## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

# AGENT SIMULATION USING OBJECT-ORIENTED METHODOLOGY

CHANGJIANG ZHANG

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

JUNE 2000

© CHANGJIANG ZHANG, 2000

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*

*Our file Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-54339-0

Canadä

# Abstract

Agent Simulation using Object-oriented Methodology

Changjiang Zhang

In this report, we present an ecological Agent Simulation System, which simulates the interactions between ecological agents and shows the statistics of simulation results. We discuss how we design and implement it using Object-oriented analysis, design, and programming. The design phase illustrates how we apply the concepts of Object-oriented methodology to develop this software application – the Agent Simulation System. The implementation phase demonstrates the programming process by using an Object-oriented language (C++) and Software Development Kit (Visual C++). We show the simulation results with two simulation examples and analyze the statistics. We conclude by listing some advantages of Object-oriented we experienced from this project and suggest further work.

# Acknowledgments

I wish to express my sincere gratitude to my supervisor, Dr. Peter Grogono, for all his enthusiastic support, careful supervision, and consistent guidance during the development of this major report. I also wish to thank Dr. Ching Y. Suen, who kindly took the time to review the report.

Furthermore, I would like to appreciate my partner Weidong Sun, for his ideas, hard work, and collaboration.

Finally, I wish to thank my wife, Ni Li, for all her encouragement and support.

# Table of Contents

List of Figures .......... viii
List of Tables .......... ix
1 Introduction .......... 1
1.1 Aim of the Project .......... 1
1.2 Motivation .......... 1
1.3 Structure of this report .......... 2
2 Background .......... 3
2.1 Problem description .......... 3
2.1.1 Resources .......... 3
2.1.2 Strings .......... 3
2.1.3 Agents and Genes .......... 4
2.1.4 Cell, Environment, Food, and Tax .......... 4
2.1.5 Encounters between Agents .......... 5
2.1.5.1 Combat .......... 5
2.1.5.2 Trade .......... 6
2.1.5.3 Reproduction .......... 6
2.1.6 Simulation Parameters .......... 7
2.1.7 Running the simulation .......... 8
2.1.7.1 Food Distribution .......... 9
2.1.7.2 Feeding .......... 9
2.1.7.3 Encounters .......... 9
2.1.7.4 Reaping .......... 10
2.2 Related work .......... 10
3 Design .......... 12
3.1 Identifying the classes .......... 12
3.2 Agent classes .......... 12
3.2.1 Class CSimString .......... 13
3.2.1.1 Operations: .......... 13
3.2.1.2 Attributes .......... 14
3.2.2 Class CAgent .......... 14
3.2.2.1 Operations: .......... 15
3.2.2.2 Attributes: .......... 17
3.2.3 Class CAgentList .......... 17
3.2.3.1 Operations: .......... 17
3.2.3.2 Attributes: .......... 18
3.2.4 Class CCell .......... 18
3.2.4.1 Operations: .......... 18

3.2.4.2 Attributes: 20
3.2.5 Class CEnvironment 21
3.2.5.1 Operations: 21
3.2.5.2 Attributes: 22
3.2.6 Class CSimParam 23
3.2.7 Class CSummaryTable 23
3.2.7.1 Operations: 23
3.2.7.2 Attributes: 25
3.2.8 Class CSimApp 25
3.2.8.1 Operations: 26
3.2.8.2 Attributes: 26
3.3 Graphical user interface and related classes 26
3.3.1 Dialog box and controls 27
3.3.1.1 "About" dialog box 28
3.3.1.2 Simulation parameter dialog box 28
3.3.1.3 Summary cycle dialog box 28
3.3.1.4 Summary table dialog box 29
3.3.2 Menus 29
3.3.3 Help system 29
4 Implementation 31
4.1 Introduction to Visual C++, the implementation tool 31
4.2 Programming progress 32
4.3 AppWizard created classes 32
4.3.1 Documents and Views 32
4.3.1.1 Class CSimulationDoc 32
4.3.1.2 CSimulationView 34
4.4 Agent classes 35
4.4.1 Class CSimString 36
4.4.2 Class CAgent 37
4.4.3 Class CAgentList 40
4.4.4 Class CCell 41
4.4.5 Class CEnvironment 43
4.4.6 Class CSimParam 44
4.4.7 Class CSummaryTable 45
4.5 Graphical user interface: 46
4.5.1 Toolbar: 46
4.5.2 Status bar: 48
4.5.3 Dialog box and controls 49
4.5.3.1 About Simulation Dialog: 49
4.5.3.2 Simulation Parameter Dialog: 50
4.5.3.3 Summary Cycle Dialog: 51
4.5.3.4 Summary Table Dialog: 52
4.6 The help system 54
4.6.1 Presenting help 54

4.6.2 Components of the Help System ........ 55
4.6.3 Programming the help ........ 56
4.6.3.1 The help button ........ 56
4.6.3.2 The Context help ........ 57
5 Simulation result ........ 59
5.1 Example 1 ........ 59
5.2 Example 2 ........ 65
6 Conclusion ........ 70
6.1 Experience on Object-oriented Programming ........ 70
6.2 Further work ........ 71
A. Bibliography ........ 72
B. Class diagram 1 ........ 73
C. Class diagram 2 ........ 74

# List of Figures

FIGURE 4-1:Toolbar ....... 48
FIGURE 4-2:The About Dialog box ....... 49
FIGURE 4-3:Simulation parameter dialog box ....... 51
FIGURE 4-4:The Summary cycle dialog box ....... 52
FIGURE 4-5:The summary Table dialog box ....... 52
FIGURE 4-6:Help Topics ....... 54
FIGURE 4-7:Help button triggered Help window ....... 55
FIGURE 5-1:Before Simulation starts ....... 60
FIGURE 5-2:Simulation result after 50 cycles with example 1 ....... 61
FIGURE 5-3:Simulation result after 100 cycles with example 1 ....... 62
FIGURE 5-4:Simulation result after 150 cycles with example 1 ....... 63
FIGURE 5-5:Simulation result after 200 cycles with example 1 ....... 64
FIGURE 5-6:Simulation results after 50 cycles with example 2 ....... 65
FIGURE 5-7:Simulation results after 100 and 150 cycles with example 2 ....... 66
FIGURE 5-8:Simulation results after 200 and 300 cycles with example 2 ....... 67
FIGURE 5-9:Simulation results after 1000 cycles with example 2 ....... 68

# List of Tables

TABLE 2-1: Threatening Behavior........................................................................... 6

# 1 Introduction

In this chapter, we describe the project that we selected and why we chose this topic. We also introduce the background of the project and describe the structure of this report.

## 1.1 Aim of the Project

This project is about ecological Simulation. Our particular example is the Agent Simulation System. The Aim of this project is to present an Object-Oriented approach to simulating interactions among ecological agents and between agents and their environments using C++ language and the Microsoft Visual C++ toolkit. Our intention is to design and implement an Ecological Agent Simulation as realistic as possible. A graphical user interface is provided so that user can use it as a tool to input simulation parameters and learn the simulation results.

## 1.2 Motivation

Simulations are very interesting topics, which always attract people doing research or programming in computer literature because:

- They provide a good model to establish data structures and algorithms;
- They combine knowledge of specific field, artificial intelligence, and simulation strategies;

- They require rich and complex algorithms.

## 1.3 Structure of this report

Chapter 1 is an introduction. It briefly describes the aim and motivation of the project. Chapter 2 describes the background and all concepts of the project. Chapter 3 presents the design phase, including the simulation classes, the graphical user interface design, and the Help System. Chapter 4 starts with an introduction to our implementation tool and gives all the class, interface, and Help details. Chapter 5 presents the simulation results by two examples. Chapter 6concludes this project and suggests further work for this project.

# 2 Background

## 2.1 Problem description

This chapter describes the background knowledge necessary for designing and implementing the Agent Simulation System.

### 2.1.1 Resources

There are VARIETY (which is a simulation parameter, as described in *Section 2.1.6* on page 7) kinds of resource. Each resource is represented by a lower-case letter chosen from the alphabet starting from “a”. For example, if VARIETY = 3, the resource are named a, b, and c.

### 2.1.2 Strings

A string is a sequence of resources, possibly empty. For example, “abcd”, “aabbccde” are strings. The important property of a string is the number of occurrences of each resource. The terms with string:

- The *match* of two strings is a positive integer that depends on the number of resources that the strings have in common.
- The *difference* of two strings stringA and stringB is the excess of resources in stringA over those in stringB.

- A string stringA *exceeds* stringB if, for each letter *c*, the number of occurrences of *c* in stringA exceeds the number of occurrences of *c* in stringB.

### 2.1.3 Agents and Genes

An *agent* is a collection of ecological primitive components. An agent consists of nine strings: *attack, defend, beauty, combat, trade, lust, eat, give*, and *reserve*. The first eight strings are *genes*; together, they constitute the *genome* of the agent. An agent is born with a genome and the genome does not change during the lifetime of the agent.

The last string, *reserve*, represents the collection of different resources that the agent currently owns. It may change during the lifetime of the agent.

### 2.1.4 Cell, Environment, Food, and Tax

The whole ecological world is called the *environment* by the simulation. An agent lives in a *cell* that is a part of the environment. We can imagine the environment as a two-dimension grid of cells. There is a string associated with the cell called *food*. Food is distributed throughout the environment. The food an agent eats is stored in its "reserve". Each agent begins its life in a cell and has opportunity to "eat" during each simulation cycle. The "eat" gene determines how much food an agent can eat. It moves away from that cell if there is no food available for it in the cell.

An agent is also "taxed" by its cell. Taxing corresponds to metabolic processing in an animal: some resources must be returned to the environment. The tax that an agent pays depends on its size. TAX is a simulation parameter described later. Taxing can prevent

agents from becoming too large as they evolve.

### 2.1.5 Encounters between Agents

During the simulation, agents *encounter* one another. An encounter involves exactly two agents. There are four possible outcomes of an encounter: *combat*, *trade*, *reproduction*, or nothing. The agents try each of the possibilities in turn. For example, if "combat" has no effect, they attempt to trade; if "trade" has no effect, they try to "reproduce". A problem with this strategy is that there might be too much fighting and not much reproduction. One solution is to randomly choose the sequence of encounter. The sequence of combat, trading, and reproduction has $3 * 2 = 6$ possibilities:

- combat → trade → reproduce
- combat → reproduce → trade
- trade → combat → reproduce
- trade → reproduce → combat
- reproduce → trade → combat
- reproduce → combat → trade

Each time two agents encounter, one of the above sequences is randomly chosen. This will make the chances of trading, fighting, and reproduction even.

#### 2.1.5.1 Combat

Under certain conditions, an agent A will *threaten* another agent B. If agent A's "attack" gene exceeds (as described in *Section 2.1.2 Strings* on page 3) agent B's "defend" gene, we say "A threatens B". When two agents A and B encounter one another, there are four pos-

sibilities, as shown in Table 2-1, "Threatening Behavior," on page 6.

If a fight takes place, the winner adds all of the loser's resources (genome and reserve) to its "reserve". If one agent surrenders, or nothing happens, the agents proceed to the next possible kind of encounter.

***TABLE 2-1: Threatening Behavior***

| A threatens B | B threatens A | |
|---|---|---|
| | True | false |
| True | Combat | B surrenders or combat |
| False | A surrenders or combat | nothing happens |

#### 2.1.5.2 Trade

Trading between agents can take place if agent A's "trade" gene exceeds agent B's "attack" gene and agent B's "trade" gene exceeds agent A's "attack" gene. If trading takes place, resources are transferred between the agents' "reserves". The agents' "give" genes limit the size of the transfer.

#### 2.1.5.3 Reproduction

Like trading, reproduction takes place only if both agents are willing. The condition is that the match of agent A's "lust" gene and agent B's "beauty" gene exceeds MINIMATCH (a simulation parameter as described in *Section 2.1.6 Simulation Parameters* on page 7) and the match of agent B's "lust" gene and agent A's "beauty" gene exceeds MINIMATCH. If these conditions are satisfied, we call the two agents "parents". Reproduction consists of the following three steps, which are described in detail below:

- Design a new genome from the parents' genes.
- Ensure that the parents have sufficient resources to construct the genome.
- If so, construct a new agent using the new genome and resources from the reserves of the parents.

### 2.1.6 Simulation Parameters

A simulation parameter is a value that remains constant for a particular run of the simulation, but may be changed between simulations. We tested the simulation during and after the implementation phase using several sets of simulation parameters as input and analyzed the results. The values given below are intended to be typical to give interesting simulation results.

- SIDE - A default value of SIDE is 10, giving $10\times10 = 100$ cells in the environment.
- POP - The number of agents in a cell. There should be, on average, more than one agent in a cell. 500 is the default value.
- VARIETY – The number of different resources type. The default value is 3.
- TAX - Some of these resources would be returned to the environment as taxes. This might reduce the amount of food required (or, alternatively, lead to a population somewhat

larger than predicted). TAX is a fraction. The default value is 0.1.

- MINMATCH – the minimum condition which allow the engaged agents to reproduce child. Reproduction will happen only if the match (as described in *Section 2.1.2 Strings* on page 3) of two agents' "lust" and "beauty" genes is greater than MINIMATCH. The default value is 2.
- FRAC – the fraction of the remaining resources from parent agents that will be transferred to their child. The default value is 0.33. That means 1/3 of the parents' resources will be transferred to the child during reproduction.
- ACTIVE – An agent has been inactive for more than ACTIVE cycles will be removed. The default value of ACTIVE is 10.

### 2.1.7 Running the simulation

The simulation is started with a population of several hundred agents. The initial agents have random genomes and "reserves". The average size of the initial "reserve" should be roughly the same as the genome. For example, if the average length of a gene is 3 resource units, then the length of reserve is 24 resource units.

The simulation runs in cycles (also called steps). During each cycle, the following events occur:

- Food is distributed throughout the environment.

- Each agent is given a chance to eat some food.
- Agents are selected in pairs for encounters.
- Agents that appear to be inactive are removed.

#### 2.1.7.1 Food Distribution

A fixed amount of food is distributed randomly throughout the environment at each simulation cycle.This allows the total resources in the environment to increase and newly produced agents to grow.

#### 2.1.7.2 Feeding

Each agent is allowed to transfer food from its cell to its "reserve" in accordance with the rules. If an agent fails to obtain any food during a cycle, it is moved to an adjacent cell.

#### 2.1.7.3 Encounters

The simulation selects agents for encounters using the following procedure:

- Choose at random an agent that has not had an encounter during this cycle.
- Choose at random another agent in the same cell that has not had an encounter during this cycle.

This allows the agents to interact in accordance with the rules defined in *Section 2.1.5 Encounters between Agents* on page 5.

If something happens during the encounter (fighting, trading, or reproduction), the simulation records the fact that these agents have had an encounter (and therefore cannot have

any further encounters) during this cycle. If, on the other hand, nothing happens during the encounter, the agents are allowed to participate in another encounter during the same cycle.

#### 2.1.7.4 Reaping

The simulation removes agents that are inactive, returning their resources to the cell they are currently living in. The definition of "inactive" is that the agent has not had a positive outcome from an encounter during the last ACTIVE (simulation parameter) cycles. A "positive outcome" is one of: a fight that it wins; a trade in which it receives some resources units; or the birth of a child.

What will happen between the encountered parties depends on the genes inside each agent according to some kind of rules we set in our design. An agent dies when it fails in fighting or it is removed because of being inactive for a relatively long period of time.

Resources in dead agents are returned to the environment.

Our focus is on the interactions among agents and between agents and analyzing the data after a certain number of simulation cycles.

## 2.2 Related work

Peter T. Hraber and Stephanie Forrest from University of New Mexico, Terry Jones from Santa Fe Institute designed a generic ecosystem model called Echo appendix 2 on page 72, in which evolving agents were situated in a resource-limited environment.

Our supervisor, Dr. Peter Grogono, developed a Cell Simulation System, which provided

a proof of many biochemical concepts (appendix 1 on page 72). Although it was designed and implemented for biochemical purpose, it was a good model for our project. It showed the simulation results on screen using graphical curves.

# 3 Design

## 3.1 Identifying the classes

The key for designing the Agent Simulation System using Object-oriented method is to identify the objects and classes of objects in the system. As described in chapter 2, we find that nouns such as "agent", "cell", "environment", and "string" that would represent the objects in the Agent Simulation. We designed a standard Windows style graphical user interface, which is represented by the objects such as toolbar, menu, buttons, and dialog boxes. Therefore, two categories of classes are defined in the system. One category of the class is associated with non-user interface, for example string, agent, cell, environment, and simulation parameters. Another category of the classes is user interface associated, for example dialog boxes.

In this chapter, we explain the design of some classes by listing their most important operations and attributes. The operations are shown with their return type, parameter types, and a brief description. Parameter names and more details are described in Chapter 4 Implementation.

## 3.2 Agent classes

We add "C" at the beginning of each class name to make them consistent with the names of Microsoft Visual C++ created classes.

### 3.2.1 Class CSimString

As described in chapter 2, CSimString is a possibly empty sequence of resources (alphabet letters). Genes are objects of CSimString class. It is named CSimString instead of CString because CString exits in MFC.

#### 3.2.1.1 Operations:

- int ResourceSize(int): Get the number of one give type of resource in the CSimString.
- int size( ): Get the size of the CSimString. The size of the CSimString is the sum of all its resources. For example, "abccdde" has size 7.
- CString operator+(CSimString&): Concatenate two CSim-Strings.
- int operator-(CSimString&): Get the difference of two CSimStrings. The difference of two CSimStrings is the excess of resources in one CSimString over those in another.
- int operator*(CSimString&): Get the match of two CSim-Strings. The match of two CSimStrings is a positive integer that depends on the number of resources that the two CSim-Strings have in common.
- bool operator>=(CSimString &) Check if one CSimString exceeds another. If the number of each resource in first

CSimString is greater than that in the second CSimString, we say the first CSimString exceeds the second CSimString.

- void convert2cstr(char*, int): Convert a CSimString object to a C language-style string.
- bool transfer(CSimString&, CSimString&, CSimString&): Transfer resources from one CSimString to another.
- void clear( ): Clear all resources: that is, set the number of all resources to zero.
- void Mutation( ): randomly choose a resource in a CSimString, then randomly change (increase or decrease) the number of that resource.

#### 3.2.1.2 Attributes

- An array of integer storing the number of each resource in this CSimString. For example the first element of the array is the number of character "a" in the CSimString, second element is the number of "b" in the CSimString.

### 3.2.2 Class CAgent

This is the core class of our simulation. Each agent lives in a cell and usually is put on a linked list (beforelist or afterlist).

#### 3.2.2.1 Operations:

- CCell* GetCell( ): Get the cell the agent lives in.
- void SetNext(CAgent*): Set an agent's "next" attribute. The "next" attribute is usually a pointer pointing to another agent next to the current agent on the linked list.
- CAgent* GetNext( ): Get the next agent on the linked list.
- void SetActive(bool): Set its active status to "true" or "false".
- bool Initialize( ): randomly construct all genes and reserve of the agent.
- int ResourceSize(int): Get the number of specific type of resource, for example how many "a"s in the agent.
- bool Mutation( ): mutate all genes of the agent at string level.
- void Kill(CAgent*): Kill another agent and get all resources from the agent being killed.
- void Eating( ): transfer food (resources) from cell to the agent. If an agent cannot eat anything in the current cell when it has the chance to eat, it moves to a neighboring cell at a random direction.
- int Size( ): Get the total number of all resources the agent owns (eight genes and reserve).

- void Paytax( ): according to the agent's size and the simulation parameter TAX, return random type and amount of resources from reserve to the cell the agent lives in.
- void Encounter(CAgent*): Encounter with another agent, including combat, trade, and reproduce. Since the sequence of combat, trade, and reproduce will affect the simulation result, we need to make the sequence as random as possible.
- bool IsFightable(CAgent*): Decide if the encountered two agents will fight.
- void Combat(CAgent*): Fight with another agent. One agent may kill another agent, or nothing happens.
- bool IsWinner(int, int): Decide which agent will win the fight.
- bool IsTradable(CAgent*): Decide if the encountered two agents will trade.
- void Trade (CAgent*): Trade with another agent. Usually two agents transfer resources to/from each other's reserve.
- bool IsMatable(CAgent*): Decide if the encountered two agents will reproduce.
- void AddToList(CAgentList*): Add an agent to an agentlist.
- bool IsActive( ): Get the agent active status: it is active or not.

- void Reproduce(CAgent*): Create a new agent and construct its genome and reserve.
- void Die( ): return the agent's all resources to its cell
- void Move(DIRECTION): move the agent to another cell.

#### 3.2.2.2 Attributes:

- Genome (eight genes) and reserve, which are of type CSimString.
- The cell the agent lives in.
- A variable indicates the status of the agents (active or not).
- Its next agent on the agentlist.

### 3.2.3 Class CAgentList

An *agentlist* is a linked list. An agent on the list has a pointer pointing to another agent on the list. The last agent on the list points to *null*.

#### 3.2.3.1 Operations:

- void SetHeader(CAgent*): Set an agent as the header of the agentlist.
- void Initialize( ): Initialize all agents on the agentlist at agent level (each agent on the agentlist initializes).
- void AddAgent(CAgent*): Add an agent to the header of the agentlist.

- void RandomAddAgent(CAgent*): Add an agent to a random position of the agentlist.
- void RemoveAgent(CAgent*): Remove an agent from the agentlist.
- CAgent* RemoveFirstAgent( ): Remove the header of the an agentlist.
- int GetCount: Get the number of agents on the agentlist.
- int SetCount(int): Set the number of agents on the agentlist.

#### 3.2.3.2 Attributes:

- A pointer to the header agent of the agentlist.
- The number of agents on the agentlist.

### 3.2.4 Class CCell

A cell has two agentlists in it: beforelist and afterlist. Initially all agents are on the beforelist at the beginning of each simulation cycle. The beforelist contains all agents that have not been chosen to encounter with another agent in the current simulation cycle. Afterlist is the agentlist that contains all agents that have finished encountering with another agent and as well as new agents created by reproduction in the current simulation cycle.

#### 3.2.4.1 Operations:

- void SetEnvironment(CEnvironment*): Set the environment the cell belongs to.

- CEnvironment* GetEnvironment( ): Get the environment the cell belongs to.
- void Mutation( ): get the afterlist of the cell and mutate all agents on the list at agent level (let all agents "mutate").
- CSummaryTable* GetSummaryTable( ): Get the summary table of the cell (create a summary table contains all statistics based on the cell).
- CAgentList* GetBeforeList( ): Get the afterlist.
- CAgentList* GetAfterList( ): Get the beforelist.
- void Initialize( ): get the beforelist and initialize all agents on the list at agentlist level (let all agents on the list do "initialize"—as described in class CAgent).
- void SetFood(CSimString&): Distribute food to the cell.
- CSimString& GetFood( ): Get the current food in the cell.
- void Feeding( ): get the beforelist and let all agents eat food from the cell. Then move the agents from beforelist to afterlist.
- void Taxing( ): get beforelist in the cell and let all agents on the list pay tax as described in class CAgent.
- void Reaping( ): get afterlist in the cell, remove all inactive agents on the list and return their resources to the cell.

- bool Report(ostream&): Report information at cell level (for example, how many agents left in the cell) and output simulation summary results to a file.
- CCell* GetNeighbor(DIRECTION): Given a direction, get the neighboring cell in the environment.
- void Encounter( ): get the beforelist in the cell, remove first two agents from the head of beforelist and let them encounter with each other at agent level (do "encounter" as described in class CAgent). After encountering, add them to afterlist. If only one agent left (there is no more agent left to encounter), add it to afterlist.
- void PrepareNextCycle( ): get the afterlist, remove its header agent and randomly add it to beforelist for the next simulation cycle.

#### 3.2.4.2 Attributes:

- A pointer to the environment the cell belongs to.
- The summary table of the cell. Summary is on cell basis. Computation is based on each agent.
- A pointer to the beforelist in the cell.
- A pointer to the afterlist in the cell.
- The food in the cell.

### 3.2.5 Class CEnvironment

Some operations in this class simply perform a loop and ask every cell in the environment to do the same operation at cell level. Usually the cell then goes one level down again to ask every agent (or agentlist) to do the same operations at lower levels.

#### 3.2.5.1 Operations:

- CSummaryTable* GetSummaryTable( ): Get the summary table of the environment.
- CCell* GetNeigbor(CCell*, DIRECTION): Given a cell and direction, get the neighboring cell in that direction.
- void Mutation( ): let all cells in the environment mutate at cell level (each cell does "mutate" as described in class CCell).
- void Initialize( ): initialize all cells in the environment at cell level (as described in class CCell).
- void UpdateSummary( ): Update the summary table of the environment with the sum of the summary tables in all cells.
- void DistributeFood( ): Distribute food to all cells in the environment.
- void Reaping( ): reap all cells in the environment at cell level.

- void Report(ostream&): let all cells in the environment do "report" at cell level (as described in class CCell) and out put to a file.
- int GetCycle( ): Get the number of cycles that the simulation has already run.
- void SetCycle(int): Set the number of cycles that the simulation has already run.
- void Feeding( ): feed all cells in the environment at cell level (let all cells do "feed" as described in class Ccell).
- void Taxing( ): let all cells in the environment pay tax at cell level (as described in class CCell).
- void Encounter( ): let all cells in the environment encounter at cell level (as described in class CCell).
- void PrepareNextCycle( ): let all cells in the environment do "prepare next cycle" (as described in class CCell) at the cell level.

#### 3.2.5.2 Attributes:

- A 2-dimensional array of cells in the environment.
- Number of cells in the environment. There are SIDE*SIDE cells in the environment.
- Number of cycles that have been run so far.

- A summary table. The summary information is on Environment basis, for example, how many reproductions happened in the Environment so far. Computation is based on Cells.

### 3.2.6 Class CSimParam

This class does not have any operations.

Simulation parameters are used as global variables. We define all these simulation parameters in this class as static attributes, which are common to all objects of the class. Although we do not need to create any objects while using static attributes, the encapsulation of these simulation parameters demonstrates the Object-oriented concept. Simulation parameters are described in *Section 2.1.6* on page 7.

### 3.2.7 Class CSummaryTable

This class contains the simulation results. It works as a virtual table in memory instead of on disk. There will be one CSummaryTable object for the environment and SIDE*SIDE objects for cells in memory when the simulation is running.

#### 3.2.7.1 Operations:

- void incAgentNumber(int): Increase the agent number when new agent is created.
- void DecreaseAgent(int): Decrease the agent number when agent dies.
- unsigned long GetAgentNumber( ): Get the agent number.

- unsigned long GetCombatNumber( ): Get the number of combats happened.
- unsigned long GetDeathNumber( ): Get the number of agents that died.
- unsigned long GetDrawnNumber( ): Get the number of combats drawn.
- unsigned long GetNewBirth( ): Get the number of new births.
- unsigned long GetTradeNumber( ): Get the number of trades that occurred.
- void IncCombatNumber(int): Increase number of combats when combat happens.
- void IncTradeNumber(int): Increase number of trades when trade happens.
- void IncReproductionNumber(int): Increase number of reproduction when reproduction happens.
- void IncNewBirth(int): Increase number of new birth when new agent is created. Reproduction does not guarantee that new birth will happen since there probably has not enough resource for the child. So the number of new birth and number of reproduction are usually different.
- void IncDeathNumber(int): Increase number of dead agents when agent dies.

- void IncDrawnNumber(int): Increase number of drawn combats when a draw of combat happens.
- void IncWinNumber(int): Increase number of combat won when one agent wins the combat.
- CSummaryTable operator+(CSummaryTable&): Add two summary tables (usually one cycle is finished, add the summary table of the current simulation cycle with the summary table before this cycle).

#### 3.2.7.2 Attributes:

- Number of agents.
- Number of combat drawn.
- Number of newly produced agents.
- Number of agents that died.
- Number of combat won.
- Number of combat happened.
- Number of trade happened.
- Number of reproduction happened.

### 3.2.8 Class CSimApp

This class creates the CEnvironment object, which then creates cells, beforelist, and agents.

#### 3.2.8.1 Operations:

- void Initialize( ): initialize the environment at environment level.
- int GetCycleNumber( ): Get the current number of simulation cycles that has been run.
- CSummaryTable* GetSummaryTable( ): Get the summary table of the environment.
- void Report( ostream& ): let the environment report at environment level (for example, how many agents left in the whole environment).
- void RunOneCycle( ): run one cycle of the simulation starting from environment level, including distributing food, feeding, taxing, encountering, reaping, mutation, update summary table, and prepare next cycle. This function shows the sequence of one simulation cycle.

#### 3.2.8.2 Attributes:

- A variable shows the simulation state is paused or not.
- The output file.
- The environment.

## 3.3 Graphical user interface and related classes

To make the simulation interactive, we need to provide a friendly Graphical User Interface

so the user does not need to spend much time on learning how to use the simulation. Since Microsoft Windows is the most popular Desktop Operating System, we designed our GUI follow the standard Windows style. It looks like most Windows applications so if the user is familiar with windows applications, the user will have no difficulty running our simulation application.

### 3.3.1 Dialog box and controls

Common Windows applications have several dialog boxes, each designed to retrieve a different type of information from the user. With Microsoft Visual C++, for each dialog box that appears on screen, there are two entities need to be developed: a dialog box resource and a dialog box class.

The dialog box resource is used to draw the dialog box and its controls on the screen. The class holds the values of the dialog box. A member function of the class causes the dialog box to be drawn on the screen. They work together to achieve the overall effect, making communication with the program easier for user.

In our project, we add the following controls into dialog boxes for different purposes:

- Static text controls which show some information to user. The user cannot modify the information.
- Edit box controls which show values to the user. The user can also edit (change) the value. Sometimes we can "lock" the edit box to prevent its value from being modified

- Buttons which are clicked by the user to confirm/cancel the modification. The current dialog box is usually closed or another window is popped up when the button is clicked.

Usually the value of the control (for example the value shown in a edit box) in a dialog box is matched to a member variable of the dialog box class.

#### 3.3.1.1 "About" dialog box

The "About" dialog box shows the author of the project and the software version. User clicks a menu item to show the dialog box and clicks an "OK" button to close it. No further operations and attributes are needed here.

#### 3.3.1.2 Simulation parameter dialog box

We need a dialog box to show the user the default (or current values if user has changed the default values) simulation parameters. The user is allowed to modify the values shown. The user can confirm the modification or cancel the modification. In case of cancel, the simulation parameter should keep the current values. Since users may need to understand the meanings of the simulation parameters to help them set their desired values, a way to invoke the help system from the dialog box is required.

#### 3.3.1.3 Summary cycle dialog box

We offer this dialog box to the user so the user can decide how frequent to show the simulation results (summary dialog box as described later). For example, if the user sets the value to 20, then the simulation results (summary dialog box) will pop up each time 20

simulation cycles have been run. The user can confirm or cancel the value entered. In case of cancellation, the current value is reserved.

#### 3.3.1.4 Summary table dialog box

This dialog box shows the simulation results, which is the statistics information we are interested in. The user is not allowed to modify the information shown. When the summary dialog box is shown, the user clicks on the OK button to let the program continue to run.

### 3.3.2 Menus

Menus in our simulation project provide the following functions:

- Bring up the dialog boxes described above.
- Start/stop/pause/resume the application.
- Show/hide toolbars.
- Show/hide status bars.
- Bring up helps.
- Exit the application.

Cares must be taken to that when some functions are not allowed in certain cases, the related menu items must be disabled.

### 3.3.3 Help system

In Windows applications, there are a number of ways to bring up help:

- By choosing an item from the Help menu

- By pressing F1
- By clicking on What's This? button on a toolbar and then clicking something else
- By choosing What's This? from the Help menu and then clicking on something
- By clicking a Question button on dialog box and then clicking part of the dialog box
- By right-clicking something and choosing What's This? from the pop-up menu

In our application, we choose the following two approaches:

- The help menu, which brings the help topics window.
- The help button on a Dialog box, which pops up a help window

# 4 Implementation

## 4.1 Introduction to Visual C++, the implementation tool

The 32-bit applications for Windows are often far larger and more complex than their predecessors for 16-bit Windows, or older programs that did not use a graphical user interface. Yet as program size and complexity has grown, programmer effort has actually decreased, at least for programmers who are using the right tools. After having implemented several course projects using Microsoft Visual Studio, we found that Visual C++ is one of the right tools. With its code-generating Wizards, it can produce the shell of working Windows application in seconds. The class library included with Visual C++, the Microsoft Foundation Classes (MFC), has become the industry standard for Windows software development in a variety of C++ compilers. The visual editing tools make layout of menus and dialog boxes much easier. The time we invested in learning to use this product has paid itself back on our project.

Visual C++ doesn't just compile code. It generates code. Using the tool called AppWizard provided by Visual C++, we could select the options we want and let it create a starter application. It copies code into our application that almost all Windows applications need, for example basic menus, minimize and maximize buttons. Then we can add/remove menus and modify the generated code. The next step is to write the code of our simulation classes.

## 4.2 Programming progress

We first use Visual C++'s AppWizard to create a skeleton application. Then, according to our design, we use ClassWizard to generate the class declaration for simulation classes for example class CAgent, CCell, etc. This step includes the declaration of class member functions and member variables. The next step is to define simulation classes we just added to the application. We also need to modify the code and user interface (menus, dialog boxes) that created by AppWizard to make the user interface work well with the simulation classes.

## 4.3 AppWizard created classes

### 4.3.1 Documents and Views

MFC's document/view architecture separates an application's data from the way the user actually views and manipulates that data. Simply, the document object is responsible for storing, loading, and saving the data, whereas the view object (which is just another type of window) enables the user to see the data on-screen and to edit that data in a way that is appropriate to the application.

#### 4.3.1.1 Class CSimulationDoc

The document class CSimulationDoc is created by AppWizard and is derived from class CDocument. We add our own functions and attributes according to our requirements.

#### 4.3.1.1.1 Operations: Some important member functions are listed below:

- int CSimulationDoc::GetSummaryCycle( ): return the number of cycles to show the summary table.
- void CSimulationDoc::OnSimulationParameter( ): when the user clicks the menu item to show the simulation parameter dialog box, this function generates a parameter dialog box object and assigns the simulation parameter values to the member variables of the dialog box. When the user clicks the OK button of the dialog box, this function gets the values currently shown (either modified by user or not) and assign the values back to the simulation parameters.
- void CSimulationDoc::OnSummaryCycle( ): when the user clicks a menu item to show the simulation summary cycle dialog box, this function generates a summary cycle dialog box object and assigns the summary cycle value (number of cycles to show the summary table) to the member variables of the dialog box. When the user clicks the OK button of the dialog box, this function gets the value currently shown (either modified by user or not) and assign the value back to the summary cycle.
- void CSimulationDoc::OnSummarytable( ): when the summary table is to shown, this function gets the simulation summary table and hence simulation results in the table (for

example number of agents left). It then sets the simulation results to the member variables of the summary dialog box.

- void CSimulationDoc::Start( ): When the user clicks the start menu item or the toolbar, this function creates an object of class CSimApp. The CSimApp object then calls its function "Initialize". After the initialization, it starts a thread to run the simulation.
- UINT CSimulationDoc::ThreadFunc(LPVOID p): This function is called by Start. It lets the CSimApp object to run the function RunOneCycle( ). It counts the simulation cycle and pops up the summary table dialog box by calling function OnSummarytable every specified number of cycles.

#### 4.3.1.2 CSimulationView

CSimulationView is derived from class CView. As mentioned previously, the view class displays the data stored in the document object and enables the user to modify this data. The view object keeps a pointer to the document object, which it uses to access the document's member variables in order to display or modify them.

##### 4.3.1.2.1 Operations:

- void CSimulationView::OnStart( ):display running status of the application (starting, running), get the pointer to the

document object, and call the document objects function Start.

- SimulationDoc* CSimulationView::GetDocument( ): get a pointer to the document object.
- void CSimulationView::DisplayInfo(CDC* pDC, char *info): This function displays the information on the screen. It finds out the Device Context, create new font, save the old font, get the client area, draw the text in that area, and then restore the old font. It is called by functions OnDraw, On Start, OnStop, etc.
- void CSimulationView::DisplayInfo(CDC* pDC, char *info): This function displays the information on the screen. It finds out the Device Context, create new font, save the old font, get the client area, draw the text in that area, and then restore the old font. It is called by functions OnDraw, On Start, OnStop, etc.

## 4.4 Agent classes

We summarize the main member functions of the agent simulation classes with their input/output and a brief description in associated classes in this section. Some class diagrams are listed in appendix B on page 73 and appendix C on page 74. For more detailed information, please refer to appendix 5 on page 72.

### 4.4.1 Class CSimString

- void CSimString::Clear( ):clear the resource array, set all resource numbers to zero.
- void CSimString::convert2cstr(char* buffer, int size): convert the CSimString to a C language string and put it in "buffer".
- void CSimString::Decrease(int type, int number): decrease a specific type of resource by a specific number.
- void CSimString::Mutation( ): randomly decrease the number of one type of resource.
- int CSimString::operator*(CSimString & x): return the number of matched resources in two CSimStrings.
- CSimString CSimString::operator+(CSimString & x): combine two CsimStrings and add all their resources together.
- int CSimString::operator-(CSimString & x): returns the number of all resources that the current CSimString exceeds the given CSimString.
- CSimString & CSimString::operator=(CSimString & theString): assign the given CSimString to the current CSimString.
- bool CSimString::operator>=(CSimString & x): returns true if the number of each type of resource in the current CSim-

String is greater than the number of that resource in the given CSimString

- int & CSimString::operator[](int r): get the number of certain resource.
- int CSimString::ResourceSize(int type): return the number of specific type of resource.
- int CSimString::size( ): return the total number of all resources.

### 4.4.2 Class CAgent

- void CAgent::AddToList(CAgentList* theList): add the agent to an agentlist (beforelist or afterlist).
- void CAgent::Combat(CAgent* agentB): fight with the given agent. One may kill another.
- CSimString CAgent::ConstructGene(CSimString& LeftGene, CSimString& RightGene): given two genes from parent agents, construct a new gene for the child. Called by function GiveBirth.
- void CAgent::ContributeResources(CAgent* AgentB, CAgent* Child): contributes resources from parents' reserves to child's reserve. Called by function GiveBirth.

- void CAgent::Die( ):The agent dies and contributes all its resources to the cell it lives in as food. Update the summary table of the cell.
- void CAgent::Eating( ): transfers the food from the cell. If the agent could not get any food, move the agent to a neighboring cell randomly.
- void CAgent::Encounter(CAgent* theAgent): encounter with a given agent. The sequence of trade, reproduction, and combat has 3 * 2 = 6 possibilities. To make it even, we randomly choose a number (0 to 5) each time to decide the sequence of trade, combat, and reproduction.
- CCell* CAgent::GetCell( ): return a pointer to the cell the agent lives in.
- CAgent* CAgent::GetNext( ): return a pointer to the next agent of the current agent on an agentlist.
- CAgent* CAgent::GiveBirth(CAgent* agentB): Act as parent agents with the given agent to create a child agent. The parents construct the genes and reserve of the child agent. Called by function Reproduce.
- bool CAgent::Initialize( ): create the genes and reserve of the agent.
- bool CAgent::IsActive( ): check the status of the agent (active or not).

- bool CAgent::IsFightable(CAgent* theAgent): Check if two agents will combat with each other according to their scores.
- bool CAgent::IsMatable(CAgent* theAgent): check if two agents will reproduce according to their Beauty and Lust genes.
- bool CAgent::IsSurrender(int part, int total): decide if the combat is drawn. Called by functions IsFightable and IsWinner.
- bool CAgent::IsThreaten(CAgent* theAgent): compares the Attack gene of the current agent and the Defend gene of a given agent. Called by function IsFightable.
- bool CAgent::IsTradable(CAgent* theAgent): compares the Attack and trade gene of two agents.
- bool CAgent::IsWinner(int part, int total): called by function Combat.
- void CAgent::Kill(CAgent* agentB): get all resources from the give agent and save the resources to reserve. Then remove the given agent from memory. Called by function Combat when on agent wins the combat.
- void CAgent::Move(DIRECTION direction): find a neighboring cell and change the agent's cell to that cell.

- bool CAgent::Mutation( ): mutate all genes of the agent by calling the function CString::Mutation.
- CAgent& CAgent::operator=(CAgent& theAgent): over-load the = operator for class CAgent.
- void CAgent::PayTax( ): decrease resources from reserve.
- void CAgent::Reproduce(CAgent* agentB): call Function GiveBirth to create new agent. Update summary table.
- int CAgent::ResourceSize(int type): return the total size of a give type of resource in the agent.
- void CAgent::SetActive(bool type): set the agent's status to active or not active.
- void CAgent::SetNext(CAgent* theAgent): set the agents next agent on the agentlist.
- int CAgent::Size( ): return the total size of all type of resources in the agent.
- void CAgent::Trade(CAgent* agentB): transfer resources between two agents' reserve.

### 4.4.3 Class CAgentList

- void CAgentList::AddAgent(CAgent* theAgent): set the given agent to the list of the agentlist.
- int CAgentList::GetCount( ): return the number of agents on the list.

- CAgent* CAgentList::GetHeader( ): return a pointer t the header agent of the agentlist.
- void CAgentList::Initialize( ): let all agents on the agentlist call the CAgent::Initialize.
- void CAgentList::RandomAddAgent(CAgent* theAgent): add the given agent to a random position of the agentlist.
- void CAgentList::RemoveAgent(CAgent* theAgent): remove the given agent from the agentlist.
- CAgent* CAgentList::RemoveFirstAgent( ): remove the header agent from the agentlist.
- void CAgentList::SetCount(int count): set the value of "number of agents on the agentlist".
- void CAgentList::SetHeader(CAgent* theAgent): set the given agent to the header of the agentlist.

### 4.4.4 Class CCell

- void CCell::AddFood(CSimString & fd): add the given string to the cell as food.
- void CCell::Encounter( ): remove every first two agents from the cell's beforelist and let them encounter (CAgent::Encounter). If only one agent left, move it to the afterlist.

- void CCell::Feeding( ): let all agents on beforelist eat food in the cell.
- CAgentList* CCell::GetAfterList( ): return a pointer to the afterlist of the cell.
- CAgentList* CCell::GetBeforeList( ): return a pointer to the beforelist of the cell.
- CEnvironment* CCell::GetEnvironment( ): return a pointer to the environment the cell belongs to.
- CSimString & CCell::GetFood( ): get the food the cell has.
- CCell* CCell::GetNeighbor(DIRECTION theDirection): given a direction, get a pointer to the neighboring cell.
- CSummaryTable* CCell::GetSummaryTable( ): return a pointer to the summary table of the cell.
- void CCell::Initialize( ): initialize the beforelist (CAgnetList::Initialize).
- void CCell::Mutation( ): mutate all agents on the afterlist of the cell (call CAgent::Mutation for each agent).
- void CCell::PrepareNextCycle( ): remove agents from afterlist and randomly add them to beforelist. The beforelist is to be used for the next simulation cycle.
- void CCell::Reaping( ): go through the afterlist and remove all inactive agents. Removed agents die (CAgent::die).

- bool CCell::Report(ostream& fout): output the summary table of the cell to a file.
- void CCell::SetEnvironment(CEnvironment* pEnv): set the environment of the cell.
- void CCell::SetFood(CSimString & fd): set the food of the cell to the given string.
- void CCell::Taxing( ): let all agents on the beforelist pay tax (all agents run CAgent::PayTax).

### 4.4.5 Class CEnvironment

- void CEnvironment::DistributeFood( ): create food and add food to all cells in the environment.
- void CEnvironment::Encounter( ): let all cells in the environment do encounter (call CCell::Encounter).
- void CEnvironment::Feeding( ): let all cells in the environment do feeding (call CCell::Feeding).
- int CEnvironment::GetCycle( ): return the number of cycles have been run so far.
- CCell* CEnvironment::GetNeigbor(CCell* theCell, DIRECTION direction): given a direction, return a pointer to the neighboring cell on the direction.
- CSummaryTable* CEnvironment::GetSummaryTable( ): return a pointer to the summary of the environment.

- void CEnvironment::Initialize( ): initialize all cells in the environment.
- void CEnvironment::Mutation( ): mutate all cells in the environment.
- void CEnvironment::PrepareNextCycle( ): let all cells in the environment prepare the next cycle.
- void CEnvironment::Reaping( ): reap all cells (all cells call CCell::Reaping).
- void CEnvironment::Report(ostream& fout): let all cells in the environment output the summary to a file.
- void CEnvironment::SetCycle(int cycle): set the number of cycles has been run.
- void CEnvironment::Taxing( ): tax all cells in the environment (all cells call CCell::Taxing).
- void CEnvironment::UpdateSummary( ): add the summary tables of all cells together.

### 4.4.6 Class CSimParam

This class only has static class member variables with default values:

- int CSimParam::ACTIVE = 10
- float CSimParam::FRAC = 0.33;
- int CSimParam::MINMATCH = 2;
- int CSimParam::POP = 500;

- int CSimParam::SIDE = 10;
- float CSimParam::TAX = 0.1;
- int CSimParam::VARIETY = 5;

### 4.4.7 Class CSummaryTable

- void CSummaryTable::DecreaseAgent(int count): decrease the number of agents.
- unsigned long CSummaryTable::GetAgentNumber( ): return the number of agents.
- unsigned long CSummaryTable::GetCombatNumber( ): return the number of combats happened.
- unsigned long CSummaryTable::GetDeathNumber( ): return the number of agents died.
- unsigned long CSummaryTable::GetDrawnNumber( ): return the number of drawn combats.
- unsigned long CSummaryTable::GetNewBirth( ); return the number of new birth.
- unsigned long CSummaryTable::GetReproductionNumber( ): return the number of reproduction happened.
- unsigned long CSummaryTable::GetTradeNumber( ): return the number of trade happened.
- unsigned long CSummaryTable::GetWinNumber( ): return the number of combats won.

- void CSummaryTable::IncAgentNumber(int count): increase the number of agents.
- void CSummaryTable::IncCombatNumber(int count): increase the number of combatshappened.
- void CSummaryTable::IncDeathNumber(int count): increase the number of agents died.
- void CSummaryTable::IncDrawnNumber(int count): increase the number of drawn combats happened.
- void CSummaryTable::IncNewBirth(int count): increase the number of new birth.
- void CSummaryTable::IncTradeNumber(int count): increase the number of trade happened.
- void CSummaryTable::IncWinNumber(int count): increase the number of combats won.
- CSummaryTable CSummaryTable::operator+(CSummaryTable& x): enable the addition of two summary tables.

## 4.5 Graphical user interface:

### 4.5.1 Toolbar:

The buttons on a toolbar correspond to commands, just as the items on a menu do. Although we can add a toolbar to the application with AppWizard, we still need to use a little programming polish to get things just right. This is because every application is dif-

ferent, and AppWizard can create only the most generally useful toolbar for most applications. So in our application, we needed to add and delete some Toolbar items to support our application's unique command set.

For example, AppWizard created a toolbar with buttons:

- New
- Open
- Save
- Cut
- Paste
- Print

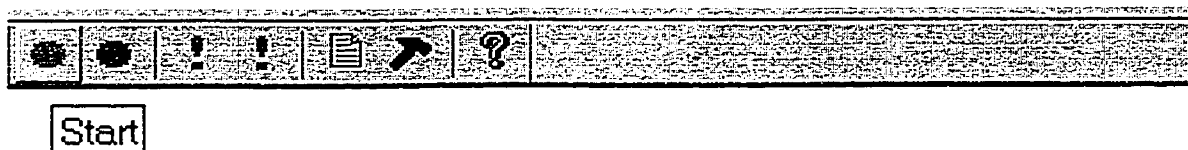
These buttons are not needed in our application. So we deleted these buttons and added our own buttons:

- Start
- Stop
- Resume
- Pause
- Summary Table
- Simulation Parameter

To add toolbar buttons, we first draw the button's icon (bit map) using the tool provided by Microsoft Visual Studio. Then we match the button with its command (usually associate it with a menu command so it does the same thing as a menu item). We also provide each button with its ToolTip and description. The ToolTip appears whenever the user leaves the mouse pointer over the button for a second or two and acts as a reminder of the

button's purpose. The description appears in the message area of the application's status bar, as described in the next section. The toolbar is shown in *Figure 4-1: Toolbar* on page 48.

***FIGURE 4-1: Toolbar***




### 4.5.2 Status bar:

Status bars are mostly benign objects that sit at the bottom of the application's window, doing whatever MFC instructs them to do. This consists of displaying command descriptions and the status of various keys on the keyboard, including the Caps Lock and Scroll Lock keys. In fact, status bars are so mundane from the programmer's point of view that they aren't even represented by a resource that we can edit like a toolbar.

A status bar, like a toolbar, must reflect the interface needs of the specific application. For that reason, the CStatusBar class features a set of methods with which we can customize the status bar's appearance and operation.

The status bar has several parts, called panes, which display certain information about the status of the application and the system. These panes include indicators for the Caps Lock, Num Lock, and Scroll Lock keys, as well as a message area for showing status text and command descriptions. Although all these are implemented in our project, only the message area is particularly meaningful. When user places the mouse pointer over the toolbars, the message area changes the text in it, showing that the function of the toolbar being pointed.

### 4.5.3 Dialog box and controls

We build a dialog box resource with the resource editor, adding controls to it and arranging them to make the control easy to use. Class Wizard then helps us to create a dialog box class, typically derived from the MFC class CDialog, and to connect the resource to the class. Usually each control on the dialog box resource corresponds to one member function of the class. To set the control values to defaults before displaying the dialog box, or to determine the values of the controls after the user is finished with the box, we use the member variables of the class.

There are four dialog boxes implemented in our project:

#### 4.5.3.1 About Simulation Dialog:

When user clicks menu Help, About Simulation…, this dialog box pops up. as shown in *Figure 4-2* . It has an OK button and some labels (static text), which illustrate the name, version and authors of the application.

*FIGURE 4-2: The About Dialog box*

#### 4.5.3.1.1 Class CAboutDlg:

No special functions and attributes are needed.

### 4.5.3.2 Simulation Parameter Dialog:

When user clicks menu Setup, Simulation Parameter, this dialog box pops up as shown in *Figure 4-3* on page 51. It has some labels, edit boxes, and some buttons. User can set the simulation parameters in the edit boxed and confirm by click the OK button or cancel by clicking the Cancel button. If user confirms the modification, get the values user entered. Otherwise nothing happens with the current data. There is a button labeled Help, which triggers the help system.

#### 4.5.3.2.1 Class CSimParaDlg

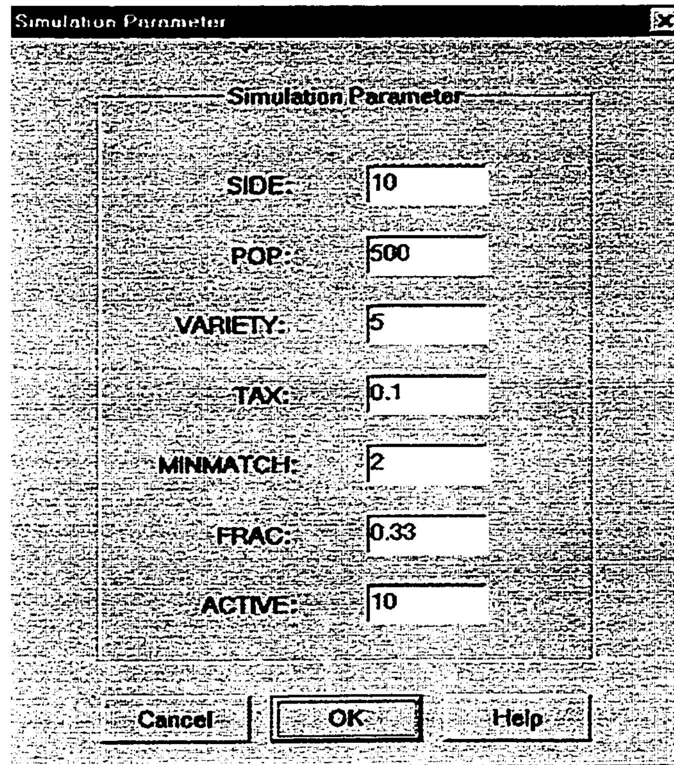
Operations:

- void OnOK( ):Verify the simulation is running or not. If it is not running, change the parameters. Changing parameters for a running simulation is not allowed.
- afx_msg void OnParamterBtnHelp( ): Bring up the help window for simulation parameters.

Attributes:

- bool m_running: A variable shows the status of the simulation: running or not.
- Seven variables matching the seven simulation parameter edit boxes.

FIGURE 4-3: *Simulation parameter dialog box*




#### 4.5.3.3 Summary Cycle Dialog:

When user clicks on menu Setup, Summary Cycle, this dialog box pops up as shown in *Figure 4-4* on page 52. User can set the number of simulation cycles to show the Summary Dialog box by modifying the value in the edit box. The default value is 50.

##### 4.5.3.3.1 Class CSummaryCycleDlg

Operations:

- If user confirms the modification, get the values user entered. Otherwise nothing happens with the current data.

Attributes:

- UINTm_cyclenumber: Number of cycles to show the simulation results (summary dialog box), related to the value of the summary cycle edit box.

*FIGURE 4-4: The Summary cycle dialog box*

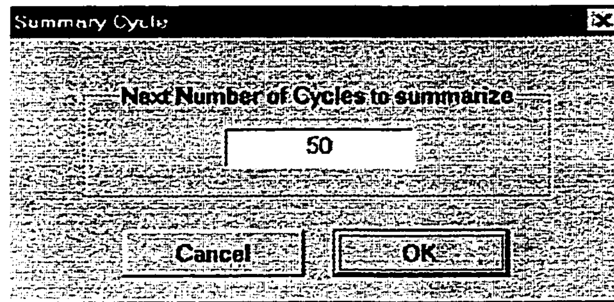
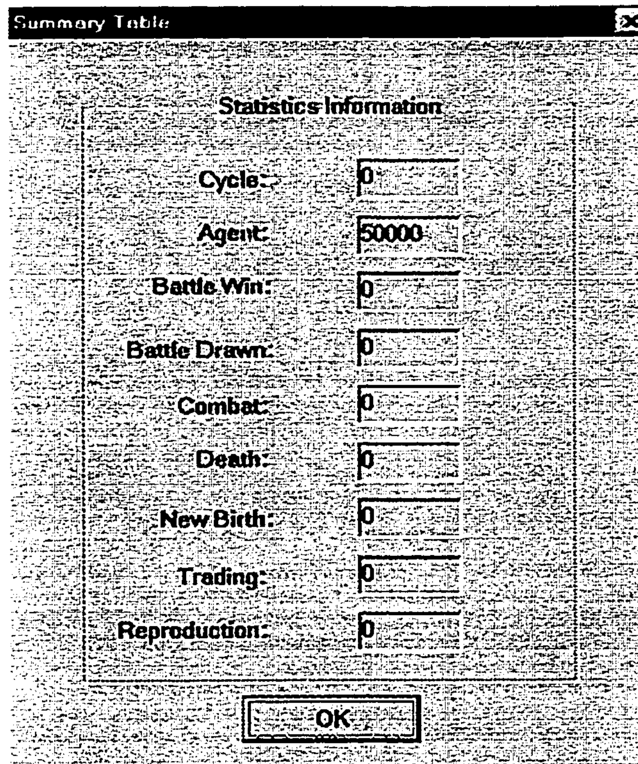



*FIGURE 4-5: The summary Table dialog box*




### 4.5.3.4 Summary Table Dialog:

When user clicks menu Simulation, Summary Table, this dialog box pops up as shown in

*Figure 4-5* on page 52. Or, every 50 cycles (default value), it pops up and shows the intermediate data.

#### 4.5.3.4.1 Class CSummarytableDlg

Operations:

- No specific operations needed.

Attributes:

- UINTm_cycle: the current cycle number, related to the value of a static text field.
- UINTm_agent: the number of agents alive, related to a locked edit box.
- UINTm_birth: the number of new birth, related to a locked edit box.
- UINTm_combat: the number of combats happened, related to the value of a locked edit box.
- UINTm_drawn: the number of combat won, related to the value of a locked edit box.
- UINTm_drawn: the number of combat drawn, related to the value of a locked edit box.
- UINTm_reproduction: the number of reproduction, related to the value of a locked edit box.
- UINTm_death: the number of agents died, related to the value of a locked edit box.

# 4.6 The help system

## 4.6.1 Presenting help

We offer two approaches to launch the help system in our project:

- Help Topics dialog box (as shown in *Figure 4-6* on page 54): allows user to scroll through an index, look at a table of contents, or find a word within the Help text.
- The help button on Simulation Parameter Dialog box (as shown in *Figure 4-3* on page 51), which pops up a help window (as shown in *Figure 4-7* on page 55) illustrating the meaning of all simulation parameters.

***FIGURE 4-6: Help Topics***

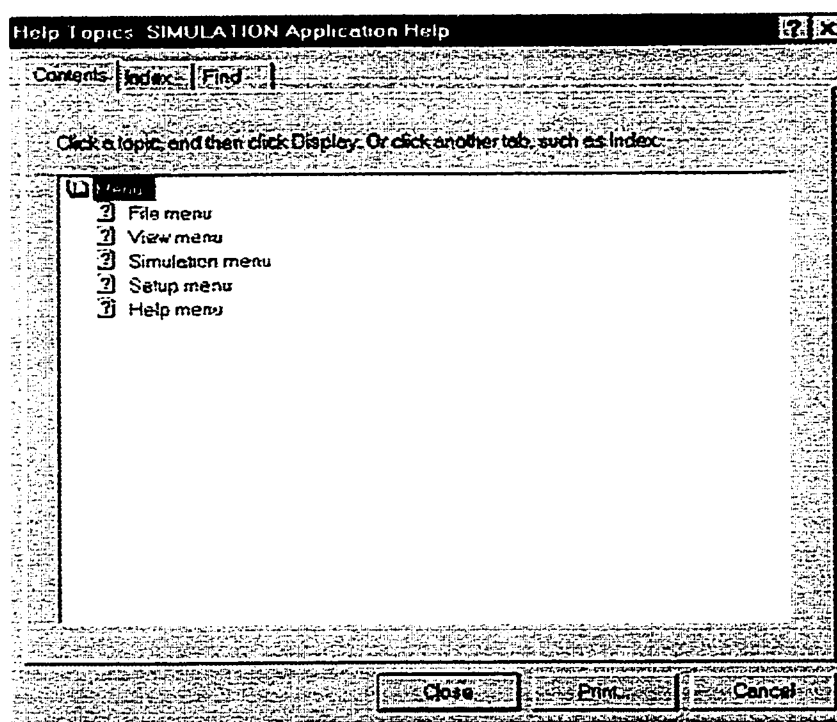
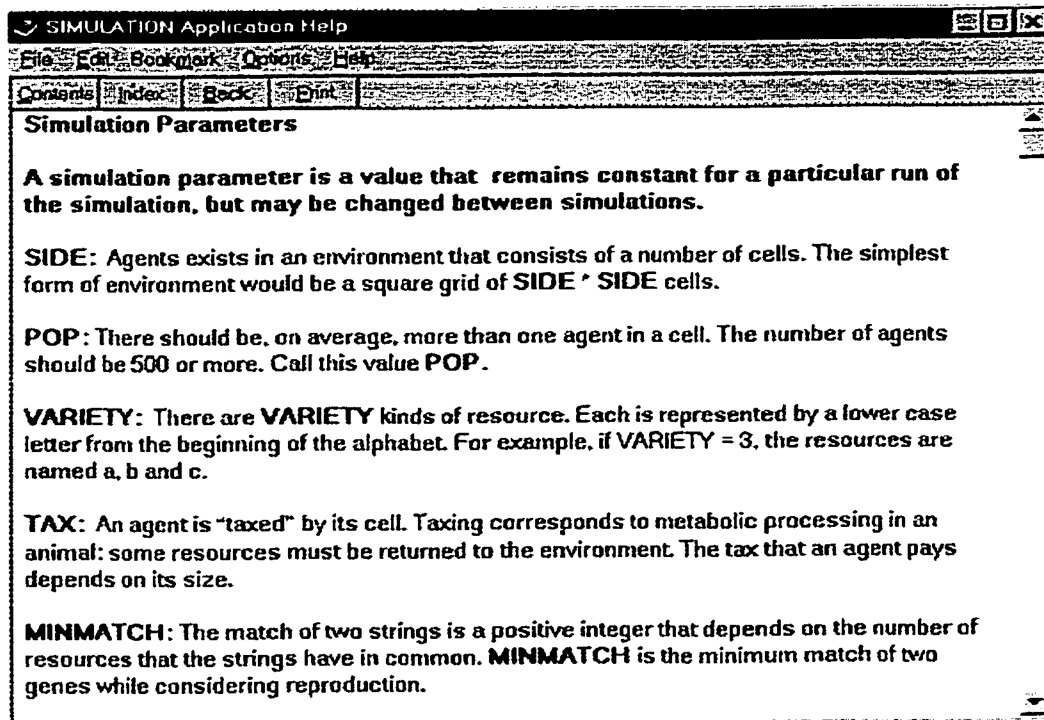


*FIGURE 4-7: Help button triggered Help window*




### 4.6.2 Components of the Help System

From the point of view of the developer, there are a large number of files interact to make the on-line Help work. The final product, which is delivered to the user, is the Help file, with the .HLP extension. The component files produced by Visual C++'s Application Wizard are as follows:

- .h files: define resource Ids and Help topic Ids for use with the C++ code.
- .hm files: are called Help Mapping files that define Help topic IDs.

- .rtf files: Rich text Format files that contain the Help text for each Help topic. Eight kinds of footnotes (for example #, $, and K) are used to link help topics.
- .cnt file: table of contents file that is used to create the contents tab of the Help Topics dialog box.
- .hpj file: it pulls together .hm and .rtf files to produce, when compiled, an .hlp file.

When being used, the Help system generates other files. These files need to be removed in case user uninstalls the application:

- .gid file: is a configuration file, typically hidden.
- .fts file: is a full text search file, generated when user does a Find through the Help text.
- .ftg file: is a full text search group list, also generated when user does a Find.

### 4.6.3 Programming the help

#### 4.6.3.1 The help button

To link the button Help in the Simulation parameter dialog box, we need to define a Help ID in file CSimParaDlg.h as below:

#define HID_PARAMETERS 0x01

Then we need to add a help mapping entry in a new file named Simulationx.hm as:

HID_PARAMETERS 0x01

Next, we edit the Help project file Simulation.hpj and add a line:

*#include < Simulationx.hm >*

Now both the Help system and the compiler know about this new Help topic ID.

We add the function in class CSimParaDlg, as described in *Section 4.5.3.2.1* on page 50.

```
void CSimParamDlg::OnParamterBtnHelp( )
{
        WinHelp(HID_PARAMETERS);
}
```

and a message mapping:

```
BEGIN_MESSAGE_MAP(CSimParamDlg, CDialog)
        //{{AFX_MSG_MAP(CSimParamDlg)
        ON_BN_CLICKED(IDC_PARAMTER_BTN_HELP, OnParamterBtnHelp)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP( )
```

The message mapping is not explained here. For more information, please refer to appendix 4 on page 72

The real help text is edited as described in the next section.

#### 4.6.3.2 The Context help

All help text is written in the .rtf file. The AppWizard creates a boilerplate Help file named afxcore.rtf. This file needs to be modified as below:

Add the name of our program ("Simulation") to the <<YourApp>> position all over the file

Remove all help entries that are not shown in our program such as "Print" menu item, "Window" menu.

Then we need to create our own .rtf file named Simulation.rtf and add our own help entries and help texts, for example help for menu “Simulation”.

We use the tool Help Workshop provided by Microsoft Visual Studio to adjust the help contents shown each time we bring the context help.

# 5 Simulation result

When we first run the simulation, we observed that no matter what the simulation parameters we chose, all agents would die after a number of cycles. We investigated it and found the reason was that we used a fixed sequence of agent interaction: fighting first, then trading, and reproducing at last. So the fighting had higher probability than trading and reproduction. Only if the combat was drawn could trading and reproduction happen. But most agents were killed during combat and the agent population would always decrease. We changed our design to give a random sequence of interaction: when two agents encounter, the possibilities of fighting, trading and reproduction were made as even as possible.

In this chapter, we use two examples describe the results that the Agent Simulation is capable of producing. Note here that even we use the same set of simulation parameter values to run the simulation the results will be always different. The reason is that the interactions and the initialization of agents are implemented as randomly as possible.

## 5.1 Example 1

The first example will present a simulation process using all the default simulation parameters.

The default simulation parameters and their values are shown in *Section 4.4.6* on page 44.

The summary table displays every 50 simulation cycles.

Step 1. Start the program:

The user interface starts as shown in *Figure 5-1: Before Simulation starts* on page 60. It dis-

plays "Simulation is stopped" in the middle of the window.

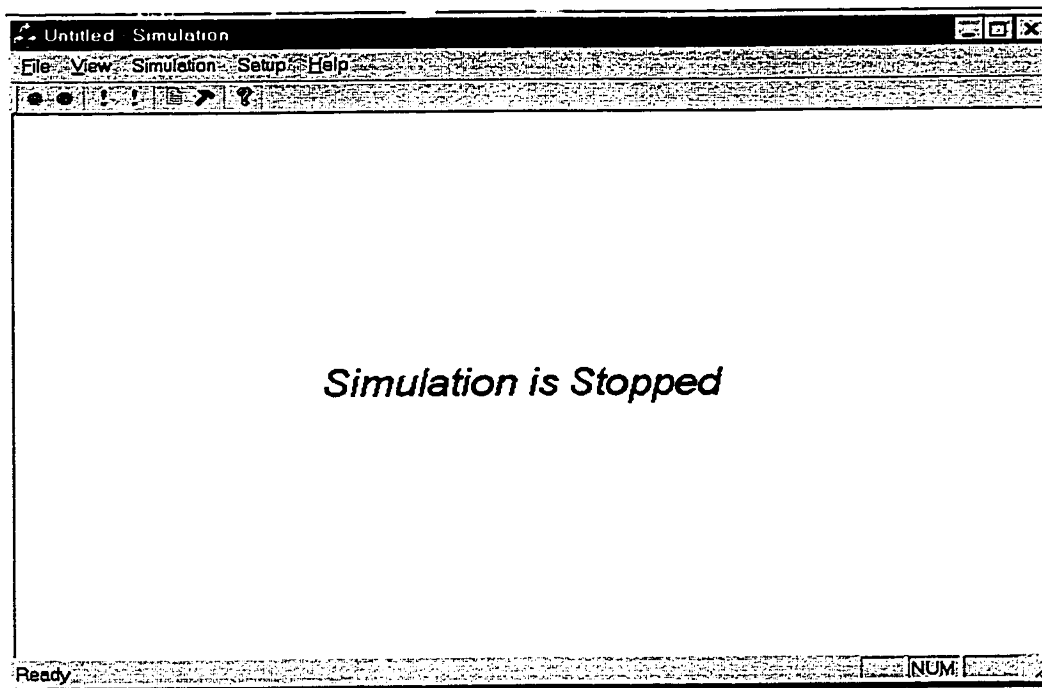
Step 2. Checking default simulation parameters:

Click on menu Setup, Simulation Parameter, the Simulation Parameter dialog box is shown with all default parameters on it as shown in *Figure 4-3: Simulation parameter dialog box* on page 51. Click OK. The dialog box disappears.

Step 3. Start the simulation

Click on the menu Simulation, Start or the Toolbar button to start the simulation. The message on the windows changes to "Starting the simulation, Please wait..."

***FIGURE 5-1: Before Simulation starts***



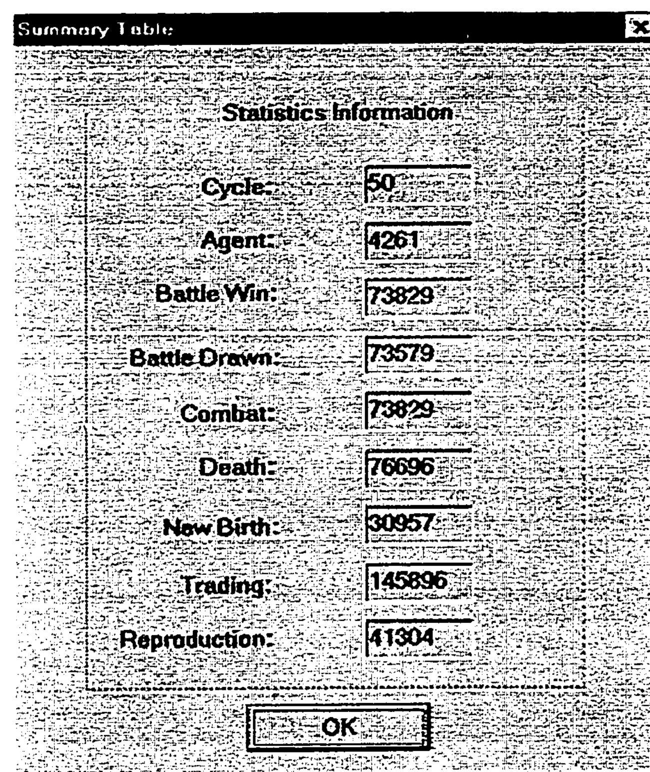

Step 4. The summary table dialog box is shown with the initial statistic information as shown in *Figure 4-5* on page 52.

Step 5. Click on OK to hide the summary table dialog box. The message on the window shows “Simulation is running”.

Step 6. After every 50 cycles, a new summary table dialog box appears. Observe the statistics and click on OK button.

Step 7. After 200 simulation cycles, the agent number increases dramatically. Quit the program by clicking on the “stop” button on the toolbar. The simulation results are captured and shown in *Figure 5-2* .

***FIGURE 5-2: Simulation result after 50 cycles with example 1***




The simulation result after 50 cycles is shown in *Figure 5-2* . Initially there were 10*10*500=50000 agents. After 50 simulation cycles, although there were 41394 occur-

rences of reproduction, only 30957 new agents were produced. That means that some parent agents did not have enough resource for making a child. "73829 battle win" means 73829 agents were killed after fighting with other agents. "73579 battle drawn" means although during these battles the agents did not kill each other and they might proceed to trade or reproduce.

*FIGURE 5-3: Simulation result after 100 cycles with example 1*

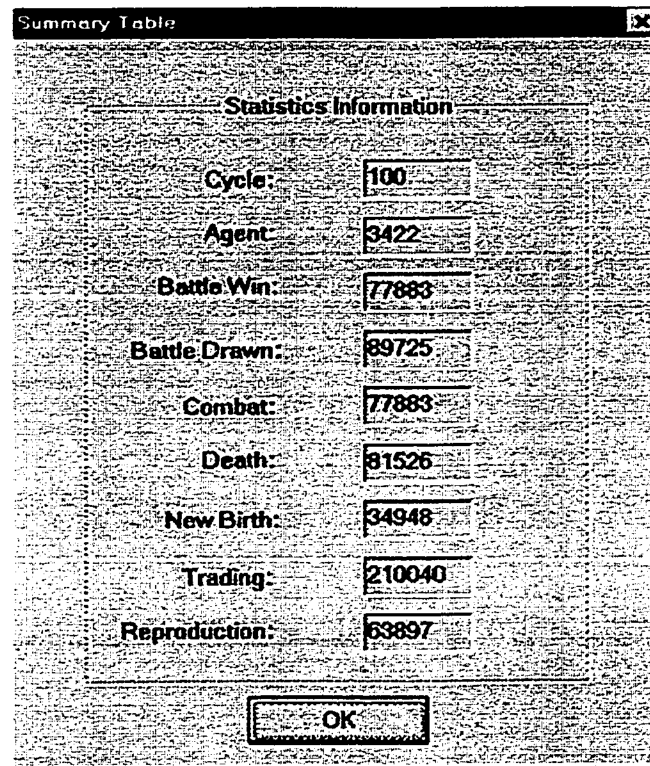



*Figure 5-3* shows that after anther 50 cycles, the agent number did not reduce too much. Fighting did not happen a lot but trading and reproduction did. It seemed that the living agents like peace. This was determined by their genes.

*FIGURE 5-4: Simulation result after 150 cycles with example 1*

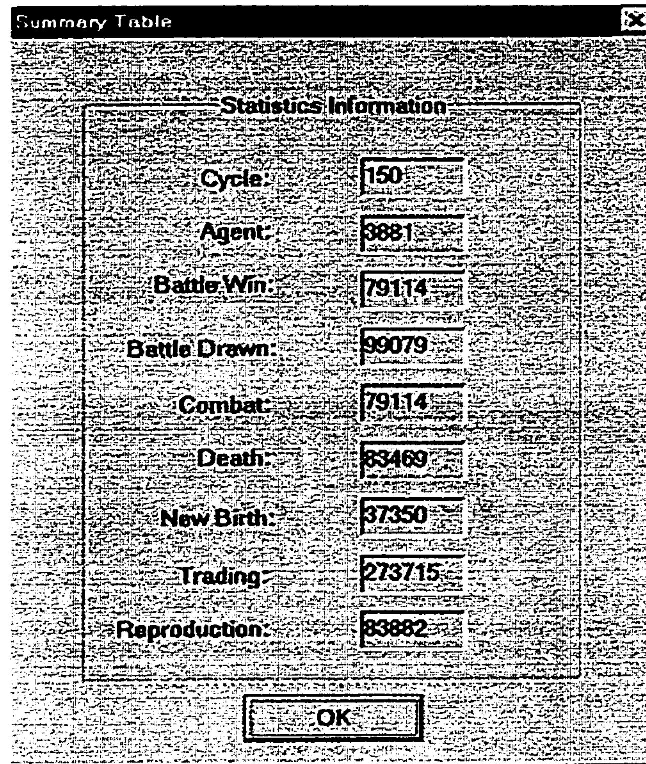



*Figure 5-4* shows the number of agents started to increase at the end of 150 simulation cycles. The living agents have enough resources to produce new agents and they are hard to die.

*FIGURE 5-5: Simulation result after 200 cycles with example 1*

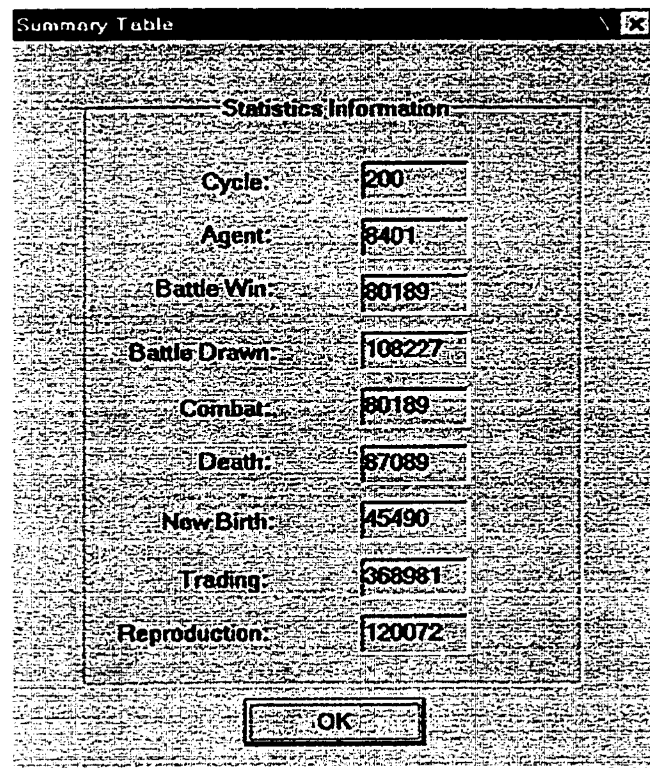



*Figure 5-5* shows during these 50 cycles, the number of agents was more than doubled. Fighting did not happen too often but trading and reproduction happened frequently. Continue running the simulation took a long time and the PC memory could hardly handle the large number of agents.

Although we cannot see the genomes of the surviving agents, according to the conditions of fighting, trading and reproduction, we can easily deduce that, from cycle 150 to cycle 200 :

- Most agents had short "attack" gene compared to "defend" gene. They did not threaten each other to make fighting occur.
- A lot of agents had long "trade" genes to incur trading.

- Most agents' "lust" gene matched other agents' "beauty" gene. They "attracted" each other.
- The agents were "rich". They had enough resources to give to the children during reproduction.

# 5.2 Example 2

With the second example we present a simulation process using some user entered simulation parameters:

- POP = 300: the initial population is decreased
- TAX = 0.2: the tax rate is increased

Other simulation parameters keep their default values.

*FIGURE 5-6: Simulation results after 50 cycles with example 2*

Summary Table

Statistics Information

| | |
|---|---|
| Cycle: | 50 |
| Agent: | 650 |
| Battle Win: | 38781 |
| Battle Drawn: | 34775 |
| Combat: | 38781 |
| Death: | 39532 |
| New Birth: | 10182 |
| Trading: | 71621 |
| Reproduction: | 17376 |

OK

***FIGURE 5-7: Simulation results after 100 and 150 cycles with example 2***

Summary Table

Statistics Information

| | |
|---|---|
| Cycle: | 100 |
| Agent: | 384 |
| Battle Win: | 39008 |
| Battle Drawn: | 36788 |
| Combat: | 39008 |
| Death: | 39823 |
| New Birth: | 10207 |
| Trading: | 80228 |
| Reproduction: | 19126 |

OK

Summary Table

Statistics Information

| | |
|---|---|
| Cycle: | 150 |
| Agent: | 328 |
| Battle Win: | 39035 |
| Battle Drawn: | 38047 |
| Combat: | 39035 |
| Death: | 39880 |
| New Birth: | 10208 |
| Trading: | 87040 |
| Reproduction: | 20176 |

OK

***FIGURE 5-8: Simulation results after 200 and 300 cycles with example 2***

Summary Table

Statistics Information

| | |
|---|---|
| Cycle: | 200 |
| Agent: | 301 |
| Battle Win: | 39048 |
| Battle Drawn: | 39164 |
| Combat: | 39048 |
| Death: | 39907 |
| New Birth: | 10208 |
| Trading: | 93048 |
| Reproduction: | 21016 |

OK

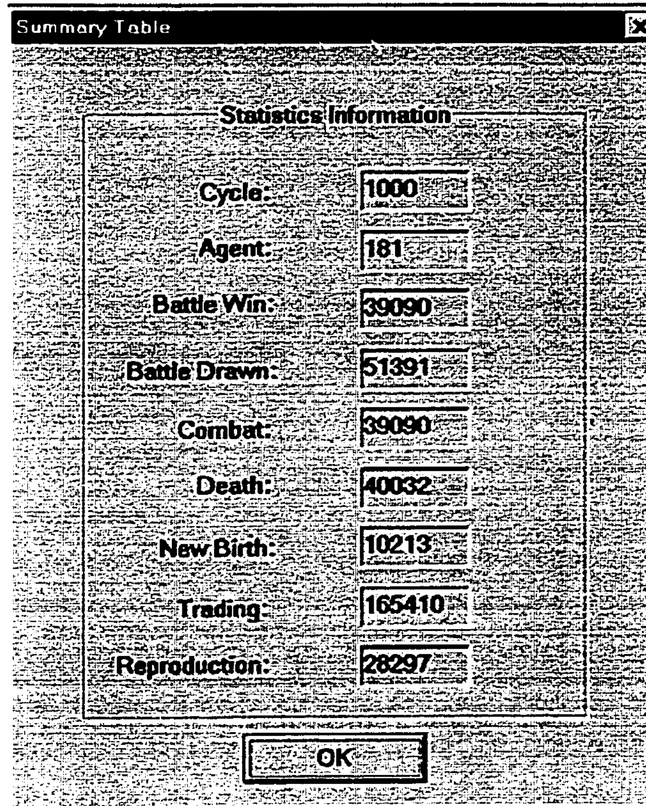
Summary Table

Statistics Information

| | |
|---|---|
| Cycle: | 300 |
| Agent: | 274 |
| Battle Win: | 39059 |
| Battle Drawn: | 41149 |
| Combat: | 39059 |
| Death: | 39934 |
| New Birth: | 10208 |
| Trading: | 104461 |
| Reproduction: | 22433 |

OK

*FIGURE 5-9: Simulation results after 1000 cycles with example 2*




From above figures we see that after cycle 300, trading occurred frequently. Fighting and reproduction did not happen too much. The occurrence of new birth was much lower than that of reproduction. The reason could be that since tax rate had been increased, agents did not have that much extra resources for child agents. But because the surviving agents were very strong (hard to die during fighting), the agent number decreased very slowly.

Similarly, we can tell the genomes of the surviving agents after cycle 300:

- The surviving agents had short "attack" genes compared to their "defend" genes
- The agents had long "trade" genes compared to their "attack" genes.

- The agents did not attract each other too much because their "lust" gene and "beauty" gene did not match enough.
- The agents were "poor". They paid a lot of their resources to the environment so they did not have enough resources for the child agents during reproduction.

# 6 Conclusion

Running the two examples above demonstrates that the implementation of Agent Simulation, the Graphical user interface and the Help system, runs well and meets all our requirements. It offers user a convenient and efficient way to observe the interactions among agents. The results are interesting and worth to be learned.

## 6.1 Experience on Object-oriented Programming

Traditional programming languages separated data from functionality. Typically, data was aggregated into structures that then were passed among various functions that created, read, altered, and otherwise managed that data.

C++, as an Object-oriented language, is concerned with the creation, management, and manipulation of objects. An object encapsulates data and the methods used to manipulate that data.

Object-oriented programming offers a new and powerful model for writing computer software. It speeds the development of new programs, improves the maintenance, reusability, and modifiability of software. Object-oriented programming focuses on the creation and manipulation of objects, such as agents, agent lists, cells, environments. This type of programming gives us a greater level of abstraction; we, the programmers can concentrate on how the objects interact without having to focus on the details of the implementation of

the object.

## 6.2 Further work

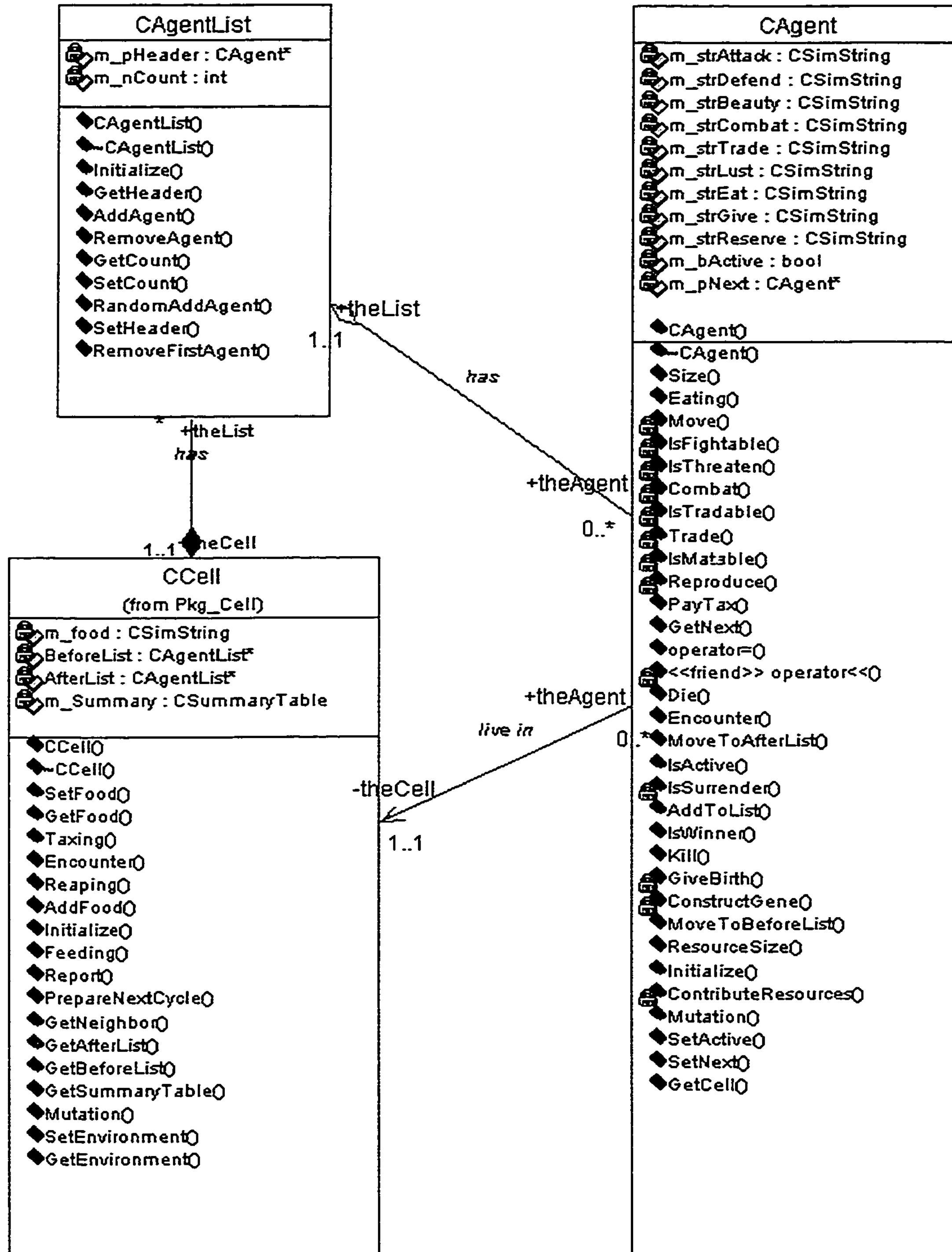
In this project we present the design and implementation of the Agent Simulation System with Object-oriented methodology and Microsoft Visual C++ toolkits. One of the limitations of the system is that it only provides the results (as numbers) of the whole environment. If user wants to learn the statistics on a cell basis, he/she has to read the result file. Therefore we suggest the following further works:

- The user will be able to select simulation results of any individual cell from the interface.
- The simulation results can be shown as graphical charts or curves, which are more challenging but more user friendly.
- Tools will be provided to analyze the simulation results for user.

# A. Bibliography

1. Peter Grogono. The Cell Simulation version 0.1, Department of Computer Science, Concordia University, 1999
2. Peter T. Hraber, Terry Jones, Stephanie Forrest. The Ecology of Echo. Massachusetts Institute of Technology, 1997
3. Peter Grogono. Evolving Agents, An outline, Department of Computer Science, Concordia University, 1999
4. Kate Gregory. Special Edition Using Visual C++ 5.QUE, 1997.
5. Weidong Sun. Cell Simulation Using Object-Oriented Methodology, master's major report, Department of Computer Science, Concordia University, 2000.

## B. Class diagram 1



## C. Class diagram 2