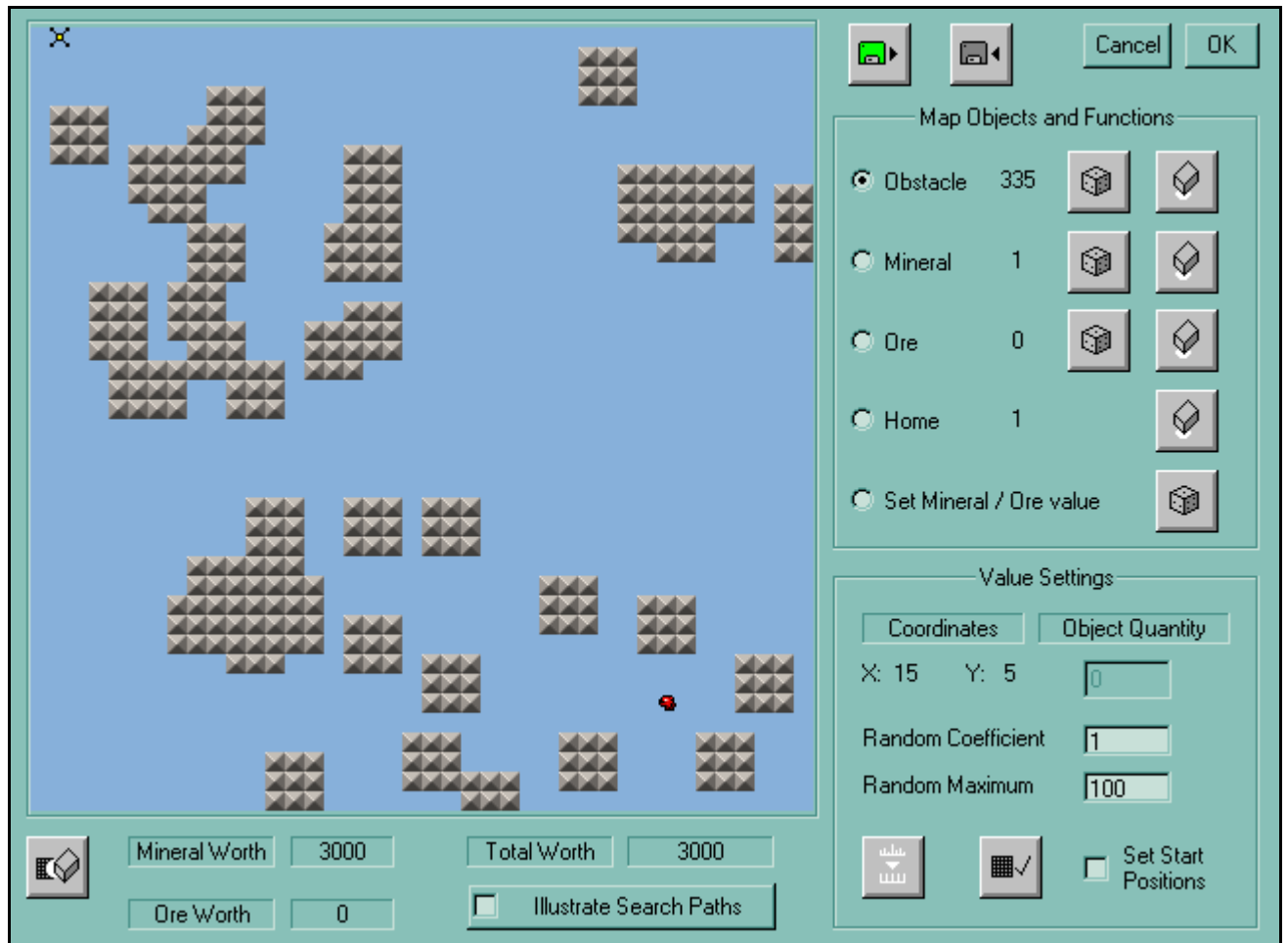


Fig 3.29 The Chain-Mining Experiment – Experiment 3.3 Environment

We examined the scenarios from this experiment set and found that a similar effect was in action as that described in Experiment 3.2. There existed alternative routes and the addition of obstacles had broken up the clarity of any chains produced. We have selected scenario 27, frame 767 as our illustrative example. Using the Path Density Algorithm, we have illustrated the “hotspots” equal to or over a density of 1000. Notice that the “links” do not correspond well with the paths taken by the agents. Although some major “links” have been identified in red, minor “links” do exist elsewhere.

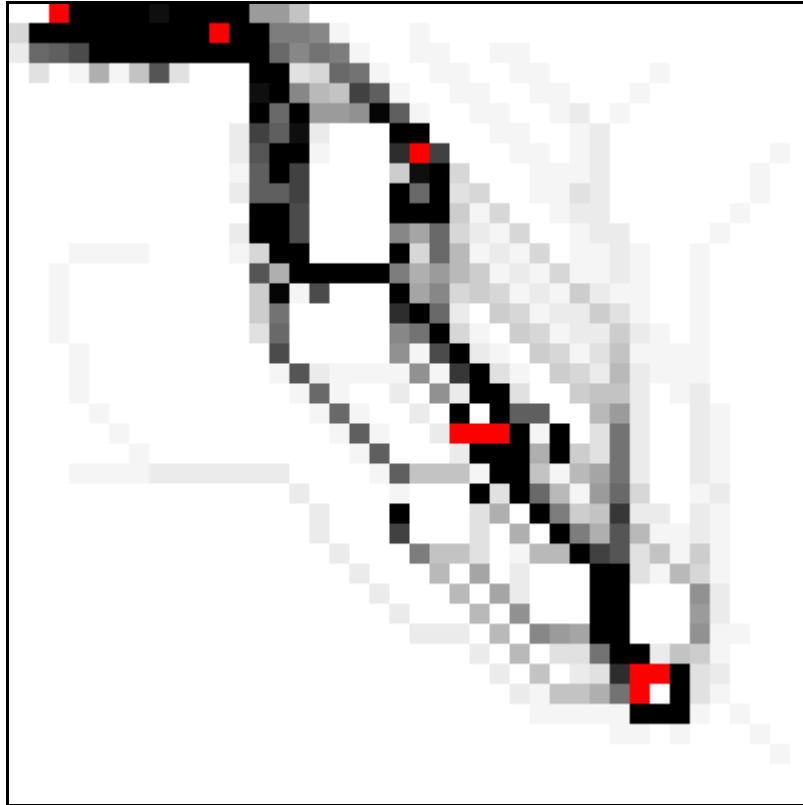


Fig 3.30 The Chain-Mining Experiment – Experiment 3.3, Scenario 27, Frame 767,
Path Densities

Experiment 3.4

In this experiment, we have demonstrated the shifting group behaviour of these chain-mining agents according to the proximity of ore to the home base. Due to the limitations of our simulation system, we developed a rather “artificial” pseudo-environmental situation, which is illustrated below:

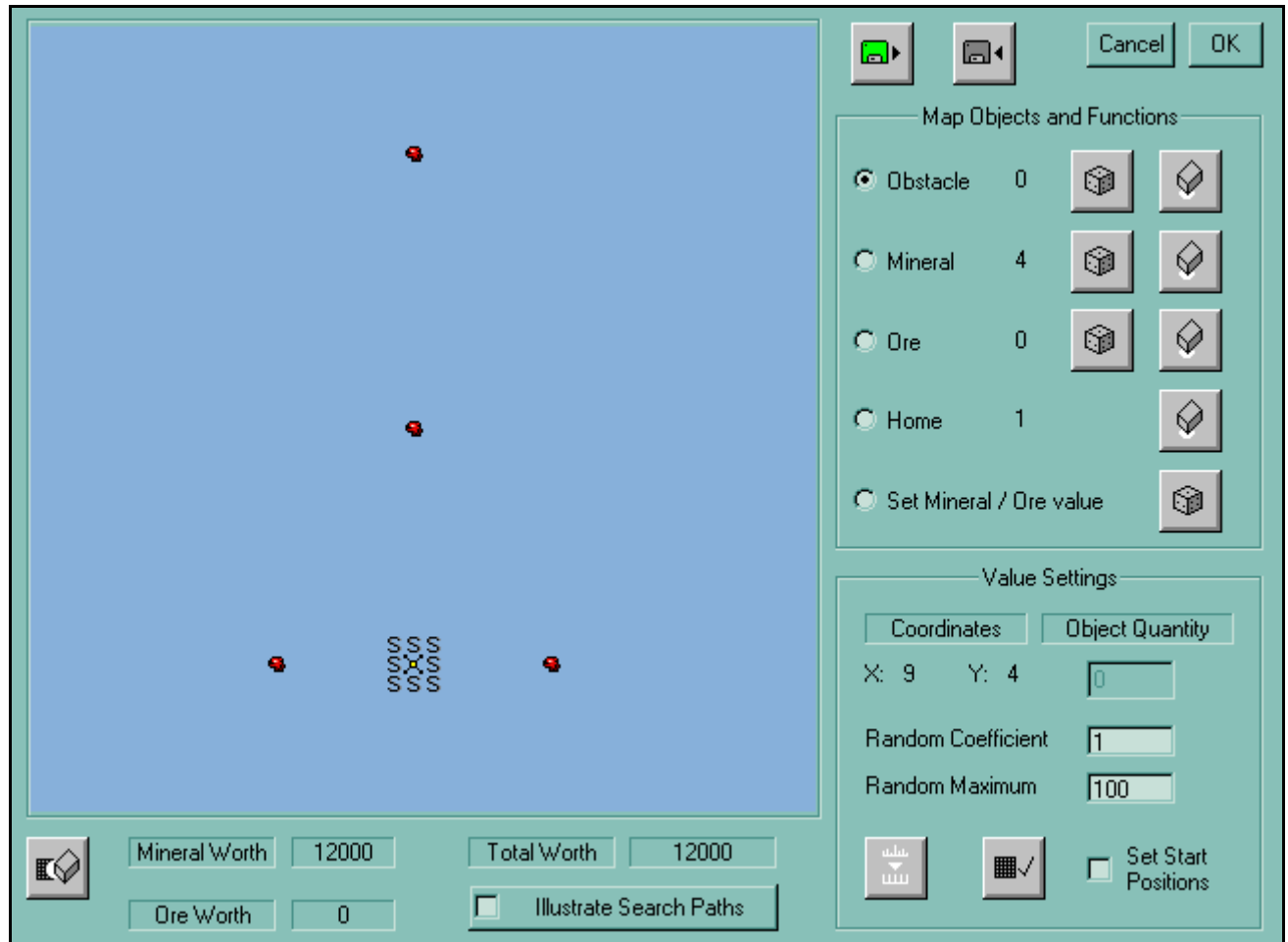


Fig 3.31 The Chain-Mining Experiment – Experiment 3.4 Environment

As you will notice, three of the mineral deposits are near the home base. The fourth is some distance away. The “S” markers on this map indicate that agents will be placed at these locations when a scenario is begun. The objective of this experiment was to demonstrate that the agents mine the nearby mineral resources in the same manner as the agents described in Experiment 1. However, when they encounter the mineral that is farthest away, their group behaviour will change to chain-mining.

By frame 340, the agents had depleted one of the nearest mineral resources and there were no signs of chain-mining:

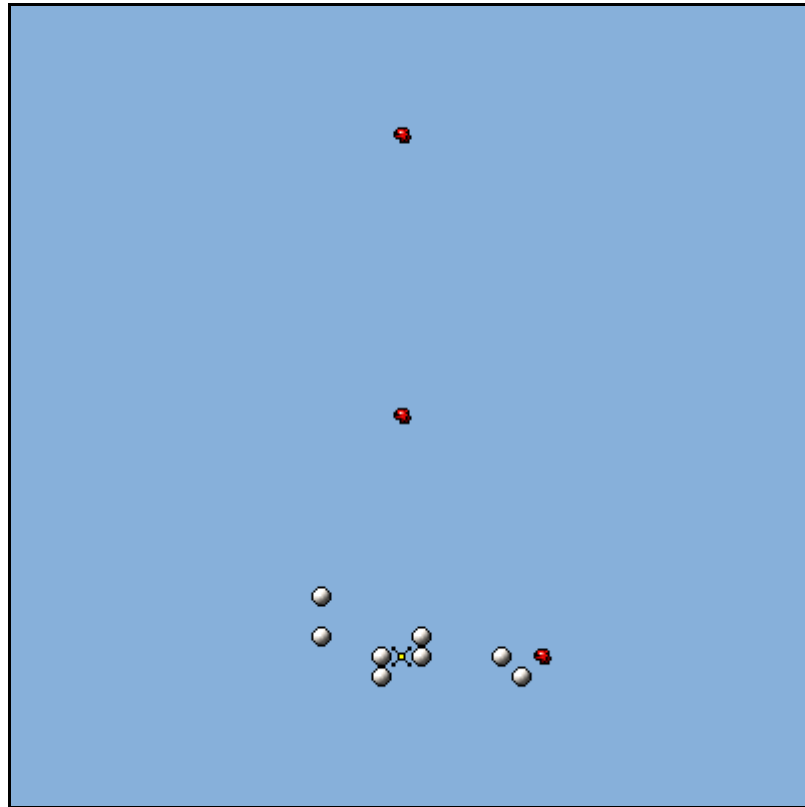


Fig 3.32 The Chain-Mining Experiment – Experiment 3.4, Frame 340

By frame 690, both nearby mineral resources had been mined and the agents had almost finished with the mid-positioned mineral:

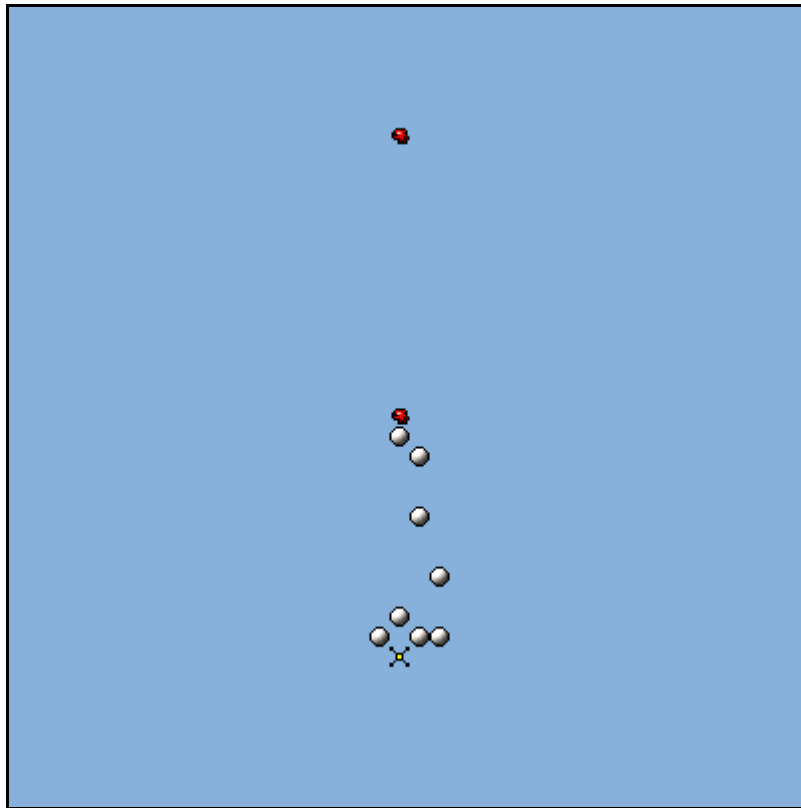


Fig 3.33 The Chain-Mining Experiment – Experiment 3.4, Frame 690

By frame 921, the agents had started mining the most distant mineral and their group behaviour had altered to that of chain-mining:

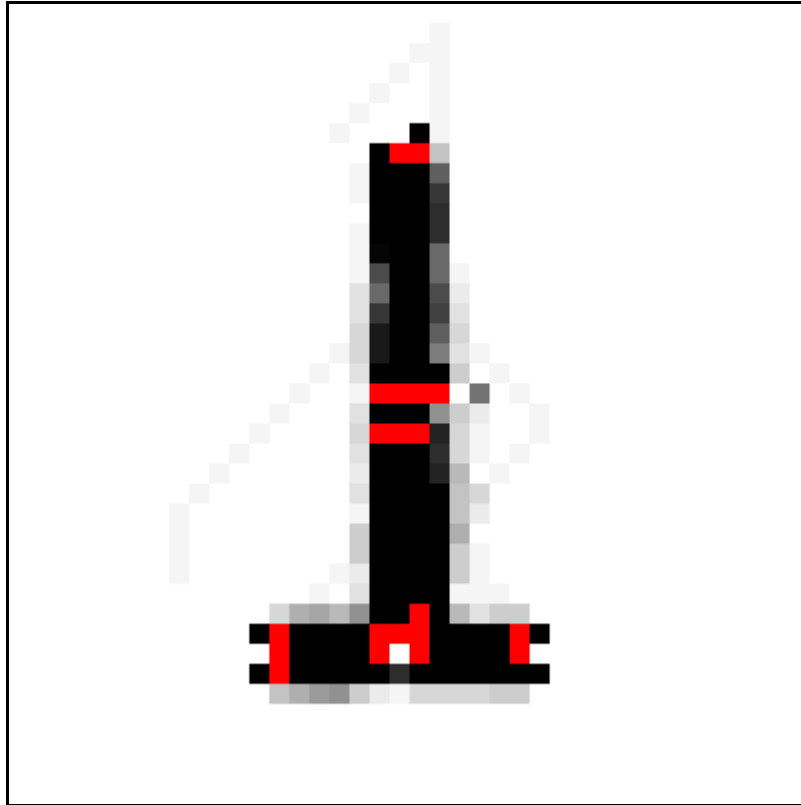


Fig 3.35 The Chain-Mining Experiment – Experiment 3.4, Frame 1361, Path Densities

Notice that we have two red “bars” at the mid-point. This is because the agents not only mined a mineral in this area but also formed a “link” in a chain at this point.

We have therefore demonstrated that these agents’ group behaviour can be determined by the environmental circumstances. Their group activity “adapts” through emergent phenomena to the situation in hand.

3.3.5 Conclusions

We have been able to generate a simple chaining phenomena that occurs when the conditions are right. If these chain-mining agents inhabit a world where the mineral resources are distributed at varying distances from the home base, they will mine the closest minerals to the home base in the same way that our ordinary mining agents would in Experiment 1. However, if the resources are further than their energy allows, they will form “transport chains”, depending on availability of other agents. So the emergent behaviour of these agents will change to reflect the varying circumstances. Unfortunately, our system architecture was not fully appropriate to demonstrate all of the situations that we had tested, but we are given an insight into the ideas proposed and do gain an adequate understanding of the issues involved.

We have satisfied our original hypothesis by generating “transport chains”. The chains are not perfect, there is no orderly arrangement of interactions between the agents, but their emergent behaviour is similar to the real-world analogy that we discussed at the beginning of this experiment. Most importantly, perhaps, is that these agents are furnished with “community behaviour adaptation”, through emergent phenomena. The chain-mining behaviour also has the appearance of a globally organised plan, yet the agents only possess local information about the world they inhabit and they do not communicate. This is similar to the behaviours observed in various species of ants and evidences an ability of such colonies to perform in a co-ordinated manner, despite any organised form of global control. Drogoul (1995) carried out various experiments along a similar theme to develop emergent tactical strategies in games.

3.4 Experiment 4 - The Crowd Dynamics Experiment

3.4.1 Introduction

We designed this experiment to illustrate a possible real-world application of emergent phenomena. We are not claiming that our simulation system has any practical worth. We are merely using its features as an example of how such a system may be used to solve a real problem.

The issue that we are faced with in this experiment is adjusting the layout of a building, such that in the event of an emergency, the occupants may escape quickly and safely. Generally, we have assumed that safety decreases as congestion increases during an emergency situation involving the general public. This is a rough approximation of the hysteria and panic that normally accompanies the human instinct to survive. Therefore, we focus all of our attention on the traffic flow of persons through a building to the available exits.

We created a new Crowd agent possessing the following production system:

```
IF not targeting a home base AND knows of home base
    IF can reach the closest home base possible
        THEN seek that home base
        FINISH
    ENDIF
ENDIF
```

```
IF targeting anything
    IF can reach that target
        THEN seek that target
        FINISH
    ENDIF
ENDIF
```

```
IF can wander somewhere  
    THEN seek the wander location  
    FINISH  
ENDIF
```

```
do nothing
```

```
FINISH
```

These agents have one major goal. They have to reach the nearest home base, if they know of one. Otherwise, they wander until one is found. Ideally, these agents should have pseudo-personalities with varied masses, speeds, degrees of knowledge about their environment and not have the ability to see over walls. We decided to keep this example simple and not attempt to model more than our simulator allowed. The home bases are now representative of exits from the building and we have used the Miner graphic to animate the Crowd agents. The code for Crowd agents may be found in Appendix C.9.



C.9 - The Crowd Class - Page 309

3.4.2 Investigations

We tested varying parameters for this experiment and found that the simplest building arrangements carried the best illustrative impact within the constraints of our model.

3.4.3 Experiment Plan

We created an imaginary scenario wherein we were responsible for checking the layout of a building in order to test its safety in the event of a crisis. Each improvement to the building would be tested with the simulator and the results of the Path Density Algorithm would be used to predict where the best alterations to the building should be carried out.

The parameters for these agents were set as follows for every experiment:

- **93 crowd agents - split into different “rooms”**
- **Turns unlimited**
- **Visual range 6**
- **Exploring off**
- **All path density hot-spots set at values that equal or exceed 250**

All of these agents know where the exits are. This is unrealistic but sufficient for this example.

3.4.4 Experiment Results

We begin with the starting layout of our building which is illustrated in the following screenshot:

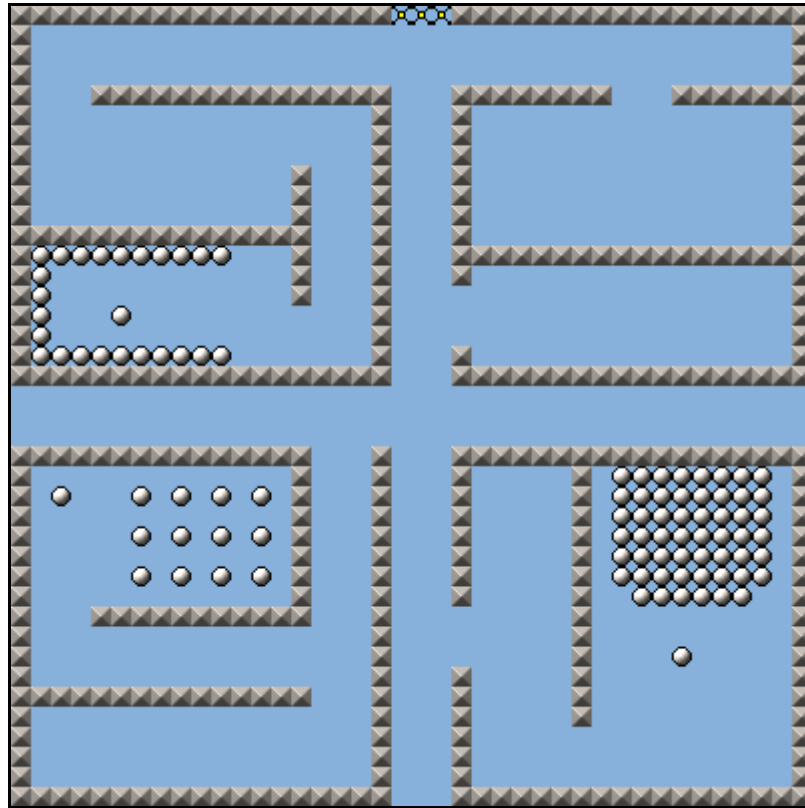


Fig 3.36 The Crowd Dynamics Experiment – Building 1

There is only one exit from this building that lies to the North. The arrangement and density of the agents in each room reflects that room's average intended use. We ran the simulation until all agents had exited from the building and then analysed the paths of the agents using the Path Density Algorithm:

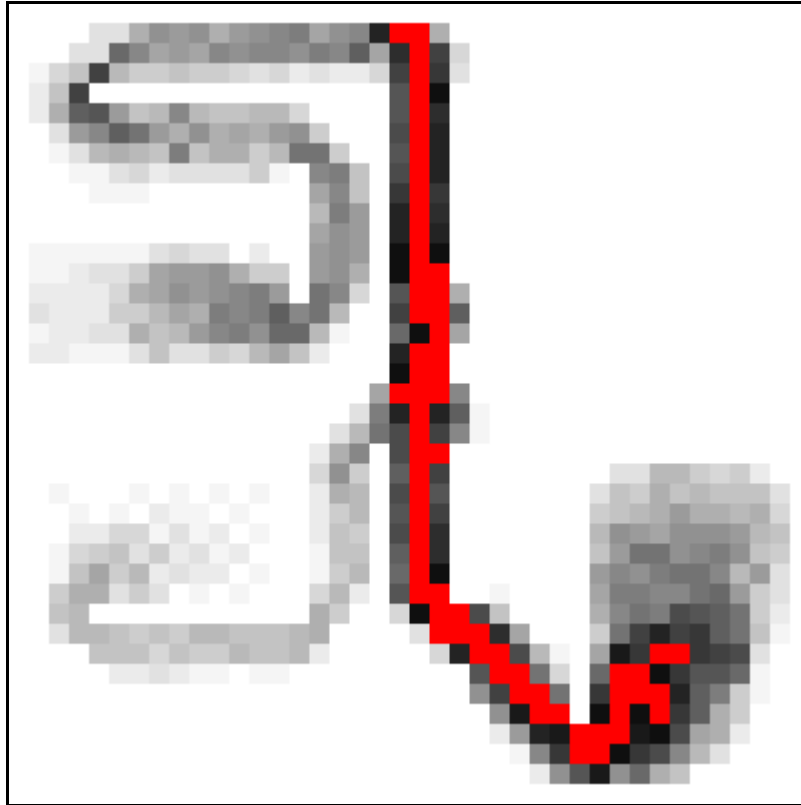


Fig 3.37 The Crowd Dynamics Experiment – Building 1, Frame 111, Path Densities

All agents had exited by frame 111. Notice that quite a considerable traffic load stretches from the bottom right of Fig. 3.37 to the top, indicated in red. We deemed this to be an unacceptable amount of time under this imaginary situation and were concerned by the possible injuries that may be caused by having such heavy traffic.

The building layout was adjusted to provide an alternative exit to the West. This is illustrated in the next screen-shot:

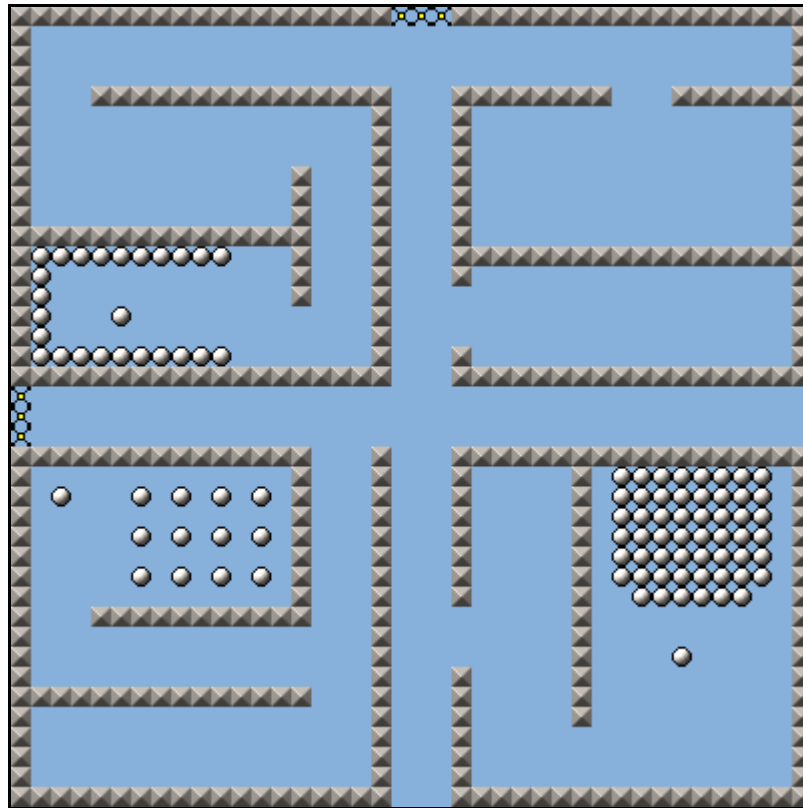


Fig 3.38 The Crowd Dynamics Experiment – Building 2

We ran the simulation again and obtained the path densities:

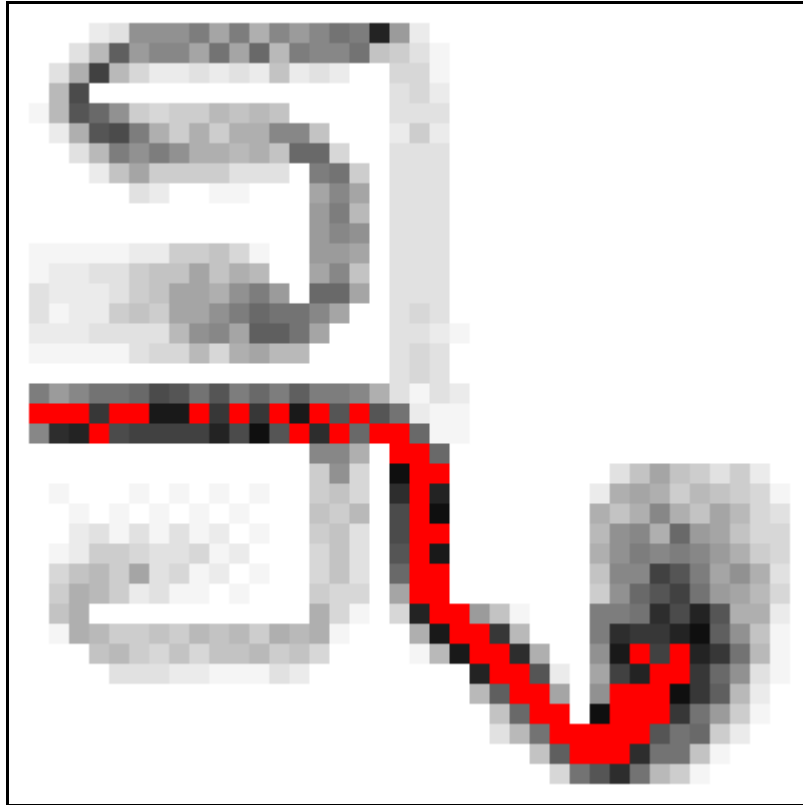


Fig 3.39 The Crowd Dynamics Experiment – Building 2, Frame 115, Path Densities

Although the traffic had been reduced, we had mostly diverted the problem elsewhere. It also took 115 turns for all the agents to exit.

Another alteration is made to the building in the form of an additional exit to the South which is illustrated in the next screen-shot:

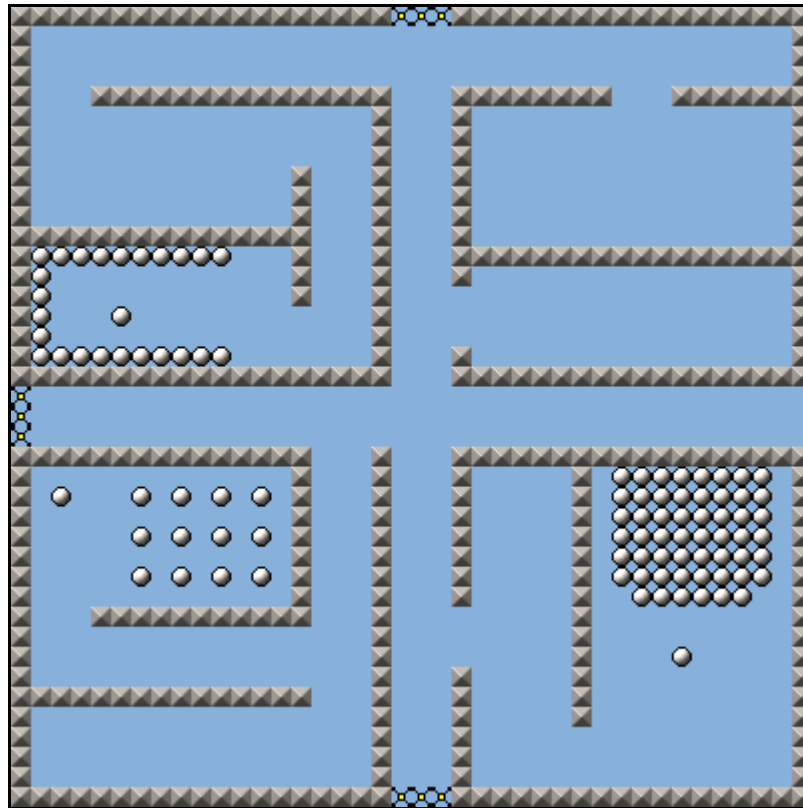


Fig 3.40 The Crowd Dynamics Experiment – Building 3

We ran the simulation again and extracted the path densities:

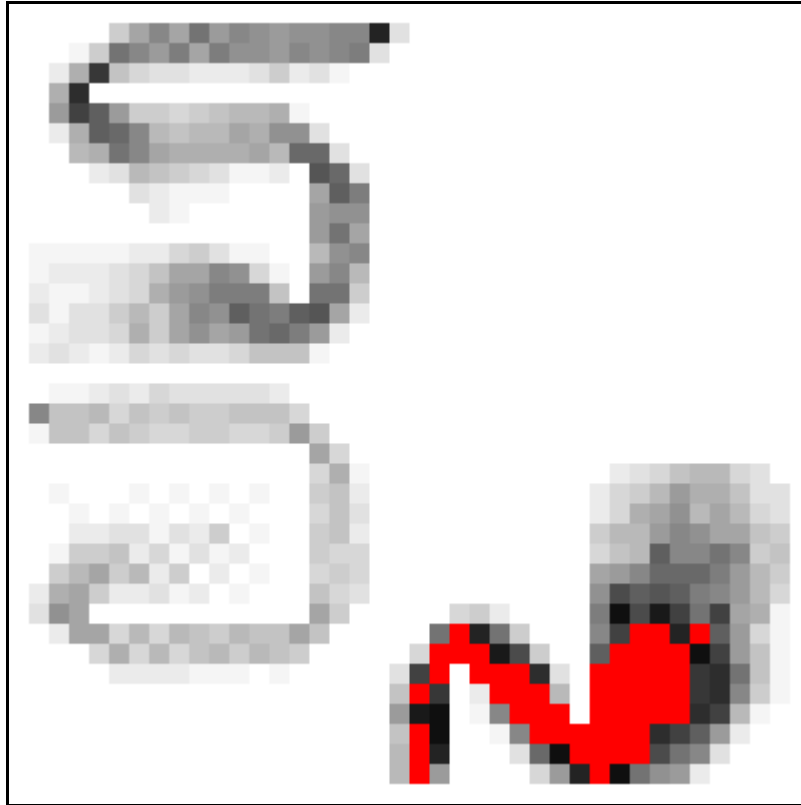


Fig 3.41 The Crowd Dynamics Experiment – Building 3, Frame 89, Path Densities

The agents took 89 turns to exit and we have significantly reduced the traffic problem. However, due to the amount of “people” that we expect to be present at any time in the Southeast room, a congestion problem still exists here.

Further alterations were made by moving the exit for this particular room so that it aligns with the main corridor exit to the West of this room. This is illustrated in the next screenshot:

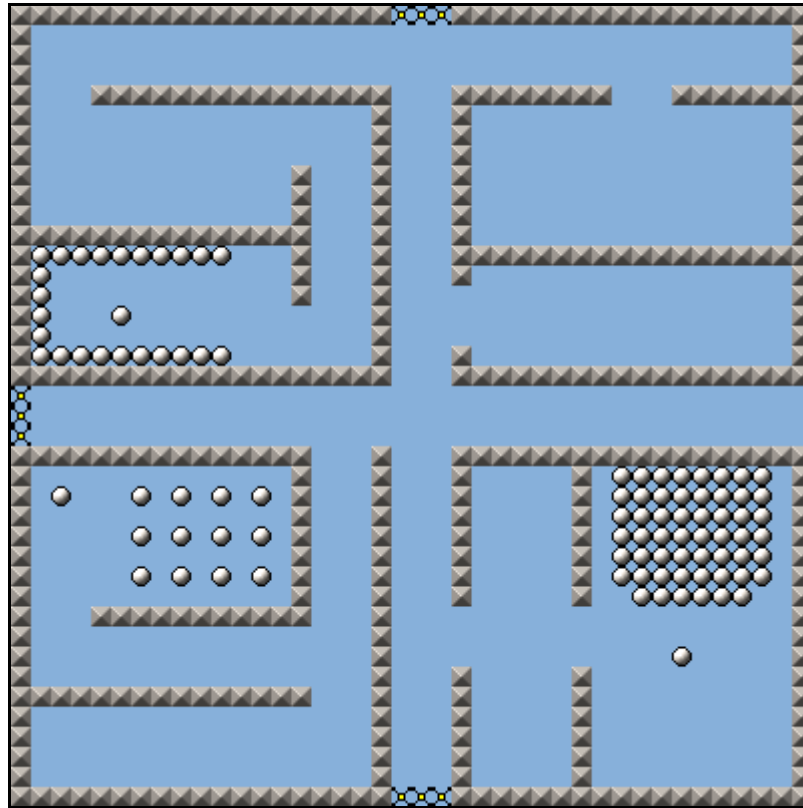


Fig 3.42 The Crowd Dynamics Experiment – Building 4

We ran the simulation again and extracted the path densities:

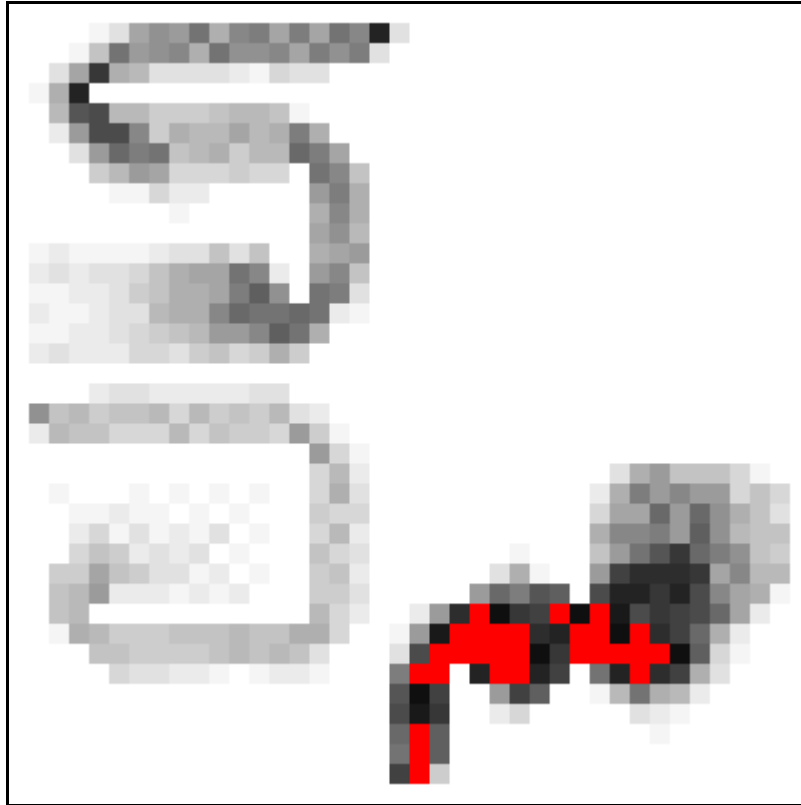


Fig 3.43 The Crowd Dynamics Experiment – Building 4, Frame 80, Path Densities

All agents had exited by frame 80, so we were heading towards a more acceptable exit time. Also, the traffic problem had been reduced by relocating the exit from the Southeast room.

However, we were not entirely satisfied so we created a new Northbound exit from this room and placed an additional building exit to the East. This is illustrated in the next screenshot:

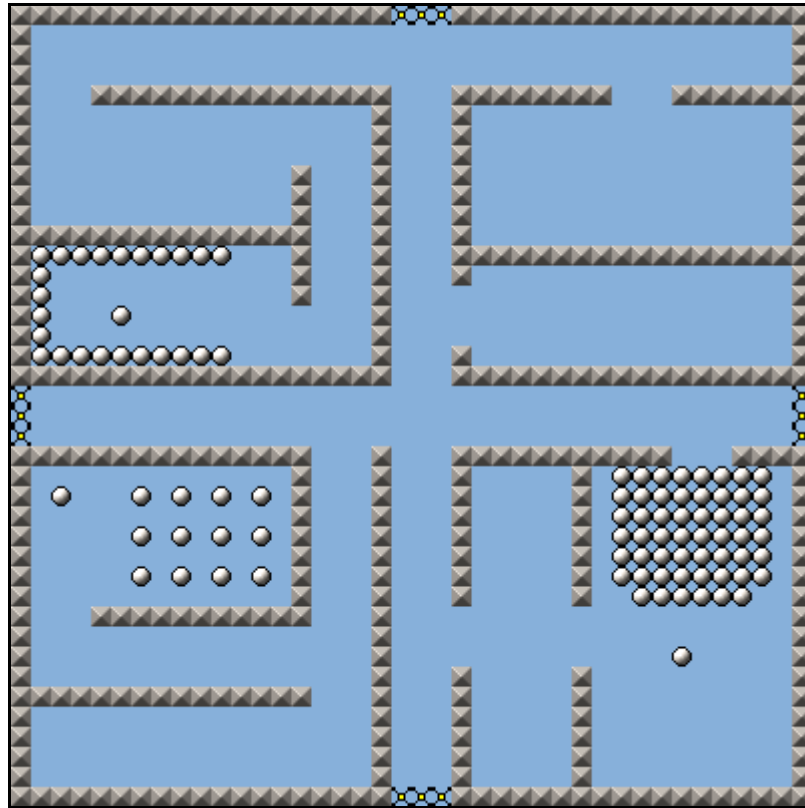


Fig 3.44 The Crowd Dynamics Experiment – Building 5

We ran the simulation again and extracted the path densities:

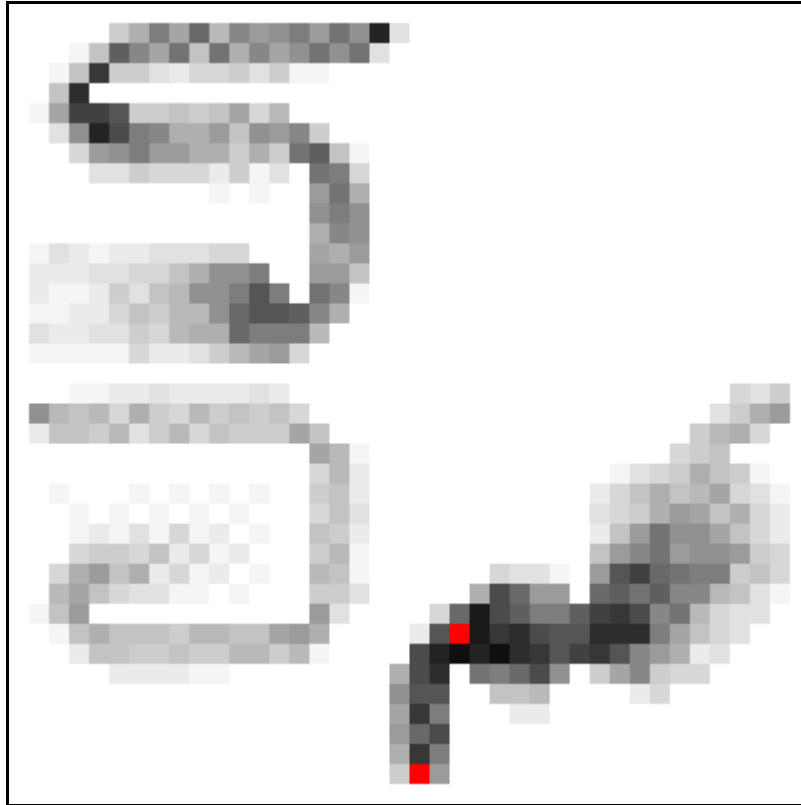


Fig 3.45 The Crowd Dynamics Experiment – Building 5, Frame 80, Path Densities

All agents had exited by frame 80. We have almost eliminated the traffic problem in the Southeast room and thus arrive at a satisfactory building layout, based on its intended “person load”.

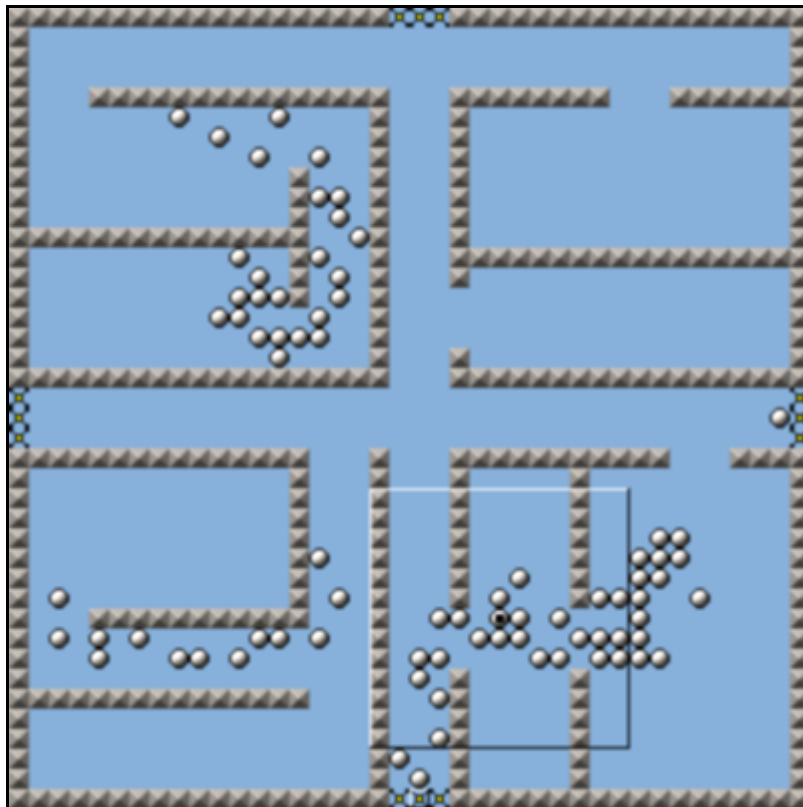
3.4.5 Conclusions

We conjured an imaginary problem of reducing traffic flow in a building. The scenario was heavily simplified, but we have been able to appreciate how agents may be used to simulate people within a confined domain. The emergent path structures of these agents revealed areas of heavy traffic and congestion. This information was useful in allowing us to make decisions about effective alterations to the building in order to improve safety.

It would be very difficult to estimate the traffic flow without the use of agents. These agents have autonomy, in that they make decisions based on their current knowledge of their world. This makes agents crudely similar to people and by modelling people as agents, with varying attributes, we could expect to gain a better understanding of their emergent behaviour as a group within a model domain.

CHAPTER 4

SYSTEM APPLICATIONS & LIMITATIONS



4.1 Introduction

In this chapter, we highlight the possible applications of a simulator system for emergent phenomena by suggesting a use in industry. We also address the fundamental limitations of our simulator system.

4.2 Applications of the System

In Section 3.4, we discussed a simple experiment that illustrated how a simulator could be used to model the flow of people through a building. Our simulator is far too simple to achieve the task of modelling human behaviour. However, we are able to appreciate the possible applications of a more advanced system.

Such a system would require the expert involvement of psychologists, engineers, architects and others. Ideally, the simulator would form a part of the overall software system for designing the building so that three-dimensional floor-plans could not only be used by the architects and construction engineers, but also by the simulator to model emergent phenomena within the building. Indeed, approaches similar to this have already been used in industry.

Modelling emergent phenomena need not only be restricted to buildings. For instance, modelling the movements of crew during simulated combat and emergency exercises onboard a battleship.

4.3 Limitations of the System

When we are faced with the task of modelling a domain, perhaps the most fundamental question that we must ask ourselves is at what level of detail should we finish describing things and arrive at an atomic symbolism. For instance, in our model, an object within the environment is represented as two numbers. The first number is a unique identification that marks what type of object is present. The second number represents how much of it is present. We say nothing else about the object but we cannot prove that more or less information is either better or worse. This is the problem of granularity and produced a particular side effect in our model that surfaced in Experiment 3. Due to the “physics” of our domain, the shortest route from one location to another is not necessarily a straight line. This prevents us from effectively carrying out experiments where straight lines are useful.

The following illustration displays a grid world with a black line drawn between two points. The line is straight and is also intuitively the shortest route between the points. However, we may only move from one square to another either vertically, horizontally or diagonally. The path cost for a horizontal or vertical move is 10 and the path cost for a diagonal move is 14. The distances vary due to the nature of Pythagoras's Theorem for right-angled triangles. We have drawn a possible route in green that would represent the shortest path from the bottom left to the top-right between the end-points of this line:

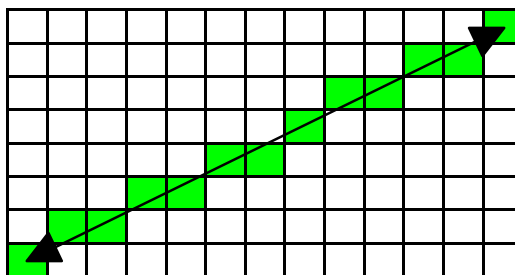


Fig 4.1 A Minimum-Cost Path Within a Grid

The shortest route is a total cost that must include 7 diagonal moves towards the upper-right and 5 horizontal moves towards the right, from the starting point. To the human observer, this line looks as straight as possible and fits our intuitions about the shortest path between points A and B, being along a straight line. However, due to the restricted movement directions of our domain, this is not the only combination of moves that yields the shortest path.

Any of the four red lines in the next illustration are also permissible, because they contain the same combination of moves and thus the same total path cost:

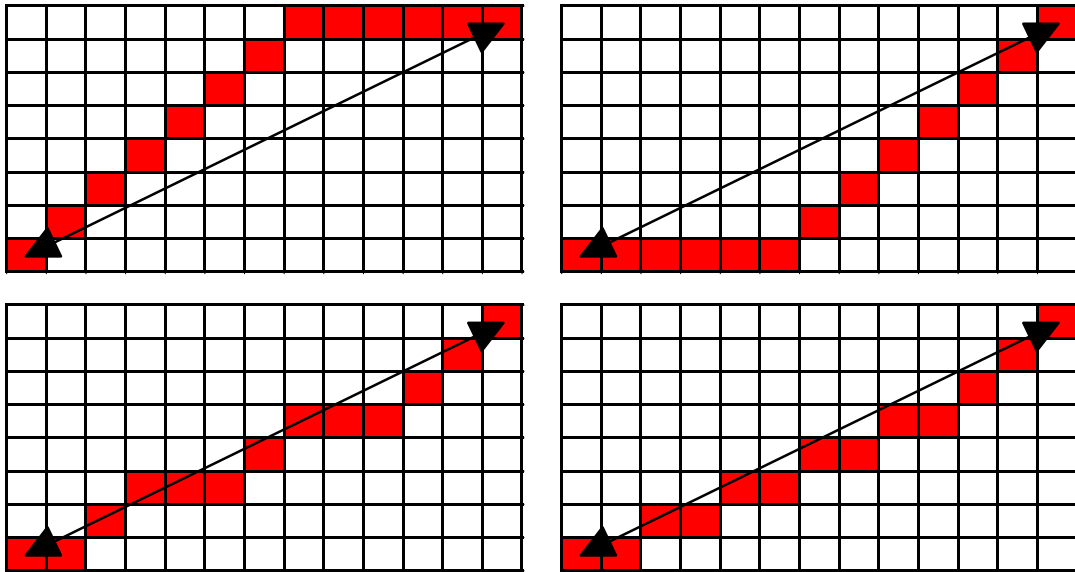


Fig 4.2 Further Minimum-Cost Path Examples within a Grid

There are many more permutations that yield the same minimum cost path. It turns out that the top two examples in Fig. 4.2 are the sort of paths produced by our Path-Searching Algorithm. Although we would like straight lines to represent the shortest path between any two points in our model, there are only certain conditions where they exist. Namely, points along the same horizontal, vertical or diagonal vector. We haven't made an error, it is just the scale at which we have built our model obeying its own laws and our intuitive beliefs failing to grasp the dynamics of the level of granularity and abstraction.

We can also encounter side effects that are concerned directly with the level of abstraction that we have attributed to something. Sometimes, our agents fall into deadlock situations when pursuing a target in their environment. Consider the following screen-shot of an initial environmental state that eventually leads to a deadlock situation:

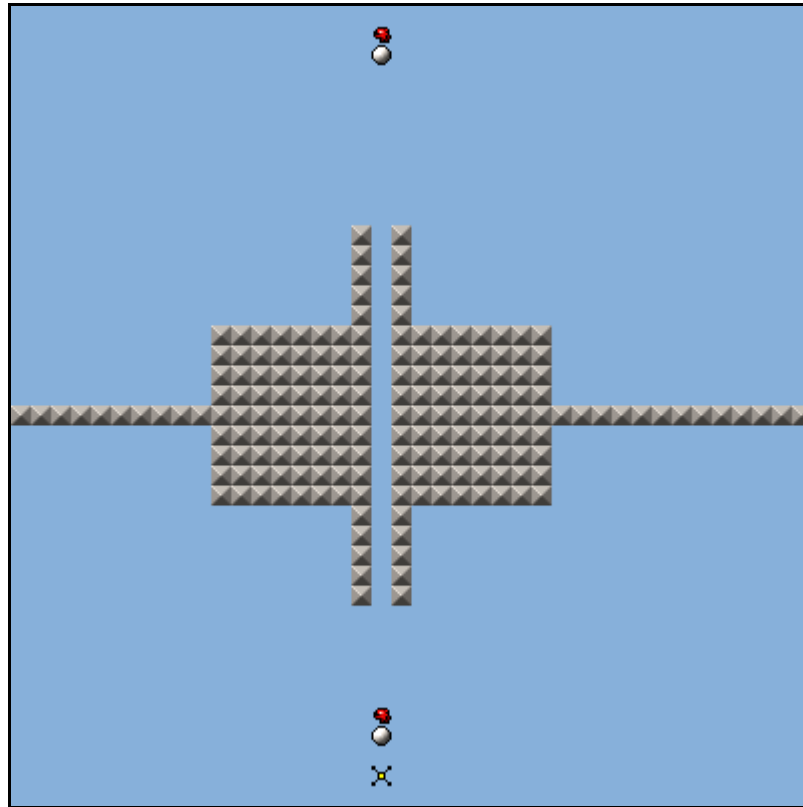


Fig 4.3 Mining Agents and Deadlock – Initial State

Both mining agents have a complete knowledge of their environment from the start but their visual ranges are small. The mineral values are so small that they are both mined immediately and fill each agent's backpack. The agent at the bottom deposits the contents of its backpack at the home base and heads up to collect the other mineral at the top. Meanwhile, the top agent heads down to the home base at the bottom to empty its backpack. The bottom agent doesn't know that the mineral at the top has already been mined because that region does not fall under its visual field. The top agent doesn't know that the bottom agent is heading its way and will block its route down to the home base:

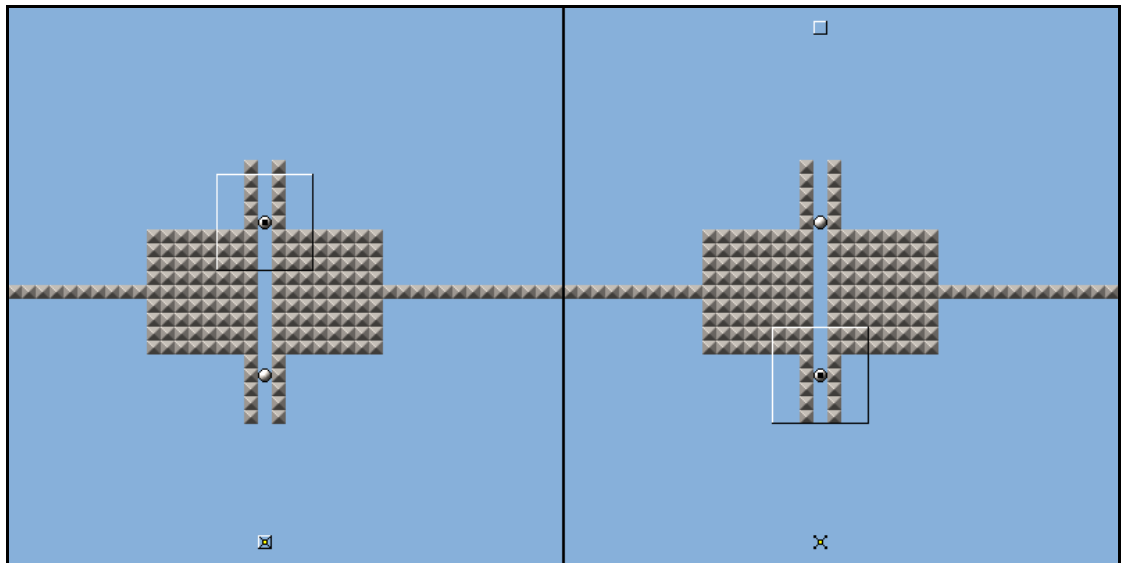


Fig 4.4 Mining Agents and Deadlock – Agents Seek Targets

Both agents encounter each other through their visual fields:

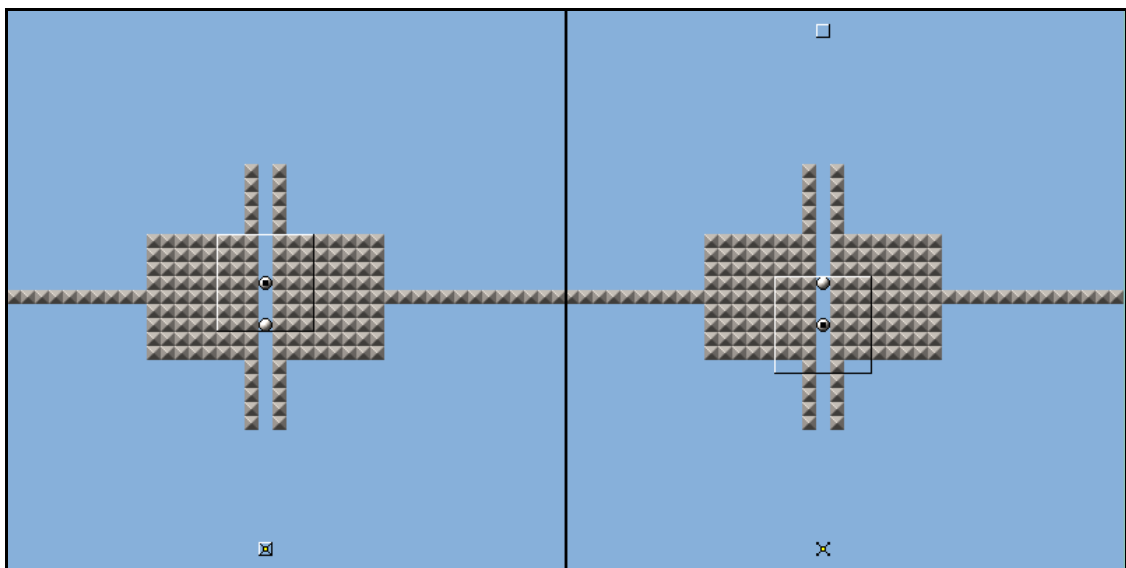


Fig 4.5 Mining Agents and Deadlock – Agents Encounter

They are unable to reach their targeted locations so they both head off in opposite directions. They wander to a random location that is reachable from their current positions:

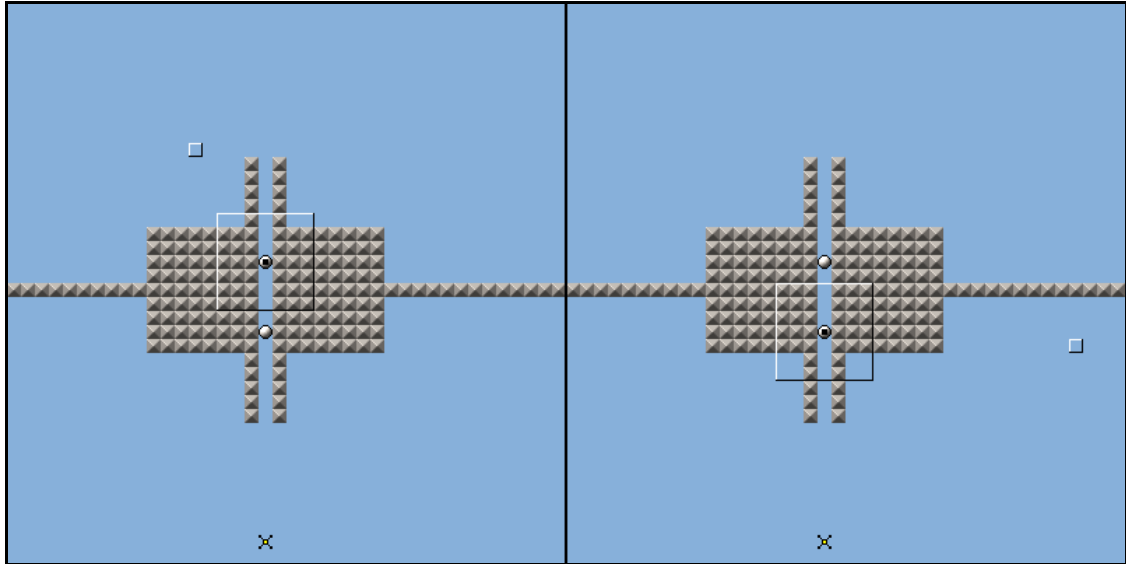


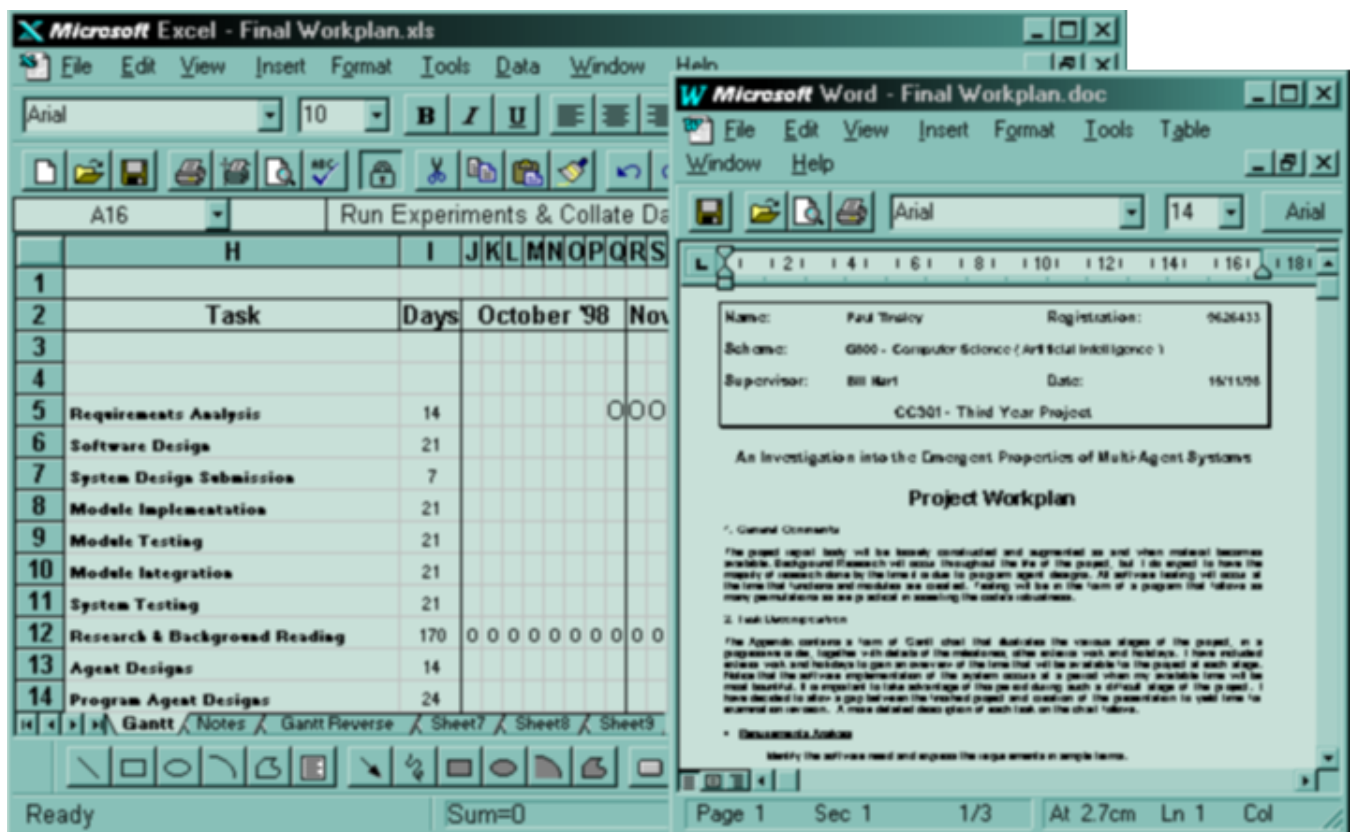
Fig 4.3 Mining Agents and Deadlock – Agents Wander Away

However, once these agents are outside each other's visual range, they "forget" about the existence of the agent that blocked their path. The rule that states the action that should be performed according to their current memory state is executed which results in them continually encountering each other, then moving away again to a random location. They become stuck in infinite deadlock.

There is no simple solution to this problem that we have been able to identify that does not involve either inter-agent communication or complicated planning. Fundamentally, the problem is that these agents are clones and lack any distinct individuality. Any anti-deadlock strategy that we give to an agent is also possessed by all other agents of the same type. We could perhaps give them a memory deterioration rate that is random for every agent, rather than make them "forget" all other agents that fall out of their visual field. However, deciding the scale of this random number in order to make it effective would depend on the dynamic and thus unique situation within the environment at that moment. How do we use an agent's knowledge of the environmental situation to generate such a number? This is the realm of agent planning that does not form an aspect of this project.

CHAPTER 5

PROJECT MANAGEMENT



5.1 Changes to the Plan

In this section, we reveal the major aspects of the project management plan that correspond with the creation of this report and discuss the changes made. The actual plan was submitted much earlier in this study and should be referred to in order to gain a full understanding of the specific tasks involved. The main goals were as follows:

- Design and implement a multi-agent simulation system
- Design and implement various software agents
- Run experiments using the simulator and agent designs
- Find appropriate research material

The project report was to be constructed and augmented as research and experiment material became available. The tasks in the original plan were arranged so that each stage lead to the next, with milestones set at specific points.

We maintained a diary throughout the most difficult stages of this project, in order to keep track of our progress. From this diary, we have selected the most salient events that either inspired us, proved very difficult or added extra workload that had not been factored into the original plan.

The Design Document

We had prototyped much of our core system design and wasted little time transferring our ideas onto paper. However, four assignments were running alongside our work on the design document and they were draining much of the available time. We had highlighted this external load in our plan and were ready for the extra effort. However, a week of illness made this task that much harder and we only finished just in time.

Research

We found the task of finding appropriate research material for this project far more labour intensive than we had imagined. Most of the time, the abstracts held misleading keywords or we had misunderstood the topic and had wasted effort. There were many abstracts that we never managed to find papers for. Although quite a lot of the papers that we found reported emergent phenomena, hardly any of them investigated the notion of emergence itself. We were dissatisfied with the quality of our research effort. There comes a time when you simply have to stop the research and get on with the rest of the project. We reached this point with few examples in our possession that reflected the effort invested. It has not prevented us from carrying out this study effectively, but we are certain that we have not been able to obtain full benefit from those persons who have gone before us.

Designing a Path-Searching Algorithm

We spent considerable effort in creating a fast searching technique. Our most profound moment of inspiration occurred on the 16th January, when we had finally eliminated our avenues of enquiry and the solution had shone through. We don't believe that we have

discovered anything new, but the personal gratification was most rewarding and is as equally significant as our failures and difficulties.

Adding a Graphical User Interface

Right from the beginning of this project, we were keen to develop an interface. As a strategic move, which is perhaps hazardous to admit, we did not include an interface in our design document as we did not know how to implement one properly in the programming language that we had chosen. It may also have proven too far a distraction away from the experiments that we had hoped to achieve. However, an excellent Spring term course in Windows Programming suddenly glued together all that we had learnt beforehand and allowed us to achieve an interface within a reasonably acceptable time. We took two full weeks out of our plan, around the time that we should have been developing different agent designs, to develop the interface. Various odd moments thereafter were spent adjusting and improving its behaviour. This was only possible due to the fact that we had spent almost the entire Christmas vacation getting the core system code finished. All we had to do was slide the code under the interface and connect up the relevant points. It was rather a big “gamble”, but we forced our way through and still found time to generate some interesting agent designs and experiments. In fact, most of the experiments we produced were more than we had originally planned.

5.2 The Project Overall

Despite our disappointments with research, we found that our original plan was well designed and allowed us to flow through the stages in a methodical and encouraging manner. We enjoyed this project tremendously and were surprised at how much we had achieved within such a relatively short period. Much has been learnt about our own personal strengths and weaknesses which has given us the foresight to consider with more confidence how we might tackle projects in the future.

Appendix – Volume 2

Technical Notes

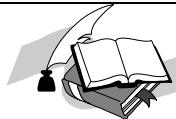
T.1 The Base Class Agent	152
T.2 The Grid Class Implementation	160
T.3 The Arbitration Process	161
T.4 The Path-Searching Algorithm	164
T.5 The Animate Main Interface	173
T.6 The Animate Map Editor	179
T.7 The Hunting Experiment Data	183

T.1 The Base Class Agent

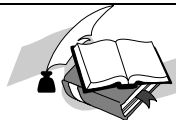
In this appendix we take a pragmatic approach in describing the attributes of the base class agent, which implicitly defines derived variants, using references to program data structures and functions. This will be of particular interest to students and experts in the field of Artificial Intelligence. First, we introduce the core data structures and illustrate what they represent. Then we discuss the core functionality, with particular attention as to how the production system is put into effect by the Administrator class, including arbitration.

The base class agent code may be found in Appendix C.4. Derived agent classes run from Appendices C.5 to C.9. Any references to actual program data structures and functions

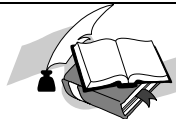
occur in brackets and correspond to definitions in the base class agent and derived variants. Definitions of data structures with the prefix “ag_” may be found in the base class agent code whereas “ad_” refers to the Administrator class code, Appendix C.11. Words in upper case refer to the objects and actions enumerated in the header file for the Grid class code, Appendix C.1. These atoms transcend the entire simulation system.



C.4 - The Agent Class - Page 231



C.11 - The Administrator Class - Page 323



C.1 - The Grid Class - Page 192

The base class agent is fundamentally:

- A simple reflex agent with internal state, similar to that illustrated by Russell & Norvig (1995)
- Uses an antecedent / consequent production system that activates from working memory. There are no special rule priority implementations such as refractoriness or specificity. All rules have a natural ordering from beginning to end of the rule set
- Is primarily geared towards searching for minimum-cost paths and thus inhabits a dynamic pseudo-environment that typically involves extensive “travel”

- Designed to operate within a turn-based simulation system for which it generates symbolic action objects containing atomic action tokens and coordinates that are arbitrated to resolve possible agent conflicts
- “Self-oriented,” in that their production rules reflect a hierarchy of goals that only “satisfy” the agent and nothing else
- Non-communicative with other agents
- Maintain a subjective view of their domain through the use of memory

All derived agent classes adhere to the same theme. The “internal state” that we refer to is in the form of targeting data and memory. However, these agents do not have explicit rules that calculate the outcome of actions as a means of future prediction. Most rules generate a target location that may be used by the path-searching algorithm to generate a movement action object (ag_intent) that gets the agent closer to the target object atom. Targets persist in memory until a production rule, percept or effector overwrites its content. Targeting consists of four data structures:

- | | | | |
|------------------|---------|---|--------------------------------------|
| 1. ag_target | boolean | - | true if target has been set |
| 2. ag_target_obj | integer | - | symbolic atom representing an object |
| 3. ag_target_x | integer | - | the target x-coordinate |
| 4. ag_target_y | integer | - | the target y-coordinate |

Actions that the derived agents may perform are divided into two classes:

Class “C_INHABIT” *inhabit a new location, not always the agent (DROP)*

MOVE JUMP DROP

Class “C_INTERACT” *interact with an adjacent location*

GRAB POUNCE

There also exists an action atom “NOTHING” that is used to either prevent the agent from performing an action, should it “lose” in arbitration, or purely to signal that no action will be performed in this turn so it may be ignored by arbitration.

The data structure used to record information received from percepts (ag_m) is in the form of a two-dimensional grid that has the same dimensions as that of the pseudo-environment (ag_env). This provides an injective or one-to-one function between memory and the pseudo-environment. Any symbolic atoms measured by percepts are simply copied to memory at their relative coordinates. This greatly simplifies the task of providing an associative spatial relationship between objects recorded by percepts in memory.

An additional grid is used to record the outcome of path-searching (ag_s). This again has the same dimensions as the pseudo-environment and is tightly connected to the contents of memory (ag_m). Before a search is executed, every symbolic atom in memory is filtered into the search memory by the following procedure:

1. Set all search memory elements to the value 0
2. For every element in memory, if it is not a SPACE, set the corresponding element in search memory to the value –1, unless it is the agent’s current location.

Thus, everything is an obstacle unless it is a SPACE atom or the agent. A quite profound simplification. It is important to not reflect the agent's location as impenetrable as this will yield poor path opportunities if this agent is within a tight corridor, such as a one square wide channel.

There are two forms of path searching that utilise the same algorithmic process. The first form enumerates all possible paths in search memory to the goal location. The second form will enumerate all paths until the current location of the agent is reached, whereupon the algorithm terminates. The first form is useful where it is necessary to compare all atoms of a certain type within memory in order to obtain the closest atom to the agent's current location. The second form is useful for enumerating a path to a target location, where all paths are not desired. Additionally, the first form is useful in determining whether a location is accessible to the agent or the agent's location is accessible from some other location. This is because the result of searching loads elements in the search memory with values that represent a path cost to the goal location. If the path cost is 0 at a particular location in search memory, then no path exists. In order to steer the agent towards the goal location, all it need do is select the adjacent location in search memory to its current location in memory that has the lowest path cost. Typically, due to the dynamic properties of the pseudo-environment, a new search is conducted in each turn to reflect any changes in memory due to the invocation of percepts. Thus, the path-searching algorithm ideally needs to be fast, so that we do not have to wait too long for each simulation turn to finish. There is an option in the simulator to run an animated scenario in real-time and it would therefore be rather disappointing to an observer if the agents took seconds or even minutes to perform their actions in each turn. A special searching technique has been developed as a unique solution to our path-finding problem that is discussed in Appendix T.4.



This technique has allowed us to perform up to 100 partially enumerated searches per second in a 40 by 40 pseudo-environment, during real-time bitmap-animated simulation, with an obstacle density of roughly 25%, using 100 Crowd Agents on our 200 MHz 80586 notebook computer under the Windows 95/98™ interface messaging system.

We now take a look at an example simulation turn and observe the mechanisms that generate, arbitrate and execute an agent's action.

STEP 1

The Administrator invokes every agent's production system (virtual void think()). This will typically have the following effect:

- a function (void expire_moving_objects()) removes all atoms representing other agents that have the capability of movement from memory (ag_m)*
- the percept function (void look()) is fired that copies pseudo-environmental data (ag_env) into memory (ag_m) based on the visual field (ag_v_range) relative to the agent's current location (ag_x and ag_y).
- the production system is interrogated and generates an action within an Intent class object (ag_intent)

** If an agent does not “forget” the existence of moving objects, it may never again explore a region of the pseudo-environment if such an object blocked access to that region at the time percepts “observed” that region, unless it “chances” upon that region again.*

Code for the Intent class may be found in Appendix C.3.



C.3 - The Intent Class - Page 220

STEP 2

The Administrator then transfers the list of agents’ Intents (ad_intents) to the Arbitrator (ad_arb). The Arbitrator maps all action atoms in each Intent object to one of the action classes, either C_INHABIT or C_INTERACT. The action objects are then arbitrated and if any matching action classes occur at the same location, a “winner” is selected, either randomly or by priority, with all “losers” having their actions cancelled by replacing the action atom in the intent object with the action atom NOTHING.

A more enlightening example of arbitration may be found in Appendix T.3. Code for the Arbitrator class may be found in Appendix C.10.



T.3 - The Arbitration Process - Page 161



C.10 - The Arbitrator Class - Page 314

STEP 3

If any special rules have been defined in the Administrator to handle actions that involve agents affecting other agents, the rule is fired here and any adjustments are made to the intent objects, or a function is fired in the agent that is the successful result of the rule. For instance, forcing a Prey agent to change its state to a CARCASS object as a result of the successful arbitration and scoring of the action atom POUNCE from an adjacent Predator agent.

STEP 4

The Administrator then copies back the arbitrated intent objects to each agent (`ag_intent`). It then executes every agent's action execution function (`virtual void execute_action()`). This contains a switch statement that fires functions within each agent on the basis of the action atom and coordinates contained in their intent object (`ag_intent`), which has now been arbitrated and overwritten by the Administrator. Agents directly interfere with the pseudo-environment through their effectors (`ag_env`).

The art of making these agents perform something “useful” very much hinges upon the design of their production systems. The structure of the base class agent is general so that derived types may be created easily. The structure of the simulation process is also general enough to allow changes to be effected easily. There are some side effects of using such a simplistic model that are discussed in section 4.3.

T.2 The Grid Class Implementation

In this class, we have developed a simple mapping that allows a cartesian coordinate pair to be mapped to a unique number within the boundaries of a contiguous list. This mapping is defined as follows:

Let E be the unique number or element index on the list

Let X be the x-coordinate in the cartesian pair

Let Y be the y-coordinate in the cartesian pair

Let D be the dimension of the two-dimensional grid

then,

$$E = X + (Y \times D)$$

By providing a lookup table for the (Y x D) portion of the function, we reduce the processing required to generate the mapping index number. As vectors represent contiguous lengths of memory of the objects they contain, the mapping provides a sort of address offset to fetch or set the appropriate vector member. We can use array notation “[]” to address vectors, thus we arrive at an extremely fast method of obtaining data from the vector within the realms of object-orientation. This was of great significance to this project, as we required rapid grid access due to its intensive use.

As our confidence grew with this class, we added faster access and retrieval functions that did not perform any coordinate or value checks. These are the functions that have the suffix “_f”. As many accesses to these grid instances were via controlled iterative loops, the

possible coordinate and value permutations were confined and, once testing was complete, it became safe to use the faster function forms. If for any reason an error was occurring somewhere, the original function could be replaced swiftly.

We have also made use of many STL functions. Most of our research into STL functionality was obtained from Breymann (1998). Examples of using predicates within STL containers were found in Stroustrup (1997). The persistent storage method of grid data is not efficient, but has the advantage of being editable within any standard text editor.

T.3 The Arbitration Process

Here, we provide an example scenario involving arbitration. Additionally, we provide details of the original arbitration atoms and changes that were made during the life-cycle in order to improve the arbitration method.

Example

Suppose there exists agents A, B, C, D and E. There also exist locations x, y and z.

The agents wish to perform the following actions at some point in time:

- A wants to grab from location x and has a priority of 4.
- B wants to grab from location x and has a priority of 0.
- C wants to grab from location x and has a priority of 0.

- D wants to move to location y and has a priority of 1.
- E wants to move to location y and has a priority of 1.

The Arbitrator has already matched these agents' actions against the actions intended and executed during the last turn, incrementing priorities for each agent where the same action is being repeated at the same location.

There is a conflict between agents A, B and C as they wish to perform the same action at the same location. Also, there is a conflict between agents D and E. A fair method of deciding who gets to execute an action is required.

A has a higher priority than B and C, so A should be allowed to act, whereas B and C should have their actions cancelled. However, B and C should have their priorities incremented by 1 should they wish to perform the same action again in the next turn, because they failed to execute their action in this turn. A should have its priority reset to zero because it will successfully execute its action.

The Arbitrator should select a winner at random from D and E because their priorities match, setting the winner's priority to zero and incrementing the loser's priority by 1 in the next turn, should the action and location be the same.

Let's say that from D and E, D wins and E loses.

Now suppose in the next turn, the agents' intended actions are as follows:

- A wants to grab from location x and has a priority of 1.
- B wants to grab from location x and has a priority of 2.

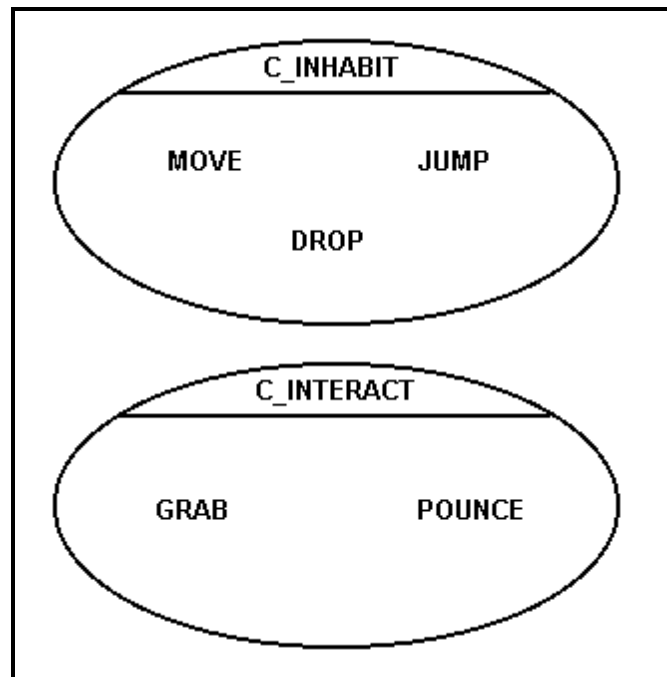
- C wants to grab from location x and has a priority of 2.
- D wants to move to location z and has a priority of 0.
- E wants to move to location y and has a priority of 2.

Because A wants to perform the same action as in the last turn, the Arbitrator increments its priority before arbitration. Likewise, B and C also want to perform the same action again so their priorities have been incremented. D wants to move to a new location so its priority is reset and therefore remains at zero. E still wants to move to location y, so its priority is incremented.

The Arbitrator should fail A as its priority is the lowest of A, B and C who all want to perform the same action at the same location. The Arbitrator must select a random winner from B and C because they have the highest priorities and the priorities are equal. D has no conflict, so the Arbitrator sets its priority to zero and allows it to move. E finally gets to move to location y, because it is the only one wishing to perform that action at that location and has its priority reset to zero.

The Original Arbitration Atoms

Originally, we designed the Arbitrator to match an action atom and a coordinate mapping. This would typically yield something rather like (MOVE,234). It became apparent during the experimental phases of this project that classes of action were more appropriate. For instance, the action of moving to a location is fundamentally the same as dropping a new object at that location. Both want to acquire the location, but only one may succeed. Therefore, upon completion of the system, the following classes crystallised to encapsulate different clusters of actions:



An intention supplied to the Arbitrator has its action atom matched against the relevant class category. The arbitration function then arbitrates the intentions using these categories and the coordinate mapping. After arbitration, any action that failed is altered to the fail atom, otherwise the original action atom remains as it was originally.

T.4 The Path-Searching Algorithm

Effective path planning is at the very core of every one of our agents. We felt that our agents should always utilise optimal paths to their goal locations. Much of the functionality would rely upon path building, so we were challenged with finding or creating an algorithm that could cater for our needs.

We had developed quite a demanding “wish-list” for our search algorithm:

- rapid computation time - to cope with a large number of searches within each simulation turn on our notebook machine
- range - every search should enumerate all possible paths to the goal
- simplicity - easy to obtain the route once the search was complete
- access selection - to be able to query a set of objects in memory and select the closest by only searching once
- availability - to be able to query a location in memory and know whether it is accessible from where the agent is currently located

Immediately our minds turned towards Dijkstra's Algorithm, which will compute every path to a set location. Aho & Ullman (1995) describe this algorithm and many others like it. However, the implementations that were described in this book were rather time consuming for our specific purposes and did not satisfy all of our needs.

Taking together all that we had researched and learnt during this degree, we developed the following method that is very similar in concept to Dijkstra's Algorithm. The following C++ code was used in this project for enumerating paths. We provide a pseudo-code description of its features after the code.

```
void Agent::execute_search_flood(short x, short y)
{
    short value = 10;           // path costs
    short c1,c2;                // allowable move counters
    short cx,cy;                // relative positioning
    short put_value;            // paths costs
    short temp;                 // value comparison
    short count = 0;            // loop counter
    bool hypotenuse = true;     // a toggle that represents a diagonal move
    triple t(x,y,value);        // for recording algorithm progress
    ag_search.clear();           // clear the search list
    ag_search.push_back(t);      // set first value
    ag_s.put_f(x,y,value);       // record first value in search grid
```

```

// whilst we are not at the end of search list
while(count < ag_search.size())

{

    // get this value
    ag_search[count].get(x,y,value);

    // look at neighbours
    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)

    {

        c2 = c1 + 1;
        cx = x + moves[c1];           // make new position
        cy = y + moves[c2];           // make new position
        hypotenuse = !hypotenuse;     // toggle hypotenuse

        // if this coordinate is valid
        if(ag_s.check(cx,cy))

        {

            // if this coordinate is not blocked
            if((temp = ag_s.get_f(cx,cy)) != -1)

            {

                put_value = value;      // make path value

                // if diagonal transition
                if(hypotenuse)

                {

                    // note lazy evaluation so order of statements
                    // is crucial due to assign and test
                    // if new path cost < this value or a space
                    if(((put_value += 14) < temp) || temp == SPACE)

                    {

                        // record new value and push onto back of list
                        ag_s.put_f(cx,cy,put_value);
                        t.put(cx,cy,put_value);
                        ag_search.push_back(t);

                    }

                }

            }

            else

            {

                if(((put_value += 10) < temp) || temp == SPACE)

                {

                    ag_s.put_f(cx,cy,put_value);
                    t.put(cx,cy,put_value);
                    ag_search.push_back(t);
                }

            }

        }

    }

}

```

```

        }
    }
}

count++;
};
}

```

The following pseudo-code offers a less knowledge-dependant explanation:

LET there be a two-dimensional grid G.
 LET there be a vector of triples V.
 LET there be a path cost P.
 LET there be a counter C.
 LET there be coordinates X and Y.
 LET neighbours of X and Y be X' and Y' respectively.
 LET the notation (a,b,c) be (x-coord,y-coord,value at x-coord and y-coord)

PREPARE G so that all spaces are 0 and everything else is -1

C is 0.
 P is 10.
 V is empty but has been allocated ample size
 X is the goal x-coordinate
 Y is the goal y-coordinate

SET (X,Y,P) in G, push (X,Y,P) onto the back of V.

WHILE C is less than the size of V

GET the value (X,Y,P) from V at index position C.

FOR ALL adjacent neighbours (X',Y') to (X,Y,P)

IF (X',Y') are valid coordinates for G

IF ? of (X',Y',?) in G is not an obstacle

P' is P

IF X',Y' is a diagonal transition from X,Y

IF ? of (X',Y',?) in G > (P' is P' + 14) OR ? is a space

SET ? of (X',Y',?) in G to P'

```

        PUSH (X',Y',P') onto the back of V
    ENDIF
ELSE
    IF ? of (X',Y',?) in G > (P' is P' + 10) OR ? is a space
        SET ? of (X',Y',?) in G to P'
        PUSH (X',Y',P') onto the back of V
    ENDIF
ENDIF
ENDIF
ENDIF
END FOR
C is C + 1
EXTENT OF WHILE

```

We have chosen the value 14 as a basic representation of the hypotenuse distance in a right-angled triangle, using Pythagoras's Theorem, with both other sides set to 10. This provides a more accurate distance measurement for diagonal transitions between locations, which is ideal for our purposes but by no means perfect.

The most significant benefits of this algorithm is that there is no back-tracking, as the algorithm is always "seeking" forward through the vector. There are no trees to trickle down in order to select paths once the search is complete and the cost of pushing elements onto the back of a vector is fast, i.e. $O(1)$, as long as the allocated size of the vector has enough capacity. This implementation provides a very rapid process that fills up a two-dimensional grid with "special" values that represent a path cost to the goal location. All accesses to a grid are $O(1)$, because we are using vector position lookup tables and pseudo-address-

arithmetic to obtain the correct mapping from cartesian coordinate pairs to an element location in the vector.

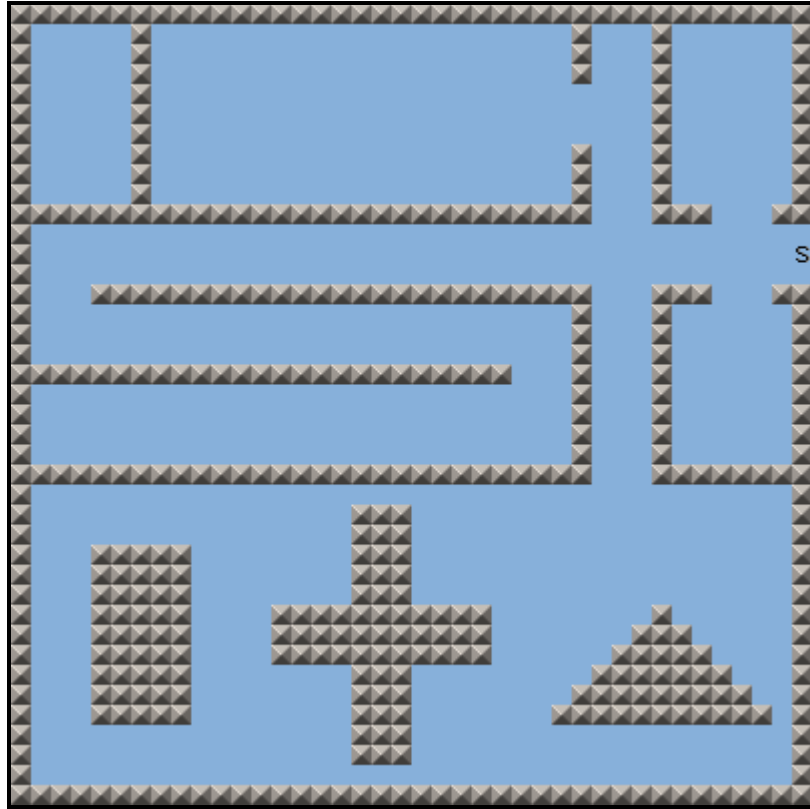
Altogether, we obtain the following useful features:

1. Fast computation time - relative to our needs
2. A search grid that maps directly against agent memory, containing all the path costs to a goal location. All we need do at any moment is pick the adjacent location to the agent's current location with the lowest path value in order to steer the agent down a minimum cost path to the goal location.
3. Any location in the grid with a value of 0 means that the location cannot be reached.
4. Any location in the grid with a value of -1 is an obstacle.
5. The ability to perform partial searches by cancelling the search as soon as the algorithm encounters the agent's current location, thus saving some computation time.
6. The ability to query a set of objects in memory, obtain the lowest path value that is adjacent to each one, thus providing the closest member of that set according to the agent's current location. We do this by setting the goal in the search as the agent's current location.
7. The algorithm is guaranteed to terminate - there are no heuristics.
8. Real search time is a measure of how many obstacles are present in the grid. The more obstacles there are, the faster the search time.
9. Storage of the paths is extremely simple - we do not have to traverse complicated tree structures and waste computation time by transferring countless addresses through the CPU.

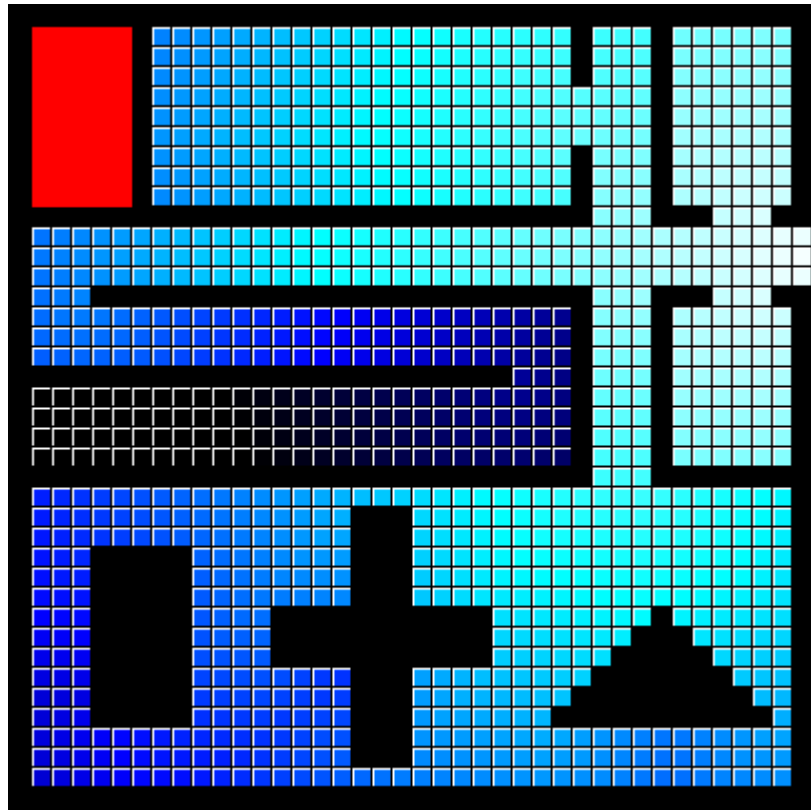
We have not carried out any formal time analysis on this procedure, but have found it to be very rapid and feel that its merits are worth illustrating due to its importance in this project and the effort that was invested in its creation.

A Visual Search Example

We have provided some special functionality within the map editing feature of the simulation system to demonstrate path searching. The following screen-shot from the Map Editor is of an environment where obstacles have been arranged so as to produce rooms and shapes:

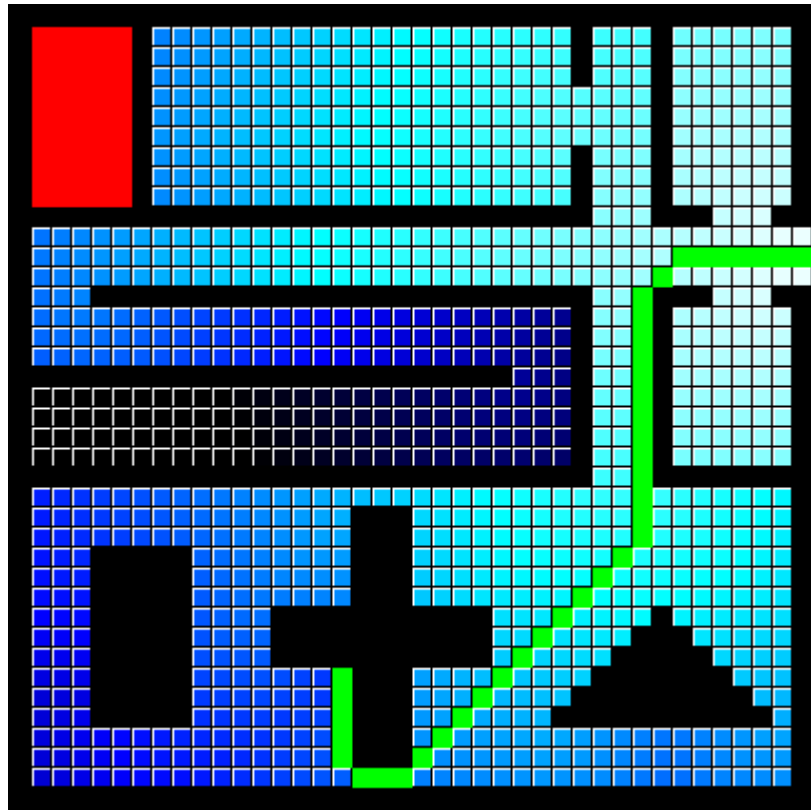


The “S” marked on this map indicates the entrance to this strange building. We are able to invoke the path algorithm and visually appreciate all of the paths from any position in the map by locating the mouse cursor over a particular location and clicking the left mouse button. For this illustration, we clicked at location “S” and produced the next screen-shot. A form of “colour temperature” is provided that runs from white to black through green and blue. Therefore, locations that are closer are lighter and locations that are further away are darker. The colours enclosed in three-dimensional rectangles represent the path costs from every location in the map that can be reached from the location that we clicked at. The red colour indicates an inaccessible location and the black colour represents impenetrable obstacles.



This is not only visually appealing and fun but also demonstrates how the path-planning algorithm works. We have now enumerated every possible path in the map to the location “S”. If this map were the contents of an agent’s memory and the agent was positioned at “S”, according to its subjective viewpoint, it may query any location in its memory and “know” how far away it is. It would also “know” that one region is inaccessible.

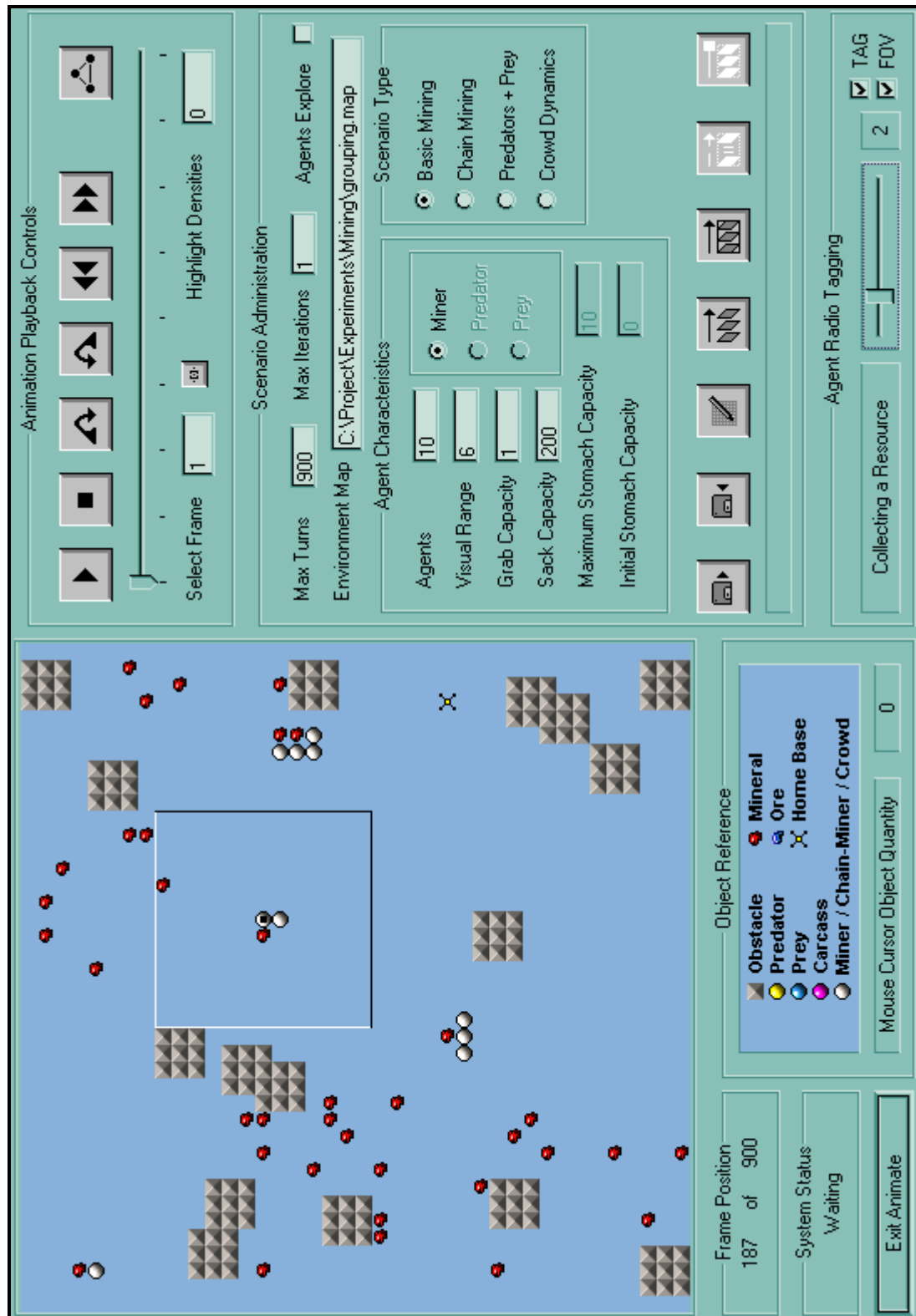
In order to follow a path, all an agent need do is pick the adjacent square with the lowest path cost in each turn. We are also able to demonstrate particular paths using the Map Editor and this is illustrated in the next screen-shot. In this example, we have set the target location under the left arm of the cross in the map and the current location as the entrance marked by “S”. The system simulates a turn-based movement and draws a green line representing the path that would be taken by an agent if the environment remained constant until the target location was reached.



These unusual features of the Map Editor have allowed us to inspect environments and appreciate the distances and paths involved between certain objects. If a certain phenomenon arose out of an experiment, we were able to use the Map Editor to query the paths that existed and begin to understand the reasons why such a phenomenon occurred.

T.5 The Animate Main Interface

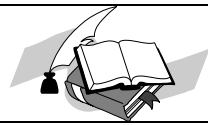
Here, we illustrate in detail the features of the simulation system called “Animate”.



Animation Window

The top-left portion of the screen contains the Animation Window. This is where the environment is animated using a series of memory-mapped bitmaps and a special vector-

based script containing state data about every object in the environment. Code details on this script mechanism may be found in Appendix C.12.



C.12 - The Special Data Structures - Page 352

Bitmap animation is not featured in any of our degree courses and we had to carry out some quite extensive research into this topic. (Kruglinski, 1997) and (Prosise, 1997) both proved highly valuable in getting this feature to work correctly and efficiently.

Object Reference

This is simply a key that provides a legend for the objects that may appear in the Animation Window. The “Mouse Cursor Object Quantity” feature will output the quantity associated with an agent or object that the user places the mouse cursor over in the Animation Window.

Frame Position

This feature keeps a record of the total number of frames and the current frame being viewed or processed within a simulation or playback mode. This was useful for recording the frame positions where interesting experimental observations occurred.

System Status

This is a remnant of the testing process. We used this to output messages during testing. Rather than remove it once testing was complete, we decided to alter its purpose to that of informing the user about fundamental system processes that are running.

Animation Playback Controls

These controls are similar to those that one may find on a video cassette player. They allow a scenario script to be controlled in various ways:



Play an animation script from the current frame position



Stop animation playback



Step forward to the next animation frame



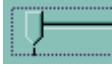
Step backward to the previous animation frame



Jump to the first animation frame



Jump to the last animation frame



Set the animation playback speed or simulation speed to a value between approx. 50 milliseconds and 3 seconds



Select a frame from the frame set



Jump to a frame selected by the edit box "Select Frame"



Generate a Trail or Path Density analysis from frame 1 to the frame position indicated by "Select Frame"



If a Path Density Analysis, highlight all densities equal to or exceeding this value in red in the Animation Window

The Trail and Path Density algorithms are discussed in section 2.4.2.

Scenario Administration

This is the area where experiments may be set up and their parameters adjusted:



Set the maximum number of turns for each scenario simulation



Set the number of times to repeat the scenario

Agents Explore ☐

Set whether agents have to explore their environment

Environment Map

Define the source of the environment map being used

Scenario Type and Agent Characteristics are quite self-explanatory. The following buttons control the execution of a simulation:



Load an animation script



Save an animation script



Invoke the Map Editor



Run an experiment without real-time animation



Run an experiment with real-time animation



Pause execution of an experiment



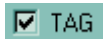
Cancel execution of an experiment

Agent Radio Tagging

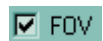
These tools enable specific agents to be monitored during an experiment or scenario script playback:



Select an agent to tag



Activate agent tagging

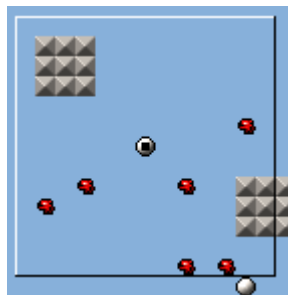


If tagging, show agent's field of view



If tagging, show tagged agent's current action

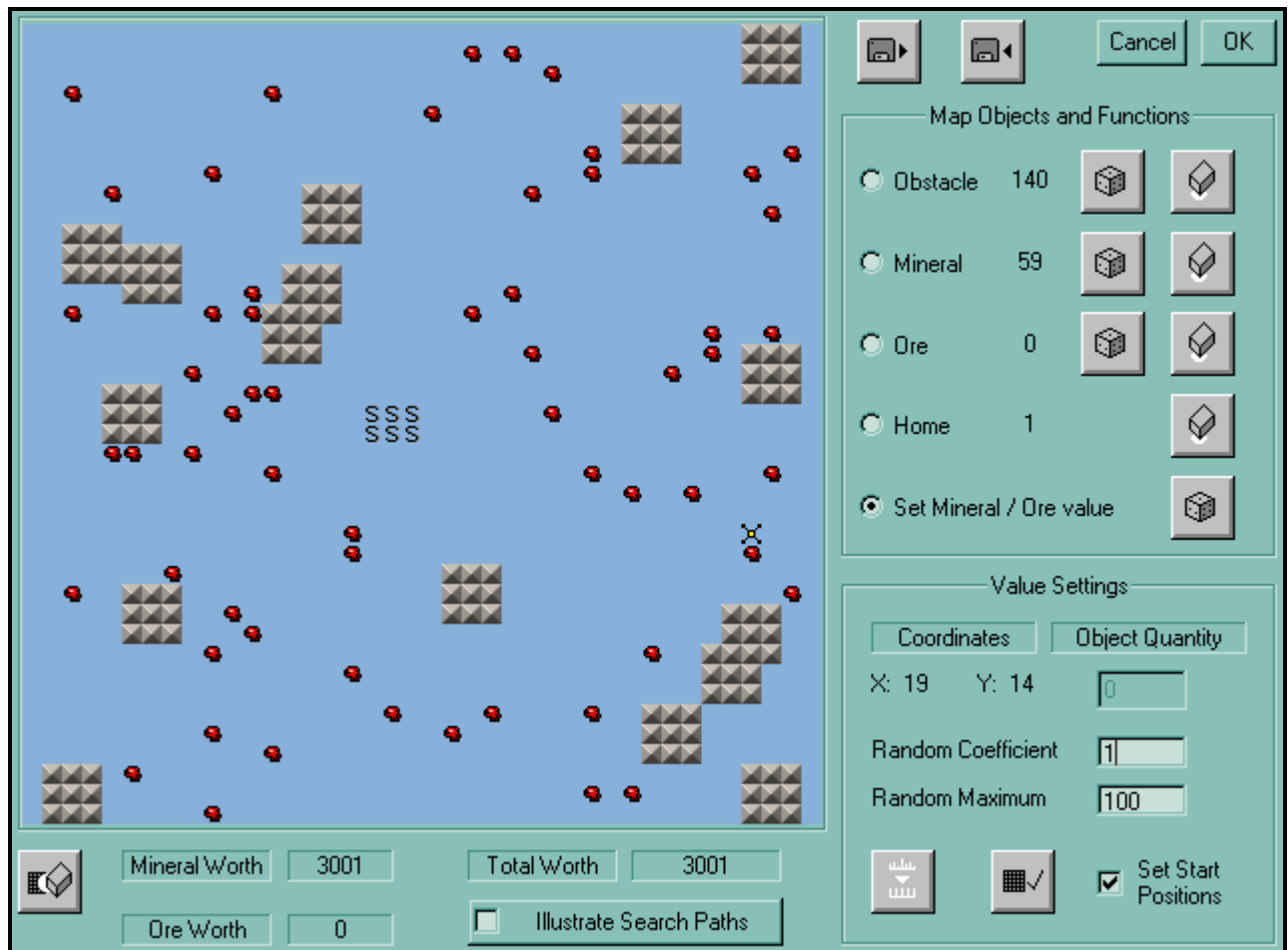
The effect of tagging an agent places a black box inside the agent in the Animation Window. This allows the user to visually track the tagged agent. If “FOV” is checked, a three-dimensional rectangle is drawn over the tagged agent in the Animation Window to represent its field of view:



Tagging is also discussed in section 2.4.2.

T.6 The Animate Map Editor

Here, we illustrate in detail the features of the Map Editor within the “Animate” simulation system. First, we take a look at the editor window and then we discuss each feature.



Drawing Window

The area with the blue background provides a “canvass” upon which various map objects and object quantities may be set by using the mouse cursor.

Map Objects and Functions

The various radio buttons allow certain objects to be placed onto the map by using the mouse. The system records a count of how many objects of that type are present in the environment alongside each radio selection. Additionally, buttons adjacent to each radio selection provide the following functionality:



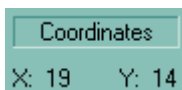
Randomise the adjacent category of object over the map using the current randomisation settings



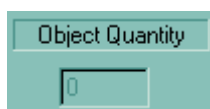
Erase all instances of the adjacent category from the map

Value Settings

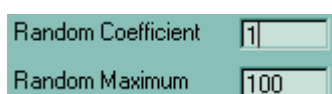
This section allows the quantities of objects to be set and also provides some miscellaneous information:



This displays the coordinates of the mouse cursor when it is positioned over the Drawing Window



If the radio button “Set Mineral / Ore Value” is selected, this edit box allows the user to place the mouse cursor over a mineral or ore object and set its quantity.



This allows the user to state the parameters for randomisation. The coefficient represents the minimum

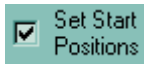
number



According to whether a mineral or ore object radio button is selected and a value has been entered into the “Object Quantity” edit box, the system sets all objects of that category to have the value specified in the edit box

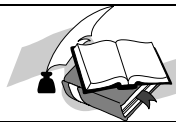


This checks the map for any regions that may be inaccessible and highlights the regions with white circles



This has a two-fold purpose. The user may define specific starting locations where agents will be positioned at the beginning of an experiment. These are the “S” objects in the Map pictured at the beginning of this appendix item. Also, it is used to demonstrate the path-searching algorithm

A demonstration of the Path-Searching Algorithm using Animate’s Map Editor occurs in Appendix T.4.



T.4 - The Path-Searching Algorithm - Page 164

Items below the Drawing Window

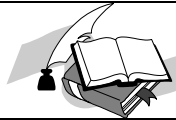
Three boxes provide dynamic information about the total mineral and ore quantities present in the map. There is also a button and a check box that possess the following functionality:



Erase the entire map



If selected and the user clicks the mouse cursor within the Drawing Window, a total search is illustrated using the current map configuration. See Appendix T.4.



T.4 - The Path-Searching Algorithm - Page 164

There is also a load and save button at the top-right of the Drawing Window that allows map files to be stored and retrieved.

T.7 The Hunting Experiment Data

Four experiments were conducted:

1. One Predator agent and one Prey agent
2. Two Predator agents and one Prey agent
3. Four Predator agents and one Prey agent
4. Eight Predator agents and one Prey agent

Data during the experiments was collected into ASCII text files. These files were then parsed into Microsoft Excel and were arranged in tables. The column totals were calculated as follows:

Let the data in column “Kill Turn” = KT

Let the data in column “Total Stomachs” = TS

Let the data in column “Flock Kill?” = FK

Let the data in column “Flock + Surround?” = FS

Let the data in column “Surround?” = S

Let the data in column “Individual?” = I

Let the data in the box at the foot of column “Scenarios” = Kills Made = KM

Let Scenarios Run = 50 = SR

Let Maximum Turns = 500 = MT

Let Maximum Possible Total Stomach Contents = MS (No. of Predators * 20)

- Kills Made = $\sum(KT < 500)$
- Kill Time % = $1 - ((1/(SR * MT)) * \sum(KT))$
- Feeding % = $(1/(SR * MS)) * \sum(TS)$
- Flock Involved = $((100/(KM * 2)) * (\sum(FK) * 2))/100$
- Flock + Surrounding = $((100/(KM * 2)) * (\sum(FS) * 2))/100$
- Surrounding = $((100/(KM * 2)) * (\sum(S) * 2))/100$
- Individual = $((100/(KM * 2)) * (\sum(I) * 2))/100$

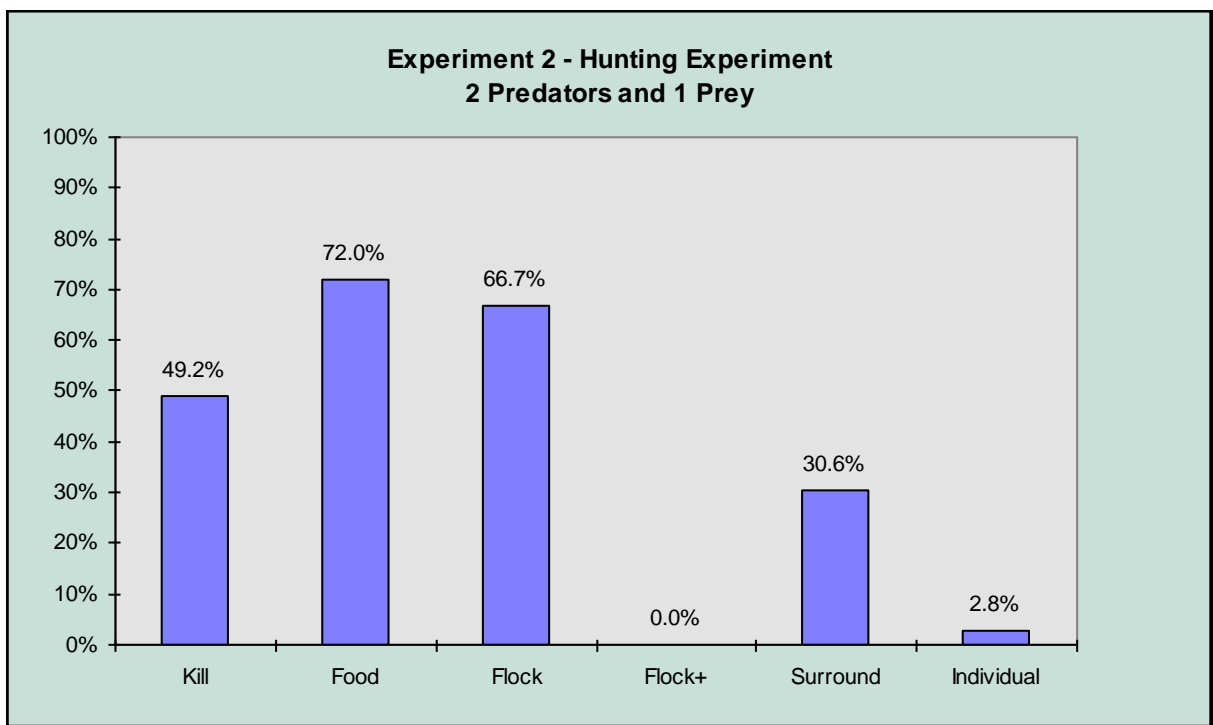
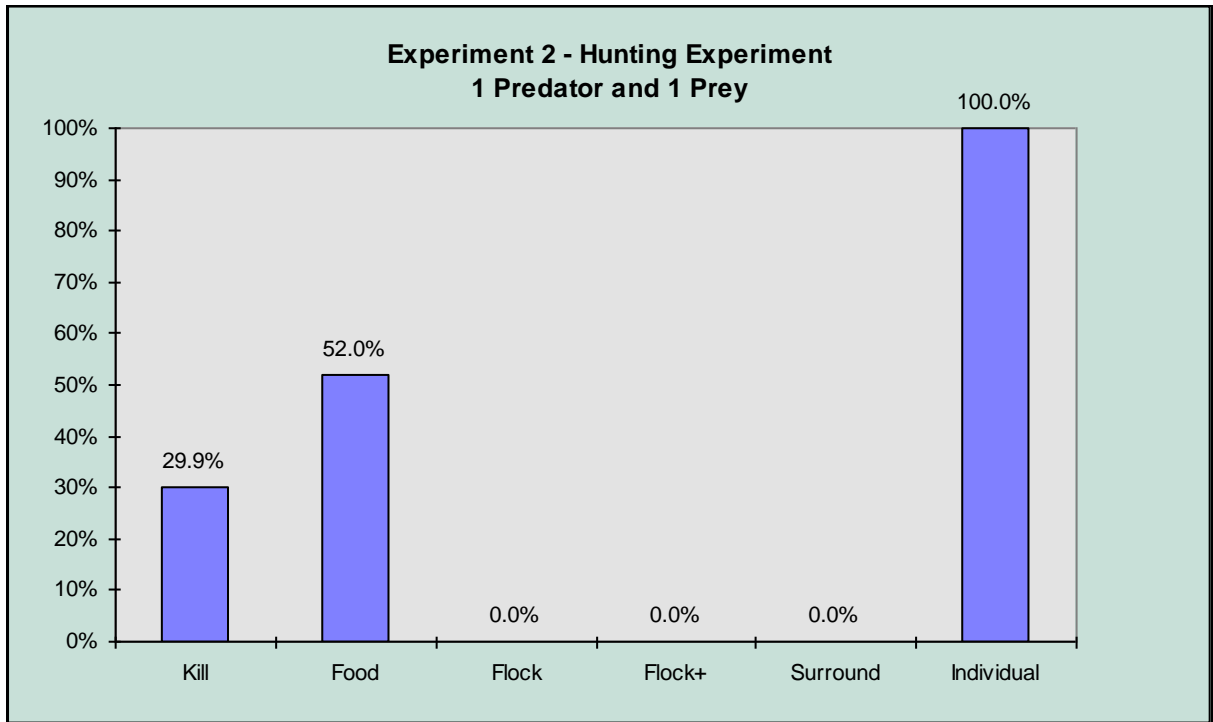
One on One						
Scenario	Kill Turn	Total Stomachs	Flock Kill?	Flock + Surround?	Surround?	Individual?
1	40	20				1
2	500	0				
3	500	0				
4	500	0				
5	221	20				1
6	326	20				1
7	412	20				1
8	199	20				1
9	500	0				
10	500	0				
11	1	20				1
12	500	0				
13	500	0				
14	84	20				1
15	31	20				1
16	439	20				1
17	158	20				1
18	500	0				
19	456	20				1
20	240	20				1
21	269	20				1
22	142	20				1
23	387	20				1
24	35	20				1
25	500	0				
26	141	20				1
27	157	20				1
28	500	0				
29	500	0				
30	500	0				
31	20	20				1
32	500	0				
33	500	0				
34	43	20				1
35	500	0				
36	495	20				1
37	500	0				
38	500	0				
39	500	0				
40	500	0				
41	392	20				1
42	360	20				1
43	108	20				1
44	500	0				
45	500	0				
46	500	0				
47	500	0				
48	331	20				1
49	500	0				
50	39	20				1
Kills Made	29.9%	52.0%	0.0%	0.0%	0.0%	100.0%
26	Kill Time %	Feeding %	Flock Involved	Flock + Surrounding	Surrounding	Individual
Experiment: 2 Scenario Type: Predators & Prey Predators: 1 Predator Vision: 12 Predator Stomach Max: 20 Predator Stomach Init: 0 Predator Ingest Value: 10 units of carcass per turn Prey: 1 Prey Carcass Value: 200 Prey Vision: 4 Map Name: hunting.map Scenario Turns: 500 Scenario Iterations: 50 Agents Explore?: No						

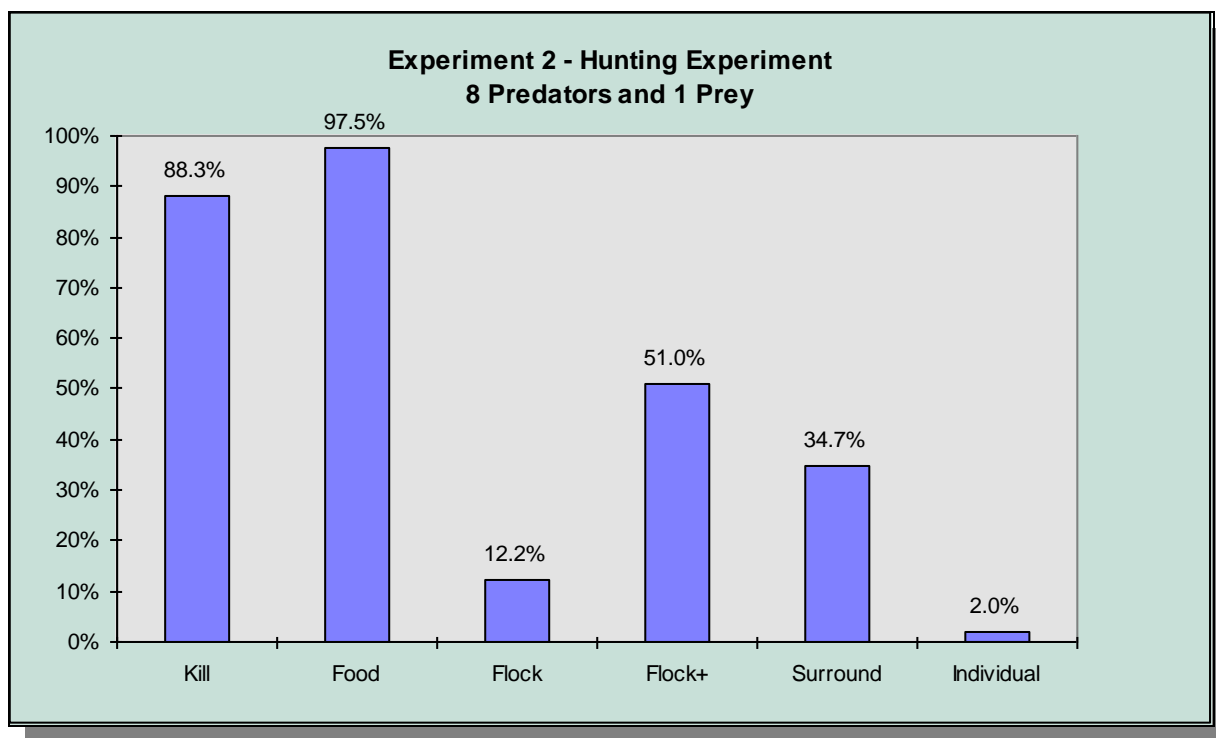
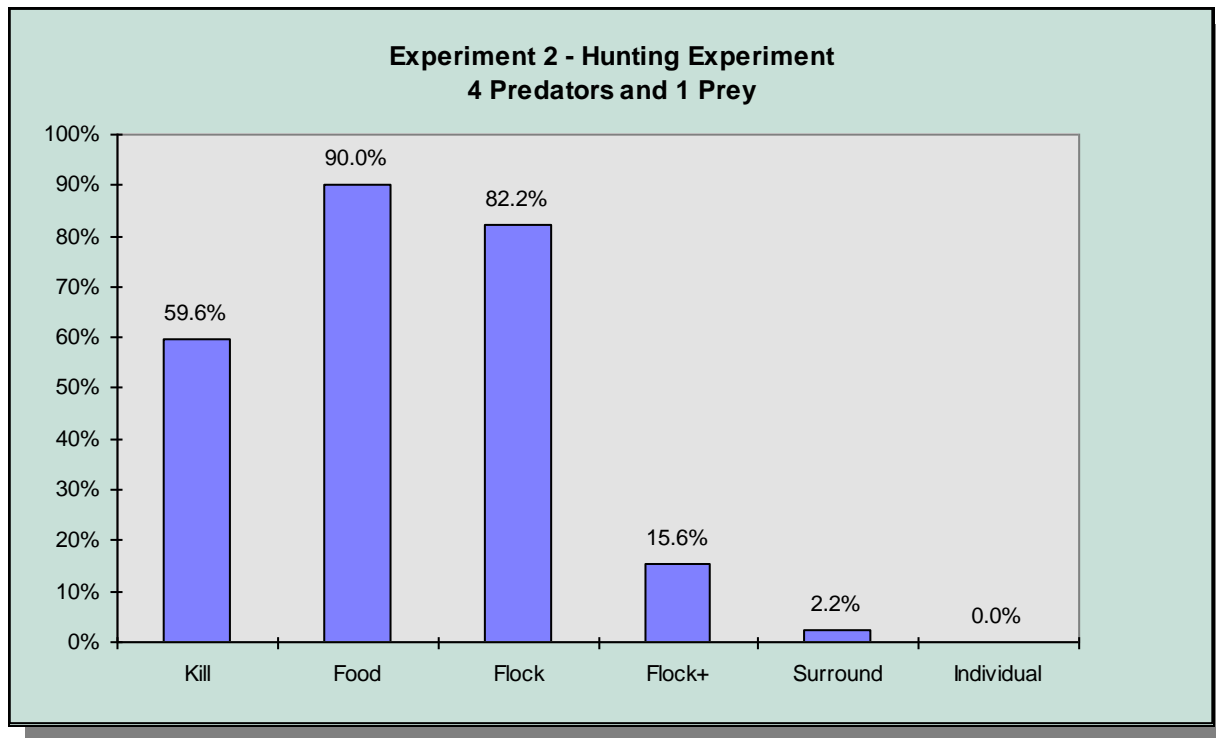
Two on One						
Scenario	Kill Turn	Total Stomachs	Flock Kill?	Flock + Surround?	Surround?	Individual?
1	500	0				
2	500	0				
3	339	40	1			
4	344	40	1			
5	83	40	1			
6	36	40			1	
7	44	40			1	
8	500	0				
9	16	40			1	
10	500	0				
11	265	40	1			
12	84	40	1			
13	16	40			1	
14	14	40			1	
15	500	0				
16	254	40	1			
17	500	0				
18	383	40	1			
19	284	40	1			
20	24	40				1
21	480	40	1			
22	42	40	1			
23	247	40	1			
24	500	0				
25	136	40	1			
26	40	40	1			
27	500	0				
28	35	40	1			
29	360	40	1			
30	388	40	1			
31	500	0				
32	500	0				
33	15	40			1	
34	135	40	1			
35	213	40	1			
36	500	0				
37	61	40			1	
38	42	40			1	
39	500	0				
40	271	40	1			
41	500	0				
42	65	40			1	
43	32	40			1	
44	54	40	1			
45	193	40	1			
46	103	40	1			
47	500	0				
48	375	40	1			
49	208	40	1			
50	17	40			1	
Kills Made	49.2%	72.0%	66.7%	0.0%	30.6%	2.8%
36	Kill Time %	Feeding %	Flock Involved	Flock + Surrounding	Surrounding	Individual
Experiment: 2 Scenario Type: Predators & Prey Predators: 2 Predator Vision: 12 Predator Stomach Max: 20 Predator Stomach Init: 0 Predator Ingest Value: 10 units of carcass per turn Prey: 1 Prey Carcass Value: 200 Prey Vision: 4 Map Name: hunting.map Scenario Turns: 500 Scenario Iterations: 50 Agents Explore?: No						

Four on One						
Scenario	Kill Turn	Total Stomachs	Flock Kill?	Flock + Surround?	Surround?	Individual?
1	133	80	1			
2	211	80	1			
3	36	80	1			
4	71	80	1			
5	144	80	1			
6	144	80	1			
7	80	80	1			
8	473	80	1			
9	192	80	1			
10	8	80			1	
11	83	80	1			
12	500	0				
13	114	80		1		
14	500	0				
15	296	80	1			
16	25	80		1		
17	294	80	1			
18	312	80	1			
19	156	80	1			
20	42	80	1			
21	329	80	1			
22	178	80	1			
23	260	80	1			
24	270	80	1			
25	38	80		1		
26	84	80	1			
27	388	80		1		
28	44	80	1			
29	43	80		1		
30	413	80	1			
31	127	80	1			
32	135	80	1			
33	126	80	1			
34	500	0				
35	500	0				
36	167	80	1			
37	194	80	1			
38	287	80	1			
39	303	80	1			
40	30	80		1		
41	372	80	1			
42	500	0				
43	240	80	1			
44	181	80	1			
45	169	80	1			
46	76	80	1			
47	56	80		1		
48	74	80	1			
49	43	80	1			
50	171	80	1			
Kills Made	59.6%	90.0%	82.2%	15.6%	2.2%	0.0%
45	Kill Time %	Feeding %	Flock Involved	Flock + Surrounding	Surrounding	Individual
Experiment: 2 Scenario Type: Predators & Prey Predators: 4 Predator Vision: 12 Predator Stomach Max: 20 Predator Stomach Init: 0 Predator Ingest Value: 10 units of carcass per turn Prey: 1 Prey Carcass Value: 200 Prey Vision: 4 Map Name: hunting.map Scenario Turns: 500 Scenario Iterations: 50 Agents Explore?: No						

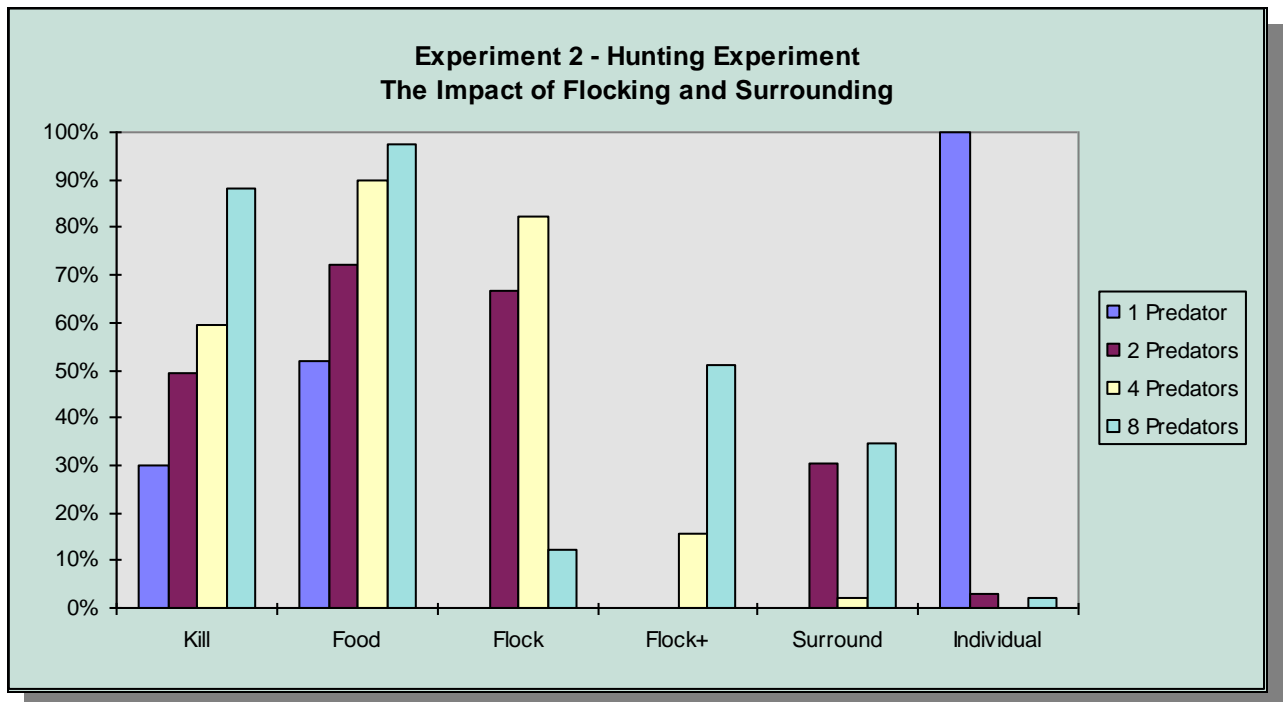
Eight on One						
Scenario	Kill Turn	Total Stomachs	Flock Kill?	Flock + Surround?	Surround?	Individual?
1	8	160			1	
2	18	160			1	
3	7	160			1	
4	5	160			1	
5	11	160		1		
6	178	160		1		
7	29	160		1		
8	32	160	1			
9	187	160	1			
10	10	160			1	
11	21	160		1		
12	14	160			1	
13	12	160			1	
14	35	160		1		
15	37	160		1		
16	27	160		1		
17	147	160		1		
18	239	140		1		
19	93	160		1		
20	18	160			1	
21	26	160		1		
22	18	160			1	
23	19	160		1		
24	32	160			1	
25	57	160		1		
26	180	160		1		
27	28	160		1		
28	20	160				1
29	26	160		1		
30	111	140	1			
31	62	160		1		
32	7	160			1	
33	62	160		1		
34	43	160		1		
35	12	160			1	
36	11	160			1	
37	27	160		1		
38	10	160			1	
39	75	160	1			
40	71	160		1		
41	160	160		1		
42	32	160		1		
43	11	160			1	
44	9	160			1	
45	24	160		1		
46	29	160	1			
47	500	0				
48	47	160		1		
49	17	160			1	
50	62	160	1			
Kills Made	88.3%	97.5%	12.2%	51.0%	34.7%	2.0%
49	Kill Time %	Feeding %	Flock Involved	Flock + Surrounding	Surrounding	Individual
Experiment: 2 Scenario Type: Predators & Prey Predators: 8 Predator Vision: 12 Predator Stomach Max: 20 Predator Stomach Init: 0 Predator Ingest Value: 10 units of carcass per turn Prey: 1 Prey Carcass Value: 200 Prey Vision: 4 Map Name: hunting.map Scenario Turns: 500 Scenario Iterations: 50 Agents Explore?: No						

The charts for each table follow:





The combined chart follows:



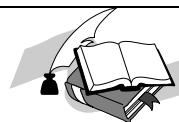
Appendix - Volume 2

System Code

C.1 The Grid Class	192
C.2 The Environment Class	207
C.3 The Intent Class	220
C.4 The Agent Class	231
C.5 The Miner Class	258
C.6 The Predator Class	271
C.7 The Prey Class	280
C.8 The Chain-Miner Class	288
C.9 The Crowd Class	309
C.10 The Arbitrator Class	314
C.11 The Administrator Class	323
C.12 The Special Data Structures	352

C.1 The Grid Class

Here, we provide the system code for the Grid class. It's features are discussed in Appendix T.2.



T.2 - The Grid Class Implementation - Page 160

// Filename: Grid.h


```
//      Version:          Final
//      Author:           Paul Tinsley (pmbtin)
//      Date:             27 February, 1999
//      Platform:         Visual C++ 5.00

//      Symbology:        Numbers, functions and data variables are enclosed in
//                        square parentheses "[ ]". This is to remove any
//                        ambiguity within the text of comments. All private data
//                        member variables are prefixed by "g_". Any function with
//                        the suffix "_f" is a faster version of that function as
//                        there is no range / error checking.

//      Errors:           All error messages appended to "errlog.txt".

//      Purpose:          This class implements a two-dimensional grid with
//                        square dimensions that may be accessed by cartesian
//                        parameters using an implementation of the STL [vector]
//                        container. Any n*n grid of [short] types may be defined
//                        within the range [10]*[10] up to a maximum size of [40]*[40].
//                        The origin of the grid is ([0],[0]) up to a maximum
//                        reference of ([s]-[1],[s]-[1]), [s] being the dimension
//                        of the grid. The origin is perceived as starting at the
//                        top left and writing in rows of [s] down the "screen".
```

```
// Function Comment Style:
```

```
// Purpose:              Purpose of the function.
// Preconditions:         What states or conditions must be met before execution
//                        of the function.
// Returns:              The object or primitive type that the function returns.
// Side Effects:          Effects that the function may have on private data that
//                        are not immediately obvious to the nature of the function.
//                        This section may also contain advice and miscellaneous
//                        comments.
// Errors:               What type of error is generated, why it is generated and
//                        its effects.
```

```
#ifndef __GRID_H
#define __GRID_H
```

```
#include <vector>          // to contain [Grid] elements
#include <fstream>         // output / input / from files
#include <stddef.h>        // for program error exit
#include <algorithm>       // generic container functions
```

```
//      *** TYPE DEFINITIONS / GLOBALS / COMPILER FLAGS ***
```

```
using namespace std; // standard namespace directive
```

```
// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
```

```
#pragma warning(disable:4786)
```

```
typedef vector<short> SHORTVECTOR; // the vector type
typedef SHORTVECTOR::iterator SHORTVECTOR_IT; // iterator over SHORTVECTOR
const short MAX_DIMENSION = 40; // largest Grid dimension
const short MIN_DIMENSION = 10; // smallest Grid dimension
const short MAX_VALUE = 32700; // largest Grid location value
const short MIN_VALUE = -32700; // smallest Grid location value
```

```
//      *** OBJECT / ACTION ENUMERATIONS ***
```

```

/*      The following enumerations provide a symbology to the programmer for types
of objects, actions and action classes. Further details occur in the report.
*/

enum {SPACE,OBSTACLE,MINER,PREDATOR,PREY,CARCASS,MINERAL,ORE,HOME,UNKNOWN,MOVE,
NOTHING,GRAB,DROP,POUNCE,JUMP,ASTART,C_INHABIT,C_INTERACT};

//      NOTES:      Order is important. All agents must fall after MINER and before
//                  MINERAL. All non-agent and non-space and non-obstacle items must
//                  fall after MINERAL.

//      *** CLASS DEFINITION ***

class Grid

{

    public:

        //      *** CONSTRUCTORS ***

        Grid();

        // Purpose:      Default constructor.
        // Preconditions: None.
        // Returns:      A [Grid] instance.
        // Side Effects:  The Grid is not fully constructed. Note
        //                  that the Grid dimension may be redefined
        //                  and the Grid rebuilt by using the member
        //                  function [metamorphose(const short s)].
        // Errors:      None.

        Grid(const short s);

        // Purpose:      Creates a [Grid] instance based on dimension [s]
        //                  and initialises all contained values to [0].
        // Preconditions: [s] >= [MIN_DIMENSION] && [s] <= [MAX_DIMENSION].
        // Returns:      A [Grid] instance.
        // Side Effects:  None.
        // Errors:      [errlog] error message if precondition violation.
        //                  Program then terminates.

        //      *** COPY CONSTRUCTOR ***

        Grid(const Grid &g);

        // Purpose:      Creates a copy of a [Grid] instance based on
        //                  the [Grid] parameter [g].
        // Preconditions: None.
        // Returns:      A [Grid] instance.
        // Side Effects:  None.
        // Errors:      None.

        //      *** OPERATORS ***

        Grid & operator =(const Grid &g);

        // Purpose:      Destructively assigns [g] to the current
        //                  context.
        // Preconditions: None.
        // Returns:      None.
        // Side Effects:  Destructive assignment.

```

```

// Errors:                None.

const bool operator ==(const Grid &g);

// Purpose:                Deep equality operator.
// Preconditions: None.
// Returns:                [bool].
// Side Effects:           None.
// Errors:                None.

//      *** PUBLIC FUNCTION PROTOTYPES ***

void put(const short x, const short y, const short v);
void put_f(const short x, const short y, const short v);

// Purpose:                Sets element at coordinate ([x],[y]) to
//                          value specified by [v].
// Preconditions: If ![g_state], function exits without putting.
//               [x] < [g_SIZE] && [x] >= [0] && [y] < [g_SIZE]
//               && [y] >= [0]. [v] >= [MIN_VALUE] && [v] <=
//               [MAX_VALUE].
// Returns:                None.
// Side Effects:           None.
// Errors:                [errlog] error message if precondition violated.
//                          No action is performed. [g_state] = [false].

const short get(const short x, const short y);
const short get_f(const short x, const short y);

// Purpose:                Returns value held at coordinate ([x],[y]).
// Preconditions: If ![g_state], function exits without getting.
//               [x] < [g_SIZE] && [x] >= [0] && [y] < [g_SIZE]
//               && [y] >= [0].
// Returns:                [short]. Value held at ([x],[y]) in the Grid.
// Side Effects:           None.
// Errors:                [errlog] error message if precondition violated.
//                          [g_state] = [false]. Returns [0] if error
//                          occurs. Returns [0] if ![g_state].

const bool get_state();

// Purpose:                Returns the current state of the class.
// Preconditions: None.
// Returns:                [bool] [g_state]. If [false], an error has
//                          occurred.
// Side Effects:           None.
// Errors:                None.

void set_state(const bool s);

// Purpose:                Set the state of this class [g_state] to [s].
//                          Use extreme caution with this function as you
//                          have the ability to force the error state to
//                          a possible illogical value.
// Preconditions: None.
// Returns:                None.
// Side Effects:           None.
// Errors:                None.

const short get_dim();

// Purpose:                Returns dimension of [g_list], [g_SIZE].
// Preconditions: None.

```

```

// Returns:          [short] [g_SIZE].
// Side Effects:      None.
// Errors:            None.

void wipe(const short v);

// Purpose:          Sets every element of the Grid to value [v].
// Preconditions: If ![g_state], function exits without wiping.
//                  [v] <= [MAX_VALUE] && [v] >= [MIN_VALUE].
// Returns:           None.
// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.
//                  [g_state] = [false].

void replace_all_instances(const short v, const short r);

// Purpose:          Replace every element of [g_list] that has
//                  the value [v], with the value [r].
// Preconditions: If ![g_state], function exits without replacing.
//                  [v] <= [MAX_VALUE] && [v] >= [MIN_VALUE].
//                  [r] <= [MAX_VALUE] && [r] >= [MIN_VALUE].
// Returns:           None.
// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.
//                  [g_state] = [false].

void replace_all_non_zero_instances(const short r);

// Purpose:          Replace every element of [g_list] that satisfies
//                  the unary predicate [is_greater_zero()] with the
//                  the value [r].
// Preconditions: If ![g_state], function exits without replacing.
//                  [r] <= [MAX_VALUE] && [r] >= [MIN_VALUE].
// Returns:           None.
// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.
//                  [g_state] = [false].

void expire_moving();

// Purpose:          Replace every element of [g_list] that satisfies
//                  the unary predicate [is_moving_object()] with
//                  [SPACE].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

const short count_all_instances(const short v);

// Purpose:          Counts how many elements in [g_list] that have
//                  the value [v].
// Preconditions: If ![g_state], function exits without counting
//                  and returns [0].
//                  [v] <= [MAX_VALUE] && [v] >= [MIN_VALUE].
// Returns:           [short]. The amount of elements.
// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.
//                  [g_state] = [false].

const bool find_instance(const short v);

// Purpose:          If [v] is in [g_list], returns [true], else

```

```

//          returns [false].
// Preconditions: If ![g_state], function exits without finding
//               and returns [false].
//               [v] <= [MAX_VALUE] && [v] >= [MIN_VALUE].
// Returns:      [bool]. Whether element found.
// Side Effects:  None.
// Errors:       [errlog] error message if precondition violated.
//               [g_state] = [false].

const void output(ofstream &out);

// Purpose:      Sends contents of Grid to the supplied
//               file output stream [out]. Data is written out
//               in [g_SIZE] rows.
// Preconditions: Valid, active stream. If ![g_state], function
//               exits without data output.
// Returns:      None.
// Side Effects:  The stream receives data.
// Errors:       if [out] failure, function exits with [errlog]
//               message. [g_state] = [false].

void input(ifstream &in);

// Purpose:      Sets the [g_list] to the contents of the file.
// Preconditions: If ![g_state], function exits without input.
//               The file must be ANSI text.
//               Values in the file must be arranged in [g_SIZE]
//               columns and [g_SIZE] rows. Each value must be
//               seperated by a space character [" "]. The
//               stream must be freshly opened and located at
//               the first data item (excluding whitespace /
//               returns).
// Returns:      None.
// Side Effects:  None.
// Errors:       If ![g_state], function exits without processing
//               the file. If stream failure or end of file
//               detected before completion, a [errlog] message
//               is generated. [g_state] = false.

const bool check(const short x, const short y);

// Purpose:      Analyses whether [x] and [y] are valid
//               cartesian coordinates for [g_list].
// Preconditions: None.
// Returns:      [bool], [true]/[false].
// Side Effects:  None.
// Errors:       None.

void metamorphose(const short s);

// Purpose:      Destroy the Grid and re-create it using the
//               dimension [s].
// Preconditions: [s] >= MIN_DIMENSION && [s] <= MAX_DIMENSION.
// Returns:      None.
// Side Effects:  Every new location value is set to [0].
//               If succesful, [g_state] = [true].
// Errors:       [errlog] error message if precondition violation.

```

private:

```

//      *** PRIVATE MEMBER VARIABLES ***

SHORTVECTOR g_list;      // the vector list of coordinate values

```

```

SHORTVECTOR g_lookup;    // lookup table for y reference
short g_SIZE;            // the dimension of the square Grid
bool g_state;            // an error state variable, [false] == error

//      *** PRIVATE FUNCTION PROTOTYPES ***

void create_lookup();

// *** Suggested by Mark Roper ***

// Purpose:              Generates a lookup table for the position of a
//                        y-referenced coordinate. This is used by the
//                        getters and putters to more rapidly compute
//                        locations in the vector on the basis of the
//                        mapping: element E = x + (y * g_SIZE). This
//                        function computes the multiplication in advance
//                        and stores the result in a lookup table referenced
//                        by the y parameter. Therefore, all future vector
//                        location computation only involves addition.
// Preconditions: This must be called before any references are made
//                to grid coordinate locations.
// Returns:             None.
// Side Effects:        None.
// Errors:              None.

};

//      *** UNARY PREDICATES ***

//      The following simple predicate is used in the function
//      [replace_all_non_zero_instances(const short r)]

class is_greater_zero

{
public:

    const bool operator()(const short &v)

    {

        return(v > 0);

    }

};

//      The following simple predicate is used in the function
//      [expire_moving()]

class is_moving_object

{
public:

    const bool operator()(const short &v)

    {

        return(v >= MINER && v < MINERAL);

    }

};

```

```

    }

};

#endif

//      Filename:          Grid.cpp
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              27 February, 1999
//      Platform:          Visual C++ 5.00

//      Purpose:           See Grid.h.

#include "stdafx.h"
#include "Grid.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** CONSTRUCTORS ***

Grid::Grid() : g_SIZE(40), g_state(false)
{
    g_list.clear();
    g_lookup.clear();
}

Grid::Grid(const short s): g_SIZE(s), g_state(true)
{
    g_list.clear();
    g_lookup.clear();

    if(g_SIZE > MAX_DIMENSION || g_SIZE < MIN_DIMENSION)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::Grid(s), a size [s] of " << g_SIZE;
        err << " is out of range.\nThe Grid cannot be set!\n";
        err << "FORCED EXIT!\n";
        err.close();
        exit(1);
    }

    const short limit = (g_SIZE * g_SIZE);

    // allocate vector memory
    g_list.reserve(limit);

    // tidy up the vector to save memory
    g_list.resize(limit);

    // set all elements to spaces

```

```

        wipe(0);

        // create a lookup table
        create_lookup();
    }

Grid::Grid(const Grid &g)
{
    g_list = g.g_list;
    g_SIZE = g.g_SIZE;
    g_state = g.g_state;
    g_lookup = g.g_lookup;
}

//      *** OPERATORS ***

Grid & Grid::operator =(const Grid &g)
{
    g_list = g.g_list;
    g_SIZE = g.g_SIZE;
    g_state = g.g_state;
    g_lookup = g.g_lookup;

    return(*this);
}

const bool Grid::operator ==(const Grid &g)
{
    return(g_list == g.g_list);
}

//      *** PUBLIC FUNCTIONS ***

void Grid::put(const short x, const short y, const short v)
{
    if(!g_state)
    {
        return;
    }

    if(!check(x,y))
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::put(x,y,v), [x] or [y] is out of range.\n";
        err << "The put operation of " << v << " to (" << x << ", " << y << ")\n";
        err << "failed for this Grid of dimension " << g_SIZE << ".\n";
    }
}

```



```

        err.close();
        g_state = false;
        return;
    }

    if(v < MIN_VALUE || v > MAX_VALUE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::put(x,y,v), [v] is out of range.\n";
        err << "The put operation of " << v << " to (" << x << ", " << y << ") \n";
        err << "failed for this Grid of dimension " << g_SIZE << ".\n";
        err.close();
        g_state = false;
        return;
    }

    // map cartesian coordinates to contiguous and unique list coordinates
    g_list[x + g_lookup[y]] = v;
}

void Grid::put_f(const short x, const short y, const short v)
{
    g_list[x + g_lookup[y]] = v;
}

const short Grid::get(const short x, const short y)
{
    if(!g_state)
    {
        return(0);
    }

    if(!check(x,y))
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::get(x,y), [x] or [y] is out of range.\n";
        err << "The get operation from coordinates (" << x << ", " << y << ") \n";
        err << "failed for this Grid of dimension " << g_SIZE << ".\n";
        err.close();
        g_state = false;
        return(0);
    }

    return(g_list[x + g_lookup[y]]);
}

```

```

const short Grid::get_f(const short x, const short y)
{
    return(g_list[x + g_lookup[y]]);
}

const bool Grid::get_state()
{
    return(g_state);
}

void Grid::set_state(const bool s)
{
    g_state = s;
}

const short Grid::get_dim()
{
    return(g_SIZE);
}

void Grid::wipe(const short v)
{
    if(!g_state)
    {
        return;
    }

    if(v < MIN_VALUE || v > MAX_VALUE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::wipe(v), [v] is out of range.\n";
        err << "The wipe operation has failed.\n";
        err.close();
        g_state = false;
        return;
    }

    // this is an STL container algorithm
    fill(g_list.begin(),g_list.end(),v);
}

void Grid::replace_all_instances(const short v, const short r)

```

```

{

    if(!g_state)

    {

        return;

    }

    if(v < MIN_VALUE || v > MAX_VALUE || r < MIN_VALUE || r > MAX_VALUE)

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::replace_all_instances(v,r),\n";
        err << "[v] or [r] is out of range. The replace operation has failed.\n";
        err.close();
        g_state = false;
        return;

    }

    // an STL container algorithm
    replace(g_list.begin(),g_list.end(),v,r);

}

void Grid::replace_all_non_zero_instances(const short r)

{

    if(!g_state)

    {

        return;

    }

    if(r < MIN_VALUE || r > MAX_VALUE)

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::replace_all_non_zero_instances(r),\n";
        err << "[r] is out of range. The replace operation has failed.\n";
        err.close();
        g_state = false;
        return;

    }

    // an STL container algorithm
    replace_if(g_list.begin(),g_list.end(),is_greater_zero(),r);

}

void Grid::expire_moving()

{

```

```

        // an STL container algorithm
        replace_if(g_list.begin(),g_list.end(),is_moving_object(),SPACE);
    }

    const short Grid::count_all_instances(const short v)
    {
        if(!g_state)
        {
            return(0);
        }

        if(v < MIN_VALUE || v > MAX_VALUE)
        {
            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, Grid::count_all_instances(v), [v]\n";
            err << "is out of range. The count operation has failed.\n";
            err.close();
            g_state = false;
            return(0);
        }

        // an STL container algorithm
        return(short(count(g_list.begin(),g_list.end(),v)));
    }

    const bool Grid::find_instance(const short v)
    {
        if(!g_state)
        {
            return(0);
        }

        if(v < MIN_VALUE || v > MAX_VALUE)
        {
            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, Grid::find_instance(v), [v]\n";
            err << "is out of range. The find operation has failed.\n";
            err.close();
            g_state = false;
            return(0);
        }

        // an STL container algorithm
        SHORTVECTOR_IT f = find(g_list.begin(),g_list.end(),v);

```

```

        return(f != g_list.end());

    }

    const void Grid::output(ofstream &out)
    {
        if(!g_state)
        {
            return;
        }

        if(out.fail())
        {
            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, Grid::output(out), stream failure!\n";
            err << "The data could not be output.\n";
            err.close();
            g_state = false;
            return;
        }

        short count;

        for(count = 0 ; count < (g_SIZE * g_SIZE) ; count++)
        {
            if((count % g_SIZE) == 0)
            {
                out << endl;
            }

            out << g_list[count] << " ";
        }
    }

    void Grid::input(ifstream &in)
    {
        if(!g_state)
        {
            return;
        }
    }

```

```

if(in.fail())
{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Grid::set_map(in), stream failure!\n";
    err << "The data from the file could not be loaded.\n";
    err.close();
    g_state = false;
    return;
}

SHORTVECTOR_IT locator;
short value;

for(locator = g_list.begin() ; locator != g_list.end() ; ++locator)
{
    in >> value;

    if(in.fail() || in.eof())
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::set_map(in), stream failure!\n";
        err << "The data from the file could not be loaded correctly.\n";
        err.close();
        g_state = false;
        return;
    }

    *locator = value;
}

}

const bool Grid::check(const short x, const short y)
{
    return((x < g_SIZE) && (x >= 0) && (y < g_SIZE) && (y >= 0));
}

void Grid::metamorphose(const short s)
{
    if(s > MAX_DIMENSION || s < MIN_DIMENSION)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Grid::metamorphose(s), [s] is out of range.\n";
        err << "The metamorphose of this Grid has failed.\n";
        err.close();
        g_state = false;
        return;
    }
}

```

```

    }

    // erase old
    g_list.clear();

    // set new dimension
    g_SIZE = s;

    // reset error state
    g_state = true;

    // calculate vector limit
    const short limit = (g_SIZE * g_SIZE);

    // allocate vector memory
    g_list.reserve(limit);

    // tidy up the vector to save memory
    g_list.resize(limit);

    // set all elements to zero
    wipe(0);

    // create a lookup table
    create_lookup();
}

//      *** PRIVATE FUNCTIONS ***

void Grid::create_lookup()
{
    g_lookup.clear();
    for(short y = 0 ; y < g_SIZE ; y++)
    {
        g_lookup.push_back(y * g_SIZE);
    }
    g_lookup.resize(g_SIZE);
}

```

C.2 The Environment Class

First we provide the code for the Environment class. Then we briefly discuss its most salient features.

```

//      Filename:      Environment.h
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          28 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:     Numbers, functions and data variables are enclosed in
//                      square parentheses "[ ]". This is to remove any
//                      ambiguity within the text of comments. All private data
//                      member variables are prefixed by "e_". Any function with
//                      the suffix "_f" is a faster version of that function as
//                      there is no range/error checking.

//      Errors:        All error messages are appended to the file "errlog.txt"

//      Purpose:       This class utilises the functionality of the class [Grid]
//                      to provide a two-dimensional pseudo-environment for the
//                      artificial agents to inhabit. This is achieved by having
//                      two grids - one for object types and another to hold the
//                      quantities related to those objects.

// Function Comment Style:

// Purpose:           Purpose of the function.
// Preconditions:     What states or conditions must be met before execution
//                      of the function.
// Returns:           The object or primitive type that the function returns.
// Side Effects:      Effects that the function may have on private data that
//                      are not immediately obvious to the nature of the function.
//                      This section may also contain advice and miscellaneous
//                      comments.
// Errors:           What type of error is generated, why it is generated and
//                      its effects.

#ifndef __ENVIRONMENT_H
#define __ENVIRONMENT_H

#include "Grid.h"

//      *** CLASS DEFINITION ***

class Environment
{
    public:

        //      *** CONSTRUCTORS ***

        Environment() : e_object(40), e_value(40){};

        // Purpose:      Default constructor.
        // Preconditions: None.
        // Returns:       An Environment instance.
        // Side Effects:  The Environment is not correctly constructed, unless
        //                a dimension of [40] is required. The member function
        //                [metamorphose(const short s)] may be used to define
        //                the dimensions of [e_object] and [e_value].

        Environment(const short s): e_object(s), e_value(s) {};

```



```
// Purpose:          Creates an Enviornment instance based on dimension [s]
//                  and initialises all contained values to [SPACE].
// Preconditions:[s] >= [MIN_DIMENSION] && [s] <= [MAX_DIMENSION].
// Returns:          An Environment instance.
// Side Effects:     None.
// Errors:           [errlog] message from [Grid] if precondition violation.
//                  Program then terminates.
```

```
//      *** OPERATORS ***
```

```
Environment & operator =(const Environment &e);
```

```
// Purpose:          Destructively assigns [e] to the current
//                  context.
// Preconditions:None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.
```

```
//      *** PUBLIC FUNCTION PROTOTYPES ***
```

```
void input_objects(ifstream &in);
```

```
// Purpose:          Loads [e_object] and [e_value] with the data contained
//                  in the file located by the stream [in].
// Preconditions:[e_object.get_state()] == [true].
//                  The file must be ANSI text.
//                  Values in the file must be arranged in
//                  [e_object.g_SIZE] columns and [e_object.g_SIZE]
//                  rows, first, a set of data must exist for [e_object]
//                  then a newline and a data set for [e_value]. Each
//                  value must be seperated by a space character [" "].
//                  The stream must be freshly opened and located at
//                  the first data item (excluding whitespace / returns).
// Returns:          None.
// Side Effects:     None.
// Errors:           If ![e_object.g_state], function exits without
//                  processing the file. If stream failure or end of file
//                  detected before completion, an [errlog] message
//                  is generated. [e_object.g_state] = false.
```

```
const void output_objects(ofstream &out);
```

```
// Purpose:          Sends contents of [e_object] and [e_value] to the
//                  file located by the stream [out].
// Preconditions:[e_object.get_state()] == [true].
//                  See [Grid::output(ofstream &out)].
// Returns:          None.
// Side Effects:     The file stream.
// Errors:           If ![e_object.g_state], function exits without
//                  processing the file. An [errlog] message is
//                  generated.
```

```
void put_object(const short x, const short y, const short i, const short q);
void put_object_f(const short x, const short y, const short i, const short q);
```

```
// Purpose:          Place [i] at the location ([x],[y]) in [e_object] and
//                  record its quantity [q] at the same location in
//                  [e_value].
// Preconditions:If (![e_object.g_state] || ![e_value.g_state]), function
//                  exits without putting values.
//                  [x] < [e_object.g_SIZE] && [x] >= [0] && [y] <
```

```
// [e_object.g_SIZE] && [y] >= [0]. [i] >= [MIN_VALUE] &&
// [i] <= [MAX_VALUE]. [q] >= [0] && [q] <= [MAX_VALUE].
// Returns: None.
// Side Effects: None.
// Errors: [errlog] error message if precondition violation.
// No action is performed. [e_object.g_state] or
// [e_value.g_state] may be set to [false].
```

```
const short get_object(const short x, const short y);
const short get_object_f(const short x, const short y);
```

```
// Purpose: Returns value held at the location ([x],[y]) in
// [e_object].
// Preconditions: If ![e_object.g_state], function exits without
// getting and returns [0].
// [x] < [e_object.g_SIZE] && [x] >= [0] && [y] <
// [e_object.g_SIZE] && [y] >= [0].
// Returns: [short]. Value held at ([x],[y]) in [e_object].
// Side Effects: None.
// Errors: [errlog] error message if precondition violated.
// [e_object.g_state] = [false]. Returns [0] if error
// occurs. Returns [0] if ![e_object.g_state].
```

```
const void get_object(const short x, const short y, short &i, short &q);
const void get_object_f(const short x, const short y, short &i, short &q);
```

```
// Purpose: Loads [i] with value held at the location ([x],[y])
// in [e_object] and [q] with the quantity held at the
// same location in [e_value].
// Preconditions: If (![e_object.g_state] || ![e_value.g_state]),
// function exits without getting all values.
// [x] < [e_object.g_SIZE] && [x] >= [0] && [y] <
// [e_object.g_SIZE] && [y] >= [0].
// Returns: None.
// Side Effects: None.
// Errors: [errlog] error message if precondition violated.
// [e_object.g_state] or [e_value.g_state] may be set
// to [false].
```

```
void put_quantity(const short x, const short y, const short q);
void put_quantity_f(const short x, const short y, const short q);
```

```
// Purpose: Place [q] at the location ([x],[y]) in [e_value].
// Preconditions: If ![e_value.g_state], function exits without
// putting value.
// [x] < [e_value.g_SIZE] && [x] >= [0] && [y] <
// [e_value.g_SIZE] && [y] >= [0]. [q] >= [0] &&
// [q] <= [MAX_VALUE].
// Returns: None.
// Side Effects: None.
// Errors: [errlog] error message if precondition violation.
// No action is performed. [e_value.g_state] = [false].
```

```
const short get_quantity(const short x, const short y);
const short get_quantity_f(const short x, const short y);
```

```
// Purpose: Returns value held at the location ([x],[y]) in
// [e_value].
// Preconditions: If ![e_value.g_state], function exits without
// getting and returns [0].
// [x] < [e_value.g_SIZE] && [x] >= [0] && [y] <
// [e_value.g_SIZE] && [y] >= [0].
// Returns: [short]. Value held at ([x],[y]) in [e_value].
```

```

// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.
//                   [e_value.g_state] = [false]. Returns [0] if error
//                   occurs. Returns [0] if ![e_value.g_state].

```

```

short take_quantity(const short x, const short y, const short a);
short take_quantity_f(const short x, const short y, const short a);

```

```

// Purpose:          If sufficient quantity in [e_value] exists that
//                   exceeds [a], returns [a] and deducts [a] from
//                   the quantity held at [e_value]. Otherwise returns
//                   a value that is less than [a], returns that value
//                   and sets the value at ([x],[y]) in [e_object] to
//                   [SPACE] in order to record the fact that the object
//                   no longer exists.
// Preconditions: If ![e_value.g_state], function exits without
//                   getting and returns [0].
//                   [e_object.get([x],[y])] != [SPACE].
//                   [x] < [e_value.g_SIZE] && [x] >= [0] && [y] <
//                   [e_value.g_SIZE] && [y] >= [0].
// Returns:           [short]. Portion or all of value held at ([x],[y])
//                   in [e_value].
// Side Effects:      See Purpose.
// Errors:            [errlog] error message if precondition violated.
//                   [e_value.g_state] = [false]. Returns [0] if error
//                   occurs. Returns [0] if ![e_value.g_state].

```

```

bool add_quantity(const short x, const short y, const short i, const short a);
bool add_quantity_f(const short x, const short y, const short i, const short a);

```

```

// Purpose:          If sufficient quantity in [e_value] exists that
//                   allows [a] to be added to it, returns [true] and
//                   adds [a] to the quantity held at [e_value].
//                   Otherwise returns [false].
// Preconditions: If ![e_value.g_state], function exits without
//                   putting and returns [false].
//                   [e_object.get([x],[y])] == [i] || [e_object.get([x],[y])
//                   == SPACE) || [e_object.get([x],[y]) == [HOME] ||
//                   [e_object.get([x],[y]) == [ORE] || [e_object.get([x],[y])
//                   == [CARCASS].
//                   [x] < [e_value.g_SIZE] && [x] >= [0] && [y] <
//                   [e_value.g_SIZE] && [y] >= [0].
// Returns:           [bool].
// Side Effects:      None.
// Errors:            [errlog] error message if [x] or [y] precondition violated.
//                   [e_value.g_state] and [e_object.g_state] may be set to
//                   [false]. Returns [false] if error occurs.

```

```

void rand_all_quantities(const short o, const short min, const short max);

```

```

// Purpose:          For all objects [o] found in [e_object], sets their
//                   quantities in [e_value] to a random value between
//                   [min] and [max].
// Preconditions: If (![e_object.g_state] || ![e_value.g_state]), function
//                   exits without putting values.
//                   [o] >= [MIN_VALUE] && [o] <= [MAX_VALUE].
//                   [min] >= [MIN_VALUE] && [min] <= [MAX_VALUE].
//                   [max] >= [MIN_VALUE] && [max] <= [MAX_VALUE].
//                   [max] > [min].
// Returns:           None.
// Side Effects:      None.
// Errors:            [errlog] error message if precondition violated.

```

```

void set_all_quantities(const short o, const short v);

// Purpose:          For all objects [o] found in [e_object], sets their
//                   quantities in [e_value] to a the value [v].
// Preconditions: If (![e_object.g_state] || ![e_value.g_state]), function
//                   exits without putting values.
//                   [o] >= [MIN_VALUE] && [o] <= [MAX_VALUE].
//                   [v] >= [MIN_VALUE] && [v] <= [MAX_VALUE].
// Returns:          None.
// Side Effects:     None.
// Errors:           [errlog] error message if precondition violated.

void metamorphose(const short s);

// Purpose:          Destroy [e_object] and [e_value].
//                   Then re-create them using the dimension [s].
// Preconditions: [s] >= MIN_DIMENSION && [s] <= MAX_DIMENSION.
// Returns:          None.
// Side Effects:     Every new location value is set to [0].
//                   If succesful, [e_object.g_state] = [true] and
//                   [e_value.g_state] = [true].
// Errors:           [errlog] error message if precondition violation.
//                   [e_object.g_state] = [false] and [e_value.g_state]
//                   = [false].

void clear_objects(const short o);

// Purpose:          Sets all values in [e_object] that are type [o]
//                   to [0] and sets their corresponding values in
//                   [e_value] to [0].
// Preconditions: If ![e_object.g_state], function exits without
//                   clearing.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

void clear_all();

// Purpose:          Sets all values in [e_object] and [e_value]
//                   to [0].
// Preconditions: If ![e_object.g_state], objects will not clear.
//                   If ![e_value.g_state], values will not clear.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const short get_dim();

// Purpose:          Returns the dimension of the environment.
// Preconditions: None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

const bool check_coord(const short x, const short y);

// Purpose:          Analyses whether [x] and [y] are valid
//                   cartesian coordinates for the environment.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           [errlog] error message if invalid coordinates.

```

```

void get_all_objects(Grid &g);

// Purpose:          Copies entire contents of [e_object] to [g].
// Preconditions: None.
// Returns:          None.
// Side Effects:     Caution re grids not same dimension.
// Errors:           None.

const bool get_state();

// Purpose:          If [e_object.get_state()] == [false] ||
//                  [e_value.get_state()] == [false], returns
//                  [false], else returns [true].
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

const bool contains_object(const short v);

// Purpose:          If [e_object] contains [v], returns [true], else
//                  returns [false].
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

private:

//      *** PRIVATE MEMBER VARIABLES ***

Grid e_object;      // objects in the environment
Grid e_value;       // quantity of objects in the environment

};

#endif

//      Filename:      Environment.cpp
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          28 February, 1999
//      Platform:      Visual C++ 5.00

//      Purpose:       See Environment.h for details.

#include "stdafx.h"
#include "Environment.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** OPERATORS ***

Environment & Environment::operator =(const Environment &e)
{

    e_object = e.e_object;
    e_value = e.e_value;

```

```

        return *this;
    }

//      *** PUBLIC FUNCTIONS ***

void Environment::input_objects(ifstream &in)
{
    e_object.input(in);
    e_value.input(in);
}

const void Environment::output_objects(ofstream &out)
{
    e_object.output(out);
    out << endl;
    e_value.output(out);
}

void Environment::put_object(const short x, const short y, const short i, const short q)
{
    e_object.put(x,y,i);
    e_value.put(x,y,q);
}

void Environment::put_object_f(const short x, const short y, const short i, const short q)
{
    e_object.put_f(x,y,i);
    e_value.put_f(x,y,q);
}

const short Environment::get_object(const short x, const short y)
{
    return(e_object.get(x,y));
}

const short Environment::get_object_f(const short x, const short y)
{
    return(e_object.get_f(x,y));
}

const void Environment::get_object(const short x, const short y, short &i, short &q)
{

```

```

        i = e_object.get(x,y);
        q = e_value.get(x,y);
    }

const void Environment::get_object_f(const short x, const short y, short &i, short &q)
{
    i = e_object.get_f(x,y);
    q = e_value.get_f(x,y);
}

void Environment::put_quantity(const short x, const short y, const short q)
{
    e_value.put(x,y,q);
}

void Environment::put_quantity_f(const short x, const short y, const short q)
{
    e_value.put_f(x,y,q);
}

const short Environment::get_quantity(const short x, const short y)
{
    return(e_value.get(x,y));
}

const short Environment::get_quantity_f(const short x, const short y)
{
    return(e_value.get_f(x,y));
}

short Environment::take_quantity(const short x, const short y, const short a)
{
    if(e_object.get(x,y) == SPACE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Environment::take_quantity(x,y,a), An object does.\n";
        err << "not exist at this location. Get operation failed.\n";
        err.close();
        return(0);
    }
}

```

```

short q = e_value.get(x,y);

if(q <= a)

{
    // all used up so adjust environment and record destroyed object

    e_object.put(x,y,SPACE);
    e_value.put(x,y,SPACE);
    return(q);

}

else

{

    q -= a;
    e_value.put(x,y,q);
    return(a);

}

}

```

short Environment::take_quantity_f(const short x, const short y, const short a)

```

{

    short q = e_value.get_f(x,y);

    if(q <= a)

    {
        // all used up so adjust environment and record destroyed object

        e_object.put_f(x,y,SPACE);
        e_value.put_f(x,y,SPACE);
        return(q);

    }

    else

    {

        q -= a;
        e_value.put_f(x,y,q);
        return(a);

    }

}

```

bool Environment::add_quantity(const short x, const short y, const short i, const short a)

```

{

    short t = e_object.get(x,y);
    short q = e_value.get(x,y);

    if(t == i || t == SPACE || t == HOME || t == ORE || t == CARCASS)

    {

```



```

        if((q+a) <= MAX_VALUE)
        {
            e_value.put(x,y,q+a);
            if(t == SPACE)
            {
                e_object.put(x,y,i);
            }
            return(true);
        }
        else
        {
            return(false);
        }
    }
    else
    {
        return(false);
    }
}

bool Environment::add_quantity_f(const short x, const short y, const short i, const short a)
{
    short t = e_object.get_f(x,y);
    short q = e_value.get_f(x,y);

    if((q+a) <= MAX_VALUE)
    {
        e_value.put_f(x,y,q+a);
        if(t == SPACE)
        {
            e_object.put_f(x,y,i);
        }
        return(true);
    }
}

```

```

else
{
    return(false);
}
}

void Environment::rand_all_quantities(const short o, const short min, const short max)
{
    if(o < MIN_VALUE || o > MAX_VALUE || min < MIN_VALUE || min > MAX_VALUE ||
        max < MIN_VALUE || max > MAX_VALUE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Environment::rand_all_quantities(o,min,max), An\n";
        err << "argument is out of range. Randomisation of all quantities failed.\n";
        err.close();
        return;
    }

    if((max - min) <= 0)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Environment::rand_all_quantities(o,min,max), [max]\n";
        err << "must be greater than [min]. Randomisation of all quantities failed.\n";
        err.close();
        return;
    }

    short rand_e = max - min;
    short c1,c2;
    short dim = e_object.get_dim();

    for(c1 = 0 ; c1 < dim ; c1++)
    {
        for(c2 = 0 ; c2 < dim ; c2++)
        {
            if(e_object.get_f(c1,c2) == o)
            {
                e_value.put_f(c1,c2,(rand() % rand_e) + min);
            }
        }
    }
}

```

```

}

void Environment::set_all_quantities(const short o, const short v)
{
    if(o < MIN_VALUE || o > MAX_VALUE || v < MIN_VALUE || v > MAX_VALUE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Environment::set_all_quantities(o,v), An\n";
        err << "argument is out of range. Setting of all quantities failed.\n";
        err.close();
        return;
    }

    short c1,c2;
    short dim = e_object.get_dim();

    for(c1 = 0 ; c1 < dim ; c1++)
    {
        for(c2 = 0 ; c2 < dim ; c2++)
        {
            if(e_object.get_f(c1,c2) == o)
            {
                e_value.put_f(c1,c2,v);
            }
        }
    }
}

void Environment::metamorphose(const short s)
{
    e_object.metamorphose(s);
    e_value.metamorphose(s);
}

void Environment::clear_objects(const short o)
{
    set_all_quantities(o,0);
    e_object.replace_all_instances(o,0);
}

void Environment::clear_all()

```

```

{
    e_object.wipe(0);
    e_value.wipe(0);
}

const short Environment::get_dim()
{
    return(e_object.get_dim());
}

const bool Environment::check_coord(const short x, const short y)
{
    return(e_object.check(x,y));
}

void Environment::get_all_objects(Grid &g)
{
    g = e_object;
}

const bool Environment::get_state()
{
    return(e_object.get_state() && e_value.get_state());
}

const bool Environment::contains_object(const short v)
{
    return(e_object.find_instance(v));
}

```

This is a “wrapper” class and utilises much of the functionality attached to the Grid instances that it possesses. As in the Grid class, we provided faster access and retrieval functions once our confidence had grown to a significant level.

C.3 The Intent Class

First we provide the code for the Intent class. Then we briefly discuss its most salient features.

```
//      Filename:          Intent.h
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

//      Symbology:         Numbers, functions and data variables are enclosed in
//                          square parentheses "[ ]". This is to remove any
//                          ambiguity within the text of comments. All private data
//                          member variables are prefixed by "i_".

//      Errors:            All error messages appended to "errlog.txt"

//      Purpose:           Generic intention structure for agents. After an agent has
//                          decided its action, it loads its [Intent] with the action. The
//                          action may then be arbitrated. [i_key] is a special sorting key
//                          that is generated by the class based on [i_x] and [i_y]. In
//                          order to sort a list of [Intent]'s, one requires a unique mapping
//                          from an ([x],[y]) pair to a number [n]. This number allows comparison
//                          between two [Intent] objects on the basis of their [i_x] and [i_y]
//                          coordinates. If they have the same coordinates, they will have the
//                          same [n], otherwise [i_key] will differ. This mapping allows a finite
//                          contiguous list from [0] to [i_const] * [i_const].

// Function Comment Style:

// Purpose:                Purpose of the function.
// Preconditions:          What states or conditions must be met before execution
//                          of the function.
// Returns:                The object or primitive type that the function returns.
// Side Effects:           Effects that the function may have on private data that
//                          are not immediately obvious to the nature of the function.
//                          This section may also contain advice and miscellaneous
//                          comments.
// Errors:                What type of error is generated, why it is generated and
//                          its effects.

//      *** PRECONDITIONS WITHIN THE SCOPE OF THIS CLASS ***

// All variables and parameters are assumed to have positive values, despite the fact
// that they may have negative values due to their type. Using negative numbers may
// have unpredictable or undesired results.

#ifndef __INTENT_H
#define __INTENT_H

#include <fstream>          // error logs
#include <stdlib.h>         // for exit from the program
#include "Grid.h"          // for calculating [MAX_KEY]

//      *** GLOBALS ***

const short MAX_PRIORITY = 3000;                                // maximum [i_priority]
```

```

const short MAX_KEY = MAX_DIMENSION * MAX_DIMENSION;           // the maximum [i_key] value

//      *** CLASS DEFINITION ***

class Intent
{
    public:

    //      *** CONSTRUCTORS ***

    Intent(): i_name(0), i_action(0), i_priority(0), i_x(0), i_y(0),
              i_key(0), i_const(0){};

    // Purpose:           Default constructor.
    // Preconditions:None.
    // Returns:           An [Intent] object.
    // Side Effects:       Caution, because [i_const] has not been defined,
    //                     [i_key] is nonsense. Use [put] to set all of the
    //                     attributes of the object before any further use
    //                     of the class instance.
    // Errors:            None.

    Intent(short n, short a, short p, short x, short y, short c);

    // Purpose:           Constructs an intent object setting members as
    //                     follows: [i_name] gets [n], [i_action] gets [a]
    //                     [i_priority] gets [p], [i_x] gets [x], [i_y] gets [y]
    //                     [i_const] gets [c]. The member [i_const] allows the class
    //                     to generate a unique key in [i_key] based on [i_x]
    //                     and [i_y].
    // Preconditions:[c] >= MIN_DIMENSION && [c] <= MAX_DIMENSION.
    //                     [p] >= MAX_PRIORITY && [p] >= [0].
    //                     As [n], [a], [x] and [y] should already be implicitly
    //                     range checked in the associated classes, no checking
    //                     of their values occurs here.
    // Returns:           An [Intent] object.
    // Side Effects:       [i_key] is generated, see private function [set_key()].
    // Errors:            [errlog] error message if precondition violation.
    //                     Program then terminates.

    Intent(const Intent &i);

    // Purpose:           Copy constructor. Destructively assigns [i] to the
    //                     current context.
    // Preconditions:None.
    // Returns:           An [Intent] object.
    // Side Effects:       None.
    // Errors:            None.

    //      *** OPERATORS ***

    Intent & operator =(const Intent &i);

    // Purpose:           Destructive assignment.
    // Preconditions:None.
    // Returns:           None.
    // Side Effects:       None.
    // Errors:            None.

    const friend bool operator ==(const Intent &a, const Intent &b)

```

```

// Purpose:          Deep equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    // intents are equal if same action, in same location
    // with same coordinate key

    return(a.i_action == b.i_action && a.i_key == b.i_key);

}

const friend bool operator <(const Intent &a, const Intent &b)

// Purpose:          Less-than operator. An Intent is less than another
//                    Intent if its action is lower in value. Further sorting
//                    predicates are defined after the class definition in
//                    this file.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.i_action < b.i_action);

};

const friend bool operator >(const Intent &a, const Intent &b)

// Purpose:          Greater-than operator. An Intent is greater than another
//                    Intent if its action is higher.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.i_action > b.i_action);

};

// *** PUBLIC FUNCTION PROTOTYPES ***

const void get(short &n, short &a, short &p, short &x, short &y);

// Purpose:          [n] gets [i_name], [a] gets [i_action],
//                    [p] gets [i_priority], [x] gets [i_x],
//                    [y] gets [i_y].
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

const void get(short &a, short &x, short &y);

// Purpose:          [a] gets [i_action], [x] gets [i_x], [y] gets [i_y].
// Preconditions: None.

```

```

// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const void get(short &x, short &y);

// Purpose:          [x] gets [i_x] and [y] gets [i_y].
// Preconditions:     None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const short get_name();

// Purpose:          Returns [i_name].
// Preconditions:     None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

const void get_action(short &a, short &p, short &k);

// Purpose:          [a] gets [i_action], [p] gets [i_priority],
//                   [k] gets [i_key].
// Preconditions:     None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const short get_action();

// Purpose:          Returns [i_action].
// Preconditions:     None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

const short get_priority();

// Purpose:          Returns [i_priority].
// Preconditions:     None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

const short get_key();

// Purpose:          Returns [i_key].
// Preconditions:     None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

void put(const short n, const short a, const short p, const short x,
        const short y, const short c);

// Purpose:          [n] puts [i_name], [a] puts [i_action],
//                   [p] puts [i_priority], [x] puts [i_x],
//                   [y] puts [i_y], [c] puts [i_const].
// Preconditions:     [p] >= MAX_PRIORITY && [p] >= [0].
//                   [c] >= MIN_DIMENSION && [c] <= MAX_DIMENSION.
// Returns:          None.
// Side Effects:     [i_key] is generated, see private function

```



```

//          [set_key()].
// Errors:   See [set_key()].

void put(const short a, const short x, const short y);

// Purpose:   [a] puts [i_action], [x] puts [i_x], [y] puts [i_y].
// Preconditions: None.
// Returns:   None.
// Side Effects: [i_key] is generated, see private function
//          [set_key()].
// Errors:   See [set_key()].

void inc_priority();

// Purpose:   Add one to [i_priority].
// Preconditions: None.
// Returns:   None.
// Side Effects: None.
// Errors:   Caution [errlog] message if [i_priority] exceeds
//          [MAX_PRIORITY]. [i_priority] becomes 0.

void set_name(const short n);

// Purpose:   Changes [i_name] to [n].
// Preconditions: None.
// Returns:   None.
// Side Effects: None.
// Errors:   None.

void set_action(const short a);

// Purpose:   Changes [i_action] to [a].
// Preconditions: None.
// Returns:   None.
// Side Effects: None.
// Errors:   None.

void reset_priority();

// Purpose:   Sets [i_priority] to 0.
// Preconditions: None.
// Returns:   None.
// Side Effects: None.
// Errors:   None.

private:

//      *** PRIVATE VARIABLES ***

short i_name;          // agent's name
short i_action;        // agent's intended action
short i_priority;      // priority of that action
short i_x;             // x-coordinate where action is to take place
short i_y;             // y-coordinate where action is to take place
short i_key;           // unique number based on i_x,i_y and i_const
// generated by the class
short i_const;         // max value that i_x and i_y may have

//      *** PRIVATE FUNCTION PROTOTYPES ***

void set_key();

// Purpose:   Generate the value for [i_key] based on [i_x],

```

```

//                                     [i_y] and [i_const].
// Preconditions: None.
// Returns:      None.
// Side Effects:  None.
// Errors:       If [i_key] is greater than [MAX_KEY], program
//               terminates. [errlog] message generated.

};

//      *** COMPARISON PREDICATES ***

// The following classes provide additional comparison predicates involved
// in sorting a collection of [Intent] objects.

class compare_intent_names
{
public:
    bool operator () (Intent &a, Intent &b)
    {
        return(a.get_name() < b.get_name());
    }
};

class compare_intent_keys
{
public:
    bool operator () (Intent &a, Intent &b)
    {
        return(a.get_key() < b.get_key());
    }
};

class compare_intent_priorities
{
public:
    bool operator () (Intent &a, Intent &b)
    {
        return(a.get_priority() > b.get_priority());
    }
};

#endif

```

```

//      Filename:          Intent.cpp
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00
//      Purpose:           See Intent.h

#include "stdafx.h"
#include "Intent.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** CONSTRUCTORS ***

Intent::Intent(const Intent &i)

{
    i_name = i.i_name;
    i_action = i.i_action;
    i_priority = i.i_priority;
    i_x = i.i_x;
    i_y = i.i_y;
    i_key = i.i_key;
    i_const = i.i_const;
}

Intent::Intent(short n, short a, short p, short x, short y, short c)
: i_name(n), i_action(a), i_priority(p), i_x(x), i_y(y), i_const(c)

{
    if(p < 0 || p > MAX_PRIORITY)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Intent::Intent(n,a,p,x,y,c), [p] is out\n";
        err << "of range. FORCED EXIT!\n";
        err.close();
        exit(1);
    }

    if(c < MIN_DIMENSION || c > MAX_DIMENSION)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Intent::Intent(n,a,p,x,y,c), [c] is out\n";
        err << "of range. FORCED EXIT!\n";
        err.close();
        exit(1);
    }

    set_key();
}

```

```

}

//      *** OPERATORS ***

Intent & Intent::operator =(const Intent &i)
{
    i_name = i.i_name;
    i_action = i.i_action;
    i_priority = i.i_priority;
    i_x = i.i_x;
    i_y = i.i_y;
    i_key = i.i_key;
    i_const = i.i_const;

    return(*this);
}

//      *** PUBLIC FUNCTIONS ***

const void Intent::get(short &n, short &a, short &p, short &x, short &y)
{
    n = i_name;
    a = i_action;
    p = i_priority;
    x = i_x;
    y = i_y;
}

const void Intent::get(short &a, short &x, short &y)
{
    a = i_action;
    x = i_x;
    y = i_y;
}

void Intent::reset_priority()
{
    i_priority = 0;
}

const void Intent::get(short &x, short &y)
{
    x = i_x;
    y = i_y;
}

const short Intent::get_name()

```

```

{
    return(i_name);
}

const void Intent::get_action(short &a, short &p, short &k)
{
    a = i_action;
    p = i_priority;
    k = i_key;
}

const short Intent::get_action()
{
    return(i_action);
}

const short Intent::get_priority()
{
    return(i_priority);
}

const short Intent::get_key()
{
    return(i_key);
}

void Intent::put(const short n, const short a, const short p, const short x,
                const short y, const short c)
{
    i_name = n;
    i_action = a;
    i_priority = p;
    i_x = x;
    i_y = y;
    i_const = c;
    set_key();
}

void Intent::put(const short a, const short x, const short y)
{
    i_action = a;
    i_x = x;
    i_y = y;
}

```

```

        set_key();
    }

void Intent::inc_priority()
{
    i_priority++;

    if(i_priority > MAX_PRIORITY)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\nCAUTION, Intent::inc_priority(), i_priority exceeding ";
        err << MAX_PRIORITY << " .. set to 0.\n";
        err.close();
        i_priority = 0;
    }
}

void Intent::set_name(const short n)
{
    i_name = n;
}

void Intent::set_action(const short a)
{
    i_action = a;
}

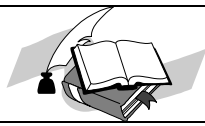
//      *** PRIVATE FUNCTIONS ***

void Intent::set_key()
{
    i_key = (i_x + (i_y * i_const));

    if(i_key > MAX_KEY)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Intent::set_key(), coordinates exceed maximum range.\n";
        err << "FORCED EXIT!\n";
        err.close();
        exit(1);
    }
}

```

This class provides a definition for an action that an agent “wishes” to perform. We have used the same coordinate mapping described in Appendix T.2, to enable coordinate locations to be sorted and compared easily. Deciding the highest priority that an action may have is a “black art”. The value of 100 that we set for MAX_PRIORITY resulted from experimentation. Much of the functionality for this class is in the form of getting and putting functions. Essentially, the idea was to provide a structure that could be passed around the system and be used by various components.



T.2 - The Grid Class Implementation - Page 160

C.4 The Agent Class

First we provide the code for the Agent class, which is the base class for all agents. Then we discuss its most salient features. A fairly detailed description of agents and the simulation process occurs in Appendix T.1.



T.1 - The Base Class Agent - Page 152

```
//      Filename:      Agent.h
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:     Numbers, functions and data variables are enclosed in
//                      square parentheses "[ ]". This is to remove any
```

```
//
//      ambiguity within the text of comments. All private data
//      member variables are prefixed by "ag_".

//      Errors:      All error messages appended to "errlog.txt".

//      Purpose:      Base class of agents. This class contains the attributes and
//                    functionality that are inherent in every agent. All agents
//                    must derive from this class.
```

```
// Function Comment Style:
```

```
// Purpose:      Purpose of the function.
// Preconditions: What states or conditions must be met before execution
//               of the function.
// Returns:      The object or primitive type that the function returns.
// Side Effects: Effects that the function may have on private data that
//               are not immediately obvious to the nature of the function.
//               This section may also contain advice and miscellaneous
//               comments.
// Errors:      What type of error is generated, why it is generated and
//               its effects.
```

```
#ifndef __AGENT_H
#define __AGENT_H
```

```
#include "Environment.h" // for agents memory, environment access, object/action
                        // symbology
#include "Intent.h"      // agents' Intent structure for arbitration
#include "DataStructures.h" // various useful data structures
#include <fstream>        // errors
#include <stddef.h>       // null definition re pointers and exit
#include <stdlib.h>       // random numbers
#include <algorithm>      // STL container functions
```

```
//      *** GLOBALS / TYPE DEFINITIONS ***
```

```
// The following are the eight possible movement transitions from any
// coordinate in the grid. [moves_f] represents a faster move
```

```
const short MOVE_DIM = 16;
const short moves[MOVE_DIM] = {0,-1,1,-1,1,0,1,1,0,1,-1,1,-1,0,-1,-1};
const short moves_f[MOVE_DIM] = {0,-2,2,-2,2,0,2,2,0,2,-2,2,-2,0,-2,-2};
```

```
// UP = X+0, Y-1
// UP-RIGHT = X+1, Y-1
// RIGHT = X+1, Y+0
// DOWN-RIGHT = X+1, Y+1
// DOWN = X+0, Y+1
// DOWN-LEFT = X-1, Y+1
// LEFT = X-1, Y+0
// UP-LEFT = X-1, Y-1
```

```
//      *** CLASS DEFINITION ***
```

```
class Agent
```

```
{
```

```
    public:
```

```
    //      *** CONSTRUCTORS ***
```

```
    Agent() : ag_x(0), ag_y(0), ag_env(NULL), ag_m(40), ag_s(40), ag_name(0),
```



```

        ag_v_range(3), ag_type(MINER), ag_alive(true), ag_target_x(0),
        ag_target_y(0), ag_target(false), ag_target_obj(0),
        ag_orig_type(MINER), ag_dim(40), ag_search_prep(false){};

// Purpose:          Default constructor.
// Preconditions: None.
// Returns:          None.
// Side Effects:      It is important to define values and alter the dimension
//                   of the Agent's memory before using the Agent. See
//                   [reset_environment(args)]. Also see [record_position(args)]
//                   and [init(args)].
// Errors:           None.

Agent(const short s, const short x, const short y, Environment *e, const short n,
      const short r, const short t);

// Purpose:          Creates an Agent. [ag_m] and [ag_s] get [s] as their dimension,
//                   [ag_x] gets [x], [ag_y] gets [y], [ag_env] gets [e], [ag_dim]
//                   gets [s], [ag_name] gets [n], [ag_v_range] gets [r], [ag_type]
//                   gets [t]. The Agent's [ag_intent] is initialised to its current
//                   coordinates and name with the action [NOTHING]. All elements in
//                   [ag_m] are set to [UNKNOWN]. [erase_target()] is invoked.
//                   [ag_alive] is set to [true].
// Preconditions: [s] == [[e] -> get_dim()].
//                   [x] >= 0 && [x] < [[e] -> get_dim()].
//                   [y] >= 0 && [y] < [[e] -> get_dim()].
//                   [e] -> get_object([x],[y]) == [SPACE].
// Returns:          [Agent].
// Side Effects:      Agent has not recorded its position in the environment or its memory.
//                   Use [record_position()].
// Errors:           [errlog] error message if precondition violation. Possible
//                   program exit.

Agent(const Agent &a);

// Purpose:          Copy constructor.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

//      *** OPERATORS ***

virtual Agent & operator =(const Agent &a);

// Purpose:          Destructively assigns [a] to the current context.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

const friend bool operator ==(const Agent &a, const Agent &b)

// Purpose:          Equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.ag_name == b.ag_name);
}

```

```

}

const friend bool operator !=(const Agent &a, const Agent &b)

// Purpose:          Non-equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const Agent &a, const Agent &b)

// Purpose:          Less-than operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.ag_name < b.ag_name);

};

const friend bool operator >(const Agent &a, const Agent &b)

// Purpose:          Greater-than operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.ag_name > b.ag_name);

};

//      *** PUBLIC FUNCTION PROTOTYPES ***

//      ** ADMINISTRATION FUNCTIONS **

virtual void record_position();

// Purpose:          Agent records its position in the environment and its memory
//                   on the basis of [ag_x] and [ag_y]. It also sets its intent
//                   to [NOTHING] with its current location. Note that the agent
//                   will record its quantity as [1] in [ag_env].
// Preconditions: [ag_x] >= 0 && [ag_x] < [[ag_env] -> get_dim()].
//                   [ag_y] >= 0 && [ag_y] < [[ag_env] -> get_dim()].
//                   [ag_env] -> get-object([ag_x],[ag_y]) == [SPACE].
// Returns:          None.
// Side Effects:      None.
// Errors:          [errlog] error message if precondition violation. Possible
//                   program exit.

void get_intent(Intent &i);

```

```
// Purpose:          Loads [i] with [ag_intent], the Agent's intended action.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.
```

```
void put_intent(Intent i);
```

```
// Purpose:          Loads [ag_intent] with [i], the Agent's intended action.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.
```

```
Intent get_intent();
```

```
// Purpose:          Returns [ag_intent], the Agent's intended action.
// Preconditions: None.
// Returns:          [Intent].
// Side Effects:      None.
// Errors:           None.
```

```
virtual void init(const short n, const short x, const short y, const short t);
```

```
// Purpose:          Sets [ag_name] to [n], [ag_x] to [x], [ag_y] to [y] and
//                   [ag_type] to [t].
// Preconditions: None.
// Returns:          None.
// Side Effects:      The Agent's [ag_intent] is totally reset.
// Errors:           None.
```

```
virtual void init(const short x, const short y);
```

```
// Purpose:          Sets [ag_x] to [x] and [ag_y] to [y].
// Preconditions: None.
// Returns:          None.
// Side Effects:      The Agent's [ag_intent] is altered to action [NOTHING]
//                   and the Agent's new coordinates.
// Errors:           None.
```

```
void get_location(short &x, short &y);
```

```
// Purpose:          [x] gets [ag_x], [y] gets [ag_y]. The Agent's current
//                   location.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.
```

```
virtual void kill();
```

```
// Purpose:          Sets [ag_alive] to [false]. Sets [ag_type] to [CARCASS]. The
//                   value of the [CARCASS] is set to [200].
// Preconditions: None.
// Returns:          None.
// Side Effects:      Records new type and quantity in thr environment [ag_env] and
//                   sets action to [NOTHING].
// Errors:           None.
```

```
const bool is_alive();
```

```
// Purpose:          Returns [ag_alive]. Whether Agent is alive.
```

```

// Preconditions: None.
// Returns: [bool].
// Side Effects: None.
// Errors: None.

const short get_name();

// Purpose: Returns [ag_name]. The Agent's name.
// Preconditions: None.
// Returns: [short].
// Side Effects: None.
// Errors: None.

const short get_type();

// Purpose: Returns [ag_type]. The Agent's type.
// Preconditions: None.
// Returns: [short].
// Side Effects: None.
// Errors: None.

const short get_original_type();

// Purpose: Returns [ag_orig_type]. The Agent's type. Agent's type
// will change upon death. This is useful in determining what
// type it was originally.
// Preconditions: None.
// Returns: [short].
// Side Effects: None.
// Errors: None.

virtual void get_state(short &t, short &n, short &x, short &y, short &a,
                      short &q, short &tx, short &ty);

// Purpose: Loads [t] with [ag_type], [n] with [ag_name], [x] with [ag_x],
// [y] with [ag_y], [a] with the action within [ag_intent]. If
// [ag_target] == [true], loads [tx] with [ag_target_x] and [ty]
// with [ag_target_y], else both set to [-1]. [q] is set to [0]
// but can be overridden to supply a quantity of something that
// the agent is carrying or has digested etc.
// Preconditions: None.
// Returns: None.
// Side Effects: None.
// Errors: None.

void get_environment();

// Purpose: Agent loads entire environment [ag_env] into its memory [ag_m].
// Preconditions: None.
// Returns: None.
// Side Effects: None.
// Errors: None.

// ** BEHAVIOURS AND ABILITIES **

virtual void think(){ag_search_prep = false;};

// Purpose: The antecedent/consequent rule/behaviour structure of
// the Agent, defined by the user. This is the main control structure
// called by the [Administrator].
// Preconditions: You must invoke the base definition in a derived object at the start
// of the code in your definition. This allows the invariance of
// [ag_search_prep] to be maintained.

```

```

// Returns:          None.
// Side Effects:     None.
// Errors:           None.

void look();

// Purpose:          Causes the Agent to look at its environment. Agents have
//                   360 degree vision that has the boundary of a square box
//                   limited by [ag_v_range] centered on the Agent's coordinates.
//                   Agents see everything in the box, whether they know what the
//                   things are is up to you. Agents see regardless of obstacles.
// Preconditions: None.
// Returns:          None.
// Side Effects:     The Agent's memory [ag_m] is overwritten with the data fetched
//                   from [ag_env] according to the dimensions of the box. Therefore,
//                   the Agent has a subjective view of the world according to what
//                   it last looked at. If the Agent sees its currently targetted
//                   object location and the targetted object is no longer present,
//                   targetting is cancelled ([erase_target()]).
// Errors:           None.

void set_target(const short x, const short y, const short o);

// Purpose:          Makes [o] a target. [ag_target_x] gets [x], [ag_target_y] gets [y],
//                   [ag_target_obj] gets [o].
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

void erase_target();

// Purpose:          [ag_target_x] gets [-1], [ag_target_y] gets [-1],
//                   [ag_target_obj] gets [-1], [ag_target] set to [false].
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

bool seek_target(const short x, const short y, const short o);

// Purpose:          Decide a best move that gets the Agent closer to the
//                   target object [o] located at [x] and [y].
// Preconditions: None.
// Returns:          [bool]. [true] if action is a possible [MOVE], else
//                   [false].
// Side Effects:     [ag_intent] is set to the best move if returning
//                   [true]. [set_target(args)] is invoked using [x], [y]
//                   and [o] if [true].
// Errors:           None.

bool targetting();

// Purpose:          Returns [ag_target].
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

virtual void execute_move();

// Purpose:          Executes a move based on contents of [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.

```

```

// Returns:          None.
// Side Effects:     Agent sets its new position in environment and memory.
// Errors:           [errlog] message if location specified in [ag_intent] is not
//                  a [SPACE] in [ag_env]. Action fails and [ag_action] set to
//                  [NOTHING].

virtual void execute_action(){};

// Purpose:          Causes the Agent to execute an action based on the contents
//                  of its [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:          None.
// Side Effects:     Varies on inherited behaviour.
// Errors:           None.

void do_nothing();

// Purpose:          Causes Agent to set its [ag_intent] to [NOTHING] with its
//                  current coordinates [ag_x] and [ag_y]. Agent therefore
//                  intends to do nothing.
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

void expire_moving_objects();

// Purpose:          All Agents in Agent's memory are deleted and replaced with
//                  [SPACE].
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

bool wander(short &x, short &y);

// Purpose:          Agent relatively selects a reachable location in its memory
//                  and loads [x] and [y] with the target location as a place to
//                  move towards. [UNKNOWN] regions that are closest have
//                  priority.
// Preconditions: None.
// Returns:          [bool]. [true] if a reachable place is found.
// Side Effects:     None.
// Errors:           None.

bool adjacent_to(const short o);

// Purpose:          Determines whether the agent is adjacent to the
//                  object [o].
// Preconditions: None.
// Returns:          [bool]. [true] if adjacent.
// Side Effects:     None.
// Errors:           None.

bool adjacent_to(const short x, const short y);

// Purpose:          Determines whether the agent is adjacent to the
//                  coordinates specified by [x] and [y] by comparison
//                  with [ag_x] and [ag_y].
// Preconditions: None.
// Returns:          [bool]. [true] if adjacent.
// Side Effects:     None.
// Errors:           None.

```

```

bool adjacent_to(short &x, short &y, const short o);

// Purpose:          Determines whether the agent is adjacent to the
//                  coordinates specified by [x] and [y] by comparison
//                  with [ag_x] and [ag_y] and that the location contains
//                  the object [o].
// Preconditions: None.
// Returns:          [bool]. [true] if adjacent.
// Side Effects:     None.
// Errors:           None.

bool knows_of(const short o);

// Purpose:          Whether the agent knows of object [o], anywhere in
//                  its memory [ag_m].
// Preconditions: None.
// Returns:          [bool]. [true] if [o] found in [ag_m].
// Side Effects:     None.
// Errors:           None.

bool knows_of_closest(short &x, short &y, const short o);

// Purpose:          If [o] exists in [ag_m], finds the closest [o] and
//                  loads [x] and [y] with its coordinates.
// Preconditions: None.
// Returns:          [bool]. [true] if [o] found in [ag_m].
// Side Effects:     None.
// Errors:           None.

protected:

//      *** PROTECTED DATA ***

//      ** MEMORY: KNOWN OBJECTS DATA STRUCTURES **

Grid ag_m;           // object memory
Grid ag_s;           // search memory
short ag_dim;        // the dimension of memory & environment

// ** MEMORY: OBJECT TARGETTING AND SEARCH PREPARATION DATA STRUCTURES **

bool ag_target;       // whether Agent is targetting something
short ag_target_obj;  // object targetted
short ag_target_x;    // a target x-coord
short ag_target_y;    // a target y-coord
bool ag_search_prep;  // indicates whether total search flood has been
                    // performed at agents location. Used by knows_of

//      ** ENVIRONMENT DATA STRUCTURES **

Environment *ag_env;  // pointer to Agent's environment
short ag_x;           // current x-coordinate
short ag_y;           // current y_coordinate

//      ** ATTRIBUTE DATA STRUCTURES **

short ag_type;        // type of Agent
short ag_orig_type;   // copy of type - used when scenario restarted
short ag_name;        // name
short ag_v_range;     // vision range
bool ag_alive;        // whether Agent is alive
PATH ag_search;       // used in searching

```

```

//      ** INTENTION DATA STRUCTURES **

Intent ag_intent;// next intended action collected for arbitration

//      *** PROTECTED FUNCTION PROTOTYPES ***

void update_search_memory();

// Purpose:          Copies [ag_m] to [ag_s] setting all objects to -1, and all
//                   [SPACE] to [0] in [ag_s].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

void execute_search_flood(short x, short y);
void execute_partial_search_flood(short x, short y);

// Purpose:          Floods [ag_s] with values that are representative of the
//                   shortest path to [x] and [y]. The partial version will
//                   only compute as far as it is necessary to plot a path
//                   from the Agent's current coordinates [ag_x] and [ag_y].
// Preconditions: [update_search_memory()] has been called.
// Returns:          None.
// Side Effects:      None.
// Errors:            None.

bool next_to(const short x, const short y, const short o);

// Purpose:          Tests whether [o] is adjacent to [x] and [y]. This
//                   is used to find unknown areas, [UNKNOWN], in [ag_m].
// Preconditions: None.
// Returns:          [bool]. [true] if [o] is adjacent.
// Side Effects:      None.
// Errors:            None.

};

#endif

//      Filename:      Agent.cpp
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00

//      Purpose:       Base class of agents. See Agent.h

#include "stdafx.h"
#include "Agent.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** CONSTRUCTORS ***

Agent::Agent(const short s, const short x, const short y, Environment *e,
             const short n, const short r, const short t): ag_x(x), ag_y(y),
             ag_env(e), ag_m(s), ag_s(s), ag_name(n), ag_v_range(r),

```



```

        ag_type(t), ag_alive(true), ag_orig_type(t), ag_dim(s),
        ag_search_prep(false)

{

    // initialise memory to unknown
    ag_m.wipe(UNKNOWN);

    // allocate and resize search list
    ag_search.reserve((s*s)+s);
    ag_search.resize((s*s)+s);

    if(!(ag_env -> check_coord(ag_x, ag_y)))

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Agent::Agent(s,x,y,*e,n,r,t), Agent\n";
        err << ag_name << " has been placed at an invalid location.\n";
        err << "FORCED EXIT!\n";
        err.close();
        exit(1);

    }

    // set initial intent
    ag_intent.put(n,NOTHING,0,x,y,s);

    // erase targetting
    erase_target();

}

Agent::Agent(const Agent &a)

{

    ag_m = a.ag_m;           // Agent's memory
    ag_s = a.ag_s;           // search memory
    ag_x = a.ag_x;           // current x-coordinate
    ag_y = a.ag_y;           // current y_coordinate
    ag_target_x = a.ag_target_x; // Agent's target x
    ag_target_y = a.ag_target_y; // Agent's target y
    ag_env = a.ag_env;       // Agent's environment
    ag_name = a.ag_name;     // Agent's name
    ag_v_range = a.ag_v_range; // Agent's vision range
    ag_intent = a.ag_intent; // Agent's next intended action
    ag_type = a.ag_type;     // Agent's type
    ag_orig_type = a.ag_orig_type; // copy of type
    ag_alive = a.ag_alive;   // alive or dead
    ag_target = a.ag_target; // whether an object is targetted by Agent
    ag_target_obj = a.ag_target_obj; // the object targetted
    ag_dim = a.ag_dim;       // memory & environment dimension
    ag_search = a.ag_search; // search list
    ag_search_prep = a.ag_search_prep; // whether total search done at ag_x,ag_y

}

//      *** OPERATORS ***

Agent & Agent::operator =(const Agent &a)

{

```

```

    ag_m = a.ag_m;
    ag_s = a.ag_s;
    ag_x = a.ag_x;
    ag_y = a.ag_y;
    ag_target_x = a.ag_target_x;
    ag_target_y = a.ag_target_y;
    ag_env = a.ag_env;
    ag_name = a.ag_name;
    ag_v_range = a.ag_v_range;
    ag_intent = a.ag_intent;
    ag_type = a.ag_type;
    ag_orig_type = a.ag_orig_type;
    ag_alive = a.ag_alive;
    ag_target = a.ag_target;
    ag_target_obj = a.ag_target_obj;
    ag_dim = a.ag_dim;
    ag_search = a.ag_search;
    ag_search_prep = a.ag_search_prep;

    return(*this);
}

//      *** PUBLIC FUNCTION PROTOTYPES

void Agent::record_position()
{
    // check position according to [ag_x] and [ag_y]

    if(ag_env -> get_object(ag_x, ag_y) != SPACE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Agent::record_position(), Agent has been placed at a\n";
        err << "location that does not contain SPACE.\nFORCED EXIT!\n";
        err.close();
        exit(1);
    }

    // record position and set intention to NOTHING

    ag_intent.put(ag_name,NOTHING,0,ag_x,ag_y,ag_dim);
    ag_m.put(ag_x,ag_y,ag_type);
    ag_env -> put_object(ag_x,ag_y,ag_type,1);
}

void Agent::get_intent(Intent &i)
{
    i = ag_intent;
}

void Agent::put_intent(Intent i)
{

```

```

        ag_intent = i;
    }

Intent Agent::get_intent()
{
    return(ag_intent);
}

void Agent::init(const short n, const short x, const short y, const short t)
{
    ag_name = n;
    ag_x = x;
    ag_y = y;
    ag_type = t;
    ag_intent.put(ag_name,NOTHING,0,ag_x,ag_y,ag_dim);
    ag_dim = ag_m.get_dim();
}

void Agent::init(const short x, const short y)
{
    ag_x = x;
    ag_y = y;
    ag_intent.put(NOTHING,ag_x,ag_y);
    ag_dim = ag_m.get_dim();
}

void Agent::get_location(short &x, short &y)
{
    x = ag_x;
    y = ag_y;
}

void Agent::kill()
{
    ag_alive = false;
    ag_type = CARCASS;
    // make a 200 point value carcass
    ag_env -> put_object_f(ag_x,ag_y,ag_type,200);
}

const bool Agent::is_alive()
{
    return(ag_alive);
}

```

```

}

const short Agent::get_name()

{

    return(ag_name);

}

const short Agent::get_type()

{

    return(ag_type);

}

const short Agent::get_original_type()

{

    return(ag_orig_type);

}

void Agent::get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                    short &tx, short &ty)

{

    t = ag_type;
    n = ag_name;
    x = ag_x;
    y = ag_y;
    a = ag_intent.get_action();
    q = 0;

    if(ag_target)

    {

        tx = ag_target_x;
        ty = ag_target_y;

    }

    else

    {

        tx = -1;
        ty = -1;

    }

}

void Agent::get_environment()

{

    ag_env -> get_all_objects(ag_m);

```

```

}

//      ** BEHAVIOURS AND ABILITIES **

void Agent::look()
{
    short lim = ag_v_range*2;
    short x = ag_x - ag_v_range;
    short y = ag_y - ag_v_range;
    short c1,c2,cx,cy;

    for(c1 = 0; c1 <= lim ; c1++)
    {
        for(c2 = 0 ; c2 <= lim ; c2++)
        {
            cx = x + c1;
            cy = y + c2;

            if(ag_m.check(cx,cy)) // is this a valid position
            {
                ag_m.put_f(cx,cy,ag_env -> get_object_f(cx,cy));
            }
        }
    }

    // sort out targets that have gone
    if(ag_target)
    {
        if(ag_target_x >= x && ag_target_x <= x + lim &&
           ag_target_y >= y && ag_target_y <= y + lim)
        {
            if(ag_m.get_f(ag_target_x,ag_target_y) != ag_target_obj)
            {
                erase_target();
            }
        }
    }
}

void Agent::set_target(const short x, const short y, const short o)

```

```

{
    ag_target_x = x;
    ag_target_y = y;
    ag_target = true;
    ag_target_obj = o;
}

void Agent::erase_target()
{
    ag_target_x = -1;
    ag_target_y = -1;
    ag_target = false;
    ag_target_obj = -1;
}

bool Agent::seek_target(const short x, const short y, const short o)
{
    // if already adjacent to target, not worth seeking it
    if(adjacent_to(x,y))
    {
        erase_target();
        return(false);
    }

    // world is dynamic - so we should search at each iteration
    update_search_memory();

    // need to flood search grid
    execute_partial_search_flood(x,y);

    // search grid no longer ready for testing closest objects
    ag_search_prep = false;

    // if no value in [ag_s] at agent's location then target is
    // not reachable at present
    if(ag_s.get_f(ag_x,ag_y) == 0)
    {
        return(false);
    }

    short c1,c2,cx,cy;    // indices on [moves[args]]
    short gox = 0;        // a best x-coord
    short goy = 0;        // a best y-coord
    short score,temp;     // getting lowest cost score
    bool hit = false;     // whether a move is possible

```

```

score = MAX_VALUE; // we need a high value to compare against
                  // values in the search grid

// get lowest score position in search grid that is adjacent to the agent
for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
{
    c2 = c1 + 1;

    cx = ag_x + moves[c1];
    cy = ag_y + moves[c2];

    if(ag_s.check(cx,cy))
    {

        temp = ag_s.get_f(cx,cy);

        if(temp < score && temp > 0)
        {

            // record this move
            score = temp;
            gox = cx;
            goy = cy;
            hit = true;

        }

    }

}

// check that we got a move
if(hit)
{

    // record agent's intended action
    set_target(x,y,o);
    ag_intent.put(MOVE,gox,goy);
    return(true);

}

// no moves possible at present
return(false);

}

bool Agent::targetting()
{

    return(ag_target);

}

```

```

void Agent::execute_move()
{
    short a,x,y;
    ag_intent.get(a,x,y);    // retrieve the action
    short region = ag_env -> get_object_f(x,y);

    if(a != MOVE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Agent::execute_move(), this function can only handle\n";
        err << "actions that are type MOVE.\n";
        err << "Action has FAILED.\n";
        err.close();
        return;
    }

    if(region != SPACE)
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Agent::execute_move(), Agent " << ag_name << " has been allowed to\n";
        err << "move to a blocked region located at (" << x << ", " << y << ")\n";
        err << "This region contains identifier " << region << "\n";
        err << "Action has FAILED.\n";
        err << "Agent's coordinates are: " << ag_x << ", " << ag_y << "\n";
        err.close();
        return;
    }

    ag_env -> put_object_f(ag_x,ag_y,SPACE,0);    // remove Agent's presence
    ag_m.put_f(ag_x,ag_y,SPACE);                // update memory
    ag_x = x;                                   // set new position
    ag_y = y;
    ag_env -> put_object_f(ag_x,ag_y,ag_type,1); // record new location
    ag_m.put_f(ag_x,ag_y,ag_type);              // update memory
}

void Agent::do_nothing()
{
    ag_intent.put(NOTHING,ag_x,ag_y);
    ag_target = false;
}

void Agent::expire_moving_objects()
{
    ag_m.expire_moving();
}

bool Agent::wander(short &x, short &y)

```



```

{
    PATH p;
    triple t;
    short z,choice;

    if(!ag_search_prep)
    {
        update_search_memory();
        execute_search_flood(ag_x,ag_y);
        ag_search_prep = true;
    }

    // build a list of space, guarantee it can be reached

    // note that UNKNOWN areas in [ag_m] have priority recorded in [z]
    for(x = 0 ; x < ag_dim ; x++)
    {
        for(y = 0 ; y < ag_dim ; y++)
        {
            if((z = ag_s.get_f(x,y)) > 0 && !(ag_x == x && ag_y == y)) // i can reach it
            {
                if(!next_to(x,y,UNKNOWN))
                {
                    // make path cost huge
                    z = MAX_VALUE;
                }

                t.put(x,y,z);
                p.push_back(t);
            }
        }
    }

    if(!p.size())
    {
        // no places to get to!!
        return(false);
    }

    // sort into lowest path cost first
    // triple defined as sorted by third tuple
    sort(p.begin(),p.end());

```

```

// get first value
p[0].get(x,y,z);

// if no unknowns
if(z == MAX_VALUE)

{

    // take a random place
    choice = rand() % p.size();
    p[choice].get(x,y,z);
    return(true);

}

return(true);
}

bool Agent::adjacent_to(const short o)
{

    short c1,c2,cx,cy;

    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {

        c2 = c1 + 1;

        cx = ag_x + moves[c1];
        cy = ag_y + moves[c2];

        if(ag_m.check(cx,cy))
        {

            if(ag_m.get_f(cx,cy) == o)
            {

                return(true);

            }

        }

    }

    return(false);
}

bool Agent::adjacent_to(const short x, const short y)
{

    short c1,c2,cx,cy;

    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {

```

```

        c2 = c1 + 1;

        cx = ag_x + moves[c1];
        cy = ag_y + moves[c2];

        if(ag_m.check(cx,cy))
        {
            if((cx == x) && (cy == y))
            {
                return(true);
            }
        }
    }
    return(false);
}

bool Agent::adjacent_to(short &x, short &y, const short o)
{
    short c1,c2,cx,cy;

    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {
        c2 = c1 + 1;

        cx = ag_x + moves[c1];
        cy = ag_y + moves[c2];

        if(ag_m.check(cx,cy))
        {
            if(ag_m.get_f(cx,cy) == o)
            {
                x = cx;
                y = cy;
                return(true);
            }
        }
    }

    return(false);
}

```

```

bool Agent::knows_of(const short o)
{
    return(ag_m.find_instance(o));
}

bool Agent::knows_of_closest(short &x, short &y, const short o)
{
    // flood at agent's coordinates
    if(!ag_search_prep)
    {
        update_search_memory();
        execute_search_flood(ag_x,ag_y);
        ag_search_prep = true;
    }

    short x1,y1,c1,c2,cx,cy,temp;
    short path_cost = MAX_VALUE;
    bool hit = false;

    // look for accessible objects
    for(x1 = 0 ; x1 < ag_dim ; x1++)
    {
        for(y1 = 0 ; y1 < ag_dim ; y1++)
        {
            if(ag_m.get_f(x1,y1) == o)
            {
                // look at adjacent
                for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
                {
                    c2 = c1 + 1;

                    cx = x1 + moves[c1];
                    cy = y1 + moves[c2];

                    if(ag_s.check(cx,cy))
                    {
                        temp = ag_s.get_f(cx,cy);

                        // if closer and accessible
                        if(temp < path_cost && temp > 0)

                            {

```

```

        path_cost = temp;
        x = x1;
        y = y1;
        hit = true;
    }
}

}

}

}

}

return(hit);
}

//    *** PROTECTED FUNCTIONS ***

void Agent::update_search_memory()
{
    ag_s = ag_m;

    ag_s.replace_all_non_zero_instances(-1);

    ag_s.put_f(ag_x,ag_y,0);

    ag_search_prep = false;
}

void Agent::execute_search_flood(short x, short y)
{
    short value = 10;           // path costs
    short c1,c2;                // allowable move counters
    short cx,cy;                // relative positioning
    short put_value;            // paths costs
    short temp;                 // value comparison
    short count = 0;            // loop counter
    bool hypotenuse = true;      // a toggle that represents a diagonal move
    triple t(x,y,value);        // for recording algorithm progress
    ag_search.clear();           // clear the search list
    ag_search.push_back(t);      // set first value
    ag_s.put_f(x,y,value);       // record first value in search grid

    // whilst we are not at the end of search list
    while(count < ag_search.size())
    {
        // get this value
        ag_search[count].get(x,y,value);

        // look at neighbours

```

```

for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
{
    c2 = c1 + 1;
    cx = x + moves[c1];           // make new position
    cy = y + moves[c2];           // make new position
    hypotenuse = !hypotenuse;     // toggle hypotenuse

    // if this coordinate is valid
    if(ag_s.check(cx,cy))
    {
        // if this coordinate is not blocked
        if((temp = ag_s.get_f(cx,cy)) != -1)
        {
            put_value = value;     // make path value

            // if diagonal transition
            if(hypotenuse)
            {
                // note lazy evaluation so order of statements
                // is crucial due to assign and test
                // if new path cost < this value or a space
                if(((put_value += 14) < temp) || temp == SPACE)
                {
                    // record new value and push onto back of list
                    ag_s.put_f(cx,cy,put_value);
                    t.put(cx,cy,put_value);
                    ag_search.push_back(t);
                }
            }
        }
        else
        {
            if(((put_value += 10) < temp) || temp == SPACE)
            {
                ag_s.put_f(cx,cy,put_value);
                t.put(cx,cy,put_value);
                ag_search.push_back(t);
            }
        }
    }
}

```

```

        count++;

    };

}

void Agent::execute_partial_search_flood(short x, short y)
{
    short value = 10;           // path costs
    short c1,c2;                // allowable move counters
    short cx,cy;                // relative positioning
    short put_value;            // paths costs
    short temp;                 // value comparison
    short count = 0;            // loop counter
    bool hypotenuse = true;      // a toggle that represents a diagonal move
    triple t(x,y,value);        // for recording algorithm progress
    ag_search.clear();           // clear the search list
    ag_search.push_back(t);      // set first value
    ag_s.put_f(x,y,value);       // record first value in search grid

    // whilst we are not at the end of search list
    while(count < ag_search.size())
    {
        // get this value
        ag_search[count].get(x,y,value);

        // finish if at Agent's location
        if(ag_x == x && ag_y == y)
        {
            return;
        }

        // look at neighbours
        for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
        {
            c2 = c1 + 1;
            cx = x + moves[c1];    // make new position
            cy = y + moves[c2];    // make new position
            hypotenuse = !hypotenuse; // toggle hypotenuse

            // if this coordinate is valid
            if(ag_s.check(cx,cy))
            {
                // if this coordinate is not blocked
                if((temp = ag_s.get_f(cx,cy)) != -1)
                {
                    put_value = value;    // make path value

                    // if diagonal transition

```

```

        if(hypotenuse)
        {
            // if new path cost < this value or a space
            if(((put_value += 14) < temp) || temp == SPACE)
            {
                // record new value and push onto back of list
                ag_s.put_f(cx,cy,put_value);
                t.put(cx,cy,put_value);
                ag_search.push_back(t);
            }
        }
        else
        {
            if(((put_value += 10) < temp) || temp == SPACE)
            {
                ag_s.put_f(cx,cy,put_value);
                t.put(cx,cy,put_value);
                ag_search.push_back(t);
            }
        }
    }
}

count++;
};
}

bool Agent::next_to(const short x, const short y, const short o)
{
    short c1,c2,cx,cy;

    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {
        c2 = c1 + 1;

        cx = x+moves[c1];
        cy = y+moves[c2];

        if(ag_m.check(cx,cy))

```



```

        {
            if(ag_m.get_f(cx,cy) == 0)
            {
                return(true);
            }
        }
    }
    return(false);
}

```

Originally, we intended to make the base class agent a mining agent. However, with the realisation that we could produce a graphical user interface during the Spring Term, many new opportunities for experimentation involving varying scenarios erupted. Therefore, this class depicts a simple agent with attributes that we felt would be useful for all of the classes of agents that we wished to derive for our experiments.

The most interesting feature of this class is the way in which agent memory is utilised. A coordinate in the environment directly corresponds to a coordinate in the agent's memory. This simplification avoids the necessity for considering how a form of locational associative memory may be implemented.

We found through development and experimentation that it was necessary to make agents forget about moving objects that were outside their visual range each turn. For instance, if a Miner agent “notices” a Mineral resource that it “wishes” to mine, but cannot reach the location because another Miner is blocking access, the Miner agent wanders off to an unknown or random location. If the Miner agent continued to “believe” that the route to this Mineral was blocked, it may never attempt again to mine it, if the location was now outside its visual range. The Miner that was blocking access may have gone away and we now

have the problem of agents failing to mine Minerals that are accessible. There are some interesting side-effects and flaws to using this form of “forgetfulness” that are discussed in Section 4.3.

Making agents wander away if a target location is blocked also avoids the problem of agents clustering together and failing to get things done. If a narrow route exists between a resource and a deposit point, a cluster of agents blocking the path may result in the resource never being collected.

C.5 The Miner Class

The features of this class are briefly discussed in section 3.1.

```
//      Filename:      Miner.h
//      Version:      Final
//      Author:      Paul Tinsley (pmbtin)
//      Date:      26 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:      Numbers, functions and data variables are enclosed in
//                      square parentheses "[ ]". This is to remove any
//                      ambiguity within the text of comments. All private data
//                      member variables are prefixed by "m_".

//      Errors:      All error messages appended to "errlog.txt".

//      Purpose:      Basic Mining Agents. Derived from Agent.h.

// Function Comment Style:

// Purpose:      Purpose of the function.
// Preconditions:      What states or conditions must be met before execution
//                      of the function.
// Returns:      The object or primitive type that the function returns.
// Side Effects:      Effects that the function may have on private data that
//                      are not immediately obvious to the nature of the function.
//                      This section may also contain advice and miscellaneous
//                      comments.

// Errors:      What type of error is generated, why it is generated and
//                      its effects.
```

```

#ifndef __MINER_H
#define __MINER_H

#include "Agent.h"

class Miner : public Agent
{
public:

// *** CONSTRUCTORS ***

Miner(): Agent(), m_sack(0), m_sack_max(50), m_scoop(10) {};

// Purpose:           Default constructor.
// Preconditions:None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

Miner(short s, short x, short y, Environment *e, short n, short r, short sm, short sc)
    :Agent(s, x, y, e, n, r, MINER), m_sack(0), m_sack_max(sm), m_scoop(sc) {};

// Purpose:           Constructor. See base constructor definition in [Agent::Agent(args)].
//                    [m_sack_max] records the maximum sack capacity. [m_scoop] records
//                    the amount of minerals that a Miner can mine in one turn. [m_sack]
//                    is the current mineral quantity being carried.
// Preconditions:None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

//      *** OPERATORS ***

virtual Miner & operator =(const Miner &a);

// Purpose:           Destructively assigns [a] to the current context.
// Preconditions:None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

const friend bool operator ==(const Miner &a, const Miner &b)

// Purpose:           Equality operator.
// Preconditions:None.
// Returns:           [bool].
// Side Effects:       None.
// Errors:            None.

{

    return(a.ag_name == b.ag_name);

}

const friend bool operator !=(const Miner &a, const Miner &b)

// Purpose:           Non-equality operator.
// Preconditions:None.

```

```

// Returns:          [bool].
// Side Effects:      None.
// Errors:            None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const Miner &a, const Miner &b)

// Purpose:          Less-than operator. An agent is less than another
//                    agent if its name is less in value.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:       None.
// Errors:            None.

{

    return(a.ag_name < b.ag_name);

};

const friend bool operator >(const Miner &a, const Miner &b)

// Purpose:          Greater-than operator. An agent is greater than another
//                    agent if its name is greater in value.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:       None.
// Errors:            None.

{

    return(a.ag_name > b.ag_name);

};

// *** PUBLIC FUNCTION PROTOTYPES ***

virtual void get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                      short &tx, short &ty);

// Purpose:          Loads [t] with [ag_type], [n] with [ag_name], [x] with [ag_x],
//                    [y] with [ag_y], [a] with the action within [ag_intent]. [q]
//                    gets [m_sack]. If [ag_target] == [true], loads [tx] with
//                    [ag_target_x] and [ty] with [ag_target_y], else both set to [-1].
// Preconditions: None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

virtual void think();

// Purpose:          The behavioural definition rules for these Miners.
// Preconditions: None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

virtual void execute_action();

```

```
// Purpose:          Causes the Miner to execute an action based on the contents
//                   of its [ag_intent].
// Preconditions:[ag_intent] has been arbitrated.
// Returns:          None.
// Side Effects:     The Miner may effect the environment [ag_env].
// Errors:           None.
```

```
private:
```

```
//      *** PRIVATE DATA MEMBERS ***
```

```
short m_sack;        // a sack for carrying things
short m_sack_max;    // max capacity of sack
short m_scoop;       // max amount Miner can scoop in one turn
```

```
//      *** PRIVATE FUNCTION PROTOTYPES ***
```

```
void execute_mine_minerals(const short x, const short y);
```

```
// Purpose:          The Miner will mine minerals from the location
//                   ([x],[y]) increasing the quantity in [m_sack].
// Preconditions: The intended action has been arbitrated. The Miner
//                   is adjacent to ([x],[y]). [m_sack] != [m_sack_max].
//                   Minerals [MINERAL] exist at ([x],[y]).
// Returns:          None.
// Side Effects:     The Miner may effect the environment [ag_env].
// Errors:           [errlog] messages if preconditions violated. Miner's
//                   action may be cancelled and replaced with the action
//                   [do_nothing()].
```

```
void execute_empty_backpack(const short x, const short y);
```

```
// Purpose:          The Miner will empty the contents of [m_sack] in the
//                   location ([x],[y]).
// Preconditions: The intended action has been arbitrated. The Miner
//                   is adjacent to ([x],[y]). [m_sack] != 0. ([x],[y])
//                   is a [HOME].
// Returns:          None.
// Side Effects:     If insufficient space, an [errlog] message will be
//                   generated and the action will fail.
// Errors:           [errlog] messages if preconditions violated. Miner's
//                   action may be cancelled and replaced with the action
//                   [do_nothing()].
```

```
void mine_minerals(const short x, const short y);
```

```
// Purpose:          The Miner will set [ag_intent] to relect the taking
//                   of minerals from ([x],[y]).
// Preconditions: The Miner is adjacent to ([x],[y]). [m_sack] != [m_sack_max].
//                   Minerals [MINERAL] exist at ([x],[y]).
// Returns:          None.
// Side Effects:     None.
// Errors:           [errlog] messages if preconditions violated. Miner's
//                   action may be cancelled and replaced with the action
//                   [do_nothing()].
```

```
void empty_backpack(const short x, const short y);
```

```
// Purpose:          The Miner will set [ag_intent] to relect the dropping
//                   of minerals at location ([x],[y]).
// Preconditions: The Miner is adjacent to ([x],[y]). [m_sack] != 0.
//                   ([x],[y]) is a [HOME].
```

```

// Returns:          None.
// Side Effects:     If insufficient space, an [errlog] message will be
//                  generated and the action will fail.
// Errors:           [errlog] messages if preconditions violated. Miner's
//                  action may be cancelled and replaced with the action
//                  [do_nothing()].

const bool sack_full();

// Purpose:          Returns whether [m_sack] == [m_sack_max].
// Preconditions:     None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

const bool sack_empty();

// Purpose:          Returns whether [m_sack] == [0].
// Preconditions:     None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

const short sack_space();

// Purpose:          Returns space remaining in [m_sack].
// Preconditions:     None.
// Returns:          [short].
// Side Effects:     None.
// Errors:           None.

};

#endif

//      Filename:      Miner.cpp
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00

//      Purpose:       See Miner.h. Derived from base class Agent.h.

//      Refer to Agent.h for details of some functions.

#include "stdafx.h"
#include "Miner.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** OPERATORS ***

Miner & Miner::operator =(const Miner &a)

{

    // base class privates
    ag_m = a.ag_m;
    ag_s = a.ag_s;

```

```

    ag_x = a.ag_x;
    ag_y = a.ag_y;
    ag_target_x = a.ag_target_x;
    ag_target_y = a.ag_target_y;
    ag_env = a.ag_env;
    ag_name = a.ag_name;
    ag_v_range = a.ag_v_range;
    ag_intent = a.ag_intent;
    ag_type = a.ag_type;
    ag_orig_type = a.ag_orig_type;
    ag_alive = a.ag_alive;
    ag_target = a.ag_target;
    ag_target_obj = a.ag_target_obj;
    ag_dim = a.ag_dim;
    ag_search = a.ag_search;
    ag_search_prep = a.ag_search_prep;

    // Miner privates

    m_sack = a.m_sack;           // a sack for carrying things
    m_sack_max = a.m_sack_max;   // max capacity of sack
    m_scoop = a.m_scoop;         // max amount Miner can scoop in one turn

    return(*this);
}

//      *** PUBLIC FUNCTIONS **

void Miner::get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                    short &tx, short &ty)

{
    t = ag_type;                 // type
    n = ag_name;                 // name
    x = ag_x;                    // current x-coord
    y = ag_y;                    // current y-coord
    a = ag_intent.get_action();  // current action
    q = m_sack;                  // sack contents

    if(ag_target)
    {
        tx = ag_target_x;        // target x-coord
        ty = ag_target_y;        // target y-coord
    }

    else
    {
        tx = -1;
        ty = -1;
    }
}

void Miner::think()

```

```

{

    Agent::think();

    short x,y;

    // REMOVE ALL MOBILE THINGS FROM MEMORY

    expire_moving_objects();

    // OBSERVE THE ENVIRONMENT

    look();

    // DROPPING MINERALS AT A HOME BASE

    // if next to a home and has some minerals
    if(!sack_empty() && adjacent_to(x,y,HOME))

    {

        // then deposit minerals and finish this turn
        empty_backpack(x,y);
        return;

    }

    // MINING MINERALS

    // if next to minerals and sack isn't full
    if(!sack_full() && adjacent_to(x,y,MINERAL))

    {

        // then mine minerals and end this turn
        mine_minerals(x,y);
        return;

    }

    // LOCATING AND HEADING TOWARDS MINERALS

    // if not already seeking minerals and knows of minerals and sack isn't full
    if(ag_target_obj != MINERAL && !sack_full() && knows_of(MINERAL))

    {

        // if knows of a mineral and is nearest
        if(knows_of_closest(x,y,MINERAL))

        {

            // if can get to minerals
            if(seek_target(x,y,MINERAL))

            {

                // then end this turn
                return;

            }

        }

    }

}

```



```

}

// FINDING A HOME BASE TO DEPOSIT MINERALS

// if sack is full and not already targetting a home base
if(ag_target_obj != HOME && sack_full())

{
    // if knows of a home base and is nearest
    if(knows_of_closest(x,y,HOME))

    {
        // if can get to home base
        if(seek_target(x,y,HOME))

        {
            // then end this turn
            return;
        }
    }
}

// RETURNING TO A HOME BASE - NO MORE MINERALS TO MINE

// not targetting a home base and has some minerals but does not know of any more
if(ag_target_obj != HOME && !sack_empty() && !knows_of(MINERAL))

{
    // if knows of a closest home base
    if(knows_of_closest(x,y,HOME))

    {
        // if can get to home base
        if(seek_target(x,y,HOME))

        {
            // then end this turn
            return;
        }
    }
}

// CONTINUING TARGETTING

// if targetting anything
if(targetting())

{
    // if can get to it

```

```

        if(seek_target(ag_target_x,ag_target_y,ag_target_obj))
        {

            // then end this turn
            return;

        }

    }

    // STARTING WANDERING

    // if nowhere to wander to
    if(!wander(x,y))

    {

        // stop an do nothing
        do_nothing();

    }

    else

    {

        // if can get there
        if(seek_target(x,y,SPACE))

        {

            // then end this turn
            return;

        }

        else

        {

            // stop and do nothing
            do_nothing();

        }

    }

}

const bool Miner::sack_full()

{

    return(m_sack == m_sack_max);

}

void Miner::mine_minerals(const short x, const short y)

{

    if(sack_full())

```

```

{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Miner::mine_minerals(x, y), Miner\n";
    err << "wants to mine minerals but his backpack is full.\n";
    err << "Action has failed and Miner will do nothing!\n";
    err.close();
    do_nothing();
    return;
}

if(ag_m.check(x,y))
{
    if(ag_m.get_f(x,y) == MINERAL && adjacent_to(x,y))
    {
        ag_intent.put GRAB,x,y);
        // stop any targetting
        erase_target();
    }
    else
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Miner::mine_minerals(x, y), Either\n";
        err << "minerals do not exist or Miner is not adjacent\n";
        err << "to any minerals. Action has failed and Miner\n";
        err << "will do nothing!\n";
        err.close();
        do_nothing();
    }
}
else
{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Miner::mine_minerals(x, y), Invalid\n";
    err << "coordinates. Action has failed and Miner will do\n";
    err << "nothing!\n";
    err.close();
    do_nothing();
}
}

void Miner::empty_backpack(const short x, const short y)
{
    if(sack_empty())

```

```

{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Miner::empty_backpack(x, y), Miners\n";
    err << "backpack is empty.Action has failed and Miner\n";
    err << "will do nothing!\n";
    err.close();
    do_nothing();
    return;
}

if(ag_m.check(x,y))
{
    if(ag_m.get_f(x,y) == HOME && adjacent_to(x,y))
    {
        ag_intent.put(DROP,x,y);
        // stop any targetting
        erase_target();
    }
    else
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Miner::empty_backpack(x, y), Illogical\n";
        err << "location to drop ORE. Action has failed and Miner\n";
        err << "will do nothing!\n";
        err.close();
        do_nothing();
    }
}
else
{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Miner::empty_backpack(x, y), Invalid\n";
    err << "coordinates. Action has failed and Miner will do\n";
    err << "nothing!\n";
    err.close();
    do_nothing();
}
}

const bool Miner::sack_empty()
{
    return(!m_sack);
}

```

```

}

const short Miner::sack_space()

{

    return(m_sack_max - m_sack);

}

void Miner::execute_mine_minerals(const short x, const short y)

{

    short object,quantity,grab,grabbed;

    // check that minerals exist
    ag_env -> get_object(x,y,object,quantity);

    if(object == MINERAL && quantity > 0)
    {

        grab = sack_space();

        if(grab >= m_scoop)

        {

            grab = m_scoop;

        }

        // get minerals
        grabbed = ag_env -> take_quantity_f(x,y,grab);
        m_sack += grabbed;

    }

}

void Miner::execute_empty_backpack(const short x, const short y)

{

    if(ag_env -> get_object_f(x,y) == HOME)

    {

        if(ag_env -> add_quantity_f(x,y,ORE,m_sack))

        {

            m_sack = 0;

        }

        else

        {

            ofstream err("errlog.txt", ios::app);
            err << "\n\nCAUTION, Miner::execute_empty_backpack(x,y), There\n";


```

```

        err << "was not enough space to drop the objects.\n";
        err << "Action has failed!\n";
        err.close();
    }

}

else

{

    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Miner::execute_empty_backpack(x,y), Miner\n";
    err << "dropping ore on a location other than HOME.\n";
    err << "Action has failed!\n";
    err.close();

}

}

void Miner::execute_action()

{

    if(ag_alive)

    {

        short a,x,y;
        ag_intent.get(a,x,y);

        switch(a)

        {

            case NOTHING:

                break;

            case MOVE:

                execute_move();
                break;

            case GRAB:

                execute_mine_minerals(x,y);
                break;

            case DROP:

                execute_empty_backpack(x,y);
                break;

            default:

                ofstream err("errlog.txt", ios::app);
                err << "\n\nERROR, Miner::execute_action(), Miner " << ag_name << endl;
                err << "attempting to perform illogical action. Action failed!\n";
                err.close();
                break;
        }
    }
}

```

```

    }

}

}

```

C.6 The Predator Class

The features of this class are briefly discussed in section 3.2.

```

//      Filename:          Predator.h
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

//      Symbology:         Numbers, functions and data variables are enclosed in
//                          square parentheses "[]". This is to remove any
//                          ambiguity within the text of comments. All private data
//                          member variables are prefixed by "p_".

//      Errors:            All error messages appended to "errlog.txt".

//      Purpose:           Predator agents.

// Function Comment Style:

// Purpose:                Purpose of the function.
// Preconditions:          What states or conditions must be met before execution
//                          of the function.
// Returns:                The object or primitive type that the function returns.
// Side Effects:           Effects that the function may have on private data that
//                          are not immediately obvious to the nature of the function.
//                          This section may also contain advice and miscellaneous
//                          comments.

// Errors:                What type of error is generated, why it is generated and
//                          its effects.

//      Refer to Agent.h for details of some functions.

#ifndef __PREDATOR_H
#define __PREDATOR_H

#include "Agent.h"          // base class
#include <fstream>           // error logs

class Predator : public Agent
{

```

```

public:

// *** CONSTRUCTORS ***

Predator(): Agent(), p_stomach(0), p_stomach_max(10){};

// Purpose:          Default constructor.
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

Predator(const short s, const short x, const short y, Environment *e, const short n,
          const short r, const short stom, const short stom_m)
: Agent(s,x,y,e,n,r,PREDATOR), p_stomach(stom),
  p_stomach_max(stom_m){};

// Purpose:          Constructor. See base constructor. [p_stomach] gets [stom] and
//                   p_stomach_max gets [stom_m].
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

//      *** OPERATORS ***

virtual Predator & operator =(const Predator &a);

// Purpose:          Destructively assigns [a] to the current context.
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const friend bool operator ==(const Predator &a, const Predator &b)

// Purpose:          Equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(a.ag_name == b.ag_name);

}

const friend bool operator !=(const Predator &a, const Predator &b)

// Purpose:          Non-equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const Predator &a, const Predator &b)

```



```

// Purpose:          Less-than operator. An agent is less than another
//                   agent if its name is less in value.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.ag_name < b.ag_name);

};

const friend bool operator >(const Predator &a, const Predator &b)

// Purpose:          Greater-than operator. An agent is greater than another
//                   agent if its name is greater in value.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:      None.
// Errors:           None.

{

    return(a.ag_name > b.ag_name);

};

//      *** PUBLIC FUNCTION PROTOTYPES ***

virtual void get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                      short &tx, short &ty);

// Purpose:          Loads [t] with [ag_type], [n] with [ag_name], [x] with [ag_x],
//                   [y] with [ag_y], [a] with the action within [ag_intent]. [q]
//                   gets [p_stomach]. If [ag_target] == [true], loads [tx] with
//                   [ag_target_x] and [ty] with [ag_target_y], else both set to [-1].
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

virtual void think();

// Purpose:          The behavioural definition rules for these Predators.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

virtual void execute_action();

// Purpose:          Causes the Predator to execute an action based on the contents
//                   of its [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:          None.
// Side Effects:      The Predator may affect the environment [ag_env].
// Errors:           None.

void feed(const short x, const short y);

```

```

// Purpose:          If a [CARCASS] exists at ([x],[y]) in [ag_env] and
//                   Predator is adjacent to it, sets [ag_intent] to
//                   action [GRAB] at those coordinates.
// Preconditions:[ag_env -> get_object([x],[y]) == CARCASS] &&
//                   adjacent_to([x],[y]).
// Returns:          None.
// Side Effects:     [erase_target()] called if successful.
// Errors:           [errlog] message if precondition violation. [ag_intent]
//                   set to [do_nothing()].

void pounce(const short x, const short y);

// Purpose:          If a [PREY] exists at ([x],[y]) in [ag_env] and
//                   Predator is adjacent to it, sets [ag_intent] to
//                   action [POUNCE] at those coordinates.
// Preconditions:[ag_env -> get_object([x],[y]) == PREY] &&
//                   adjacent_to([x],[y]).
// Returns:          None.
// Side Effects:     [erase_target()] called if successful.
// Errors:           [errlog] message if precondition violation. [ag_intent]
//                   set to [do_nothing()].

void execute_feed(const short x, const short y);

// Purpose:          If a [CARCASS] exists at ([x],[y]) in [ag_env] then
//                   Predator feeds from it and increases [p_stomach] up
//                   to a max value of [p_stomach_max].
// Preconditions:[ag_env -> get_object([x],[y]) == CARCASS]
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

const bool hungry();

// Purpose:          Returns whether [p_stomach] != [p_stomach_max].
// Preconditions:None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

private:

short p_stomach;      // stomach contents
short p_stomach_max; // max stomach contents

};

#endif

// Filename:          Predator.cpp
// Version:           Final
// Author:             Paul Tinsley (pmbtin)
// Date:              26 February, 1999
// Platform:          Visual C++ 5.00

// Purpose:           See Predator.h

// Refer to Agent.h for details of some functions.

#include "stdafx.h"
#include "Predator.h"

#ifdef _DEBUG

```

```

#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** OPERATORS ***

Predator & Predator::operator =(const Predator &a)

{
    // base class privates
    ag_m = a.ag_m;
    ag_s = a.ag_s;
    ag_x = a.ag_x;
    ag_y = a.ag_y;
    ag_target_x = a.ag_target_x;
    ag_target_y = a.ag_target_y;
    ag_env = a.ag_env;
    ag_name = a.ag_name;
    ag_v_range = a.ag_v_range;
    ag_intent = a.ag_intent;
    ag_type = a.ag_type;
    ag_orig_type = a.ag_orig_type;
    ag_alive = a.ag_alive;
    ag_target = a.ag_target;
    ag_target_obj = a.ag_target_obj;
    ag_dim = a.ag_dim;
    ag_search = a.ag_search;
    ag_search_prep = a.ag_search_prep;

    // Predator privates

    p_stomach = a.p_stomach;           // stomach contents
    p_stomach_max = a.p_stomach_max;   // max stomach contents

    return(*this);
}

//      *** PUBLIC FUNCTIONS **

void Predator::get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                        short &tx, short &ty)

{
    t = ag_type;
    n = ag_name;
    x = ag_x;
    y = ag_y;
    a = ag_intent.get_action();
    q = p_stomach;

    if(ag_target)
    {
        tx = ag_target_x;
        ty = ag_target_y;
    }
}

```

```

        else
        {
            tx = -1;
            ty = -1;
        }
    }

void Predator::think()
{
    Agent::think();

    short x, y;

    // GET RID OF ALL OTHER MOVING TARGETS
    expire_moving_objects();

    // OBSERVE THE ENVIRONMENT
    look();

    // FEED
    if(hungry() && (adjacent_to(x,y,CARCASS)))
    {
        feed(x,y);
        return;
    }

    // POUNCE
    if(hungry() && (adjacent_to(x,y,PREY)))
    {
        pounce(x,y);
        return;
    }

    if(hungry() && knows_of(CARCASS))
    {
        if(knows_of_closest(x,y,CARCASS))
        {
            if(seek_target(x,y,CARCASS))
            {
                return;
            }
        }
    }
}

```

```

        }
    }
}
if(hungry() && knows_of(PREY))
{
    if(knows_of_closest(x,y,PREY))
    {
        if(seek_target(x,y,PREY))
        {
            return;
        }
    }
}
if(targetting())
{
    if(seek_target(ag_target_x,ag_target_y,ag_target_obj))
    {
        return;
    }
}
if(!wander(x,y))
{
    do_nothing();
}
else if(!seek_target(x,y,SPACE))
{
    do_nothing();
}
}

void Predator::pounce(const short x, const short y)
{
    if((adjacent_to(x,y)) && (ag_env -> get_object(x,y) == PREY))

```

```

{
    ag_intent.put(POUNCE,x,y);
    erase_target();
}
else
{
    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Predator::pounce(x,y), Illogical\n";
    err << "location to pounce on. Prey does not exist at this location.\n";
    err << "Action has failed and agent will do nothing!\n";
    err.close();
    do_nothing();
}
}

void Predator::feed(const short x, const short y)
{
    if((adjacent_to(x,y)) && (ag_env -> get_object(x,y) == CARCASS))
    {
        ag_intent.put(GRAB,x,y);
        erase_target();
    }
    else
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, Predator::feed(x,y), Illogical\n";
        err << "location to feed from. Carcass does not exist at this location.\n";
        err << "Action has failed and agent will do nothing!\n";
        err.close();
        do_nothing();
    }
}

void Predator::execute_feed(const short x, const short y)
{
    short object, quantity, bite, bitten;

    ag_env -> get_object(x,y,object,quantity);

    if(object == CARCASS && quantity > 0)
    {

```

```

        bite = p_stomach_max - p_stomach;

        if(bite > 10)
        {
            bite = 10;
        }

        bitten = ag_env -> take_quantity_f(x,y,bite);

        p_stomach += bitten;
    }
}

void Predator::execute_action()
{
    if(ag_alive)
    {
        short a,x,y;
        ag_intent.get(a,x,y);

        switch(a)
        {
            case NOTHING:
                break;

            case MOVE:
                execute_move();
                break;

            case GRAB:
                execute_feed(x,y);
                break;

            case POUNCE:
                break;

            default:
                ofstream err("errlog.txt", ios::app);
                err << "\n\nERROR, Predator::execute_action(), Predator " << ag_name << endl;
                err << "attempting to perform illogical action. Action failed!\n";
                err.close();
                break;
        }
    }
}

```

```

const bool Predator::hungry()

{

    return(!(p_stomach == p_stomach_max));

}

```

C.7 The Prey Class

The features of this class are briefly discussed in section 3.2.

```

//      Filename:          Prey.h
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

//      Symbology:         Numbers, functions and data variables are enclosed in
//                          square parentheses "[]". This is to remove any
//                          ambiguity within the text of comments.

//      Errors:            All error messages appended to "errlog.txt".

//      Purpose:           Prey agents. Derived from Agent.h.

// Function Comment Style:

// Purpose:                Purpose of the function.
// Preconditions:          What states or conditions must be met before execution
//                          of the function.
// Returns:                The object or primitive type that the function returns.
// Side Effects:           Effects that the function may have on private data that
//                          are not immediately obvious to the nature of the function.
//                          This section may also contain advice and miscellaneous
//                          comments.

// Errors:                What type of error is generated, why it is generated and
//                          its effects.

#ifndef __PREY_H
#define __PREY_H

#include "Agent.h"        // base class
#include <fstream>          // error logs

class Prey : public Agent

{

    public:

```



```

// *** CONSTRUCTORS ***

Prey(): Agent({});

// Purpose:          Default constructor.
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

Prey(const short s, const short x, const short y, Environment *e, const short n,
      const short r) : Agent(s,x,y,e,n,r,PREY){};

// Purpose:          Constructor. See base class definitions.
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

//      *** OPERATORS ***

const friend bool operator ==(const Prey &a, const Prey &b)

// Purpose:          Equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(a.ag_name == b.ag_name);

}

const friend bool operator !=(const Prey &a, const Prey &b)

// Purpose:          Non-equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const Prey &a, const Prey &b)

// Purpose:          Less-than operator. An agent is less than another
//                   agent if its name is less in value.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(a.ag_name < b.ag_name);

```

```

};

const friend bool operator >(const Prey &a, const Prey &b)

// Purpose:          Greater-than operator. An agent is greater than another
//                   agent if its name is greater in value.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:       None.
// Errors:            None.

{

    return(a.ag_name > b.ag_name);

};

//      *** PUBLIC FUNCTION PROTOTYPES ***

virtual void get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                      short &tx, short &ty);

// Purpose:          Loads [t] with [ag_type], [n] with [ag_name], [x] with [ag_x],
//                   [y] with [ag_y], [a] with the action within [ag_intent]. [q]
//                   gets [ag_env -> get_quantity_f(ag_x,ag_y)]. If [ag_target]
//                   == [true], loads [tx] with [ag_target_x] and [ty] with
//                   [ag_target_y], else both set to [-1].
// Preconditions: None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

virtual void think();

// Purpose:          The behavioural rules of a Prey agent.
// Preconditions: None.
// Returns:           None.
// Side Effects:       None.
// Errors:            None.

virtual void execute_action();

// Purpose:          Causes the Prey to execute an action based on the contents
//                   of its [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:           None.
// Side Effects:       The Prey may affect the environment [ag_env].
// Errors:            None.

virtual void execute_move();

// Purpose:          Executes a move based on contents of [ag_intent]. Required
//                   to handle definition of [JUMP].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:           None.
// Side Effects:       Prey sets its new position in environment and memory.
// Errors:            [errlog] message if location specified in [ag_intent] is not
//                   a [SPACE] in [ag_env]. Action fails and [ag_action] set to
//                   [NOTHING].

private:

bool avoid_predators();

```

```

        // Purpose:          If Predators detected, builds a [JUMP] move that gets the
        //                   Prey away from the closest threat.
        // Preconditions: None.
        // Returns:          [bool].
        // Side Effects:      If succesful, [ag_intent] is updated with a [JUMP] move.
        // Errors:           None.

};

#endif

//      Filename:          Prey.cpp
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

//      Purpose:           See Prey.h

//      Refer to Agent.h for details of some functions.

#include "stdafx.h"
#include "Prey.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** PUBLIC FUNCTIONS **

void Prey::get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                    short &tx, short &ty)

{

    t = ag_type;
    n = ag_name;
    x = ag_x;
    y = ag_y;
    a = ag_intent.get_action();
    q = ag_env -> get_quantity_f(ag_x,ag_y);

    if(ag_target)

    {

        tx = ag_target_x;
        ty = ag_target_y;

    }

    else

    {

        tx = -1;
        ty = -1;

    }

}

```

```

}

void Prey::think()
{
    Agent::think();

    short x,y;

    // HANDLE DEATH

    // if dead
    if(!is_alive())
    {
        // if not converted to a SPACE
        if(ag_type != SPACE)
        {
            // if quantity at Prey's location not set
            if(ag_env -> get_quantity(ag_x,ag_y) == 0)
            {
                // become a space - environment self-adjusting
                ag_type = SPACE;
            }
        }

        // don't act
        do_nothing();
        return;
    }

    // GET RID OF ALL OTHER MOVING TARGETS

    expire_moving_objects();

    // OBSERVE THE ENVIRONMENT

    look();

    // if knows of any predators
    if(knows_of(PREDATOR))
    {
        // if can avoid predators
        if(avoid_predators())
        {
            return;
        }
    }
}

```

```

// if targetting anything
if(targetting())

{

    // if can get there
    if(seek_target(ag_target_x,ag_target_y,ag_target_obj))

    {

        return;

    }

}

// if unable to wander
if(!wander(x,y))

{

    do_nothing();

}

// if cant get to wander location
else if(!seek_target(x,y,SPACE))

{

    do_nothing();

}

}

bool Prey::avoid_predators()

{

    short x,y,c1,c2,cx,cy,dest;
    short score = 0;
    bool hit = false;

    // if predators exist
    if(knows_of_closest(x,y,PREDATOR))

    {

        // flood at predators location
        update_search_memory();
        execute_search_flood(x,y);

        // set predator location to blocked
        ag_s.put_f(x,y,-1);

        // see if predator can get to prey

        if(ag_s.get_f(ag_x,ag_y) == 0)

        {

```

```

        return(false);
    }

    // pick location furthest away as a move
    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {
        c2 = c1 + 1;

        cx = ag_x + moves_f[c1];
        cy = ag_y + moves_f[c2];

        // if jump location is a valid coordinate
        if(ag_s.check(cx,cy))
        {
            // if mid point clear
            if(ag_s.get_f(ag_x + moves[c1],ag_y + moves[c2]) > 0)
            {
                // get destination
                dest = ag_s.get_f(cx,cy);

                // if destination available and furthest so far
                if(dest > score && dest > 0)
                {
                    // record this move
                    score = dest;
                    x = cx;
                    y = cy;
                    hit = true;
                }
            }
        }
    }

    // check that we got a move
    if(hit)
    {
        // record agent's intended action
        ag_intent.put(JUMP,x,y);
        // record the target - so it shows up in animation
        set_target(x,y,SPACE);
        return(true);
    }
    else

```

```

        {
            return(false);
        }
    }
else
{
    return(false);
}
}

void Prey::execute_action()
{
    if(ag_alive)
    {
        short a, x, y;
        ag_intent.get(a,x,y);

        switch(a)
        {
            case NOTHING:
                break;

            case JUMP:
                execute_move();
                break;

            case MOVE:
                execute_move();
                break;

            default:
                ofstream err("errlog.txt", ios::app);
                err << "\n\nERROR, Prey::execute_action(), Prey " << ag_name << endl;
                err << "attempting to perform illogical action. Action failed!\n";
                err.close();
                break;
        }
    }
}

void Prey::execute_move()

```

```

{

short a,x,y;
ag_intent.get(a,x,y);    // retrieve the action
short region = ag_env -> get_object_f(x,y);

if(!(a == MOVE || a == JUMP))

{

    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Prey::execute_move(), this function can only handle\n";
    err << "actions that are type MOVE or JUMP.\n";
    err << "Action has FAILED.\n";
    err.close();
    return;

}

if(region != SPACE)

{

    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, Prey::execute_move(), Prey " << ag_name << " has been allowed to\n";
    err << "move to a blocked region located at (" << x << ", " << y << ")\n";
    err << "This region contains identifier " << region << "\n";
    err << "Action has FAILED.\n";
    err << "Prey's coordinates are: " << ag_x << ", " << ag_y << "\n";
    err.close();
    return;

}

ag_env -> put_object_f(ag_x,ag_y,SPACE,0);    // remove Agent's presence
ag_m.put_f(ag_x,ag_y,SPACE);                // update memory
ag_x = x;                                   // set new position
ag_y = y;
ag_env -> put_object_f(ag_x,ag_y,ag_type,1); // record new location
ag_m.put_f(ag_x,ag_y,ag_type);              // update memory

}

```

C.8 The Chain-Miner Class

The features of this class are briefly discussed in section 3.3.

```

//      Filename:          ChainMiner.h
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

```



```

//      Refer to Agent.h for details of some functions.

//      Symbology:          Numbers, functions and data variables are enclosed in
//                          square parentheses "[]". This is to remove any
//                          ambiguity within the text of comments. All private data
//                          member variables are prefixed by "cm_".

//      Errors:             All error messages appended to "errlog.txt".

//      Purpose:            Chain-Mining Agents. Derived from Agent.h.

// Function Comment Style:

// Purpose:                Purpose of the function.
// Preconditions:          What states or conditions must be met before execution
//                          of the function.
// Returns:                The object or primitive type that the function returns.
// Side Effects:           Effects that the function may have on private data that
//                          are not immediately obvious to the nature of the function.
//                          This section may also contain advice and miscellaneous
//                          comments.

// Errors:                What type of error is generated, why it is generated and
//                          its effects.

#ifndef __CHAINMINER_H
#define __CHAINMINER_H

#include "Agent.h"
#include <fstream>

class ChainMiner : public Agent
{
    public:

    // *** CONSTRUCTORS ***

    ChainMiner(): Agent(), cm_sack(0), cm_sack_max(50), cm_scoop(10),
                  cm_endurance(12), cm_facing(0){};

    // Purpose:            Default constructor.
    // Preconditions:       None.
    // Returns:            None.
    // Side Effects:       None.
    // Errors:            None.

    ChainMiner(short s, short x, short y, Environment *e, short n, short r,
               short sm, short sc) : Agent(s,x,y,e,n,r,MINER), cm_sack(0),
                                   cm_sack_max(sm), cm_scoop(sc), cm_endurance(12), cm_facing(0){};

    // Purpose:            Constructor. See base class constructor. [cm_sack] is
    //                      the ChainMiner's current backpack contents, [cm_sack_max]
    //                      is the maximum backpack contents, [cm_scoop] is the
    //                      maximum amount that can be grabbed in a turn and
    //                      [cm_endurance] is the current energy level.
    // Preconditions:       None.
    // Returns:            None.
    // Side Effects:       None.
    // Errors:            None.

```

```

//      *** OPERATORS ***

virtual ChainMiner & operator =(const ChainMiner &a);

// Purpose:          Destructively assigns [a] to the current context.
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

const friend bool operator ==(const ChainMiner &a, const ChainMiner &b)

// Purpose:          Equality operator.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

{

    return(a.ag_name == b.ag_name);

}

const friend bool operator !=(const ChainMiner &a, const ChainMiner &b)

// Purpose:          Non-equality operator.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const ChainMiner &a, const ChainMiner &b)

// Purpose:          Less-than operator. An agent is less than another
//                   agent if its name is less in value.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

{

    return(a.ag_name < b.ag_name);

};

const friend bool operator >(const ChainMiner &a, const ChainMiner &b)

// Purpose:          Greater-than operator. An agent is greater than another
//                   agent if its name is greater in value.
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

```

```

{

    return(a.ag_name > b.ag_name);

};

//      *** PUBLIC FUNCTION PROTOTYPES ***

virtual void get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                      short &tx, short &ty);

// Purpose:          Loads [t] with [ag_type], [n] with [ag_name], [x] with [ag_x],
//                   [y] with [ag_y], [a] with the action within [ag_intent]. [q]
//                   gets [cm_sack]. If [ag_target] == [true], loads [tx] with
//                   [ag_target_x] and [ty] with [ag_target_y], else both set to [-1].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

virtual void think();

// Purpose:          The behavioural definition rules for these ChainMiners.
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

virtual void execute_action();

// Purpose:          Causes the ChainMiner to execute an action based on the contents
//                   of its [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:           None.
// Side Effects:      The ChainMiner agent may affect the environment [ag_env].
// Errors:            None.

private:

void execute_mine_minerals(const short x, const short y);

// Purpose:          The ChainMiner will mine minerals from the location
//                   ([x],[y]) increasing the quantity in [cm_sack].
// Preconditions: The intended action has been arbitrated. The ChainMiner
//                   is adjacent to ([x],[y]). [cm_sack] != [cm_sack_max].
//                   Minerals [MINERAL] exist at ([x],[y]).
// Returns:           None.
// Side Effects:      The Chain Miner may affect the environment [ag_env].
// Errors:            [errlog] messages if preconditions violated. ChainMiner's
//                   action may be cancelled and replaced with the action
//                   [do_nothing()].

void execute_grab_ore(const short x, const short y);

// Purpose:          The ChainMiner will grab ore from the location
//                   ([x],[y]) increasing the quantity in [cm_sack].
// Preconditions: The intended action has been arbitrated. The ChainMiner
//                   is adjacent to ([x],[y]). [cm_sack] != [cm_sack_max].
//                   Ore [ORE] exist at ([x],[y]).
// Returns:           None.
// Side Effects:      The Chain Miner may affect the environment [ag_env].
// Errors:            [errlog] messages if preconditions violated. ChainMiner's
//                   action may be cancelled and replaced with the action

```

```

//                                [do_nothing()].

void execute_empty_backpack(const short x, const short y);

// Purpose:                      The ChainMiner will empty the contents of [m_sack] in the
//                                location ([x],[y]).
// Preconditions: The intended action has been arbitrated. The ChainMiner
//                                is adjacent to ([x],[y]). [cm_sack] != 0. ([x],[y])
//                                is a [HOME] or [ORE] or [SPACE].
// Returns:                      None.
// Side Effects:                  If insufficient space, an [errlog] message will be
//                                generated and the action will fail.
// Errors:                       [errlog] messages if preconditions violated. ChainMiner's
//                                action may be cancelled and replaced with the action
//                                [do_nothing()].

void mine_minerals(const short x, const short y);

// Purpose:                      The ChainMiner will set [ag_intent] to select the taking
//                                of minerals from ([x],[y]).
// Preconditions: The ChainMiner is adjacent to ([x],[y]). [cm_sack] !=
//                                [cm_sack_max].
//                                Minerals [MINERAL] exist at ([x],[y]).
// Returns:                      None.
// Side Effects:                  None.
// Errors:                       [errlog] messages if preconditions violated. ChainMiner's
//                                action may be cancelled and replaced with the action
//                                [do_nothing()].

void empty_backpack(const short x, const short y);

// Purpose:                      The ChainMiner will set its [ag_intent] to select the
//                                dropping of minerals at location ([x],[y]).
// Preconditions: The Miner is adjacent to ([x],[y]). [cm_sack] != 0.
//                                ([x],[y]) is a [HOME] or [SPACE] or [ORE].
// Returns:                      None.
// Side Effects:                  If insufficient space, an [errlog] message will be
//                                generated and the action will fail.
// Errors:                       [errlog] messages if preconditions violated. ChainMiner's
//                                action may be cancelled and replaced with the action
//                                [do_nothing()].

void grab_ore(const short x, const short y);

// Purpose:                      The ChainMiner will set [ag_intent] to select the taking
//                                of ore from ([x],[y]).
// Preconditions: The ChainMiner is adjacent to ([x],[y]). [cm_sack] !=
//                                [cm_sack_max].
//                                Ore [ORE] exist at ([x],[y]).
// Returns:                      None.
// Side Effects:                  None.
// Errors:                       [errlog] messages if preconditions violated. ChainMiner's
//                                action may be cancelled and replaced with the action
//                                [do_nothing()].

const bool sack_full();

// Purpose:                      Returns whether [cm_sack] == [cm_sack_max].
// Preconditions: None.
// Returns:                      [bool].
// Side Effects:                  None.
// Errors:                      None.

```

```

const bool sack_empty();

// Purpose:          Returns whether [cm_sack] == [0].
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

const short sack_space();

// Purpose:          Returns space remaining in [cm_sack].
// Preconditions: None.
// Returns:           [short].
// Side Effects:      None.
// Errors:            None.

const bool tired();

// Purpose:          Returns whether [cm_endurance] <= [0].
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

const bool recovered();

// Purpose:          Returns whether [cm_endurance] == [12].
// Preconditions: None.
// Returns:           [bool].
// Side Effects:      None.
// Errors:            None.

void tire();

// Purpose:          Reduces [cm_endurance] by 1. Minimum is [0].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

void recover();

// Purpose:          Increases [cm_endurance] by 1. Maximum is [12].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

void set_facing(const short x, const short y);

// Purpose:          Sets [cm_facing] according to [x] and [y].
// Preconditions: None.
// Returns:           None.
// Side Effects:      None.
// Errors:            None.

const bool facing_is_space(short &x, short &y);

// Purpose:          If ChainMiner is facing a space according to
//                   [cm_facing], [x] and [y] are loaded with the
//                   coordinates of that adjacent space. Otherwise
//                   a random adjacent space is loaded. Otherwise
//                   returns [false]

```

```

// Preconditions: None.
// Returns:      [bool].
// Side Effects:  None.
// Errors:       None.

// *** PRIVATE MEMBER VARIABLES ***

short cm_sack;      // a sack for carrying things
short cm_sack_max;  // max capacity of sack
short cm_scoop;     // max amount Miner can scoop in one turn
short cm_endurance; // getting tired
short cm_facing;    // index over MOVES[] that records a heading

};

#endif

//      Filename:      ChainMiner.cpp
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00 Console Application, release build

//      Purpose:       See ChainMiner.h

//      Refer to Agent.h for details of some functions.

#include "stdafx.h"
#include "ChainMiner.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** OPERATORS ***

ChainMiner & ChainMiner::operator =(const ChainMiner &a)

{

    // base class privates
    ag_m = a.ag_m;
    ag_s = a.ag_s;
    ag_x = a.ag_x;
    ag_y = a.ag_y;
    ag_target_x = a.ag_target_x;
    ag_target_y = a.ag_target_y;
    ag_env = a.ag_env;
    ag_name = a.ag_name;
    ag_v_range = a.ag_v_range;
    ag_intent = a.ag_intent;
    ag_type = a.ag_type;
    ag_orig_type = a.ag_orig_type;
    ag_alive = a.ag_alive;
    ag_target = a.ag_target;
    ag_target_obj = a.ag_target_obj;
    ag_dim = a.ag_dim;
    ag_search = a.ag_search;
    ag_search_prep = a.ag_search_prep;

    // ChainMiner privates

```

```

        cm_sack = a.cm_sack;                // a sack for carrying things
        cm_sack_max = a.cm_sack_max;        // max capacity of sack
        cm_scoop = a.cm_scoop;              // max amount Miner can scoop in one turn
        cm_endurance = a.cm_endurance;      // getting tired
        cm_facing = a.cm_facing;            // heading

        return(*this);
    }

//    *** PUBLIC FUNCTIONS **

void ChainMiner::get_state(short &t, short &n, short &x, short &y, short &a, short &q,
                           short &tx, short &ty)

{
    t = ag_type;
    n = ag_name;
    x = ag_x;
    y = ag_y;
    a = ag_intent.get_action();
    q = cm_sack;

    if(ag_target)
    {
        tx = ag_target_x;
        ty = ag_target_y;
    }
    else
    {
        tx = -1;
        ty = -1;
    }
}

void ChainMiner::think()
{
    Agent::think();

    short x,y;

    // REMOVE ALL MOBILE THINGS FROM MEMORY
    expire_moving_objects();

    // OBSERVE THE ENVIRONMENT
    look();

    // HANDLE TIRING

```

```

if(tired())
{
    if(!sack_empty())
    {
        // deposit at home base
        if(adjacent_to(x,y,HOME))
        {
            empty_backpack(x,y);
            return;
        }

        // deposit on existing ore
        if(adjacent_to(x,y,ORE))
        {
            empty_backpack(x,y);
            return;
        }

        // deposit on a space, facing preferred
        if(facing_is_space(x,y))
        {
            empty_backpack(x,y);
            return;
        }

        do_nothing();
        erase_target();
        return;
    }

    do_nothing();
    erase_target();
    return;
}

// if not recovered and sack empty
if(!recovered() && sack_empty())
{
    do_nothing();
    return;
}

// DROPPING MINERALS AT A HOME BASE

// if next to a home and has some minerals

```



```

if(!sack_empty() && adjacent_to(x,y,HOME))
{
    // then deposit minerals and finish this turn
    empty_backpack(x,y);
    return;
}

// MINING MINERALS

// if next to minerals and sack isn't full
if(!sack_full() && adjacent_to(x,y,MINERAL))
{
    // then mine minerals and end this turn
    mine_minerals(x,y);
    return;
}

// LOCATING AND HEADING TOWARDS MINERALS

// if knows of minerals and sack isn't full and not targeting minerals
if(ag_target_obj != MINERAL && !sack_full() && knows_of(MINERAL))
{
    // if can get to closest
    if(knows_of_closest(x,y,MINERAL))
    {
        // if can get to minerals
        if(seek_target(x,y,MINERAL))
        {
            // then end this turn
            return;
        }
    }
}

// FINDING A HOME BASE TO DEPOSIT MINERALS

// if sack is full and not already targetting a home base
if(ag_target_obj != HOME && sack_full())
{
    // if knows of a home base and is nearest
    if(knows_of_closest(x,y,HOME))
    {
        // if can get to home base
        if(seek_target(x,y,HOME))

```

```

        {

            // then end this turn
            return;

        }

    }

}

// RETURNING TO A HOME BASE - NO MORE MINERALS TO MINE

// if has some minerals but does not know of any more and not targeting minerals
if(ag_target_obj != HOME && !sack_empty() && !knows_of(MINERAL))

{

    // if knows of a closest home base
    if(knows_of_closest(x,y,HOME))

    {

        // if can get to home base
        if(seek_target(x,y,HOME))

        {

            // then end this turn
            return;

        }

    }

}

// GRABBING ORE

// if next to ore and sack isn't full and recovered
if(!sack_full() && recovered() && adjacent_to(x,y,ORE))

{

    // then mine ore and end this turn
    grab_ore(x,y);
    return;

}

// LOCATING AND HEADING TOWARDS ORE

// if knows of ore and sack isn't full and not targeting ore and not knows
// of any minerals
if(recovered() && ag_target_obj != ORE && !sack_full() &&
    knows_of(ORE) && !knows_of(MINERAL))

{

    // if can get to closest
    if(knows_of_closest(x,y,ORE))

```

```

    {

        // if can get to ore
        if(seek_target(x,y,ORE))

        {

            // then end this turn
            return;

        }

    }

}

// CONTINUING TARGETTING

// if targetting anything
if(targetting())

{

    // if can get to it
    if(seek_target(ag_target_x,ag_target_y,ag_target_obj))

    {

        // then end this turn
        return;

    }

}

// STARTING WANDERING

// if nowhere to wander to
if(!wander(x,y))

{

    do_nothing();

}

else

{

    if(seek_target(x,y,SPACE))

    {

        // then end this turn
        return;

    }

    else

    {

```

```

        do_nothing();
    }

}

const bool ChainMiner::sack_full()
{
    return(cm_sack == cm_sack_max);
}

void ChainMiner::mine_minerals(const short x, const short y)
{
    if(sack_full())
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::mine_minerals(x, y), Miner\n";
        err << "wants to mine minerals but his backpack is full.\n";
        err << "Action has failed and Miner will do nothing!\n";
        err.close();
        do_nothing();
        return;
    }

    if(ag_m.check(x,y))
    {
        if(ag_m.get_f(x,y) == MINERAL && adjacent_to(x,y))
        {
            ag_intent.put(GRAB,x,y);
            // stop any targetting
            erase_target();
        }
        else
        {
            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, ChainMiner::mine_minerals(x, y), Either\n";
            err << "minerals do not exist or Miner is not adjacent\n";
            err << "to any minerals. Action has failed and Miner\n";
            err << "will do nothing!\n";
            err.close();
            do_nothing();
        }
    }
}

```

```

    }

    else

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::mine_minerals(x, y), Invalid\n";
        err << "coordinates. Action has failed and Miner will do\n";
        err << "nothing!\n";
        err.close();
        do_nothing();

    }

}

void ChainMiner::grab_ore(const short x, const short y)

{

    if(sack_full())

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::grab_ore(x, y), Miner\n";
        err << "wants to pick up ORE but his backpack is full.\n";
        err << "Action has failed and Miner will do nothing!\n";
        err.close();
        do_nothing();
        return;

    }

    if(ag_m.check(x,y))

    {

        if(ag_m.get_f(x,y) == ORE && adjacent_to(x,y))

        {

            ag_intent.put(GRAB, x, y);
            // stop any targetting
            erase_target();

        }

        else

        {

            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, ChainMiner::grab_ore(x, y), Either\n";
            err << "ORE does not exist or Miner is not adjacent\n";
            err << "to any ORE. Action has failed and Miner\n";
            err << "will do nothing!\n";
            err.close();
            do_nothing();

        }

    }

}

```

```

    }

    else

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::grab_ore(x, y), Invalid\n";
        err << "coordinates. Action has failed and Miner will do\n";
        err << "nothing!\n";
        err.close();
        do_nothing();

    }

}

void ChainMiner::empty_backpack(const short x, const short y)
{

    if(sack_empty())

    {

        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::empty_backpack(x, y), Miners\n";
        err << "backpack is empty.Action has failed and Miner\n";
        err << "will do nothing!\n";
        err.close();
        do_nothing();
        return;

    }

    if(ag_m.check(x,y))

    {

        short region = ag_m.get_f(x,y);

        if((region == HOME || region == ORE || region == SPACE) && adjacent_to(x,y))

        {

            ag_intent.put(DROP,x,y);
            // stop any targetting
            erase_target();

        }

        else

        {

            ofstream err("errlog.txt", ios::app);
            err << "\n\nERROR, ChainMiner::empty_backpack(x, y), Illogical\n";
            err << "location to drop ORE. Action has failed and Miner\n";
            err << "will do nothing!\n";
            err.close();
            do_nothing();

        }

    }

}

```

```

    }

}

else

{

    ofstream err("errlog.txt", ios::app);
    err << "\n\nERROR, ChainMiner::empty_backpack(x, y), Invalid\n";
    err << "coordinates. Action has failed and Miner will do\n";
    err << "nothing!\n";
    err.close();
    do_nothing();

}

}

const bool ChainMiner::sack_empty()

{

    return(cm_sack == 0);

}

const short ChainMiner::sack_space()

{

    return(cm_sack_max - cm_sack);

}

void ChainMiner::execute_mine_minerals(const short x, const short y)

{

    short object, quantity, grab, grabbed;
    ag_env -> get_object(x, y, object, quantity);
    if(object == MINERAL && quantity > 0)
    {

        grab = sack_space();
        if(grab >= cm_scoop)
        {

            grab = cm_scoop;

        }

        grabbed = ag_env -> take_quantity_f(x,y,grab);
        cm_sack += grabbed;

    }

}

```

```

}

void ChainMiner::execute_grab_ore(const short x, const short y)
{
    short object, quantity, grab, grabbed;

    ag_env -> get_object(x, y, object, quantity);

    if(object == ORE && quantity > 0)
    {
        grab = sack_space();    // can pick up sack_space if no mining effort

        grabbed = ag_env -> take_quantity_f(x,y,grab);

        cm_sack += grabbed;

    }
}

void ChainMiner::execute_empty_backpack(const short x, const short y)
{
    short object = ag_env -> get_object_f(x,y);

    if((object == HOME || object == ORE || object == SPACE))
    {
        if(ag_env -> add_quantity_f(x,y,ORE,cm_sack))
        {
            cm_sack = 0;

        }
        else
        {
            ofstream err("errlog.txt", ios::app);
            err << "\n\nCAUTION, ChainMiner::execute_empty_backpack(x, y), There\n";
            err << "was not enough space to drop the objects.\n";
            err << "Action has failed!\n";
            err.close();

        }
    }
    else
    {
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::execute_empty_backpack(x, y), Miner\n";
    }
}

```



```

        err << "attempted to drop cargo at invalid region.\n";
        err << "Action has failed!\n";
        err.close();
    }
}

void ChainMiner::execute_action()
{
    if(ag_alive)
    {
        short a, x, y;
        ag_intent.get(a,x,y);
        set_facing(x,y);

        switch(a)
        {
            case NOTHING:
                recover();
                break;

            case MOVE:
                execute_move();
                if(!sack_empty())
                {
                    tire();
                }
                else
                {
                    recover();
                }
                break;

            case GRAB:
                if(ag_env -> get_object(x,y) == MINERAL)
                {
                    execute_mine_minerals(x, y);
                    recover();
                }

                else if(ag_env -> get_object(x,y) == ORE)

```

```

        {
            execute_grab_ore(x, y);
        }

        break;

    case DROP:
        execute_empty_backpack(x,y);
        break;

    default:
        ofstream err("errlog.txt", ios::app);
        err << "\n\nERROR, ChainMiner::execute_action(), Miner " << ag_name << endl;
        err << "attempting to perform illogical action. Action failed!\n";
        err.close();
        break;
    }
}

const bool ChainMiner::tired()
{
    return(cm_endurance <= 0);
}

const bool ChainMiner::recovered()
{
    return(cm_endurance == 12);
}

void ChainMiner::tire()
{
    cm_endurance--;
    if(cm_endurance < 0)
    {
        cm_endurance = 0;
    }
}

void ChainMiner::recover()
{

```

```

        cm_endurance++;

        if(cm_endurance > 12)
        {
            cm_endurance = 12;
        }
    }

    void ChainMiner::set_facing(const short x, const short y)
    {
        short c1,c2;

        for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
        {
            c2 = c1 + 1;

            // get index of this heading over moves[]
            if((x == ag_x + moves[c1]) && (y == ag_y + moves[c2]))
            {
                cm_facing = c1;
                break;
            }
        }
    }

    const bool ChainMiner::facing_is_space(short &x, short &y)
    {
        x = ag_x + moves[cm_facing];
        y = ag_y + moves[cm_facing+1];

        // if facing location is space - select it
        if(ag_m.check(x,y))
        {
            if(ag_m.get_f(x,y) == SPACE)
            {
                return(true);
            }
        }
        else
    
```

```

{
    // count spaces around agent and select by random
    short c1,c2,cx,cy;
    PATH p;
    triple t;

    for(c1 = 0 ; c1 < (MOVE_DIM - 1) ; c1+=2)
    {
        c2 = c1 + 1;
        cx = ag_x + moves[c1];
        cy = ag_y + moves[c2];

        // get list of spaces
        if(ag_m.check(cx,cy))
        {
            if(ag_m.get_f(cx,cy) == SPACE)
            {
                t.put(cx,cy,0);
                p.push_back(t);
            }
        }
    }

    if(p.size())
    {
        c1 = rand() % p.size();
        p[c1].get(x,y,c2);
        return(true);
    }
}

return(false);
}

```

This class is very similar to the Miner class. We could have derived this class from the Miner class but preferred to not get involved with any multiple-inheritance issues. Essentially, the difference from ordinary Miners is that these Chain-Miners “know” about the concept of ore and “tire” whilst carrying it.

C.9 The Crowd Class

The features of this class are briefly discussed in section 3.4.

```
//      Filename:      Crowd.h
//      Version:      Final
//      Author:      Paul Tinsley (pmbtin)
//      Date:      26 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:      Numbers, functions and data variables are enclosed in
//                      square parentheses "[]". This is to remove any
//                      ambiguity within the text of comments.

//      Errors:      All error messages appended to "errlog.txt".

//      Purpose:      Crowd Agents. Used to illustrate how crowd behaviours
//                      may be implemented and investigated.

// Function Comment Style:

// Purpose:      Purpose of the function.
// Preconditions:      What states or conditions must be met before execution
//                      of the function.
// Returns:      The object or primitive type that the function returns.
// Side Effects:      Effects that the function may have on private data that
//                      are not immediately obvious to the nature of the function.
//                      This section may also contain advice and miscellaneous
//                      comments.

// Errors:      What type of error is generated, why it is generated and
//                      its effects.

#ifndef __CROWD_H
#define __CROWD_H

#include "Agent.h"

class Crowd : public Agent
{
    public:

    // *** CONSTRUCTORS ***

    Crowd(): Agent({});

    // Purpose:      Default constructor.
    // Preconditions: None.
```

```

// Returns:          None.
// Side Effects:     None.
// Errors:           None.

Crowd(short s,short x,short y,Environment *e,short n,short r)
    :Agent(s,x,y,e,n,r,MINER){};

// Purpose:          Constructor. See base constructor definition in
//                   [Agent::Agent(args)].
// Preconditions: None.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

//      *** OPERATORS ***

const friend bool operator ==(const Crowd &a, const Crowd &b)

// Purpose:          Equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(a.ag_name == b.ag_name);

}

const friend bool operator !=(const Crowd &a, const Crowd &b)

// Purpose:          Non-equality operator.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(!(a.ag_name == b.ag_name));

}

const friend bool operator <(const Crowd &a, const Crowd &b)

// Purpose:          Less-than operator. An agent is less than another
//                   agent if its name is less in value.
// Preconditions: None.
// Returns:          [bool].
// Side Effects:     None.
// Errors:           None.

{

    return(a.ag_name < b.ag_name);

};

const friend bool operator >(const Crowd &a, const Crowd &b)

// Purpose:          Greater-than operator. An agent is greater than another
//                   agent if its name is greater in value.

```

```

// Preconditions: None.
// Returns:      [bool].
// Side Effects:  None.
// Errors:       None.

{

    return(a.ag_name > b.ag_name);

};

//      *** PUBLIC FUNCTION PROTOTYPES ***

virtual void think();

// Purpose:      The behavioural rules for these Crowd agents.
// Preconditions: None.
// Returns:      None.
// Side Effects:  None.
// Errors:       None.

virtual void execute_action();

// Purpose:      Causes the Crowd agent to execute an action based on the contents
//               of its [ag_intent].
// Preconditions: [ag_intent] has been arbitrated.
// Returns:      None.
// Side Effects:  None.
// Errors:       None.

virtual void kill();

// Purpose:      Sets [ag_alive] to [false]. Sets [ag_type] to [SPACE].
// Preconditions: None.
// Returns:      None.
// Side Effects:  Records new type and quantity in the environment [ag_env] and
//               sets action to [NOTHING].
// Errors:       None.

};

#endif

//      Filename:      Crowd.cpp
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00

//      Purpose:       See Crowd.h. Derived from base class Agent.h.

//      Refer to Agent.h for details of some functions.

#include "stdafx.h"
#include "Crowd.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** PUBLIC FUNCTIONS **

```

```

void Crowd::think()
{
    if(!is_alive())
    {
        do_nothing();
        return;
    }

    Agent::think();

    short x,y;

    // REMOVE ALL MOBILE AGENTS FROM MEMORY
    expire_moving_objects();

    // OBSERVE THE ENVIRONMENT
    look();

    // DISSAPEAR WHEN AT EXIT
    if(adjacent_to(HOME))
    {
        kill();
        do_nothing();
        return;
    }

    // GET TO THE HOME - MAKE SURE ALWAYS CLOSEST KNOWN
    if(ag_target_obj != HOME && knows_of(HOME))
    {
        if(knows_of_closest(x,y,HOME))
        {
            if(seek_target(x,y,HOME))
            {
                return;
            }
        }
    }

    if(targetting())
    {

```



```

        if(seek_target(ag_target_x,ag_target_y,ag_target_obj))
        {
            return;
        }
    }
    if(wander(x,y))
    {
        if(seek_target(x,y,SPACE))
        {
            return;
        }
    }
    do_nothing();
}

void Crowd::execute_action()
{
    if(ag_alive)
    {
        short a,x,y;
        ag_intent.get(a,x,y);

        switch(a)
        {
            case NOTHING:
                break;

            case MOVE:
                execute_move();
                break;

            default:
                ofstream err("errlog.txt", ios::app);
                err << "\n\nERROR, Crowd::execute_action(), Crowder " << ag_name << endl;
                err << "attempting to perform illogical action. Action failed!\n";
                err.close();
                break;
        }
    }
}

```

```

    }
}

void Crowd::kill()
{
    ag_alive = false;
    ag_type = SPACE;
    ag_env -> put_object_f(ag_x,ag_y,ag_type,0);
}

```

These agents are very simple. All they are designed to do is locate the nearest home base and head towards it. Upon arrival at a home base, they “dissappear” from the environment.

C.10 The Arbitrator Class

Although the concept that we have developed for arbitration is simple, its implementation is rather complicated. Appendix T.3 provides an example of the arbitration method.



T.3 - The Arbitration Process - Page 161

```

//      Filename:      Arbitrator.h
//      Version:       Final
//      Author:        Paul Tinsley (pmbtin)
//      Date:          26 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:     Numbers, functions and data variables are enclosed in
//                    square parentheses "[ ]". This is to remove any
//                    ambiguity within the text of comments. All private data
//                    member variables are prefixed by "a_".

//      Purpose:       Generic arbitration. The arbitrator simply matches all agents
//                    that wish to perform the same action at the same location
//                    and selects a "winner" via priority or randomisation.

```

```
//      Function Comment Style:

//      Purpose:          Purpose of the function.
//      Preconditions:    What states or conditions must be met before execution
//                        of the function.
//      Returns:         The object or primitive type that the function returns.
//      Side Effects:     Effects that the function may have on private data that
//                        are not immediately obvious to the nature of the function.
//                        This section may also contain advice and miscellaneous
//                        comments.
//      Errors:          What type of error is generated, why it is generated and
//                        its effects.
```

```
#ifndef __ARBITRATOR_H
#define __ARBITRATOR_H
```

```
#include "Intent.h"          // agent intentions
#include "Environment.h"     // for action enumerations
#include <vector>              // for containers of [Intent]
#include <algorithm>          // sorting containers
```

```
using namespace std;
```

```
typedef vector<Intent> INTENT_LIST;          // list of [Intent]'s
typedef INTENT_LIST :: iterator INTENT_LIST_IT;
```

```
class Arbitrator
```

```
{
```

```
    public:
```

```
    //      *** CONSTRUCTORS ***
```

```
    Arbitrator();
```

```
    // Purpose:          Default constructor.
    // Preconditions: None.
    // Returns:         [Arbitrator] object.
    // Side Effects:     None.
    // Errors:          None.
```

```
    Arbitrator(const INTENT_LIST &i, const short fail);
```

```
    // Purpose:          Constructor. [a_intents] gets [i], [a_fail_value] gets [fail].
    // Preconditions: None.
    // Returns:         [Arbitrator] object.
    // Side Effects:     None.
    // Errors:          None.
```

```
    Arbitrator(const Arbitrator &x);
```

```
    // Purpose:          Copy Constructor.
    // Preconditions: None.
    // Returns:         [Arbitrator] object.
    // Side Effects:     None.
    // Errors:          None.
```

```
    //      *** OPERATORS ***
```

```
    Arbitrator & operator =(const Arbitrator &x);
```

```

// Purpose:          Destructive assignment operator.
// Preconditions: None.
// Returns:          [Arbitrator] object.
// Side Effects:      None.
// Errors:           None.

//      *** PUBLIC FUNCTION PROTOTYPES ***

void set_fail(const short f);

// Purpose:          [a_fail_value] gets [fail]. This is the action value that
//                   the arbitrator will record in an [Intent] that fails to
//                   "win" in arbitration. This should normally be [NOTHING].
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

void set_initial_intents(INTENT_LIST &i);

// Purpose:          [a_intents] gets [i]. The arbitrator needs a copy of the
//                   agents initial intents to maintain the structure of the
//                   arbitration system. The intentions for the last turn are
//                   always matched against those of the current turn in order
//                   to calculate priorities.
// Preconditions: Done before first [arbitrate(INTENT_LIST &i)] invocation.
//                   Thereafter, never again unless restarting the experiment.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

void arbitrate(INTENT_LIST &i);

// Purpose:          Arbitrate over [i]. Then copy [i] to [a_intents].
// Preconditions: [a_intents] represents the last turns arbitrated
//                   actions.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

private:

//      *** PRIVATE DATA ***

INTENT_LIST a_intents;           // copy of last intents
INTENT_LIST a_primary;          // agents intents without classification
short a_fail_value;             // value to load into intents that lose in arbitration

//      *** PRIVATE FUNCTIONS ***

void compare_intents(INTENT_LIST &i);

// Purpose:          Increase priorities for all intents in [i] that match same action
//                   and key in [a_intents]. Sets non-matching elements to zero
//                   priority as the agent has decided to perform a different action.
// Preconditions: None.
// Returns:          None.
// Side Effects:      None.
// Errors:           None.

void set_classes(INTENT_LIST &i);

// Purpose:          Maps all actions in [i] to the correct action class.

```

```

// Preconditions:[i] has not been arbitrated.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

void remove_classes(INTENT_LIST &i);

// Purpose:          Removes the action class and replaces the original
//                   action if the action is not [NOTHING].
// Preconditions:[i] has been arbitrated.
// Returns:          None.
// Side Effects:     None.
// Errors:           None.

};

#endif

//      Filename:          Arbitrator.cpp
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:         Visual C++ 5.00

//      Purpose:          See Arbitrator.h.

#include "stdafx.h"
#include "Arbitrator.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** CONSTRUCTORS ***

Arbitrator::Arbitrator() : a_fail_value(0)

{

    a_intents.clear();

}

Arbitrator::Arbitrator(const INTENT_LIST &i, const short fail): a_fail_value(fail)

{

    a_intents = i;
    a_primary = i;

}

Arbitrator::Arbitrator(const Arbitrator &x)

{

    a_intents = x.a_intents;
    a_primary = x.a_primary;
    a_fail_value = x.a_fail_value;

}

```

```

//      *** OPERATORS ***

Arbitrator & Arbitrator::operator =(const Arbitrator &x)

{

    a_intents = x.a_intents;
    a_primary = x.a_primary;
    a_fail_value = x.a_fail_value;
    return *this;

}

//      *** PUBLIC FUNCTIONS ***

void Arbitrator::set_fail(const short f)

{

    a_fail_value = f;

}

void Arbitrator::set_initial_intents(INTENT_LIST &i)

{

    a_intents = i;
    a_primary = i;

}

void Arbitrator::arbitrate(INTENT_LIST &i)

{

    // first update priorities
    compare_intents(i);

    INTENT_LIST_IT f;
    INTENT_LIST_IT f1;
    Intent match;
    INTENT_LIST found;
    INTENT_LIST result;
    INTENT_LIST original;

    // make action classes for arbitration
    set_classes(i);

    short count,clobber,winner;

    // now look for conflicts
    while(!i.empty())

    {

        f = i.begin();
        match = *f;

        f1 = f;

        // look for matches on basis of action and key

```

```

while(f1 != i.end())
{
    if(*f1 == match)
    {
        found.push_back(*f1);
        i.erase(f1);
        f1 = i.begin();
    }
    else
    {
        f1++;
    }
};

// sort the list of matches
sort(found.begin(),found.end(),compare_intent_priorities());

// probably have a list of matched items so deal with it
if(found.size() >= 2)
{
    f1 = found.begin();

    if((f1 -> get_priority()) > ((f1+1) -> get_priority()))
    {
        // winner loses his priority
        f1 -> reset_priority();
        f1++;

        while(f1 != found.end())
        {
            // set all of the losers

            f1 -> set_action(a_fail_value);
            f1 -> inc_priority();
            f1++;

        };
    }
    else
    {
        // have to count clobbers, pick winner and set rest to NOTHING
        f1 = found.begin();
        match = *f1;
        clobber = 0;
    }
}

```

```

while((f1 != found.end()) && (f1 -> get_priority() == match.get_priority()))
{
    clobber++;
    f1++;
};

winner = rand() % clobber;
f1 = found.begin();
count = 0;

while(f1 != found.end())
{
    // go right to end cos there may be lower priorities
    if(count != winner)
    {
        f1 -> set_action(a_fail_value);
        f1 -> inc_priority();
    }
    else
    {
        // winner loses his priority
        f1 -> reset_priority();
    }
    f1++;
    count++;
};
}

else
{
    // there must only be one - he wins and loses priority
    f1 = found.begin();
    f1 -> reset_priority();
}

// copy findings to result list
f1 = found.begin();

while(f1 != found.end())
{
    result.push_back(*f1);
    f1++;
}

```



```

        };

        found.clear();

        f++;

};

// put back actual actions
remove_classes(result);
// give result to caller
i = a_intents;

}

//      *** PRIVATE FUNCTIONS ***

void Arbitrator::compare_intents(INTENT_LIST &i)
{

    short lim = a_intents.size();
    short c;

    // sort both lists by agent's name - then we can index them
    sort(a_intents.begin(), a_intents.end(), compare_intent_names());
    sort(i.begin(), i.end(), compare_intent_names());

    for(c = 0 ; c < lim; c++)
    {

        if((a_intents[c].get_action() != i[c].get_action()) &&
            (a_intents[c].get_key() != i[c].get_key()))
        {

            i[c].reset_priority();

        }

    }

    // get copy of these original intents
    a_primary = i;

}

void Arbitrator::set_classes(INTENT_LIST &i)
{

    INTENT_LIST_IT it;

    for(it = i.begin() ; it != i.end() ; ++it)
    {

        switch(it -> get_action())

```

```

    {
        case MOVE:

            it -> set_action(C_INHABIT);
            break;

        case GRAB:

            it -> set_action(C_INTERACT);
            break;

        case DROP:

            it -> set_action(C_INHABIT);
            break;

        case POUNCE:

            it -> set_action(C_INTERACT);
            break;

        case JUMP:

            it -> set_action(C_INHABIT);
            break;

        default:

            break;

    };
}

}

void Arbitrator::remove_classes(INTENT_LIST &i)
{
    // sort intents
    sort(i.begin(),i.end(),compare_intent_names());

    // run through and set fails

    short count = 0;

    while(count < a_primary.size())
    {
        if(i[count].get_action() == a_fail_value)
        {
            a_primary[count].set_action(a_fail_value);
        }

        count++;
    };
};

```

```

// copy to a_intents for matching next turn
a_intents = a_primary;

}

```

C.11 The Administrator Class

First we provide the code for the Administrator class. Then we briefly discuss its most salient features.

```

//      Filename:      Adminitrator.h
//      Version:      Final
//      Author:      Paul Tinsley (pmbtin)
//      Date:      26 February, 1999
//      Platform:      Visual C++ 5.00

//      Symbology:      Numbers, functions and data variables are enclosed in
//                      square parentheses "[]". This is to remove any
//                      ambiguity within the text of comments. All private data
//                      member variables are prefixed by "ad_".

//      Errors:      All error messages appended to "errlog.txt"

//      Purpose:      Administrates over a collection of possibly heterogeneous
//                      agents derived from a common base class. The functionality
//                      allows a turn-based control and thus the order of calls
//                      is important. Various access functions allow data to be
//                      gathered for experiments and animation. The Administrator
//                      is capable of generating a script that contains the state
//                      of all objects that may be used for animation and also
//                      information processing, such as plotting tracks of agents
//                      through the environment.

//      Function Comment Style:

//      Purpose:      Purpose of the function.
//      Preconditions:      What states or conditions must be met before execution
//                      of the function.
//      Returns:      The object or primitive type that the function returns.
//      Side Effects:      Effects that the function may have on private data that
//                      are not immediately obvious to the nature of the function.
//                      This section may also contain advice and miscellaneous
//                      comments.
//      Errors:      What type of error is generated, why it is generated and
//                      its effects.

//      Scripts:

/*

```

Scripts are vector containers of the class [chunk]. Their organisation is as follows:

TURN n:

```
info field (1,number of frames for turn n,0,0,0,0,0,0,0)
anim frame (0,ads_type,ads_name,ads_x,ads_y,ads_action,ads_quantity,ads_target_x,
            ads_target_y)
```

```
.
.
.
.
```

TURN n+1:

```
info field (1,number of frames for turn n+1,0,0,0,0,0,0,0)
anim frame .....
```

```
.
.
.
```

*/

// Example scenario control loop. The administrator instance is [admin]:

/*

**** START TURN ****

```
admin.deliberate();           // invoke [think()] in every agent
```

```
admin.get_intentions();       // collect intended actions from agents
```

```
admin.arbitrate();           // arbitrate the actions using the arbitrator class
```

```
*** SPECIAL EVENTS ***      // place special functions here to handle scenario
                             // specific events. You can intercept the writing
                             // back of agents' intentions.
```

```
admin.set_intentions();       // write back the intentions to the agents
```

```
admin.execute_actions();      // invoke [execute_action()] in every agent
```

```
admin.update_script();        // if creating an animation script, place the
                             // call here
```

**** END TURN ****

*/

```
#ifndef __ADMINISTRATOR_H
#define __ADMINISTRATOR_H
```

```
#include "Agent.h"           // base class of agents
#include "Miner.h"            // basic mining agents
#include "ChainMiner.h"       // chain mining agents
#include "Predator.h"         // predator agents
#include "Prey.h"             // prey agents
#include "Crowd.h"            // crowd agents
#include "Intent.h"           // agents intentions
#include "Arbitrator.h"       // the arbitrator
#include "DataStructures.h"   // various data structures / types
#include <time.h>              // to seed random number generator
#include <stddef.h>            // for program exit
#include <vector>              // containers of agents
```

```

using namespace std;

typedef vector<Agent*> AGENTS;           // agents derived from a base class
typedef AGENTS :: iterator AGENTS_IT;

// Note - vector contains pointers to a base class, therefore derived
// members pointed to in a filled vector are dynamically binded via
// virtual functions and the inheritance tree

//      *** CLASS DEFINITION ***

class Administrator

{

    public:

    //      *** CONSTRUCTOR ***

    Administrator();

    // Purpose:          Default constructor. All [Administrator] instances have a
    //                  two-stage construction process. Use the specific setup
    //                  function for the type of scenario being used.
    // Preconditions: None.
    // Returns:          [Administrator] object.
    // Side Effects:     1st stage of 2 stage construction.
    // Errors:           None.

    //      *** DESTRUCTOR ***

    ~Administrator();

    // Purpose:          Releases any memory allocated through the base class
    //                  [Agent] pointers in [ad_agents].
    // Preconditions: None.
    // Returns:          None.
    // Side Effects:     None.
    // Errors:           None.

    //      *** PUBLIC FUNCTION PROTOTYPES ***

    bool setup_miner1(const bool explore, const short miners, const short miner_r,
                     const short grab, const short sack, const short grid_size,
                     ifstream &m);

    // Purpose:          Second stage construction for the basic mining experiment.
    //                  [explore] reflects whether agents will already know their
    //                  environment, [true] is they do not. [miners] is the number
    //                  of basic mining agents. [miner_r] is the mining agents' visual
    //                  range. [grab] is the miners' grabbing capacity. [sack] is the
    //                  miners' sack contents capacity. [grid_size] is the dimension
    //                  of the environment contained in the stream [m].
    // Preconditions: At least one miner up to as many as you like. [m] must contain
    //                  environment map data and meet the preconditions set out in
    //                  [Grid::input(ifstream &in)].
    // Returns:          Setup will return [false] if there is not enough space in the
    //                  environment to place the miners or [miners] <= 0.
    // Side Effects:     2nd stage of 2 stage construction.
    // Errors:           [errlog] messages if not enough space for agents, or the map
    //                  file stream or map data is invalid, or [miners] <= 0. Returns
    //                  [false].

```

```

bool setup_miner2(const bool explore, const short miners, const short miner_r,
                  const short grab, const short sack, const short grid_size,
                  ifstream &m);

// Purpose:      Second stage construction for the chain-mining experiment.
//               [explore] reflects whether agents will already know their
//               environment, [true] is they do not. [miners] is the number
//               of chain-mining agents. [miner_r] is the mining agents' visual
//               range. [grab] is the miners' grabbing capacity. [sack] is the
//               miners' sack contents capacity. [grid_size] is the dimension
//               of the environment contained in the stream [m].
// Preconditions: At least one miner up to as many as you like. [m] must contain
//               environment map data and meet the preconditions set out in
//               [Grid::input(ifstream &in)].
// Returns:      Setup will return [false] if there is not enough space in the
//               environment to place the miners or [miners] <= 0.
// Side Effects: 2nd stage of 2 stage construction.
// Errors:       [errlog] messages if not enough space for agents, or the map
//               file stream or map data is invalid, or [miners <= 0]. Returns
//               [false].

bool setup_hunting(const bool explore, const short predators, const short prey,
                   const short predator_r, const short prey_r, const short stom,
                   const short stom_max, const short grid_size, ifstream &m);

// Purpose:      Second stage construction for the predator & prey experiment.
//               [explore] reflects whether agents will already know their
//               environment, [true] is they do not. [predators] is the number
//               of predator agents. [prey] is the number of prey agents.
//               [predator_r] is the predator agents' visual range. [prey_r] is
//               the prey agents' visual range. [stom] is the predators' initial
//               stomach contents. [stom_max] is the predators' maximum stomach
//               capacity. [grid_size] is the dimension of the environment
//               contained in the stream [m].
// Preconditions: At least one agent per category up to as many as you like. [m]
//               must contain environment map data and meet the preconditions
//               set out in [Grid::input(ifstream &in)].
// Returns:      Setup will return [false] if there is not enough space in the
//               environment to place the agents or [predators+prey] <= 0.
// Side Effects: 2nd stage of 2 stage construction.
// Errors:       [errlog] messages if not enough space for agents, or the map
//               file stream or map data is invalid, or [predators <= 0], or
//               [prey <= 0]. Returns [false].

bool setup_crowd1(const bool explore, const short crowders, const short crowder_r,
                  const short grid_size, ifstream &m);

// Purpose:      Second stage construction for the crowd experiment.
//               [explore] reflects whether agents will already know their
//               environment, [true] is they do not. [crowders] is the number
//               of crowd agents. [crowder_r] is the crowd agents' visual
//               range. [grid_size] is the dimension of the environment
//               contained in the stream [m].
// Preconditions: At least one crowd agent up to as many as you like. [m]
//               must contain environment map data and meet the preconditions
//               set out in [Grid::input(ifstream &in)].
// Returns:      Setup will return [false] if there is not enough space in the
//               environment to place the crowders or [crowders] <= 0..
// Side Effects: 2nd stage of 2 stage construction.
// Errors:       [errlog] messages if not enough space for agents, or the map
//               file stream or map data is invalid, or [crowders] <= 0. Returns
//               [false].

```

```

void check_hunting_POUNCE();

// Purpose:          Special function for the hunting experiment. If a Predator
//                   agent has the action [POUNCE], searches for the Prey that
//                   was pounced on and rolls random death. Predator has a 40%
//                   chance of killing Prey. If Prey killed, Administrator invokes
//                   the [kill()] function for the Prey and sets its action to
//                   [NOTHING].
// Preconditions:[ad_intents] have been arbitrated.
// Returns:          None.
// Side Effects:     See purpose.
// Errors:          None.

void deliberate();

// Purpose:          Executes the [think()] function for every agent.
// Preconditions:Agents must exist!
// Returns:          None.
// Side Effects:     Agents deliberate and set their intentions.
// Errors:          None.

void get_intentions();

// Purpose:          Collects all agents' intentions into [ad_intents].
// Preconditions:Only makes sense to use this if initial turn or just
//                   deliberated.
// Returns:          None.
// Side Effects:     [ad_intents] is loaded with new data.
// Errors:          None.

void set_intentions();

// Purpose:          Copies the [ad_intents] back to each agent.
// Preconditions:The [ad_intents] have been arbitrated.
// Returns:          None.
// Side Effects:     Agent's intentions may have been changed by arbitration.
// Errors:          None.

void execute_actions();

// Purpose:          Triggers the [execute_action()] function within every agent.
// Preconditions:None.
// Returns:          None.
// Side Effects:     Agents execute their intention and may effect the environment.
// Errors:          None.

void update_script();

// Purpose:          Gets the state of each agent and every object exluding
//                   [OBSTACLE]. Updates [ad_script] and [ad_last_script].
// Preconditions:Agents have just executed their actions [execute_actions()].
// Returns:          None.
// Side Effects:     [ad_script] gets longer, [ad_last_script] is replaced.
// Errors:          None.

void arbitrate();

// Purpose:          Hands [ad_intents] to the arbitrator for arbitration.
// Preconditions:[ad_intents] must have been loaded with each agent's intent
//                   after deliberation.
// Returns:          None.
// Side Effects:     Arbitrator may change the intents.

```

```

// Errors:                None.

void get_script(SCRIPT &s);

// Purpose:                Loads [s] with [ad_script]. This script contains the state of
//                          every object for every executed turn.
// Preconditions: None.
// Returns:                None.
// Side Effects:           None.
// Errors:                None.

void get_last_turn(SCRIPT &s);

// Purpose:                Loads [s] with [ad_last_script]. This script contains the state of
//                          every object for the last executed turn.
// Preconditions: None.
// Returns:                None.
// Side Effects:           None.
// Errors:                None.

private:

//      *** PRIVATE DATA MEMBERS ***

AGENTS ad_agents;          // pointer collection of agents
Environment ad_env;        // the environment
short ad_dim;              // environment dimension
INTENT_LIST ad_intents;    // agents' intentions
Arbitrator ad_arb;        // the arbitrator
long ad_script_pos;        // where we are up to in the script
SCRIPT ad_script;          // to hold data for animation script
SCRIPT ad_last_script;     // last turns script - only data for that turn

// private data invloved with creating animation scripts
// ordering is important when written into the type [chunk] within the script

short ads_type;            // type of object
short ads_name;            // object name if agent, else 0
short ads_x;               // current x-coord of object
short ads_y;               // current y-coord of object
short ads_action;          // current action if agent
short ads_quantity;        // quantity of object or quantity carried by agent
short ads_target_x;        // current agent targetted x-coord
short ads_target_y;        // current agent targetted y-coord

void deallocate_agents();

// Purpose:                De-allocates agents in [ad_agents].
// Preconditions: None.
// Returns:                None.
// Side Effects:           None.
// Errors:                None.

};

#endif

//      Filename:          Adminitrator.cpp
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

```



```

//      Purpose:          See Administrator.h.

#include "stdafx.h"
#include "Administrator.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//      *** CONSTRUCTOR ***

Administrator::Administrator() :ad_agents(NULL), ad_env(40), ad_arb(),
                                ad_script_pos(0), ads_type(0), ads_name(0),
                                ads_x(0), ads_y(0), ads_action(0), ads_quantity(0),
                                ads_target_x(0), ads_target_y(0), ad_dim(40)

{

    ad_agents.clear();
    ad_intents.clear();
    ad_script.clear();
    ad_last_script.clear();

}

//      *** DESTRUCTOR ***

Administrator::~Administrator()

{

    deallocate_agents();
    ad_agents.clear();
    ad_intents.clear();
    ad_script.clear();
    ad_last_script.clear();

}

//      *** PUBLIC FUNCTIONS ***

bool Administrator::setup_miner1(const bool explore, const short miners,
                                const short miner_r, const short grab,
                                const short sack, const short grid_size,
                                ifstream &m)

{

    if(miners <= 0)

    {

        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_miner1(args), there must be\n";
        err << "at least one miner agent. Setup failed.\n";
        err.close();
        return(false);

    }

    // metamorphose environment to correct size

```

```

ad_env.metamorphose(grid_size);

// get the environment dimension
ad_dim = grid_size;

// input the environment map
ad_env.input_objects(m);

// check environment state
if(!ad_env.get_state())

{

    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_miner1(args), map file stream was\n";
    err << "not freshly opened, or the data file was invalid. Unable to\n";
    err << "setup this scenario.\n";
    err.close();
    return(false);

}

PATH temp;
triple alloc;
short x,y;
bool starts_exist = false;

// check for user defined ASTART positions
if(ad_env.contains_object(ASTART))
{
    starts_exist = true;

    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object_f(x,y) == ASTART)
            {
                alloc.put(x,y,1);
                temp.push_back(alloc);
            }
        }
    }

    // are there enough start positions?
    if(temp.size() < miners)
    {

        // add all SPACE too
    }
}

```

```

        for(x = 0 ; x < ad_dim ; x++)
        {
            for(y = 0 ; y < ad_dim ; y++)
            {
                if(ad_env.get_object(x,y) == SPACE)
                {
                    alloc.put(x,y,0);
                    temp.push_back(alloc);
                }
            }
        }

        // erase ASTART markers
        ad_env.clear_objects(ASTART);
    }
else
{
    // build a list of available locations to place agents
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object(x,y) == SPACE)
            {
                alloc.put(x,y,0);
                temp.push_back(alloc);
            }
        }
    }
}

// de-allocate agents from the heap
deallocate_agents();

// erase agents
ad_agents.clear();

```

```

// erase intents
ad_intents.clear();

// erase animation scripts
ad_script.clear();
ad_last_script.clear();

// setup arbitrator fail value
ad_arb.set_fail(NOTHING);

// check that enough space for agents
if(temp.size() < miners)

{

    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_miner1(args), there was not\n";
    err << "enough space in the environment to place all of the agents.\n";
    err << "Scenario setup failed.\n";
    err.close();
    return(false);

}

// seed random number generator
srand( (unsigned)time( NULL ) );

// place agents randomly within available space
short i;                // loop counter
short ax;                // generic coordinate variable
short ay;                // generic coordinate variable
PATH_IT loc = temp.begin(); // to manipulate available spaces to place agents
short place;            // to find a place
short max;              // loop limit

// Agent names must run sequentially with vector index from
// zero to (number of agents)-1

short junk;
Miner *mr;              // pointer to a Miner

// create Miners

for(i = 0 ; i < miners ; i++)

{

    if(starts_exist)

    {

        loc = temp.begin();
        loc -> get(ax,ay,junk);

        // if we have hit a space, stop fixed alloc
        if(junk == 0)

        {

            starts_exist = false;
            max = temp.size();           // get size of available space
            place = rand() % max;        // do random selection
            loc = temp.begin() + place;  // find place

```

```

        loc -> get(ax,ay,junk);           // get coordinates
        temp.erase(loc,loc+1);          // erase this place - can't be used again
    }

    else

    {

        temp.erase(loc,loc+1);

    }

}

else

{

    // place agents randomly over available space
    max = temp.size();                  // get size of available space
    place = rand() % max;               // do random selection
    loc = temp.begin() + place;         // find place
    loc -> get(ax,ay,junk);             // get coordinates
    temp.erase(loc,loc+1);             // erase this place - can't be used again

}

mr = new Miner(ad_dim,0,0,&ad_env,0,miner_r,sack,grab);
mr -> init(i,ax,ay,MINER);             // set name and start coords
mr -> record_position();               // make agent record its position

if(!explore)                          // if agents not exploring

{

    mr -> get_environment();            // load map into memory

}

ad_agents.push_back(mr);               // make this agent

}

// tidy up expensive memory!
ad_intents.resize(miners);
ad_agents.resize(miners);

// do initial intents - should all be NOTHING
get_intentions();

// copy initial intents to arbitrator
ad_arb.set_initial_intents(ad_intents);

// set script position - used in script construction
ad_script_pos = 0;

// generate initial turn in scripts
update_script();

return(true);

}

```

```

bool Administrator::setup_miner2(const bool explore, const short miners,
                                const short miner_r, const short grab,
                                const short sack, const short grid_size,
                                ifstream &m)

{

    if(miners <= 0)

    {

        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_miner2(args), there must be\n";
        err << "at least one miner agent. Setup failed.\n";
        err.close();
        return(false);

    }

    // metamorphose environment to correct size
    ad_env.metamorphose(grid_size);

    // get environment dimension
    ad_dim = grid_size;

    // input the environment map
    ad_env.input_objects(m);

    // check environment state
    if(!ad_env.get_state())

    {

        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_miner2(args), map file stream was\n";
        err << "not freshly opened, or the data file was invalid. Unable to\n";
        err << "setup this scenario.\n";
        err.close();
        return(false);

    }

    PATH temp;
    triple alloc;
    short x,y;
    bool starts_exist = false;

    // check for user defined ASTART positions
    if(ad_env.contains_object(ASTART))

    {

        starts_exist = true;

        for(x = 0 ; x < ad_dim ; x++)

        {

            for(y = 0 ; y < ad_dim ; y++)

            {

```

```

        if(ad_env.get_object_f(x,y) == ASTART)
        {
            alloc.put(x,y,1);
            temp.push_back(alloc);
        }
    }
}

// are there enough start positions?
if(temp.size() < miners)
{
    // add all SPACE too
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object(x,y) == SPACE)
            {
                alloc.put(x,y,0);
                temp.push_back(alloc);
            }
        }
    }
}

// erase ASTART markers
ad_env.clear_objects(ASTART);
}

else
{
    // build a list of available locations to place agents
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {

```

```

        if(ad_env.get_object(x,y) == SPACE)
        {
            alloc.put(x,y,0);
            temp.push_back(alloc);
        }
    }
}

// de-allocate agents from the heap
deallocate_agents();

// erase agents
ad_agents.clear();

// erase intents
ad_intents.clear();

// erase animation scripts
ad_script.clear();
ad_last_script.clear();

// setup arbitrator fail value
ad_arb.set_fail(NOTHING);

// check that enough space for agents
if(temp.size() < miners)
{
    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_miner2(args), there was not\n";
    err << "enough space in the environment to place all of the agents.\n";
    err << "Scenario setup failed.\n";
    err.close();
    return(false);
}

// seed random number generator
srand( (unsigned)time( NULL ) );

// place agents randomly within available space
short i;                // loop counter
short ax;                // generic coordinate variable
short ay;                // generic coordinate variable
PATH_IT loc = temp.begin(); // to manipulate available spaces to place agents
short place;            // to find a place
short max;              // loop limit

// Agent names must run sequentially with vector index from
// zero to (number of agents)-1

short junk;
ChainMiner *mr;         // pointer to a Miner

// create ChainMiners

```



```

for(i = 0 ; i < miners ; i++)
{
    if(starts_exist)
    {
        loc = temp.begin();
        loc -> get(ax,ay,junk);

        // if we have hit a space, stop fixed alloc
        if(junk == 0)
        {
            starts_exist = false;
            max = temp.size();           // get size of available space
            place = rand() % max;        // do random selection
            loc = temp.begin() + place;  // find place
            loc -> get(ax,ay,junk);      // get coordinates
            temp.erase(loc,loc+1);       // erase this place - can't be used again
        }
        else
        {
            temp.erase(loc,loc+1);
        }
    }
    else
    {
        // place agents randomly over available space
        max = temp.size();           // get size of available space
        place = rand() % max;        // do random selection
        loc = temp.begin() + place;  // find place
        loc -> get(ax,ay,junk);      // get coordinates
        temp.erase(loc,loc+1);       // erase this place - can't be used again
    }

    mr = new ChainMiner(ad_dim, 0, 0, &ad_env, 0, miner_r, sack, grab);
    mr -> init(i, ax, ay, MINER);    // set name and start coords
    mr -> record_position();          // make agent record its position

    if(!explore)                    // agents not exploring
    {
        mr -> get_environment();      // load map into memory
    }

    ad_agents.push_back(mr);         // make this agent
}

```

```

// tidy up memory
ad_intents.resize(miners);
ad_agents.resize(miners);

// do initial intents - should all be NOTHING
get_intentions();

// copy initial intents to arbitrator
ad_arb.set_initial_intents(ad_intents);

// record script position
ad_script_pos = 0;

// generate initial turn in scripts
update_script();

return(true);
}

bool Administrator::setup_hunting(const bool explore, const short predators,
                                const short prey, const short predator_r,
                                const short prey_r, const short stom,
                                const short stom_max, const short grid_size,
                                ifstream &m)

{
    if(predators <= 0 || prey <= 0)
    {
        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_hunting(args), there must be\n";
        err << "at least one agent per category. Setup failed.\n";
        err.close();
        return(false);
    }

    // metamorphose environment to correct size
    ad_env.metamorphose(grid_size);

    // get environment dimension
    ad_dim = grid_size;

    // input the environment map
    ad_env.input_objects(m);

    // check state of environment
    if(!ad_env.get_state())
    {
        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_hunting(args), map file stream was\n";
        err << "not freshly opened, or the data file was invalid. Unable to\n";
        err << "setup this scenario.\n";
        err.close();
        return(false);
    }
}

```

```

PATH temp;
triple alloc;
short x,y;
bool starts_exist = false;

// check for user defined ASTART positions

if(ad_env.contains_object(ASTART))
{
    starts_exist = true;

    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object_f(x,y) == ASTART)
            {
                alloc.put(x,y,1);
                temp.push_back(alloc);
            }
        }
    }

    // are there enough start positions?
    if(temp.size() < (predators + prey))
    {
        // add all SPACE too

        for(x = 0 ; x < ad_dim ; x++)
        {
            for(y = 0 ; y < ad_dim ; y++)
            {
                if(ad_env.get_object(x,y) == SPACE)
                {
                    alloc.put(x,y,0);
                    temp.push_back(alloc);
                }
            }
        }
    }
}

```

```

    }

    // erase ASTART markers
    ad_env.clear_objects(ASTART);
}

else
{
    // build a list of available locations to place agents
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object(x,y) == SPACE)
            {
                alloc.put(x,y,0);
                temp.push_back(alloc);
            }
        }
    }
}

// de-allocate agents from the heap
deallocate_agents();

// erase agents
ad_agents.clear();

// erase intents
ad_intents.clear();

// erase animation scripts
ad_script.clear();
ad_last_script.clear();

// setup arbitrator fail value
ad_arb.set_fail(NOTHING);

// check that enough space for agents
if(temp.size() < (predators + prey))
{
    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_hunting(args), there was not\n";
    err << "enough space in the environment to place all of the agents.\n";
    err << "Scenario setup failed.\n";
    err.close();
    return(false);
}

```

```

}

// seed random number generator
srand( (unsigned)time( NULL ) );

// place agents randomly within available space
short i; // loop counter
short ax; // generic coordinate variable
short ay; // generic coordinate variable
PATH_IT loc = temp.begin(); // to manipulate available spaces to place agents
short place; // to find a place
short max; // loop limit
short name; // to maintain correct agent naming

// Agent names must run sequentially with vector index from
// zero to (number of agents)-1

short junk;
Predator *pred; // pointer to a Predator
Prey *pry; // pointer to a Prey

// create Predators
for(i = 0 ; i < predators ; i++)
{
    if(starts_exist)
    {
        loc = temp.begin();
        loc -> get(ax,ay,junk);

        // if we have hit a space, stop fixed alloc
        if(junk == 0)
        {
            starts_exist = false;
            max = temp.size(); // get size of available space
            place = rand() % max; // do random selection
            loc = temp.begin() + place; // find place
            loc -> get(ax,ay,junk); // get coordinates
            temp.erase(loc,loc+1); // erase this place - can't be used again
        }
        else
        {
            temp.erase(loc,loc+1);
        }
    }
    else
    {
        // place agents randomly over available space

```

```

        max = temp.size();           // get size of available space
        place = rand() % max;        // do random selection
        loc = temp.begin() + place;  // find place
        loc -> get(ax,ay,junk);       // get coordinates
        temp.erase(loc,loc+1);       // erase this place - can't be used again
    }

    pred = new Predator(ad_dim, 0, 0, &ad_env, 0, predator_r, stom, stom_max);
    pred -> init(i, ax, ay, PREDATOR); // set name and start coords
    pred -> record_position();          // make agent record its position

    if(!explore)                       // if agents not exploring
    {
        pred -> get_environment();      // load map into memory
    }

    ad_agents.push_back(pred);         // make this agent
}

name = i;        // must keep names contiguous

// create Prey
for(i = name ; i < (predators + prey) ; i++)
{
    if(starts_exist)
    {
        loc = temp.begin();
        loc -> get(ax,ay,junk);

        // if we have hit a space, stop fixed alloc
        if(junk == 0)
        {
            starts_exist = false;
            max = temp.size();           // get size of available space
            place = rand() % max;        // do random selection
            loc = temp.begin() + place;  // find place
            loc -> get(ax,ay,junk);       // get coordinates
            temp.erase(loc,loc+1);       // erase this place - can't be used again
        }
        else
        {
            temp.erase(loc,loc+1);
        }
    }
}

else

```

```

    {
        // place agents randomly over available space
        max = temp.size();           // get size of available space
        place = rand() % max;        // do random selection
        loc = temp.begin() + place;  // find place
        loc -> get(ax,ay,junk);       // get coordinates
        temp.erase(loc,loc+1);       // erase this place - can't be used again
    }

    pry = new Prey(ad_dim, 0, 0, &ad_env, 0, prey_r);
    pry -> init(i, ax, ay, PREY);    // set name and start coords
    pry -> record_position();        // make agent record its position

    if(!explore)                    // if not exploring
    {
        pry -> get_environment();    // load map into memory
    }

    ad_agents.push_back(pry);        // make this agent
}

// tidy up memory
ad_intents.resize((predators + prey));
ad_agents.resize((predators + prey));

// do initial intents - should all be NOTHING
get_intentions();

// copy initial intents to arbitrator
ad_arb.set_initial_intents(ad_intents);

// set script position
ad_script_pos = 0;

// generate initial turn in scripts
update_script();

return(true);
}

bool Administrator::setup_crowd1(const bool explore,const short crowders,const short crowder_r,
                                const short grid_size,ifstream &m)
{
    if(crowders <= 0)
    {
        ofstream err("errlog.txt", ios::app);
        err << "ERROR, Administrator::setup_crowd1(args), there must be\n";
        err << "at least one crowd agent. Setup failed.\n";
        err.close();
        return(false);
    }
}

```

```

}

// metamorphose environment to correct size
ad_env.metamorphose(grid_size);

// get the environment dimension
ad_dim = grid_size;

// input the environment map
ad_env.input_objects(m);

// check environment state
if(!ad_env.get_state())

{
    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_crowd1(args), map file stream was\n";
    err << "not freshly opened, or the data file was invalid. Unable to\n";
    err << "setup this scenario.\n";
    err.close();
    return(false);
}

PATH temp;
triple alloc;
short x,y;
bool starts_exist = false;

// check for user defined ASTART positions
if(ad_env.contains_object(ASTART))
{
    starts_exist = true;

    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object_f(x,y) == ASTART)
            {
                alloc.put(x,y,1);
                temp.push_back(alloc);
            }
        }
    }

    // are there enough start positions?
    if(temp.size() < crowders)

```



```

{
    // add all SPACE too
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object(x,y) == SPACE)
            {
                alloc.put(x,y,0);
                temp.push_back(alloc);
            }
        }
    }

    // erase A* markers
    ad_env.clear_objects(A*);
}

else
{
    // build a list of available locations to place agents
    for(x = 0 ; x < ad_dim ; x++)
    {
        for(y = 0 ; y < ad_dim ; y++)
        {
            if(ad_env.get_object(x,y) == SPACE)
            {
                alloc.put(x,y,0);
                temp.push_back(alloc);
            }
        }
    }
}

// de-allocate agents from the heap
deallocate_agents();

```

```

// erase agents
ad_agents.clear();

// erase intents
ad_intents.clear();

// erase animation scripts
ad_script.clear();
ad_last_script.clear();

// setup arbitrator fail value
ad_arb.set_fail(NOTHING);

// check that enough space for agents
if(temp.size() < crowders)

{
    ofstream err("errlog.txt", ios::app);
    err << "ERROR, Administrator::setup_crowd1(args), there was not\n";
    err << "enough space in the environment to place all of the agents.\n";
    err << "Scenario setup failed.\n";
    err.close();
    return(false);
}

// seed random number generator
srand( (unsigned)time( NULL ) );

// place agents randomly within available space
short i;                // loop counter
short ax;                // generic coordinate variable
short ay;                // generic coordinate variable
PATH_IT loc = temp.begin(); // to manipulate available spaces to place agents
short place;            // to find a place
short max;                // loop limit

// Agent names must run sequentially with vector index from
// zero to (number of agents)-1

short junk;
Crowd *cr;    // pointer to a Crowd agent

// create Crowd agents

for(i = 0 ; i < crowders ; i++)

{
    if(starts_exist)

    {
        loc = temp.begin();
        loc -> get(ax,ay,junk);

        // if we have hit a space, stop fixed alloc
        if(junk == 0)

        {
            starts_exist = false;

```

```

        max = temp.size();           // get size of available space
        place = rand() % max;        // do random selection
        loc = temp.begin() + place;  // find place
        loc -> get(ax,ay,junk);       // get coordinates
        temp.erase(loc,loc+1);       // erase this place - can't be used again
    }

    else

    {

        temp.erase(loc,loc+1);

    }

}

else

{

    // place agents randomly over available space
    max = temp.size();           // get size of available space
    place = rand() % max;        // do random selection
    loc = temp.begin() + place;  // find place
    loc -> get(ax,ay,junk);       // get coordinates
    temp.erase(loc,loc+1);       // erase this place - can't be used again

}

cr = new Crowd(ad_dim,0,0,&ad_env,0,crowder_r);
cr -> init(i,ax,ay,MINER);       // set name and start coords
cr -> record_position();         // make agent record its position

if(!explore)                     // if agents not exploring

{

    cr -> get_environment();// load map into memory

}

ad_agents.push_back(cr);         // make this agent

}

// tidy up expensive memory!
ad_intents.resize(crowders);
ad_agents.resize(crowders);

// do initial intents - should all be NOTHING
get_intentions();

// copy initial intents to arbitrator
ad_arb.set_initial_intents(ad_intents);

// set script position - used in script construction
ad_script_pos = 0;

// generate initial turn in scripts
update_script();

```

```

        return(true);
    }

void Administrator::update_script()
{
    // first update non-agent objects

    // erase copy of last turn
    ad_last_script.clear();

    chunk data(0,0,0,0,0,0,0,0,0);
    short count = 0;

    // set an info field
    ad_script.push_back(data);
    ad_last_script.push_back(data);

    // get non-agent object states
    for(ads_x = 0 ; ads_x < ad_dim ; ads_x++)
    {
        for(ads_y = 0 ; ads_y < ad_dim ; ads_y++)
        {
            ad_env.get_object_f(ads_x,ads_y,ads_type,ads_quantity);

            // only want objects that are not obstacles,space or agents
            if(ads_type >= MINERAL)
            {
                // ads_type, ads_name, ads_x, ads_y, ads_action,
                // ads_quantity, ads_target_x, ads_target_y

                data.put(0,ads_type,0,ads_x,ads_y,0,ads_quantity,0,0);
                // alter coords for last script
                data.screen_coords();
                ad_script.push_back(data);
                ad_last_script.push_back(data);
                count++;
            }
        }
    }

    // now get agents
    AGENTS_IT iter;

    // ads_type, ads_name, ads_x, ads_y, ads_action,
    // ads_quantity, ads_target_x, ads_target_y

    for(iter = ad_agents.begin() ; iter != ad_agents.end() ; ++iter)
    {
        (*iter) -> get_state(ads_type, ads_name, ads_x, ads_y, ads_action,
                            ads_quantity,ads_target_x, ads_target_y);
    }
}

```

```

        if(ads_type != SPACE) // agents died and been eaten up or rotted away
        {
            data.put(0,ads_type,ads_name,ads_x,ads_y,ads_action,ads_quantity,
                    ads_target_x,ads_target_y);
            //adjust coords for anim
            data.screen_coords();
            ad_script.push_back(data);
            ad_last_script.push_back(data);
            count++;
        }
    }

    // now record total objects in info fields
    data.put(1,count,0,0,0,0,0,0);
    ad_script[ad_script_pos] = data;
    ad_last_script[0] = data;
    ad_script_pos += ((long)count+1);
}

void Administrator::deliberate()
{
    // allow agents to ponder
    AGENTS_IT i;

    for(i = ad_agents.begin() ; i != ad_agents.end() ; ++i)
    {
        (*i) -> think();
    }
}

void Administrator::get_intentions()
{
    AGENTS_IT a_i;

    ad_intents.clear();

    for(a_i = ad_agents.begin() ; a_i != ad_agents.end() ; ++a_i)
    {
        ad_intents.push_back((*a_i) -> get_intent());
    }
}

void Administrator::set_intentions()
{

```

```

    INTENT_LIST_IT i_i;

    for(i_i = ad_intents.begin() ; i_i != ad_intents.end() ; ++i_i)
    {
        ad_agents[i_i -> get_name()] -> put_intent(*i_i);
    }
}

void Administrator::execute_actions()
{
    AGENTS_IT a;
    for(a = ad_agents.begin() ; a != ad_agents.end() ; ++a)
    {
        (*a) -> execute_action();
    }
}

void Administrator::arbitrate()
{
    ad_arb.arbitrate(ad_intents);
}

void Administrator::get_script(SCRIPT &s)
{
    s = ad_script;
}

void Administrator::get_last_turn(SCRIPT &s)
{
    s = ad_last_script;
}

void Administrator::check_hunting_POUNCE()
{
    // walk the intents and look for action POUNCE

    INTENT_LIST_IT i_i;
    triple t;
    PATH p;
    short x,y,z,c,x1,y1;

```

```

for(i_i = ad_intents.begin() ; i_i != ad_intents.end() ; ++i_i)
{
    if(i_i -> get_action() == POUNCE)
    {
        i_i -> get(x,y);
        t.put(x,y,0);
        p.push_back(t); // build a list of locations where a POUNCE took place
    }
}

if(!p.size())
{
    //nothing further to do
    return;
}

//look for all agents whose coords match those in p and kill them

AGENTS_IT a;
short agent_name = -1;

for(a = ad_agents.begin() ; a != ad_agents.end() ; ++a)
{
    agent_name++;

    (*a) -> get_location(x1,y1);

    for(c = 0 ; c < p.size() ; c++)
    {
        p[c].get(x,y,z);

        if(x==x1 && y==y1)
        {
            // do chance to see if pounce successful
            if((rand() % 100) > 59)
            {
                (*a) -> kill();
                // adjust intention
                ad_intents[agent_name].set_action(NOTHING);
            }
        }
    }
}

```

```

        }
    }
    void Administrator::deallocate_agents()
    {
        if(ad_agents.size())
        {
            AGENTS_IT i;
            for(i = ad_agents.begin() ; i != ad_agents.end() ; ++i)
            {
                delete(*i);
            }
        }
    }
}

```

This class is the “conductor” that orchestrates the simulation process. We have used a form of two-stage construction process for generating experiments. The first stage simply generates an Administrator instance and the choice of second stages allows a specific experiment context to be set. Although we have repeated a lot of code in the experiment set-up functions, we felt that this was acceptable due to the nature of use. If we had coded more functionality it would have become difficult to make changes easily. After all, this is a system for experiments and not a polished pseudo-commercial product.

C.12 The Special Data Structures

It was necessary to create some simple data structures in order to process the enormous data requirements of animation scripts. The following header file contains an

implementation for scripts, called “chunk” and also other data structures that were used frequently throughout the system.

```
//      Filename:          DataStructures.h
//      Version:           Final
//      Author:            Paul Tinsley (pmbtin)
//      Date:              26 February, 1999
//      Platform:          Visual C++ 5.00

//      Purpose:           Data structures used in various areas of the system.

#ifndef __DATASTRUCTURES_H
#define __DATASTRUCTURES_H

#include <vector>                                // container type

using namespace std;

// The following simple class is used in calculating minimal paths amongst
// others. It's very handy to have a triple.

class triple
{
public:
    triple() : x(0), y(0), z(0){};
    triple(const short a, const short b, const short c) : x(a), y(b), z(c) {};
    triple & operator =(const triple &a)
    {x = a.x; y = a.y; z = a.z; return *this;};
    const bool friend operator ==(const triple &a, const triple &b)
    {return(a.x == b.x && a.y == b.y && a.z == b.z);};
    const bool friend operator <(const triple &a, const triple &b)
    {return(a.z < b.z);};
    const void get(short &a, short &b, short &c)
    {a = x; b = y; c = z;};
    void put(const short a, const short b, const short c)
    {x = a; y = b; z = c;};
private:
    short x;
    short y;
    short z;
};

typedef vector<triple> PATH;    // a collection of triples, useful in path
typedef PATH::iterator PATH_IT;    // building

// this simple class is used in scripts

class chunk
{
public:
    chunk() : c1(0), c2(0), c3(0), c4(0), c5(0), c6(0), c7(0), c8(0), c9(0){};
    chunk(const short a, const short b, const short c, const short d,
          const short e, const short f, const short g, const short h,
          const short i)
        : c1(a), c2(b), c3(c), c4(d), c5(e), c6(f), c7(g), c8(h), c9(i){};
    chunk & operator =(const chunk &a)
    {c1 = a.c1; c2 = a.c2; c3 = a.c3; c4 = a.c4;
     c5 = a.c5; c6 = a.c6; c7 = a.c7; c8 = a.c8; c9 = a.c9; return *this;};
    const bool friend operator ==(const chunk &a, const chunk &b)
```

```

{return(a.c1 == b.c1 && a.c2 == b.c2 && a.c3 == b.c3 && a.c4 == b.c4
        && a.c5 == b.c5 && a.c6 == b.c6 && a.c7 == b.c7 && a.c8 == b.c8
        && a.c9 == b.c9);};
const bool friend operator <(const chunk &a, const chunk &b)
{return(a.c1 < b.c1);};

//      *** GETTERS, PUTTERS & TRANSFORMERS ***

// get all data
const void get(short &a, short &b, short &c, short &d, short &e, short &f,
               short &g, short &h, short &i)
{a = c1; b = c2; c = c3; d = c4; e = c5; f = c6; g = c7; h = c8; i = c9;};

// put all data
void put(const short a, const short b, const short c, const short d,
         const short e, const short f, const short g, const short h,
         const short i)
{c1 = a; c2 = b; c3 = c; c4 = d; c5 = e; c6 = f; c7 = g; c8 = h; c9 = i;};

// get data for trails
const void get_trails(short &a, short &b, short &c, short &d, short &e,
                     short &h, short &i)
{a = c1; b = c2; c = c3; d = c4; e = c5; h = c8; i = c9;};

// get data for tagging
const void get_tagging(short &b, short &c, short &d, short &e, short &f,
                       short &h, short &i)
{b = c2; c = c3; d = c4; e = c5; f = c6; h = c8; i = c9;};

// get data for scenario animation
const void get_anim(short &a, short &b, short &c, short &d)
{a = c2; b = c4; c = c5; d = c7;};

// get data for paths
const void get_paths(short &a, short &b, short &d, short &e)
{a = c1; b = c2; d = c4; e = c5;};

// get data for prey killed
const void get_agent(short &a, short &b, short &c)
{a = c1; b = c2; c = c3;};

// get data for predator stomachs
const void get_quantity(short &a, short &b, short &g)
{a = c1; b = c2; g = c7;};

// get data for agent locations
const void get_loc(short &a, short &b, short &c, short &d, short &e)
{a = c1; b = c2; c = c3; d = c4; e = c5;};

// get frame count for this info field
const void get_frames(short &b)
{b = c2;};

// get info field
const void get_info(short &a)
{a = c1;};

// adjust coords for screen anim
void screen_coords()
{c4*=10;c5*=10;c8*=10;c9*=10;};

```

private:

```

short c1;           // info field flag for dynamic frame size data

```

```

    short c2;           // object type or frame count if an info field
    short c3;           // agent id
    short c4;           // agent x-coord
    short c5;           // agent y-coord
    short c6;           // agent action
    short c7;           // object quantity possessed
    short c8;           // agent target x-coord
    short c9;           // agent target y-coord

};

typedef vector<chunk> SCRIPT;           // script for animation
typedef SCRIPT :: iterator SCRIPT_IT;

// to hold scenario info

struct SCENARIO_DATA
{
    int total_agents;           // total agents in scenario
    short mining_agents;       // mining agents
    short predator_agents;     // predator agents
    short prey_agents;         // prey agents
    short miner_vis;           // miner vision range
    short predator_vis;        // predator vision range
    short prey_vis;            // prey vision range
    short miner_grab;           // miner grab capacity
    short miner_sack;           // miner sack capacity
    short pred_stom_m;          // predator max stomach capacity
    short pred_stom_i;          // predator init stomach capacity
    short turns;                // turns in this scenario
    int scenario_type;          // type of scenario run
    bool exploring;             // whether agents had to explore map

    // have not included amount of scenarios run.

};

typedef vector<long> FRAMES;           // script for animation
typedef FRAMES :: iterator FRAMES_IT;

#endif

```

The implementation of these structures is rather simple, so detailed comments were not deemed appropriate. “SCENARIO_DATA” was used within the interface code to store information about the experiments as persistent disk files. All data files, apart from map files, were stored in binary form.

Some uses of the scenario script type, “chunk,” may be found in Appendix C.11. We have not provided the full interface code as our project focus is on the experimentation aspects.



Acknowledgements

The persons mentioned below played a special role in enabling this project to be completed. There are many other persons from whom we have benefitted through the degree courses. None of this would have been possible were it not for their excellent teaching of subject matter and dedication to their respective fields.

Bill Hart For providing a friendly and positive project supervision support that acted as a firm foundation from which our project progress could be managed.

Mark Roper For some code inspections, discussions about emergence and suggesting a lookup table within the Grid Class.

Prof. Jim Doran Although I have not sought the assistance of Prof. Doran in technical

matters relating to this study, his exceptional delivery of Artificial Intelligence topics during degree courses has been a considerable inspiration to me for which I am extremely grateful.

Sarah Brown For assisting with proof reading and the use of her colour printer

Bibliography

Aho, A. V. & Ullman, J. D. Foundations of Computer Science, C Edition. USA: W. H. Freeman and Company, 1995.

Breymann, U. Designing Components with the C++ STL, A New Approach to Programming. England: Addison-Wesley, 1998

Deitel, H. M. and Deitel, P. J. C++ How to Program. 2nd ed. New Jersey: Prentice-Hall, Inc., 1998.

Drogoul, A. "When Ants Play Chess - (Or Can Strategies Emerge from Tactical Behaviours?)." Lecture Notes in Artificial Intelligence 957 (1995). 13-27

Humphreys, P. "Emergence, not Supervenience." Philosophy of Science 64 (4SS) (1997). S337-S345

Kruglinski, D. J. Inside Visual C++. 4th ed. USA: Microsoft Press, 1997.

Mataric, M. J. "Designing and Understanding Adaptive Group Behaviour." Adaptive Behaviour 4 (1) (Dec. 1995). 51-80.

Pfleeger, S. L. Software Engineering: Theory and Practice. New Jersey: Prentice-Hall, Inc., 1998.

Prosise, J. Programming Windows 95 with MFC. USA: Microsoft Press, 1996.

Russel, S. and Norvig, P. Artificial Intelligence: A Modern Approach. New Jersey: Prentice-Hall, Inc., 1995.

Stroustrup, B. The C++ Programming Language. 3rd ed. USA: Addison-Wesley, 1997.