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®

SIMULATION OF OBJECT-ORIENTED TRUCKIN' UNDER WINDOWS NT

BAOSHUO CHEN

A MAJOR REPORT
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

MARCH 2000

© BAOSHUO CHEN, 2000

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
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.

0-612-47842-4

Canada

Abstract

Simulation of Object-Oriented Truckin' Under Windows NT

Baoshuo Chen

The framework of object-oriented truckin' is intended for developing programs that adapt to their environment. The competition of different trucks using different strategies constitutes an adaptive program.

In this report, we present a strategy used in this object-oriented truckin' framework, and an intuitive human-computer interface built to show the status of current situation of different trucks and the result of the competition. The truckin' simulator is implemented using the object-oriented language C++.

Important features of the simulator are the use of the object-oriented methodology to design a truckin' simulator and the use of MFC (Microsoft Foundation Class) to construct an interface for simulating the competition of trucks under Windows NT.

Acknowledgments

I would like to express my sincere gratitude to my major report supervisor, Dr. Peter Grogono. His consistent guidance and enthusiastic support during the development of this major report made the whole work a pleasant and extremely educational experience.

I am also grateful to the people who provided comments, corrections, critiques, and criticisms. These individuals include Meng Cai, Louis Harvey, Jun Zhao and Honglang Li.

Finally, I wish to thank my wife and my parents for all their encouragement and support.

Contents

List of Figures

Chapter 1

Introduction

1.1 Adaptive Programming

Adaptive Programming is viewed as a major advance in software technology based on genetic programming which can evolve solutions to difficult problems for which the answer is not obvious. This is done by the way of a fitness function. The fitness function rates the performance of a possible solution. Good solutions are combined with other good solutions to hopefully create even better solutions. A genetic program starts with a set of functions, and continually combines the good functions, replacing the bad functions with the newly created ones. By this process, the genetic program can evolve a solution.

1.2 Object-Oriented Truckin'

Object-Oriented Truckin' (OOT) is a framework for developing programs that adapt to their environment. The goal of OOT is to provide an environment that is sufficiently complex to provide interesting behavior and yet simple enough to achieve such behavior with modest programming effort.

OOT models a country (OOTLand) in which commodities are distributed by trucks. Trucks negotiate with dealers to buy and sell commodities. Competitors write code that determines the actions of a truck according to some rules; the winner is the competitor

whose truck has the capital grow fastest during a certain competition time. So we could obtain a variety of truck strategies that can be used to build an adaptive program. Although the evolution of truck strategies is the primary rationale, the game can also be seen as a competition between trucks and dealers. Dealers can evolve strategies to maximize their profits, just as trucks can. Running the simulation for an extended period of time should yield trucks and dealers of increasing sophistication.

1.2.1 Features of OOTLand

1.2.1.1 Topography

OOTLand is a square country with a grid of *highways*. *Avenues* run north/south and *streets* run east/west. All highways allow traffic to travel in both directions.

All events take place at intersections of the grid. Currently we set 10 avenues, 10 streets, and 100 intersections in "Constant.h".

A *place* is determined by two coordinates: an avenue number and a street number.

A *step* is a move between adjacent intersections.

1.2.1.2 Commodities

OOTLand contains a number of *commodities* that are traded. Three of the commodities — NONE, GARBAGE, and GAS — are treated specially, as described in the next section; the other commodities are all treated in the same way and differ only in price and quantities available.

1.2.1.3 Dealers

At each highway intersection, there is a *dealer* who trades in a particular commodity. As mentioned above, three commodities have special dealers.

- A dealer who trades in NONE will not buy or sell anything.

- A dealer who trades in GARBAGE will accept any amount of any commodity (except GAS, which is hazardous waste) and will pay a nominal amount corresponding to its scrap value. (Garbage is not implemented in the current version.)

- A dealer who trades in GAS is called a *gas-station*. A gas-station sells gas but does not buy it. Quantities are unlimited: gas-stations do not run out of gas.

For all other commodities, the dealer sets a buying price, B, and a selling price, S. In most cases, dealers will set $B<S$ in order to profit by trading. Prices vary across the country, however. For example, a mineral might be cheap in the country but expensive in the city. Conversely, a manufactured item might be cheaper in the city than in the country. For a given commodity, it should be possible to find dealers x and y such that $B_x>S_y$. Consequently, trucks can profit by buying from y and selling to x.

1.2.1.4 Trucks

Trucks travel around the country trading with dealers. At the start of the simulation, each truck has a certain amount of money (its *capital*), and a certain amount of gas. The truck attempts to increase its capital by trading. At the end of the simulation, the winner is the truck with the most capital. (Complete scoring criteria are given in Section 1.2.4.)

A truck can obtain information, travel along the highways, and trade. Each of these activities consumes resources: time, money, and gas.

1.2.2 Features of the Simulation

The simulation program models the country and its features, as described above, and includes a number of instances of classes derived from the base classes Truck and Dealer. These instances are referred to as "trucks" and "dealers".

Trucks and dealers do not have direct access to the data structures representing OOTLand. To prevent cheating, all of their actions are mediated by two other classes. Associated with every truck, there is an instance of class Control (instances are "controllers"); trucks obtain information and perform actions by sending messages to their controller. Similarly, there is an instance of class Manager (instances are "managers") associated with each dealer, and dealers can send messages only to their managers. The controllers and managers ensure that all actions are consistent with the rules of the simulation.

1.2.2.1 Time

The simulation time increases in steps of 10 minutes. The 10-minute intervals are called *time slots* or simply *slots*.

1.2.2.2 Referee

The *referee* is in charge of the simulation. The referee sends a message to each truck and each manager at the beginning of each time slot. The referee also informs each controller of the current simulation time.

1.2.2.3 Map

The unique object of class Map represents the highway system. At each highway intersection, there is a manager and a dealer. Many of the dealers are "default" dealers who trade only NONE. Trucks cannot access the map directly but can obtain information

about it from their controllers. Similarly, dealers can obtain information from their managers.

1.2.2.4 Controllers

The referee passes the simulation time to each controller at the beginning of each time slot. All other message to a controller come from its truck. The services provided by the controller on behalf of the truck are outlined here.

- A truck can obtain the time since the simulation started, the time remaining in the current slot, and the simulation time remaining (get_time()). It can also obtain its current position (get_place()), its current capital (get_capital()), its current stock of each commodity (get_stock()), and information about the dealer at the current position (get_info()). All of this information is provided without cost to the truck.

- A truck can make a telephone call to another intersection to obtain the commodity, buying price, and selling price of the dealer there (phone_inf()). A telephone call *fails* if the commodity traded at the intersection is NONE and *succeeds* otherwise. A successful telephone inquiry lasts 3 minutes and cost $3; an unsuccessful inquiry lasts 1 minute and costs $1. (These values are defined in constant.h.)

- A truck can move any number of steps in a single direction (move()). Moving one step consumes 6 minutes and 1 litre of gas (constant.h).

- A truck can attempt to buy (buy()) or sell (sell()) from the dealer at its current position. The dealer must honor its advertised buying and selling prices but is not obliged to exchange the quantity requested. Furthermore, the controller will not permit a transaction that leaves the truck with negative capital.

For example, a truck's request to buy 10 computers might fail either because the dealer is willing to sell only 5 computers (in which case the truck would receive 5 computers) or because the truck cannot afford 10 computers (in which case it would receive no computers).

- A truck can steal stock from a dealer. The theft may remain undiscovered but, with finite probability, the police will detect the theft, return the stock to the dealer, and fine the truck. (This is not implemented in the current version.)

- A truck can ask for the location of the closest dealer in a given kind of stock. For example, a truck that is low on gas might ask for the location of the closest gas-station. (This is not implemented in the current version.)

1.2.2.5 Trucks

At the beginning of each time slot, each controller sends the message play() to its truck. During the time slot, the truck can perform any of the actions provided by the controller interface. If the actions require more than 10 minutes, the truck loses some of the next time slot. For example, a truck may choose to travel two steps, which requires 12 minutes. The journey would require the entire current time slot and 2 minutes from the next time slot.

The controller ignores requests from a truck that does not have the resources required. For example, a truck that has exhausted its capital is not allowed to spend money; a truck that has run out of gas cannot travel; and a truck that has used up its time slot cannot do anything that consumes time. It is the responsibility of the truck to ensure that it has the resources required to perform a task and to check that its request achieved the desired effect.

1.2.2.6 Managers

Each manager monitors the action of a dealer. Managers receive messages from controllers. Managers ensure that dealers trade honestly and maintain positive capital and stock.

1.2.2.7 Dealers

A dealer is given an initial buying price, selling price, and stock. A dealer can change the buying and selling prices during the simulation. For example, a dealer with excessive stock might raise its buying price and lower its selling price.

Dealers may also obtain information from other parts of the country and use it to set their prices. (This is not implemented in the current version.)

1.2.3 Units and Constants

The information given in this section is subject to change. Code should always use defined constants rather than the values given here.

OOTLand has 10 avenues and 10 streets with intersections 10 kilometers apart. Trucks drive at 100 km/h and consume gas at the rate of 10 liters per 100 kilometers. They therefore require 6 minutes and 1 liter of gas to travel between intersections. The truck starts with a full tank of gas containing 50 liters.

Money is measured in cents. Each truck starts with 50,000 cents ($500.00). Gas-stations may set their own prices, but a typical price for gas would be $1/litre.

1.2.4 Rules and Scoring

The actions of trucks and dealers are restricted by the interfaces of controllers and managers. In addition, certain constants are accessible to trucks and dealers. The precise rules are:

- A truck class (e.g. MyTruck) must be derived from the class Truck or from a class derived from class Truck. The header file mytruck.h must #include the file truck.h (or the header file of the class derived from class Truck) and no other simulation classes. The implementation file mytruck.cpp must #include the file mytruck.h and no other simulation classes. Both mytruck.h and mytruck.cpp may #include standard C++ library classes.

The simulation runs for a certain time that is announced at the beginning of the run and can be obtained by a truck. When this time has elapsed, the simulation is stopped and the assets of each truck and dealer are recorded.

The only asset of a truck is its capital. Commodities on the truck, including gas, have a negative value that is the cost of buying them at the lowest buying price. For example, suppose a truck finishes the game with $50.00, 20 liters of gas, and no other commodities. If the cheapest gas available is $0.75/litre, the assets of the truck are 50 - 20 $\times$ 0.75, or $35.00.

As the end of the game approaches, trucks with a large amount of stock may seek dumps. At a dump, they can sell their stock at garbage prices, thereby avoiding the penalty of completing the game with stock. (Garbage is not implemented in the current version.)

1.3 Motivation

Object-oriented Programming (OOP) offers a new and powerful model for writing computer software. It has brought hope of increased productivity and improved reliability (help solve the software crisis) [2].

Object-oriented analysis (OOA), design (OOD) and programming (OOP) methodology work together to produce a combination that models their problem domains better than similar systems produced by traditional structured techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust and promote greater design and code reuse. OOP requires a major shift in thinking by programmers. Here we apply OOA and OOD to design one kind of truck strategy, and implement it using the object-oriented language C++.

In order to see the competition result clearly, we build an interface to show the status of the current competition among various trucks and dealers in real time. This also allows the user to dynamically interfere with the competition by changing the speed, time, and number of trucks. We use Visual C++ 5.0 to generate a standard Windows interface in Windows NT environment.

1.4 The Structure of This Report

The rest of the report is organized as followed: the second chapter introduces the background of the project. The third chapter describes the design of a truck strategy and the human interface for it. The fourth chapter is about how to implement the truck and the interface. The fifth contains the result of the interface as well as the truck competition. And the last chapter describes what I have learned and some ideas and comments for further development.

Chapter 2

Background

2.1 Object-Oriented Design

2.1.1 Introduction to the Object-Oriented Paradigm

Object-oriented technology is more than a way of programming, it is a way of thinking abstractly about a problem using real world concepts, rather than computer concepts. It provides a practical, productive way to develop high quality software for many applications. The term **object-oriented** means that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This is in contrast to conventional programming in which data structure and behavior are only loosely connected [8].

"An **object** has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable" [1]. That is, an object is anything to which a concept applies, and a concept is an idea or notion we share that applies to certain objects in our awareness.

Object-oriented programming has three main features [1]:

- "Encapsulation" is the process of hiding all of the details of an object that do not contribute to its essential characteristics.

- "Inheritance" is a relationship between classes where one class is the parent (base/superclass/ancestor/etc.) class of another. Inheritance provides for code and structural reuse. All non-private routines and structure available in the superclass are available to all subclasses. Inheritance is a natural way to model the world or a domain of discourse, and so provides a natural model for OOA and OOD (and even OOP).

- "Polymorphism" is a concept in type theory according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass, thus, any object denoted by this name is able to respond to some common set to operations in different ways.

The difference between viewing software in traditional, structured terms and viewing it from an object-oriented perspective can be summarized by a twist on a well-known quote:

Ask not what you can do to your data structure,
but rather ask what your data structures can do for you.

2.1.2 Object-Oriented Analysis and Design

OOA and OOD stand for **Object-Oriented Analysis** and **Object-Oriented Design**, respectively. OOA is the challenge of understanding the problem, and then the system's responsibilities in that light. To us, analysis is the study of a problem, leading to a specification of externally observable behavior; a complete, consistent, and feasible statement of what is needed; a coverage of both functional and non-functional operational characteristics (e.g. reliability, availability, performance).

OOD is the practice of taking a specification of externally available behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details. Unlike structured design, the process of OOD is neither top-down nor bottom-up; rather it can be best described as round-trip gestalt design, which emphasizes the incremental and iterative development of a system. A process of the OOD is described by Booch as follows [1]:

- The first step in the process of the design involves the identification of the classes and objects at a given level of abstraction and the invention of important mechanisms.

- The second step involves the identification of the semantics of these classes and objects; the important activity in this step is for the developer to act as a detached outsider, viewing each class from the perspective of its interface.

- The third step involves the identification of the relationships among these classes and objects; in this step, the ways in which things interact within the system are established, with regard to the static as well as the dynamic semantics of the key abstractions and important mechanisms.

- The fourth step involves the implementation of these classes and objects; the important activities in this step involve choosing a representation for each class and object, and allocating classes and objects to modules, and programs to processes. This step is not necessarily the last step, for its completion usually requires that we repeat the entire process, but at a lower level of abstraction.

2.2 Windows NT

In order to build the human-computer interface for this project, we choose Windows NT as our platform as Windows system offers some very desirable benefits.

Windows allows all applications run on a simulated "desktop", and each application runs in its own window. A user can easily switch from one application to another. Since several programs can be active at one time under Windows, Windows has to determine which application a given input is destined for.

The nature of the interface between a user and a Windows application is such that a range of different inputs is possible at any given time. A user may key some data, select any of a number of menu options, or click the mouse somewhere in the application window. These user actions are all regarded by Windows as **Events.** Windows records every event as a message and places the message in a message queue belonging to the program for which the message is intended. Each message will typically result in a particular piece of the program code being executed. How program execution proceeds is therefore determined by the sequence of user actions. Programs that operate in this way are referred to as **event-driven programs** [13]. A well-designed Windows application has to be prepared to deal with any type of input at any time, because there is no way of knowing in advance which type of input is going to occur. This sort of program structure can be represented as Figure 2.1.

Each event handler such as "Process Keyboard Input" in the illustration represents a piece of code written specifically to deal with a particular event. Although the program may appear to be somewhat fragmented, the primary factor welding the program into a whole is Windows itself. We can think of our Windows program as customizing Windows to provide a particular set of capabilities.

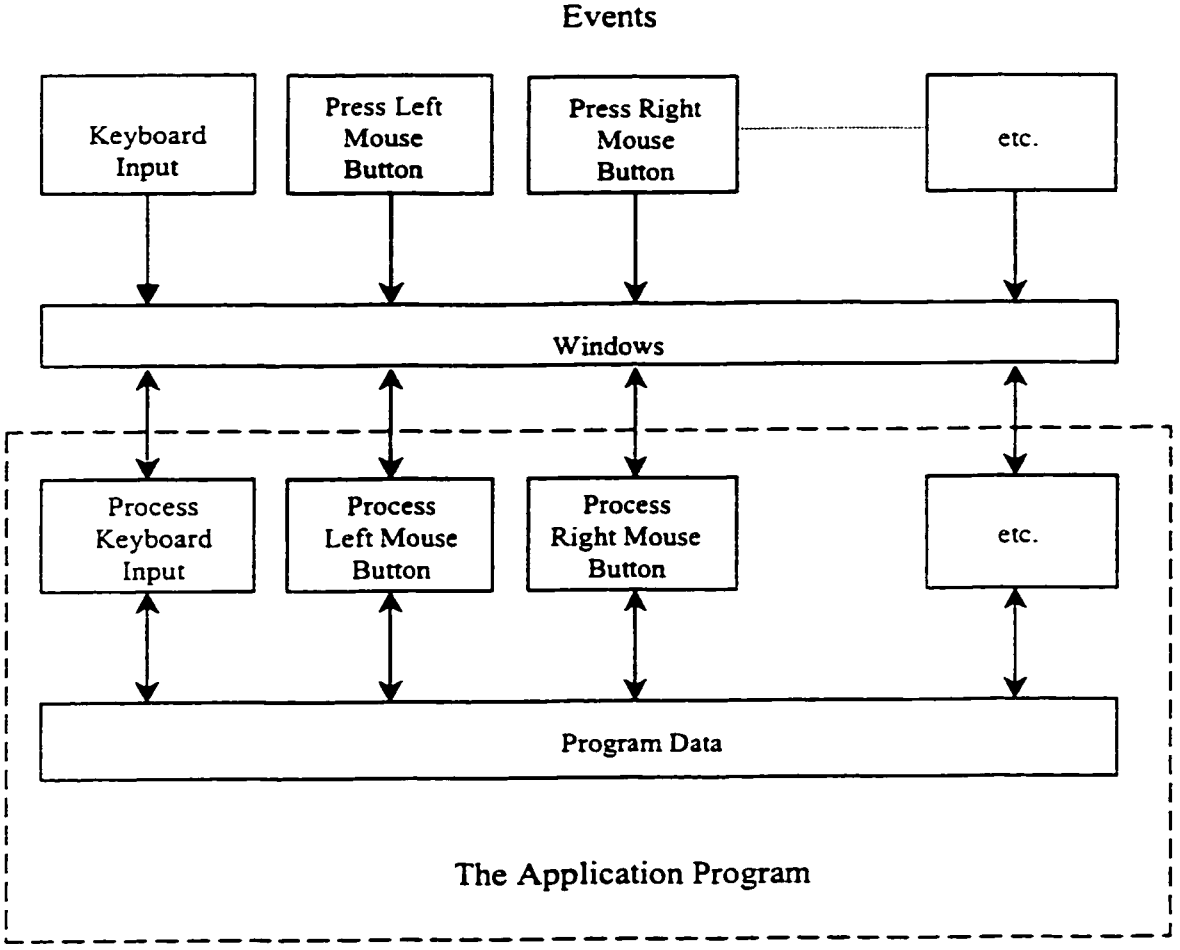

Figure 2.1 Windows Scheme

There are about 200 messages, which fall into four broad categories: Windows messages, control notifications, command messages and control messages. Users can also define their own messages. These messages process all the events that might arise during the running of an application.

Each of the messages represents an event: some action by the user or change of state within the system to which the application may wish to respond. It is up to the programmer to decide which of the many possible events that each object is going to respond to. If there is to be a response, then a piece of code must be written to carry out the required action. Not every message needs to be processed. We can filter out those that

are of interest in the program, deal with them in whatever way we want, and pass them back to Windows.

2.3 Programming Windows

2.3.1 Windows API and MFC

All of the communications between a Windows application and Windows itself use the Windows application-programming interface, otherwise known as the **Windows API** [12]. This consists of literally hundreds of functions that are provided as standard with Windows to be used by applications. It covers all aspects of the dialogue necessary between Windows and user applications. Because there is such a large number of functions, using them in the raw can be very difficult — just understanding what they all are is a task in itself. This is where MFC comes in.

MFC packages the Windows API in an object-oriented manner, and provides an easier way to use the interface with more default functionality. This takes the form of the Microsoft Foundation Classes, **MFC**. It provides an abstraction layer on which a developer can write a Windows application without needing to know the details of the native Windows API. Since MFC is a C++ object-oriented class library, the developer can then override the default behavior of the messages of interest by supplying a new method in the derived class. This vastly increases the readability of the source code and decreases the time it takes to write handlers for messages.

MFC, as figure 2.2 shows is an object-oriented encapsulation of the Windows API.

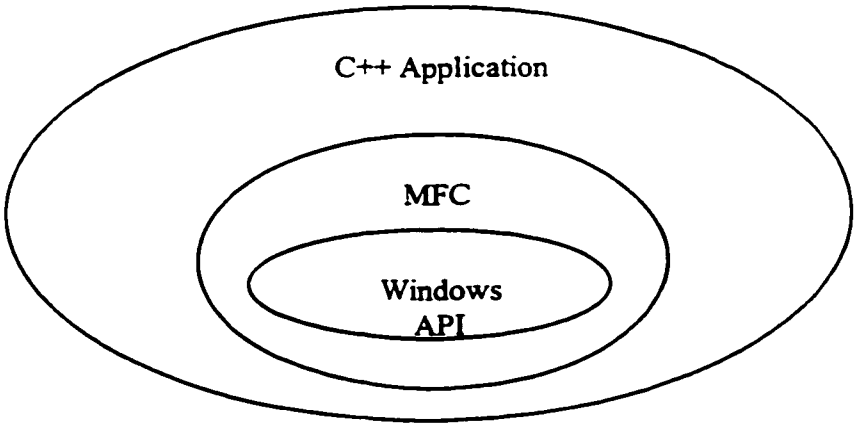

Figure 2.2 MFC Encapsulation of Windows API

By using MFC, the user can create a fully operational Windows application program with very few lines of code. However, it is quite hard to learn MFC. MFC has its own set of rules and requirements in addition to those of Windows.

2.3.2 MFC Structure and Framework

When writing application using MFC, it implies acceptance of a specific structure for the program, with application data being stored and processed in a particular way. The structure of an MFC program incorporates two application-oriented entities: a document and a view.

2.3.2.1 Document

A **document** is the name given to the collection of data in the application with which the user interacts. The term 'document' is just a convenient label for the application data in the program, treated as a unit.

The document class will be derived from the class **CDocument** in the MFC library, and add the data members to store items that the application requires, and member functions to support processing of that data. Handling application data in this way enables standard

mechanisms to be provided within MFC for managing a collection of application data as a unit and for storing and retrieving.

2.3.2.2 Document Interface

The designers have a choice as to whether the program deals with just one document at a time, or with several. The **Single Document Interface**, referred to as **SDI**, is supported by the MFC library for programs that only require one document to be open at a time. For programs needing several documents to be open at one time, you use the **Multiple Document Interface**, which is usually referred to as **MDI**.

2.3.2.3 View

A **view** always relates to a particular document object. As introduced above, a document contain of application data in the program, and a view is an object which provides a mechanism to display some or all of the data stored in a document. It defines how the data is to be displayed in a window and how the user can interact with it. Similar to the way of defining a document, a view class is derived from the MFC class **CView**. Note that a view object and the window in which it is displayed are distinct. The window in which a view appears is called a **frame window**. A view is actually displayed in its own window that exactly fills the client area of a frame window.

2.3.2.4 Relationship Among a Document, a View and a Frame Window

In Figure 2.3, the view displays only part of the data contained in the document, although a view can display all of the data in a document if that is what is required.

A document object can have multiple view objects associated with it. Each object can provide a different presentation or subset of the same document data.

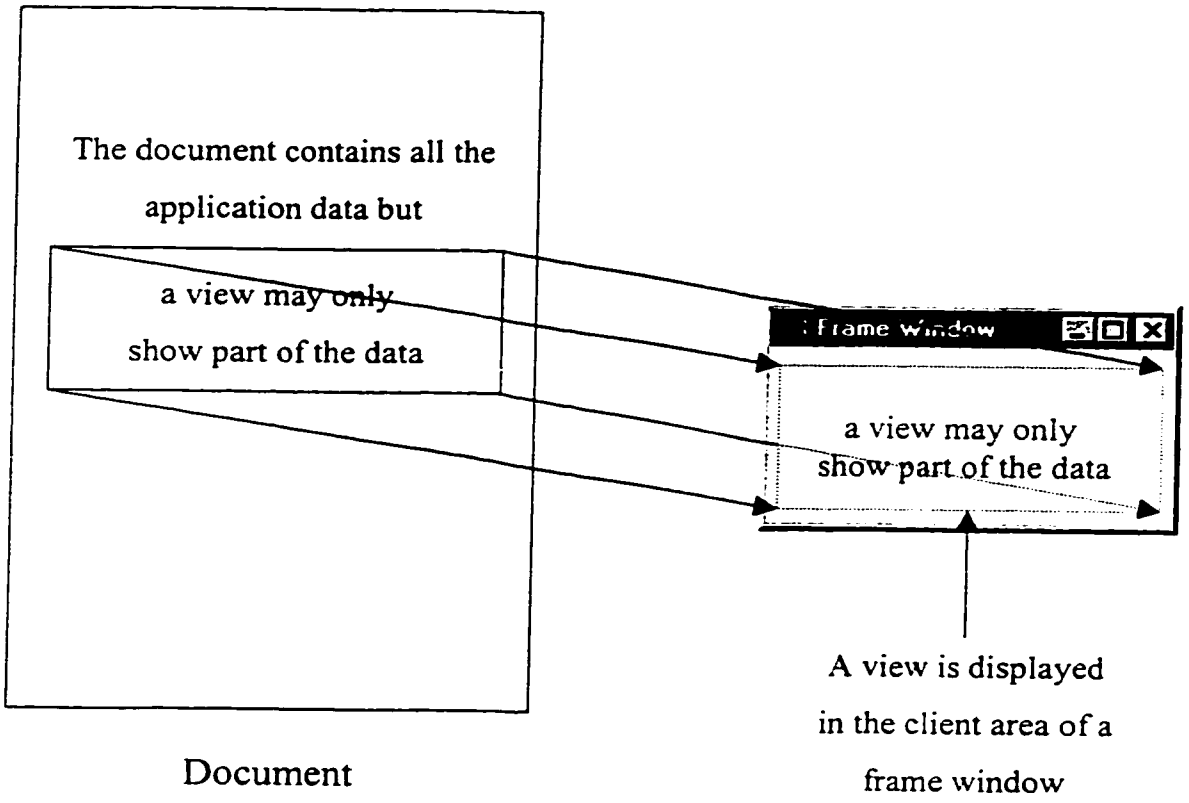

Figure 2.3 Relationship between Document, View and Frame

2.3.2.5 Linking a Document and Its Views

MFC incorporates a mechanism for integrating a document with its views, and each frame window with a currently active view. A document object automatically maintains a list of pointers to its associated views, and a view object has a data member holding a pointer to the document that it relates to. Also, each frame window stores a pointer to the currently active view object. The coordination between a document, a view and a frame window is established by another MFC class of objects called document templates.

Document Templates

A **document template** manages the document objects in the program, as well as the windows and views associated with each of them. There will be a document template for

each different kind of document in the program. If there are two or more documents of the same type, one document template is needed to manage them. To be more specific about the use of a document template, document objects and frame window objects are created by a document template object. A view is created by a frame window object. The document template object itself is created by the application object that is fundamental to any MFC application. A graphical representation of these interrelationships is shown below:

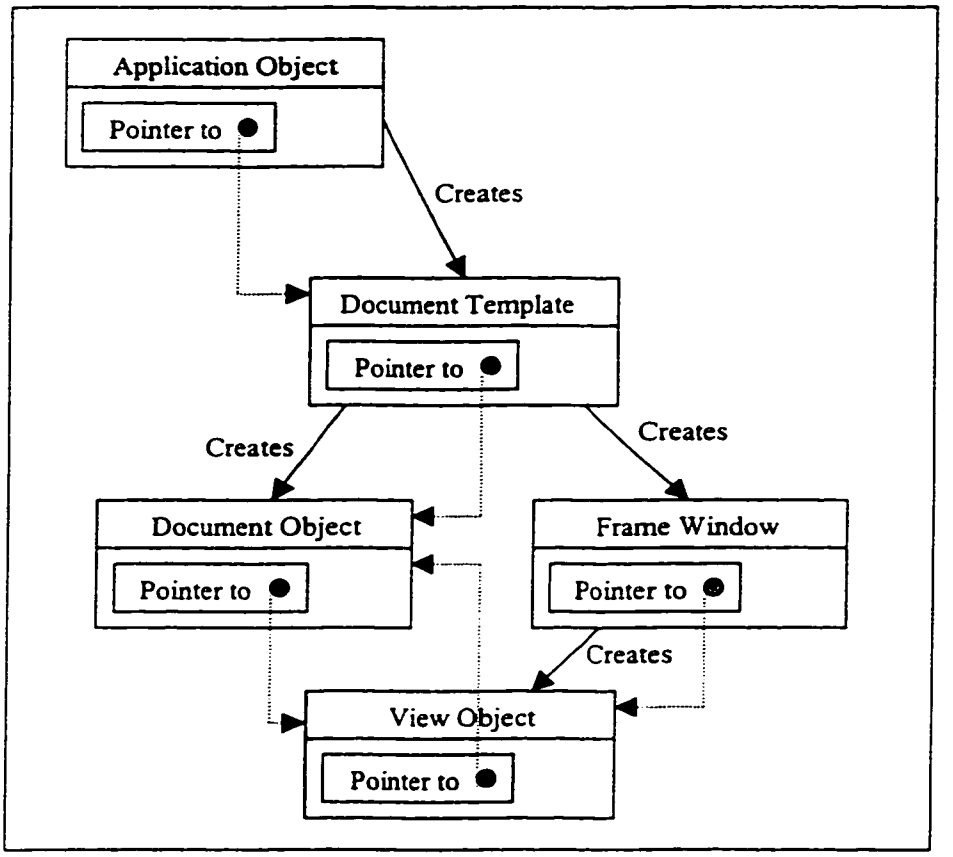

Figure 2.4 Relationship between CView, CDocument, CFrame, CDocTemplate

The diagram uses dashed arrows to show how pointers are used to relate objects. These pointers enable function members of one class object to access the **public** data or the function members in the interface of another object.

Document Template Classes

MFC has two classes for defining document templates. For SDI application, the MFC library class **CSingleDocTemplate** is used. This is relatively straightforward, since a SDI application will have only one document and usually just one view. MDI applications are rather more complicated. They have multiple documents active at one time, so a different class, **CMultiDocTemplate,** is needed to define the document template.

The Application and MFC

MFC covers a lot of ground and involves a lot of classes. It provides classes that, taken together are a complete framework for the applications, only requiring the customization necessary to make the programs do what it should do.

Chapter 3

Design

3.1 Design a Truck Strategy

3.1.1 Specification

As with the competition rules, the truck which earns the most money within the competition time will win. If a truck always makes a wise choice of the highest profit/time ratio, it has good chance to win. The profit/time ratio should be defined as follows. The trucks keep buying commodities from one dealer and selling to another to make money, and they can not be considered to be making money before they successfully sell the commodities. So the total time is calculated from when the trucks leave the current place until they finish selling the commodities, and the total profit they make is the total money they own after they sell the commodities minus the original money they owned.

To make the choice we mention earlier, the trucks need to keep information about the city, including which corner has the dealer, what the commodity the dealer is dealing with, what are the selling price and the buying price of the dealer. Every time the trucks

want to decide the next step, they search their information and decided on the best choices.

In making the decision, we should consider three things. The first is the remaining gas. If the gas is not enough, the truck can not finish the game, which is what we do not want to see. So the trucks keep track of the locations of gas stations so that they can always find out the nearest gas station no matter where they are located. In making the plan, the trucks not only consider the profit they will make; they check the remaining gas. If current gas is not enough for the truck to finish a plan (going to a dealer to buy something and then sell to another dealer) and to go to the nearest gas station after that, it needs to refill. It could be possible to refill before going to any dealer or after buying something but before selling the commodities or after selling the commodities.

The second is that according to the rules, at the end of the game, trucks with the stock will dump and the stock price would be very low since it would be seen as garbage, the only asset of the truck is the capital. We do not want to buy much stock when the time is almost finished and use up the capital. So when we consider the deal, we should also consider the remaining time. When the time is not enough to sell the stock, we do not do anything.

The third is that when the competition is running, the dealer would probably change the price without notifying the trucks, then the truck could hold the incorrect information. In order to maintain the right information, the truck should have some way to update its information periodically. One way is that every time when it passes a dealer, it gets the available information. However this has some problems. If it gets the information only from this source, it may only get the information from a set of dealers. So that is not the best choice. What we do is to keep the dealers' information, if some dealer has passed the time limit it implies that the information is out-of-date. To solve this problem, we

randomly make a phone to the dealer chosen from such dealers or from one that has never been contacted.

3.1.2 Identifying Classes and Responsibility

Here we make four classes: ChenDealerInfo, Pair, Dealer_list, and ChenTruck.

ChenDealerInfo

Here the truck should have the knowledge of every dealer, it should have a data structure to keep the message. This data structure is for the current truck to remember the information of a dealer. It also needs to verify whether the information is valid or not.

Class ChenDealerInfo	
Keep the dealer's information Check whether the information is valid	

Figure 3.1 Class ChenDealerInfo CRC Card

Pair

When the truck find a pair of dealers which sell the same commodity, it could calculate whether it is possible for it to earn some money by buying some commodities from one and then selling to another.

Class Pair	
Get the possible earn rate (money/time)	

Figure 3.2 Class Pair CRC Card

Dealer_list

In competition, every time the truck wants to make a deal, it should check with its knowledge. If it wants to find a pair of dealers that sell the same commodities, it should

search the whole dealers set which takes time. Here we add this class which stores the addresses of all known dealers dealing with the same commodities. Then in order to find a pair, it just searches in one list instead of all dealers. To maintain this class, it should have some way to add new dealer to it.

Class Dealer_list	
Add dealer to the link list	

Figure 3.3 Class Dealer_list CRC Card

ChenTruck

This class is the main class that we want to build. It should keep the information of the dealers it knows as well as the link lists of dealers selling same commodities. It should also determine which two pair dealers are selected to make a deal, what is the possible earning rate of this deal. It could do something according to the decision such as calling the dealer, going to the seller, or to the buyer, or even to a gas station, how to reach a destination. It has someway to update its information periodically.

Class: ChenTruck Base: Truck	
Decide what to do next	Pair, ChenDealerInfo, Dealer_list, Control
Make phone to inquire the dealer for information	Place, ChenDealerInfo, Control
Update the specific dealer's information	Place, ChenDealerInfo, Dealer_list, Control
Calculate the possible earn rate for a pair of dealers	Pair, Place, Control, ChenDealerInfo
Go to the destinate dealer to buy commodities	Place, Control
Go to the destinate dealer to sell	Place, Control

commodities Decide the route to go to a destination	Place, Control

Figure 3.4 Class ChenTruck CRC Card

3.1.3 Class State Transition

From the above figure we could see that classes ChenDealerInfo, Pair, Dealer_list are mainly for holding the information for the class ChenTruck uses. What we should think of is how the ChenTruck works. Below is the figure shows the state transition of the truck.

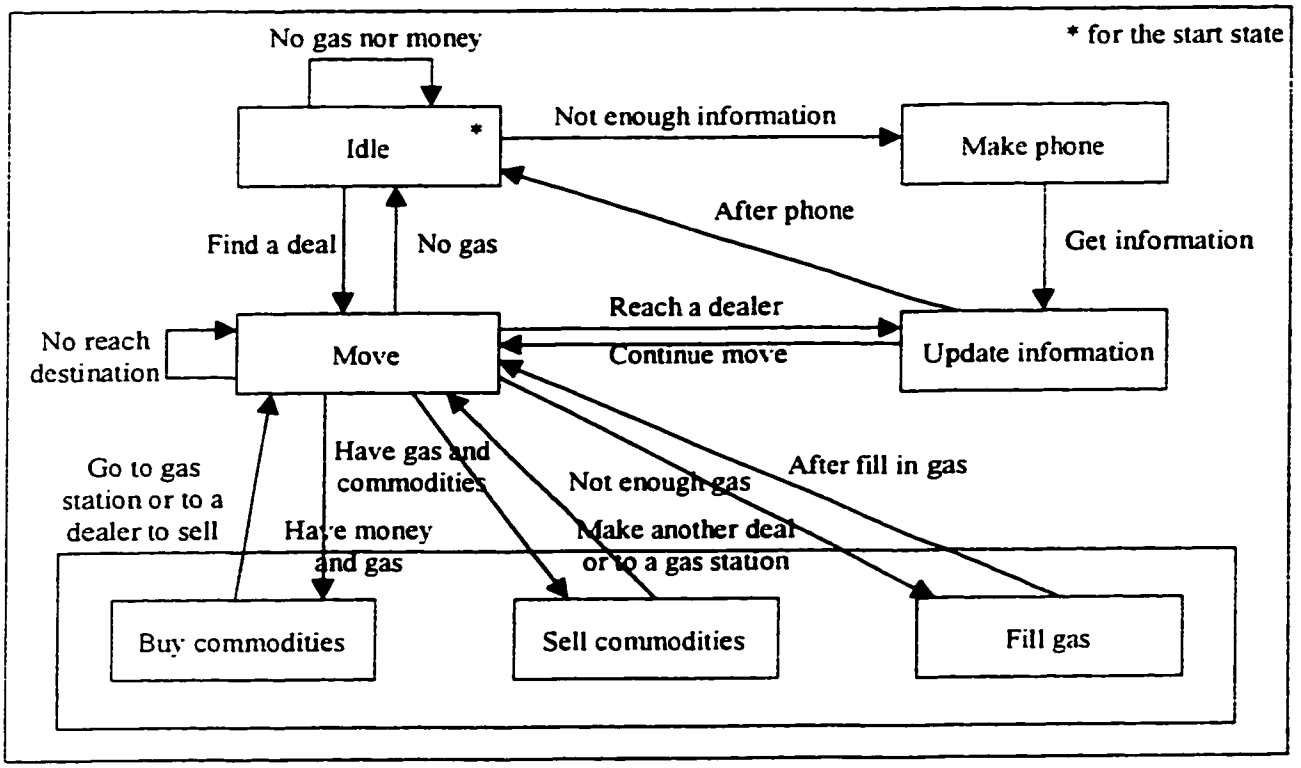

Figure 3.5 ChenTruck State Transition

3.2 User Interface Design

3.2.1 Interface Layout

As we will build the interface on Windows, we want to have our application similar to the common Windows applications. We need to put our result into a window with necessary menus and toolbars. In the client area of the window we display the data with the method we wanted.

In order to demonstrate the simulation result in real time and graphically, we need to have a frame that shows the map of the city with each dealer's location as well as what they are dealing with. When in competition, the trucks run in this map and their location are shown. The user can figure out what the trucks are dealing with from the trucks' current locations. Another thing we want to see is the total property, the remaining gas of each truck at any time and the total property for each dealer. So each truck or dealer should have its own window.

We created an application with the necessary menus and toolbars. In the client site, we built four subwindows, each just display some kind of information we have pointed out in the last paragraph. These subwindows should fill the entire client area and can be resized depending on what we are interested in most at a given time. At the beginning of the simulation, we want to see the whole map of the city in the simulation window, as well as each dealer in the right position. Each kind of dealer should be denoted by a special kind of icon. The other three windows will show nothing except some rulers. The figure below shows what we want to have our application look like:

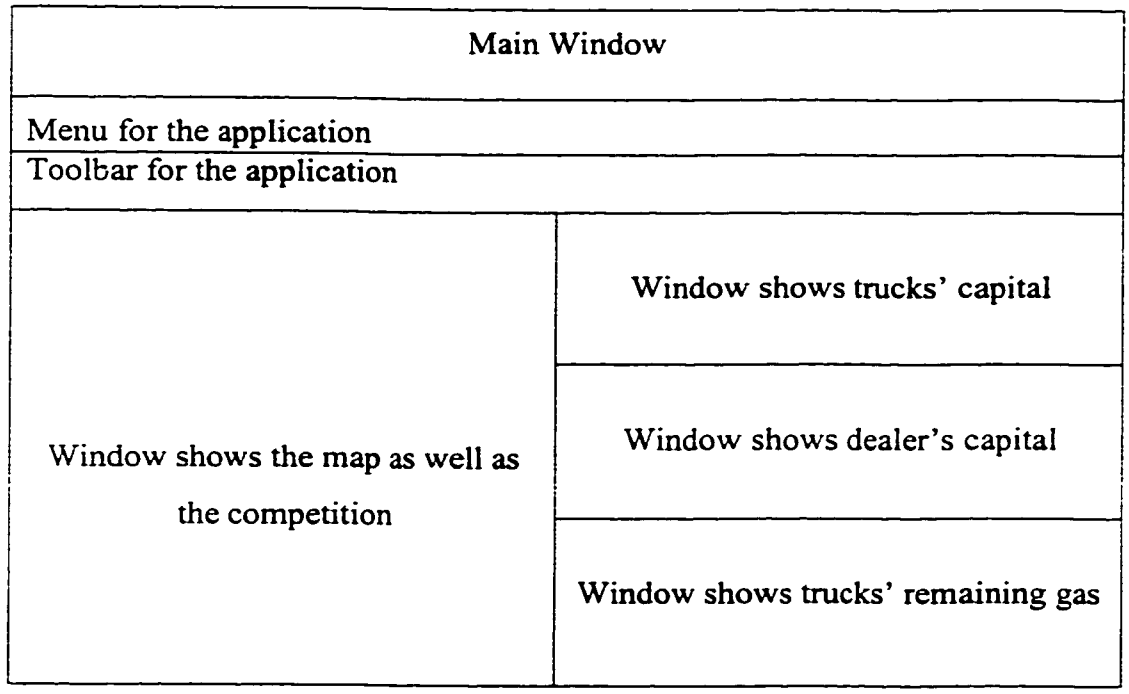

Figure 3.6 Interface Layout

3.2.2 Document and View

As we have introduced earlier, in MFC the code is separated into two parts, document and view of which the document holds the data the application needs and the view draws the document data in some ways. So we need to store our data into a class CSimulationDoc which is based on CDocument. In this class, we should handle all the competition as well as offer the information that the view needs such as the trucks' positions, the capital of the truck as well as the dealers' information. For the view site, we build four classes: CSimulationView, CTruckCapitalView, CTruckGasView, and CDealerCapitalView. Besides that, we still need to build two classes CSimulationApp and CMainFrame.

Class CSimulationDoc Base: CDocument	
According to the truck name create a kind of truck	ChenTruck, other trucks
Add a truck to the data	Control
Remove a truck from the data	Control
Get the control for a specific truck	Control
Get current time	
Get total simulation time	
Set total simulation time	
Get the map for the competition	
Set the map as well as the dealers in the map	Map, manager, dealer
Get the specific color for the truck	Control
Get the current capital for the truck	Control
Get current place for the truck	Control
Get total trucks number	
Begin simulation Pause simulation Stop simulation	

Figure 3.7 Class CSimulationDoc CRC Card

Here we need to consider one thing, as the time goes on, the curve of each information will also evolve. As the window is not large enough to hold the curves, we can scroll the window to review the previous curve. So here is the problem, each the system status, we could get the information that we need and draw it on the screen, but with the time going

on, the data was discarded so to refresh the screen becomes impossible. As a result, we should have someway to keep track of the history so that we could reuse when we redraw the view. Here we should add a data member to the dealer to hold the history of its properties and the associated functions. And another data member to the control class to hold the history of the truck capital and gas for all the information of truck is held by the control class.

Class CSimulationView Base: CView	
Get trucks' number	CSimulationDoc
Get trucks' color	CSimulationDoc
Get trucks' position	CSimulationDoc
Draw the city	
Draw the dealers in the city	
Draw the truck at the current place	
Resize and scroll the window	

Figure 3.8 Class CSimulationView CRC Card

Class CTruckCapitalView Base: CView	
Get current trucks' capital	CSimulationDoc
Get the previous trucks' capital	CSimulationDoc
Draw the truck capital at some point	
Resize and scroll the window	

Figure 3.9 Class CTruckCapitalView CRC Card

Class CTruckGasView Base: CView	
Get current trucks' gas	CSimulationDoc
Get the previous trucks' gas	
Draw the truck gas at some point	CSimulationDoc
Resize and scroll the window	

Figure 3.10 Class CTruckGasView CRC Card

Class CDealerCapitalView Base: CView	
Get current dealers' capital	CSimulationDoc
Get the previous dealers' capital	CSimulationDoc
Draw the dealer capital at some point	
Resize and scroll the window	

Figure 3.11 Class CDealerCapitalView CRC Card

Class CSimulationApp Base: CWinApp	
Create document template Start the application	

Figure 3.12 Class CSimulationApp CRC Card

Class CMainFrame Base: CFrameWnd	
Create menu and toolbar Create multiple views Enable and disable some menus and toolbars Response the menu and toolbar command Arrange the subwindow layout	 CSimulationDoc CSimulationDoc

Figure 3.13 Class CMainFrame CRC Card

3.2.3 Dialogs

Besides showing the result, we also need some way to communicate with the application. In Windows, user input is usually accomplished through dialog box.

Here our competition uses the default map with all the dealers pre-set in a map file. Since we want our program to be used in random situation, the number of dealers and positions as well as the commodities the dealer deal with could be assigned. So we should be able to respond to any request of changing map by clearing the original dealers and redrawing

the new one. We could use the MFC class CFileDialog to get the file name and open the file.

To allow this application to be used in simulating many different kinds of trucks, we need some way to randomly add the truck as well as setting the trucks' locations. We need to have a dialog CAddTruckDlg that shows the possible trucks what we may use, and the possible position to set. As there could be many trucks running, it is better to use different colors to show the truck, it is also necessary to have some way to choose the color in this dialog.

Class CAddTruckDlg Base: CDialog	
Select the truck name from the possible trucks selection	CComboBox
Get the possible address of the new truck	
Select the color for the truck	CColorDialog

Figure 3.14 Class CAddTruckDlg CRC Card

Besides adding a truck to the competition, we possibly want to pause a truck or even delete a truck from the competition at run time. Thus another dialog CDeleteTruckDlg is added to show all available trucks as well as each truck's address and color.

Class CDeleteTruckDlg Base: CDialog	
Select the truck from the available trucks	CComboBox, CSimulationDoc
Show the selected truck address	CSimulationDoc
Show the selected truck color	CSimulationDoc

Figure 3.15 Class CDeleteTruckDlg CRC Card

Before starting a competition, we may also want to modify the total competition time instead of the default time, so that we could see the results with different competition time. Here we add another dialog CSetTimeDlg.

Class CSetTimeDlg Base: CDialog	
Show the current simulation time Get the new simulation time	CSimulationDoc

Figure 3.16 Class CSetTimeDlg CRC Card

Because our program is showing the competition, and if the speed is too fast, the user could not catch up with what is going on. So we add another dialog CSetSpeedDlg that could set the clock speed, that even in the very fast machine we could also let it slow down.

Class CSetSpeedDlg Base: CDialog	
Show the default slowest speed Set the new speed	

Figure 3.17 Class CSetSpeedDlg CRC Card

Chapter 4

Implementation

4.1 Implementing the Truck

In MFC class, all objects should be based on the class CObject. CObject is the root base class for most of the Microsoft Foundation Class Library (MFC). The CObject class contains many useful features that we may want to incorporate into our own program objects, including serialization support, run-time class information, and object diagnostic output. Each class derived from CObject is associated with a CRuntimeClass structure that we can use to obtain information about an object or its base class at run time. The ability to determine the class of an object at run time is useful when extra type checking of function arguments is needed, or when we must write special-purpose code based on the class of an object.

As our classes are data members of class CSimulationDoc, CSimulationView, CTruckCapitalView, CTruckGasView, and CDealerCapitalView, which are dynamically created at run time, they also need to be created dynamically and must be derived from CObject class.

As we have mentioned before, we have to keep the information of each dealer. Class ChenDealerInfo holds the following information about a dealer: The commodity with which the dealer is dealing; current address; how many commodities it has; the buying

price and the selling price this dealer is asking; the nearest gas station within current knowledge and the distance to there; the last time we check this dealer holding the same price; approximate time period of this dealer keep the same price.

Class Pair stores two pointers which point to the dealers dealing the same commodities, the possible earning/time ratio for a given truck, and a flag indicates the steps suppose the truck selects this pair of dealers. The flag can indicate the truck to go to the gas station then to the seller or to the seller first then go to a gas station or even finish the business first then go to a gas station. All these choices are determined by the current truck situations.

The Dealer_list manages the linked list for the dealers who dealing with the same commodities. It has a pointer to the dealer as well as a link to the next Dealer_list.

Class ChenTruck should keep the information of the total dealers in the city as well as the dealers linked list which dealing with the same commodities. Because the calculation of the possible earn rate requires the information of current truck's position and gas, so we put the calculation function in class ChenTruck instead of in class Pair.

As we have mentioned the data structure we use, here we only outline some important functions. The most important thing is decision-making, it decides what to do next, to buy something or to sell something or to go to a gas station, and determines which dealer it should visit. At the beginning of the game, as the truck knows nothing about the city, it makes phone to get some information and refresh its database. After getting enough information, the truck searches its database from one kind of Dealer_list to another. For each Dealer_list, it compares the prices of any two dealers to see whether it is possible to buy from one and sell to another, and calculates what is the possible earn rate and the step of making the deal based on the current situation. From the database, it chooses the deal with the highest earn rate.

The steps to calculate the earning rate of a pair of dealers are as follows. First, get the current remaining gas. If the gas is not enough to the nearest gas station, it will run out of gas and stop at some place, so it will lose the game for sure, and there's nothing we can do. Then, calculate the possible distance we should go, if there is not much time left and we have not sold out all the commodities, the distance is that to a dealer. Otherwise, the distance is that from the current location to the location of the seller, plus the distance from the seller to the buyer. If the gas is not enough to make the deal, it must refill the gas sometime before the deal is finished. So depending on the gas remaining and the dealers' nearest gas station, we could set the flag from 1 to 5. '1' means first going to the current place's nearest gas station then to seller. '2' means first to seller's nearest gas station then to seller and then to buyer. '3' means first go to seller then to seller's nearest gas station then to buyer. '4' means first to seller then to buyer's nearest gas station then to buyer. '5' means first to here's nearest gas station then to buyer.

The third is to update the information, when the truck makes a phone or when it passes a place, it could get the latest information of that place. If this place is new to the truck, it may get the information by calling the 'control' get_info or phone_info function. If this is a regular dealer, we need to find out where is the nearest gas station. If this is a gas station, we will check all the founded dealers whether this new gas station is closer than the nearest gas station it had found before.

The source code for these functions can be found in Appendix C.

4.2 Implementing the Human Interface

In order to build the human interface using Visual C++, we can use Visual C++'s excellent wizards to create an MFC program, that would automatically creating a skeleton application upon which we can build our own specific application.

4.2.1 Build Multiple Views

As we have mentioned earlier, MFC has two kinds of structures, SDI and MDI. SDI uses only one kind of document and MDI has multiple documents with each one having its own window. Here we simply use one document CSimulationDoc to hold all the data, but we have multiple view that shows the data in different ways. So for our application we use SDI.

In order to create multiple windows, we add two intermediate windows in the CMainFrame's client site. The first one is 'm_Views' that is a splitter window and uses CMainFrame's client site as its parent window so that this window will employ the entire client site. In this window, we create two panels, the left side is for the simulation view and the right site is for all the views that show the information curves that change with the time. The benefit for this arrangement is that we could resize the competition view and the other views, and as we will mention later, the other views are very similar so that the resize of the window does not affect the viewing result. The second one is 'm_CapitalViews' which is also a splitter window. This window uses the right part of 'm_Views' as its parent window. We create three panels for this part and these three show CTruckCapitalView, CTruckGasView, and CDealerCapitalView respectively, and these three windows could also be resized within the parent window so that we could focus on the view we are interested in most at any time. Now we have built the four views, but how to show the information we hold is still unknown, we will introduce that next.

4.2.2 How to Draw the Simulation View

For the CSimulationView we have to get all the information about the city from the CSimulationDoc first, then we draw the lines to compose the whole city, then we place the associated icons in the position for each dealer.

When we add a truck, as the graph is small we could not draw much detail, we place a small circle filled with the truck color. In competition, we get the truck place from the CSimulationDoc and draw it in the right place.

Here we still have one problem. When the competition begins, the truck stops only when a competition slot is finish. During this period, we don't know the locations of the trucks. How do we interrupt the competition so that we could draw the positions in the middle? Here we add a timer that starts when the competition begins, and the timer generates an interrupt every time unit. When the time is up, we let the competition to continue one time unit and then wait. So every time we could get the truck position and draw the truck.

4.2.3 How to Draw Other Views

The other three views are similar. Here we just mention the window that draws the truck properties. First we draw the time ruler as well as the ruler for the properties. Then we set the point using the specific color to draw a point in the position based on the time and the current properties.

Just as we have mention before, we need to keep old information. The problem is how to store the information. If we just store its value in every time unit then we need huge amount of memory, and even the total time could be very large and then exceed the system memory. In our program, we use another way, because the properties only change when a deal is making, we just need to keep the time when the deal is made and the properties after dealing. When we need to draw the curve, we can get the value from the latest previous change time.

One more thing needs to be mentioned is the outline of these windows. As we want to see so many windows at the same time, each window may not occupy a large space. For the x-coordinate we could use the scrollbar to show the different time of curve, while the problem is in the y-coordinate which shows the total value of the properties. When a

truck is being simulated, the truck's capital is usually between 0 to 1000 and a dealer's capital is between 0 to 2000. So an enlarged scale is used for these ranges to get a better view. For the truck capital exceeding 1000 and dealer capital exceeding 2000, as it is rare, a reduced scale is used. We set the maximum truck capital to be 6000. If any time any truck's capital exceeds this maximum value, we use its real value for simulation but the maximum value for drawing. The same situation applies to the dealer capital except the maximum value is 12000.

4.2.4 How to Eliminate the Flash

Just doing this is not enough, when the simulation is running, the window will flash even if we slow down the competition.

After analyzing the MFC, we notice that is because the default way of drawing the window is to invalidate the whole window and then to redraw it but the computer speed is not fast enough. So here we just invalidate part of window. For example, in simulation window, we invalidate the previous truck address and restore the original color, and then draw the truck in the new position. This solves the flashing problem.

4.2.5 How to Build Dialogs

As we have analyzed before, we have to implement four dialogs: CAddTruckDlg, CSetSpeedDlg, CSetTimeDlg, and CDeleteTruckDlg. Based on the Visual C++, we could build our dialog on the following step. First we create the dialog box resource using the Visual C++ dialog box editor. This process defines the appearance of the dialog box as well as the controls that appear in the dialog box and the types of data those controls will return to the program. The second step is to write a dialog box class derived from CDialog for our dialog box, including member variables for storing the dialog box's data. Initialize the member variables in the class's constructor. The third, overload the DoDataExchange() function in our dialog box class, in the function, call the appropriate

DDX and DDV functions to perform data transfer and validation. Besides these basic steps, we need to consider some callback functions when user has done something such as change the default value, modify the scrollbar. In the CAddTruckDlg, we have more than one way to set the truck position, so that we may respond in either way to set the right value.

4.2.6 Enable and Disable Menu Items and Toolbars

The last thing we need to mention is how the menu commands and toolbars look to the user when a pop-up menu and toolbar are displayed. When the program is running, some commands are disabled at some time. For instance, before the simulation begins, if there is no truck in the city, there is no way to start the simulation. If the simulation has not started, the pause and stop commands have no meaning. So here we need to add some controls to determine when these commands should be enabled or disabled.

Use ClassWizard to connect a menu or toolbar to a command-update handler in a command-target object. It will automatically connect the menu or toolbar's ID to the ON_UPDATE_COMMAND_UI macro and create a handler in the object that will handle the update. Before displaying the menu or toolbar's commands, MFC calls each of the update-command-UI function associated with the commands. Thus we could enable or disable the menu or toolbar.

4.3 Overview of Code

We summarize the main member functions along with a brief description in associated classes.

ChenDealerInfo class

- ChenDealerInfo: Initializes a new ChenDealerInfo class with the given parameter.
- Valid: According to previous knowledge, check whether the information we keep is still valid.
- Friend class: ChenTruck, Pair.

Pair class

- Pair: Initializes a new Pair class with the given parameter.
- = : Create a new class equal to the given class.
- SetInitialize: Reset the value of Pair class.
- GetPossibleEarnRate: Get the possible earn/time ratio.
- Friend class: ChenTruck.

Dealer_list class

- Dealer_list: Create a Dealer_list object and initialize the element with a ChenDealerInfo object.
- AddToList: Create a new Dealer_list and link it to the end of current link list.

ChenTruck class

- ChenTruck: Create a new class, initialize with the given id number and the controller, as well as current place's information.
- play: Get current time, if still have slot time left, keep on trading.
- trade: According to the current plan, call the associate function to finish it.
- look_for_deal: Check each pair of dealers dealing the same commodities and find out a pair that will get the highest earn/time ratio if take this pair to deal with.
- make_phone: Randomly select a place and make phone call to update the information.

- go_buy_it/go_sell_it: Going to a dealer to buy or sell commodities; according to the choice to see whether should go to the gas station if necessary.
- go_to: Select the next step which can go to the destination as well as refresh more places' information.
- update_info: Update the designated place's information; find out the nearest gas station and the distance from that if that place is a regular dealer, otherwise reconsider all the dealers' nearest gas station.
- Possible_earning_rate: For a given pair of dealers, calculate the possible earn/time ratio. First calculate the distance between this pair, according to the remaining gas to see whether need to go to the gas station, if necessary, set the flag and recalculate the distance. See how much money it can spend to buy the commodity and how much it could earn, the time needed to finish all these steps, and then get the possible earn/time ratio.

CSimulationApp class

Important override member function:

- InitInstance: Initialize an instance of current application. Including registration the application's document templates and add to the application; parse command line for standard shell commands; dispatch commands specified on the command line.

CMainFrame class

Important override member function:

- CMainFrame: Create an object and initialize the class member.
- OnCreate: The framework calls this member function when an application requests that the Windows window be created by calling the Create or CreateEx member

function. In this function we first call the parent window's function, then create the toolbar and status bar and set the right status.

- OnCreateClient: Called by the framework during the execution of OnCreate. The default implementation of this function creates a CView object from the information provided in pContext, but here we need to create the four views. So we first create an splitter window which separate the client site into left and right parts, the left site create the CSimulationView and the right site create another splitter window, this time create three views which are CTruckCapitalView, CDealerCapitalView and CTruckGasView respectively.
- OnTimer: Response the message that system generate at every simulation time unit. This function check the state of competition first, if it is in running, it allow the trucks to run one time unit more.

Message callback functions:

- OnOptionSetSimulateSpeed: Response the command Option/SetSimulateSpeed that invokes a CSetSpeedDlg dialog and set the new simulation speed.
- OnToolBarSetSpeed: Response the toolbar button "Set Speed". This function calls OnOptionSetSimulateSpeed.
- OnRunPlay: Response the command Run/Play that starts the simulation and install a system timer.
- OnToolBarPlay: Response the toolbar button "Play". This function calls OnRunPlay.
- OnUpdateRunPlay: Called by framework before the menu "Run/Play" display. We enable or disable the menu depend on the current situation.
- OnUpdateToolBarPlay: Called by framework before the toolbar button "Play" displayed. Implementation is same as OnUpdateRunPlay.

CSimulationDoc class

- CSimulationDoc: Create an object and initialize the member variable.
- ~CSimulationDoc: Remove the trucks and controls array.
- CreateTruck: According to the truck name, call the corresponding truck constructor.
- AddATruck: According to the truck name, call the "CreateTruck" function to create a new truck and add to an array.
- DeleteATruck: Delete a truck from the truck array according to the truck ID.
- PauseATruck: Pause a truck during competition.
- GetControl: Return the specific truck's control.
- IsPlay: Check whether it is in running status.
- IsPause: Check whether it is in pause mode.
- IsLoaded: Check whether the map and the dealers' information have been loaded.
- GetCurrentTime/SetCurrentTime: Get/set current simulation time.
- GetTotalTime: Get the total simulation time.
- GetTrucksNumber: Get total trucks number.
- GetMap: Get the map for current simulation usage.
- Play: Invokes trucks to run until the specific time.
- LoadDealers: Get the dealers' information from a file.
- SetPlayFlag/SetStopFlag: Set the flag to indicate the playing or stopping mode.
- GetTruckColor: Get the specific truck's color.
- GetTruckCurrentCapital: Get the truck's capital at a specific time.
- GetTruckCurrentPlace: Get current truck address.
- GetTruckOldPlace: Get the truck address at the previous time unit.

Message callback functions:

- OnOptionChangeFile: Response the "Option/Change File" command. This function sets the new dealers' information from the user selected file.

- OnOptionSetSimulationTime: Response the "Option/Set Simulation Time" command which set the total simulation time.
- OnRunPause: Response the "Run/Pause" command which pause the simulation.
- OnRunStop: Response the "Run/Stop" command which stop the simulation.
- OnRunAddATruck: Response the "Run/Add A Truck" command which invoke the CAddATruck dialog to get the new added truck's information and add the new created truck to the truck array.
- OnRunPauseATruck: Response the "Run/Pause A Truck" command which pause a specific truck during competition.
- OnRunDeleteATruck: Response the "Run/Delete A Truck" command to delete a truck from the truck array.
- OnUpdateOptionChangeFile: Called by the framework before "Option/Change File" displayed. Determine whether this command is valid at this time.
- OnUpdateOptionSetSimulationTime: Called by the framework before "Option/Set Simulation Time" displayed. Determine whether this command is valid at this time.
- OnUpdateRunPause: Called by the framework before "Run/pause" displayed. Determine whether this command is valid at this time.
- OnUpdateRunStop: Called by the framework before "Run/Stop" displayed. Determine whether this command is valid at this time.
- OnUpdateRunAddATruck: Called by the framework before "Run/Add A Truck" displayed. Determine whether this command is valid at this time.
- OnUpdateRunPauseATruck: Called by the framework before "Run/Pause A Truck" displayed. Determine whether this command is valid at this time.
- OnUpdateRunDeleteATruck: Called by the framework before "Run/Delete A Truck" displayed. Determine whether this command is valid at this time.
- OnToolBarStop: Response the toolbar button "Stop" which stop the simulation.
- OnToolBarPause: Response the toolbar button "Pause" which pause the simulation.
- OnToolBarAddATruck: Response the toolbar button "Add A Truck" which add a truck to the truck array. Same as "OnRunAddATruck".

- OnToolBarSetTime: Response the toolbar button "Set Time" which set the total simulation time.
- OnUpdateToolBarStop: Called by the framework before toolbar button "Stop" displayed. Determine whether this command is valid at this time.
- OnUpdateToolBarPause: Called by the framework before toolbar button "Pause" displayed. Determine whether this command is valid at this time.
- OnUpdateToolBarAddATruck: Called by the framework before toolbar button "Add A Truck" displayed. Determine whether this command is valid at this time.
- OnUpdateToolBarSetTime: Called by the framework before toolbar button "Set time" displayed. Determine whether this command is valid at this time.

CSimulationView class

- DrawCity: Draw the city on the window.
- DrawTrucks: Using the trucks' color to draw the trucks on their places.
- DrawDealers: Draw the dealers on their place.

Important override member function:

- OnInitialUpdate: Called by the framework after the view is first attached to the document, but before the view is initially displayed. Here we set the initially window size.
- OnDraw: Called by the framework to render an image of the document. Here we call the DrawCity, DrawTrucks and DrawDealers functions to display the current information.
- OnUpdate: Called by the framework after the view's document has been modified. The default implementation invalidates the entire client area, marking it for painting when the next WM_PAINT message is received, here we override this

function to invalidate the previous truck address and the new truck address, so that the window will not flush.

CTruckCapitalView class

- GetTruckCapitalPosition: According to the truck capital and the current time, calculate the x and y coordinate to draw the curve.
- DrawTruckCapitalOuter: Draw the coordinate and scale for the trucks' capital window.
- DrawTrucksAssetHistory: For each time unit, calculate the x and y coordinate and draw the curve.

Important override member function:

- OnInitialUpdate: Initialize the window size.
- OnDraw: Draw the window coordinate and the trucks' capital curves.
- OnUpdate: Calculate the rectangle which to be drawn and invalidate that.

CDealerCapitalView class

- GetDealerCapitalPosition: According to the dealer capital and the current time, calculate the x and y coordinate to draw the curve.
- DrawDealerCapitalOuter: Draw the coordinate and scale for the dealers' capital window.
- DrawDealersAssetHistory: For each time unit, calculate the x and y coordinate and draw the curve.

Important override member function:

- OnInitialUpdate: Initialize the window size.
- OnDraw: Draw the window coordinate and the dealers' capital curves.
- OnUpdate: Calculate the rectangle which to be drawn and invalidate that.

CTruckGasView class

- GetTruckGasPosition: According to the truck gas and the current time, calculate the x and y coordinate to draw the curve.
- DrawTruckGasOuter: Draw the coordinate and scale for the trucks' gas window.
- DrawTrucksGasHistory: For each time unit, calculate the x and y coordinate and draw the curve.

Important override member function:

- OnInitialUpdate: Initialize the window size.
- OnDraw: Draw the window coordinate and the trucks' gas curves.
- OnUpdate: Calculate the rectangle which to be drawn and invalidate that.

CAddTruckDlg class

- CAddTruckDlg: Create an object and initialize the member variable.
- OnInitDialog: This member function is called in response to the WM_INITDIALOG message which is sent to the dialog box immediately before the dialog box is displayed. Here we need to add combobox with the currently supported truck's name.
- OnChangeColor: Response the command of "Color" button. This function invokes a CColorDialog to let user select a color and repaint the window.
- OnPaint: The framework calls this member function when Windows or an application makes a request to repaint a portion of an application's window. Here

we need to use current user selected color to repaint the edit box just below the color button.

- OnChangeEdit1/OnChangeEdit2: Response the user action that may have altered text in edit control for street and avenue respectively.
- OnCloseupCombo1: Response when the user has selected a kind of truck from the combo box. It posts a message WM_PAINT to inquire repaint the window.
- OnDeltaposSpin1/ OnDeltaposSpin2: Response the user action of clicking the spin button right of the street and avenue edit box. It reflects the action result to the edit box.

CDeleteTruckDlg class

- CDeleteTruckDlg: Create an object and initialize the member variable.
- OnInitDialog: Initialize the combobox with the current available trucks' ID.
- OnCloseupCombo2: Response when the user has selected a truck ID from the combo box. It posts a message WM_PAINT to inquire repaint the window.
- OnPaint: Response the WM_PAINT message. It gets the selected truck ID and displays the truck's place as well as the color for it.

CSetSpeedDlg class

- CSetSpeedDlg: Create an object and initialize the member variable.
- OnInitDialog: Setting the slider's range and the initial position.
- OnChangeEdit1: When change the edit box, call this function to set the slider to the corresponding position.
- OnHScroll: When move the slider, change the value in the edit box to the corresponding value.

CSetTimeDlg class

- CSetTimeDlg: Create an object and initialize the member variable.
- SetTime: Set the initial competition time.
- OnChangeEdit1: When user modify the value in the edit box, call this function to update the corresponding member variable.
- OnDeltaposSpin1: When user clicks the spin button, modify the corresponding value in the edit box.

Chapter 5

Results

In this project, we have built a simulation system of OOTLand with a graphical interface. This system has the following features:

- Built under Windows NT
- With the common windows appearance and styles.
- Only enable the command that was allowed running such as "Play" command only enable after user has added a truck.
- Real time display of the competition result.
- Allows the user to dynamically control the simulation.
- Allows the user to review the previous competition result.

Below, we will show what the project looks like and how it works. In figure 5.1 shows the project at the beginning, at this time it just load in the dealers' information from the file 'truckin.dat'. The details of truckin.dat can be found in Appendix A.

We can use the menu "option/change file" to set a new file containing the city information. The other thing we could do is to set the total simulation time by menu "option/set simulation time" or the toolbar to invoke the CSetTimeDlg (Figure 5.2). We may type in the new value or click the spin button to increase or decrease the value. The

value is between 0 to 10000, if the value exceeds this range, a warning message will pop up.

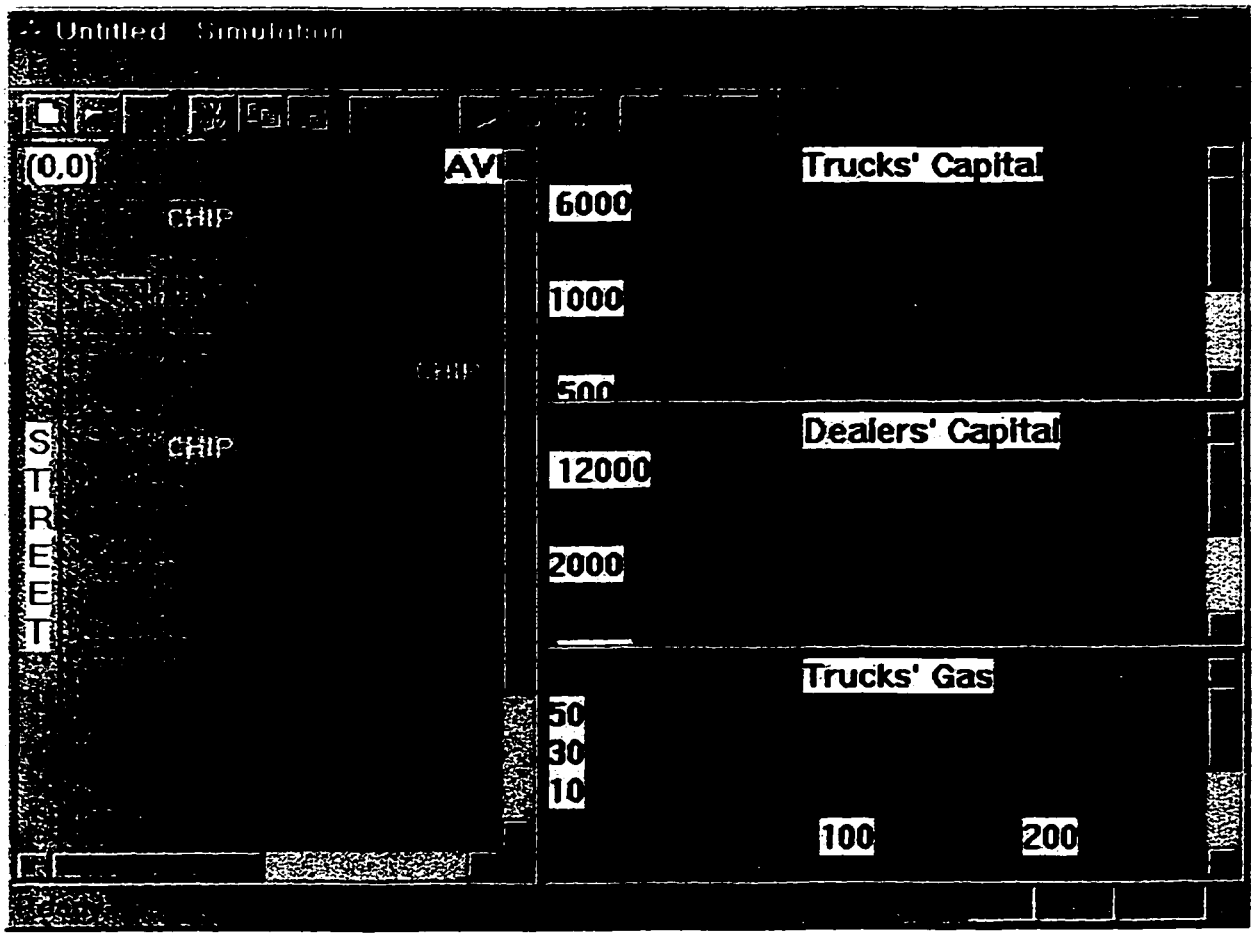

Figure 5.1 Before Simulation

Figure 5.2 CSetTimeDlg

One more thing we can do is to set the simulation speed with the menu "option/set simulation speed" or the toolbar to invoke the CSetSpeedDlg (Figure 5.3). We could type

in the speed from 1 (the slowest) to 10 (the fastest), or we could move the slider bar to set the value.

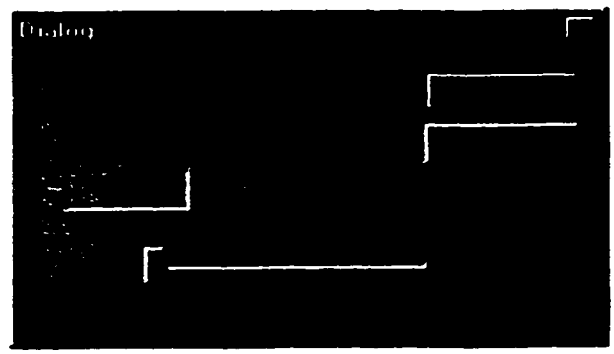

Figure 5.3 CSetSpeedDlg

Here we will see that some of the toolbars are disabled at the beginning, including those corresponds to the menu items "start running", "pause running", "stop running". That is because we can not run the simulation if there is no truck in the city.

In order to start the simulation, we have to add the truck into the city, that was done by menu "run/Add a Truck" to invoke CAddTruckDlg (Figure 5.4) to add a truck. We could set the truck position by typing in or clicking the spin button to increase or decrease the value. We could select the color for drawing the truck by clicking the "color" button, which would invoke another dialog to let you select the color you want and put the result in the field below the "color" button. The primary thing to do is to select a kind of truck, for this application is mainly for compare different kinds of trucks, so here we should be able to see the available trucks and select one. This is done by clicking the combo box and choosing one from it, here the combo box is uneditable. The chosen truck is shown on Figure 5.5.

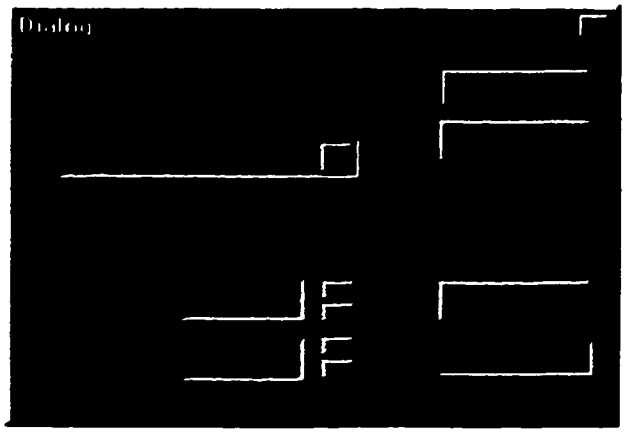

Figure 5.4 CAddTruckDlg

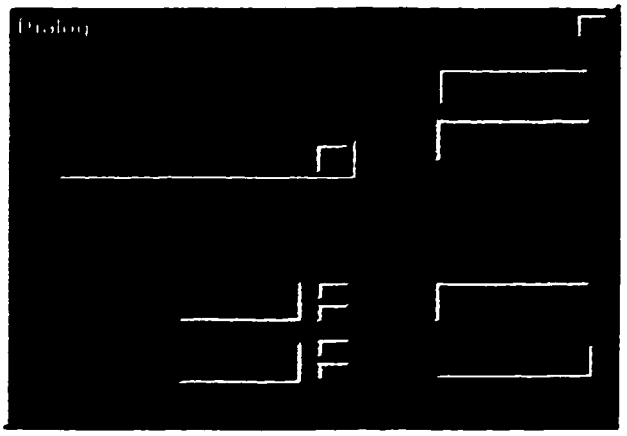

Figure 5.5 Select Truck in CAddTruckDlg

If we are unsatisfied with the selected trucks or the truck position, we could use the "run/delete a truck" to call CDeleteATruck (Figure 5.6) and delete a truck. This dialog will show the current available trucks and the selected truck's color that stands for it and its current position. By clicking the combo box, we could see the number standing for current available trucks. Since a user can not tell which truck will be deleted from its number. So when a truck is selected, the truck information is shown on the dialog including the truck position and the color. If the user selects another one, the information will automatically refresh to reflect current truck.

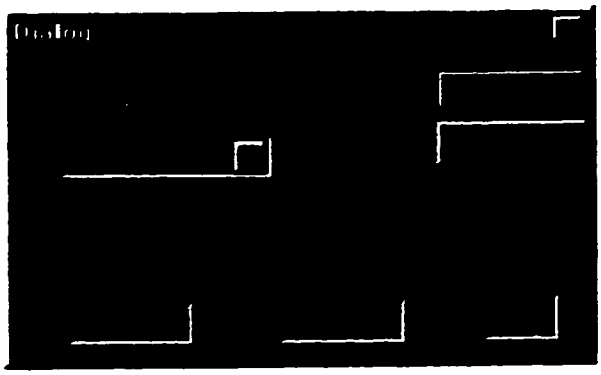

Figure 5.6 CDeleteTruckDlg

Example:

We using the default dealers' information from file truckin.dat (Appendix A) as well as the default simulation time 2000. First we add two ChenTrucks, one is putting in the place (0,0) and the color is black and the other has red color standing for it and put at the place (4,5). Then we begin to run the simulation. After some time of simulation, we pause the simulation to see how these trucks are going on. The result is showed on figure 5.7.

In the simulation window, the black truck has move to place (0,1), and red one does not shown in the picture. In the truck capital window, there are two curves that show the trucks' capital at every time unit. The black curve shows the black truck's capital and the red curve shows the red truck's. You will also see that the curve is extended with time. The same is true for the windows that show the dealer's capital and the trucks' gas. At the end of the competition, we can see that the black truck get the capital of 1022.29$ and its place is (9,5). The red one get the capital of 623.79$ and stop at (3,2).

We have also put the important message to the file so that we could know that when and what the truck or the dealer is doing. The file is in Appendix B (truckin.out).

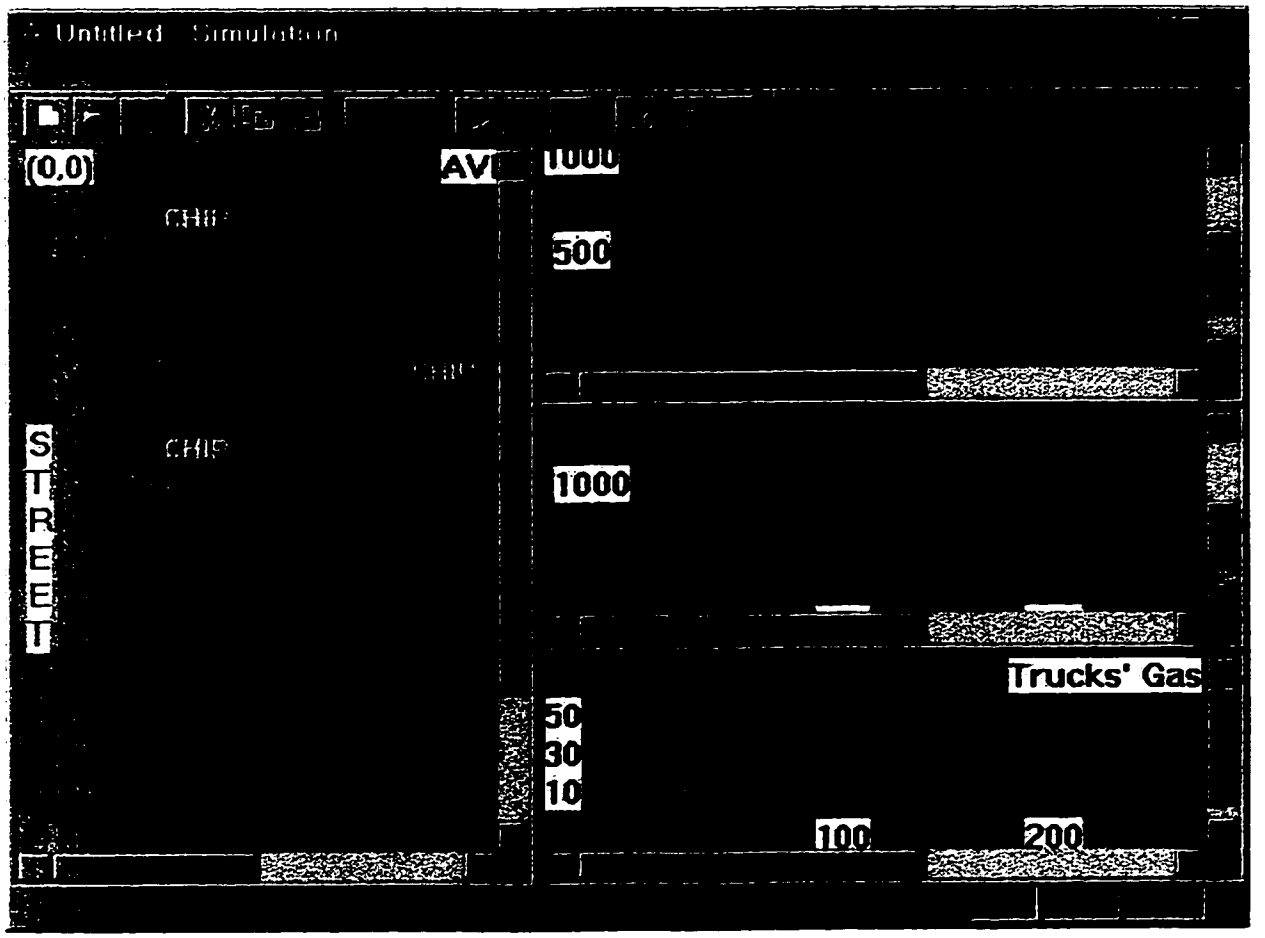

Figure 5.7 Simulation Result

Chapter 6

Conclusions and Further Work

6.1 What I Have Learned

In this application, we have used object-oriented technique to analyze, design the OOTLand simulation and applied object-oriented language C++ to implement the system. The system works in Windows environment and is fully compatible with other Windows application in terms of interface appearance and design rules. So it could be very easy for a user to learn it and use it.

Through the development of this application, I have learned that, compared with the traditional program style, object-oriented has the following advantages:

- In object-oriented design, each class stands for an object in the problem domain. This seems to be natural and appeals to human cognition.
- For the complex systems, integration would be spread out and risks would be reduced if object-oriented technique is used.
- If we want to change something in systems, we can modify only the related classes. Thus systems will change more resilient.
- We can apply the resonant similarity to techniques of thinking about problems in other domains.

- Because of similar systems have almost the same structure, frameworks could be introduced to reuse software and designs.

By studying and using the framework of Visual C++, I have got more detailed feeling of the framework and its power. It is a library of code that contains general functionality found in almost all applications of a similar nature, rather than being devoted to a specific purpose or functionality. The deriving philosophy behind application frameworks is that it is not necessary for a programmer to be constantly "reinventing the wheel". The programmer should not be the millionth person to write a linked-list implementation or a menu class. Rather, the programmer should be able to concentrate on the application-specific coding required. This is the goal of object-oriented.

6.2 Further Work

Although we have built almost everything necessary for the simulation, limitation of the functionality of the application is unavoidable. Here I suggest some directions for future work to contribute to the improvement of the application. The following are some suggestions for future work:

- Support printing the simulation result, this feature may be useful for analyzing the competition result.
- Online help can be added to assist users to learn how to user this system.
- Status bar to be added to show some help messages as well as current status during competition.
- Functions are introduced to support the analysis of the competition result.
- It is better to use the ActiveX technique to implement the system and the trucks so that new trucks can be directly plugged-in without changing the system.

Bibliography

[1] Grady Booch.
Object-Oriented Analysis and Design with Applications.
The Benjamin/Cummings Publishing Company, Inc. 1994.

[2] James Martin and James J. Odell.
Object-Oriented Analysis And Design.
Prentice Hall. 1995.

[3] Grady Booch.
Object-Oriented Design with Applications.
The Benjamin/Cummings Publishing Company, Inc. 1991.

[4] Ronald J. Norman.
Object-Oriented Systems Analysis And Design.
Prentice-Hall, Inc . 1996.

[5] Ivar Jacobson.
Object-Oriented Software Engineering: A Use Case Driven Approach.
Addison-Wesley, 1992.

[6] Timothy Budd.
An Introduction to Object-Oriented Programming.
Addison-Wesley Pub. Co. ISBN: 0201824191 1997.

[7] James P. Cohoon and Jack W. Davidson.
C++ Program Design: An Introduction to Programming and Object-Oriented Design.

Richard D. Irwin Inc. 1997.

[8] James Rumbaugh, Michael Blana, William Premerlani, Frederick Eddy, and William Lorensen.
Object-Oriented Modeling and Design.
Prentice Hall, Inc., 1991.

[9] Rick Decker and Stuart Hirshfield.
The Object Concept: An Introduction to Computer Programming Using C++.
PWS Pub. Co. ISBN: 0534204961 1995.

[10] *Programmer's Guide to Microsoft Windows 95.*
Microsoft Press. 1995.

[11] Stephen Morris.
Object-Oriented Programming Under Windows.
Butterworth Heinemann. 1995.

[12] Clayton Walnum, Paul Robichaux.
Using MFC and ATL,
Que 1997.

[13] Mark Andrews.
Visual C++ Object-Oriented Programming.
SAMS, 1994.

[14] Greg Perry.
Teach Yourself Object-Oriented Programming with Visual C++ 1.5 in 21 Days.
Sams Publishing, 1994.

[15] Mattbew Telles and Andrew Cooke.
Windows 95 API How-To.
Waite Group Inc, 1996.

Appendix A

Truckin.dat

29

1	1	1 30000	-1 105
1	8	1 30000	-1 80
3	3	1 30000	-1 105
3	6	1 30000	-1 85
6	3	1 30000	-1 85
6	6	1 30000	-1 95
8	1	1 30000	-1 80
8	8	1 30000	-1 95

0	3	2 30000	600	700
2	0	2 30000	650	725
3	2	2 30000	700	800
5	8	2 30000	500	550
7	6	2 30000	450	500
9	7	2 30000	350	450

1	0	3 30000	200	250
1	3	3 30000	220	260
4	2	3 30000	270	350

6 9 3 30000 500 570

9 5 3 30000 600 700

2 4 4 30000 200 220

3 1 4 30000 180 200

6 8 4 30000 70 100

8 5 4 30000 90 120

8 7 4 30000 80 100

0 2 5 30000 1000 1050

2 2 5 30000 1100 1200

4 4 5 30000 900 1020

7 9 5 30000 1200 1300

9 4 5 30000 1350 1400

Appendix B

Truckin.out (Part)

```
1 Dl  0  0    3000   -$0.01   -$0.01    $0.00 Set stock to maximum.
1 Dl  0  0    3000   -$0.01   -$0.01  $500.00 Set capital to maximum.
119 Dl 27  5  29968   $12.00   $13.00  $916.00 Sale completed.
128 Tr  0 at (7,9)   0  34   0   0   0  32     $57.00 Purchase completed.
120 Dl 27  5  29968   $12.00   $13.00   $10.00 Set capital to minimum.
120 Dl 27  5  29968   $12.00   $14.30   $10.00 Increased selling price.
124 Dl 13  2  29920    $3.50    $4.50  $860.00 Sale completed.
133 Tr  1 at (9,7)   0  43  80   0   0   0     $54.00 Purchase completed.
125 Dl 13  2  29920    $3.50    $4.50   $10.00 Set capital to minimum.
125 Dl 13  2  29920    $3.50    $4.95   $10.00 Increased selling price.
166 Dl 28  5  30032   $13.50   $14.00   $68.00 Purchase completed.
175 Tr  0 at (9,4)   0  27   0   0   0   0     $489.00 Sell completed.
167 Dl 28  5    500   $13.50   $14.00   $68.00 Set stock to minimum.
167 Dl 28  5    500   $12.15   $14.00   $68.00 Lowered buying price.
213 Dl  9  2  30000    $6.50    $7.25  $500.00 Short of money.
213 Dl  9  2  30076    $6.50    $7.25    $6.00 Purchase completed.
222 Tr  1 at (2,0)   0  29   4   0   0   0     $548.00 Sell completed.
214 Dl  9  2    500    $6.50    $7.25    $6.00 Set stock to minimum.
214 Dl  9  2    500    $6.50    $7.25  $500.00 Set capital to maximum.
214 Dl  9  2    500    $5.85    $7.25  $500.00 Lowered buying price.
```

.....

1730 Dl 18 3 1127 $3.44 $5.97 $3.02 Purchase completed.
1739 Tr 0 at (9,5) 0 4 0 393 0 0 $128.91 Sell completed.
1731 Dl 18 3 1127 $3.44 $5.97 $500.00 Set capital to maximum.
1731 Dl 18 3 1127 $3.09 $5.97 $500.00 Lowered buying price.
1759 Dl 5 1 29900 -$0.01 $0.95 $57.50 Sale completed.
1768 Tr 0 at (6,6) 0 50 0 393 0 0 $81.41 Purchase gas completed.
1773 Dl 14 3 29609 $2.01 $2.37 $500.00 No trade -- reduced profit.
1793 Dl 22 4 29897 $0.96 $1.00 $500.00 No trade -- reduced profit.
1800 Dl 16 3 30000 $1.96 $3.64 $10.00 Short of money.
1800 Dl 16 3 30005 $1.96 $3.64 $0.20 Purchase completed.
1809 Tr 0 at (4,2) 0 44 0 388 0 0 $91.21 Sell completed.
1801 Dl 16 3 500 $1.96 $3.64 $0.20 Set stock to minimum.
1801 Dl 16 3 500 $1.96 $3.64 $500.00 Set capital to maximum.
1801 Dl 16 3 500 $1.76 $3.64 $500.00 Lowered buying price.
1807 Dl 27 5 29968 $13.11 $13.13 $10.00 No trade -- reduced profit.
1810 Dl 23 4 28221 $0.84 $0.93 $500.00 No trade -- reduced profit.
1812 Dl 13 2 29920 $4.20 $4.21 $10.00 No trade -- reduced profit.
1835 Dl 14 3 29609 $2.01 $2.37 $500.00 Short of money.
1835 Dl 14 3 29857 $2.01 $2.37 $1.52 Purchase completed.
1844 Tr 0 at (1,0) 0 39 0 140 0 0 $589.69 Sell completed.
1836 Dl 14 3 29857 $2.01 $2.37 $500.00 Set capital to maximum.
1836 Dl 14 3 29857 $1.80 $2.37 $500.00 Lowered buying price.
1842 Dl 12 2 29747 $4.81 $5.27 $500.00 No trade -- reduced profit.
1854 Dl 28 5 500 $13.04 $13.06 $68.00 No trade -- reduced profit.
1901 Dl 9 2 500 $6.52 $6.53 $500.00 No trade -- reduced profit.
1918 Dl 20 4 2112 $1.28 $1.42 $471.29 No trade -- reduced profit.
1918 Dl 18 3 1267 $3.09 $5.97 $67.40 Purchase completed.

1927 Tr 0 at (9,5) 0 26 0 0 0 0 $1022.29 Sell completed.

1919 Dl 18 3 1267 $2.78 $5.97 $67.40 Lowered buying price.

1928 Dl 24 5 30000 $10.22 $10.23 $500.00 No trade -- reduced profit.

1928 Dl 25 5 30000 $11.47 $11.48 $500.00 No trade -- reduced profit.

1928 Dl 8 2 30000 $6.47 $6.48 $500.00 No trade -- reduced profit.

1928 Dl 15 3 30000 $2.38 $2.39 $500.00 No trade -- reduced profit.

1928 Dl 19 4 30000 $2.08 $2.09 $500.00 No trade -- reduced profit.

1928 Dl 26 5 30000 $9.58 $9.59 $500.00 No trade -- reduced profit.

1928 Dl 11 2 30000 $5.22 $5.23 $500.00 No trade -- reduced profit.

1928 Dl 21 4 30000 $0.83 $0.84 $500.00 No trade -- reduced profit.

1928 Dl 17 3 30000 $5.32 $5.33 $500.00 No trade -- reduced profit.

1938 Dl 10 2 753 $5.39 $6.03 $50.46 No trade -- reduced profit.

Appendix C

Source Code (Part)

```
void ChenTruck::look_for_deal(int& done_it) {
 Pair tmp_choice;
 int i;
 Dealer_list *tmp1,*tmp2;
 ChenDealerInfo *c1,*c2;
 long curr_time, slot_time_left, sim_time_left;

 ctl->get_time(curr_time, slot_time_left, sim_time_left);
 choice.SetInitialize();
 if( commodity[GAS] ==0){
   plan=PHONING; done_it=1;
   return;
 }

 for(i=GAS+1;i<NUM_COMM;i++){
   tmp1=commodity[i];
   while (tmp1 ){
     if( ! tmp1->company->Valid(curr_time) ){
       tmp1=tmp1->next;
       continue;
```

```
 }
 c1=tmp1->company;
 tmp2=tmp1->next;
 while(tmp2){
   if( ! tmp2->company->Valid(curr_time) ){
      tmp2=tmp2->next;
      continue;
   }
   c2=tmp2->company;
   if(c1->buy_price > c2->sell_price ){
       tmp_choice.buyer=c1;
       tmp_choice.seller=c2;
   }else if(c2->buy_price > c1->sell_price ){
       tmp_choice.buyer=c2;
       tmp_choice.seller=c1;
   }else {
      //impossible to make any profit
    tmp2=tmp2->next;
    continue;
   }
   // here has find out the possible pair
   Possible_earning_rate(tmp_choice);
   if( choice.GetPossibleEarnRate() < tmp_choice.GetPossibleEarnRate() ) {
    choice=tmp_choice;
    plan=BUYING; done_it=1;
   }
   tmp2=tmp2->next;
}
tmp_choice.seller=0;
```

```
        tmp_choice.buyer=tmp1->company;
        Possible_earning_rate(tmp_choice);
        if( choice.GetPossibleEarnRate() < tmp_choice.GetPossibleEarnRate() ) {
         choice=tmp_choice;
         plan=SELLING; done_it=1;
        }
        tmp1=tmp1->next;
      }
    }

    if ( choice.GetPossibleEarnRate()==0 ) {
     plan=PHONING ; done_it=1;
    }
  }
```

```
void ChenTruck::Possible_earning_rate(Pair &choice){
 //choose the most earning rate, set the flag_gas > 0 if need to get
 //gas between seller and buyer, if set to
 //1 means first going to here's nearst gas station then to seller,
 //2 means first to seller gas station then to seller
 //3 means first to seller then to seller's gas station
 //4 means first to seller then to buyer's gas station
 //5 means first to here's nearst gas station then to buyer

 const double deduction_rate[6]={ 0.9,0.8,0.6,0.4,0.2,0 };
 const long TIME_END=long( (MAX_GAS / GAS_MOVE)*TIME_MOVE );

 long curr_time,slot_time_left, sim_time_left;
 int k, flag_end;
 long distance;
 long time;
 long gas;
 long max_distance;
 long possible_buy_stock,possible_earn,keep_for_gas;
 Place here;

 ctl->get_place(here);
 gas=ctl->get_stock(GAS);
 max_distance=gas * GAS_MOVE;
 if(max_distance < info[here.st][here.av]->gas_distance){
  choice.possible_earn_rate= 0;
  //plan=IDLE;
```

```
  return;
}
ctl->get_time(curr_time,slot_time_left,sim_time_left);
if( sim_time_left < TIME_END ){
  flag_end=1;
  k=int( ( TIME_END - sim_time_left ) / (TIME_END / )+1;
}
else { flag_end=0; k=0;}

if( choice.seller==0 ) {
  distance= here - choice.buyer->here;
  if ( ! distance ) {  //the case when just sell something and stay in this place
     choice.possible_earn_rate= 0;
     return;
  }
}
else distance= choice.seller->here - choice.buyer->here+(choice.seller->here -here);

choice.flag_gas=0;        // don't need to go to gas station
if( max_distance < (distance + choice.buyer->gas_distance ) ){
  //gas is not enough to the destinated buyer
  long distance2;
  if ( choice.seller ){    // from here to seller to buyer pass a gas station
     //first suppose going to here's nearst gas station
    distance=info[here.st][here.av]->gas_distance +
     ( choice.seller->here - info[here.st][here.av]->gas_station ) +
     (choice.seller->here -choice.buyer->here);
    choice.flag_gas=1;
    // consider first going to the seller's nearst gas station then to seller
```

```
 if ( max_distance > ( distance2=here-choice.seller->gas_station ) &&
  ( distance2 += choice.seller->gas_distance +
  ( choice.seller->here - choice.buyer->here )) < distance ){
     distance=distance2;
     choice.flag_gas=2;
 }
 // consider first goint to seller then to seller's nearst gas station
 if( max_distance > ( distance2= here-choice.seller->here +
  choice.seller->gas_distance) &&
  ( distance2 += ( choice.seller->here - choice.buyer->gas_station )) <
   distance ) {
     distance=distance2; choice.flag_gas=3;
 }
 //consider first going to seller then to buyer's nearst gas station
 if( max_distance > ( distance2= long ( here-choice.seller->here +
  (choice.seller->here- choice.buyer->gas_station))) &&
  (distance2 += choice.buyer->gas_distance) < distance ){
     distance=distance2; choice.flag_gas=4;
 }
} else {     // directly from here to gas station then to buyer
 ChenDealerInfo *here_info=info[here.st][here.av];
 // suppose first to here's nearst gas station then to buyer
 distance=here_info->gas_distance +
     ( here_info->gas_station - choice.buyer->here ) ;
 choice.flag_gas=5;
 //consider first going to buyer's nearst gas station then to buyer
 if( max_distance > (distance2=here- choice.buyer->gas_station) &&
  ( distance2 += choice.buyer->gas_distance) < distance ){
     distance=distance2; choice.flag_gas=4;
```

```
    }
   }
  }

  keep_for_gas= ( choice.flag_gas ) ? MAX_GAS * GAS_PRICE : 0;
  if( choice.seller ){
   possible_buy_stock =( ( ctl->get_capital() - keep_for_gas ) /
          choice.seller->sell_price );
   if( possible_buy_stock > choice.buyer->stock - ctl->get_stock(choice.buyer->kind) )
    possible_buy_stock = choice.buyer->stock - ctl->get_stock(choice.buyer->kind);
   possible_earn= long ( possible_buy_stock * ( choice.buyer->buy_price -
                choice.seller->sell_price )
               + ctl->get_stock( choice.buyer->kind) *
               ( choice.buyer->buy_price -
               deduction_rate[k] * choice.seller->sell_price ) );
  }else {
   possible_earn= long( ctl->get_stock( choice.buyer->kind)
          * choice.buyer->buy_price * (1- deduction_rate[k] ) );
  }
  time=distance * TIME_MOVE;
 f (flag_end && sim_time_left < distance * TIME_MOVE + TIME_TRANS *3 ) {
     choice.possible_earn_rate=0;
  } else choice.possible_earn_rate= double(possible_earn) /time;
}
```

```
void ChenTruck::update_info (Place here) {
 // Update information about the dealer here.
 //need to consider the oil station and the citatuation of none dealer;
 Place here2;   //actually place
 long curr_time,slot_time_left, sim_time_left;
 ChenDealerInfo *tmp_info;
 Dealer_list *tmp_list;

 ctl->get_place(here2);
 tmp_info=info[here.st][here.av];
 ctl->get_time(curr_time,slot_time_left,sim_time_left);
 if (tmp_info == 0){     //new dealer point
  tmp_info = new ChenDealerInfo(here);
  info[here.st][here.av]=tmp_info;
  tmp_info->here=here;
  tmp_info->gas_distance=MAX_DIST;
  if(here2.av != here.av || here2.st != here.st )
    ctl->phone_info( here.av, here.st,tmp_info->kind,
          tmp_info->buy_price,
          tmp_info->sell_price ,tmp_info->stock );
  else ctl->get_info(tmp_info->kind,
          tmp_info->stock,
          tmp_info->buy_price,
          tmp_info->sell_price );
  if ( tmp_info->kind==GAS ){
   tmp_info->gas_distance=0;
   tmp_info->gas_station=here;
```

```
    for( int i=0;i< NUM_ST;i++)
     for( int j=0; j< NUM_AV;j++)
       if( info[i][j] && info[i][j]->gas_distance > abs( here.st -i) + abs( here.av -j ) ){
          info[i][j]->gas_distance=abs( here.st -i) + abs( here.av -j );
          info[i][j]->gas_station=here;
       }
   }else{   //need to calculate current point's nearst gas station distance and the place
    Dealer_list *tmp=commodity[GAS];
    while( tmp){
      if( tmp_info->gas_distance > here - tmp->company->here ){
       tmp_info->gas_distance=here - tmp->company->here;
       tmp_info->gas_station=tmp->company->here;
      }
      tmp=tmp->next;
    }
   }
   tmp_list=new Dealer_list(tmp_info);
   if(commodity[tmp_info->kind] ) commodity[ tmp_info->kind ]->AddToList(tmp_list);
   else commodity[ tmp_info->kind ] = tmp_list;
  }else{       //check the dealer point that has beed checked before
   long previous_buy_price,previous_sell_price;
   long previous_check_time;

   previous_buy_price=tmp_info->buy_price;
   previous_sell_ price=tmp_info->sell_price;
   previous_check_time=tmp_info->last_time_check;
   if(here2.av != here.av || here2.st != here.st )
    ctl->phone_info( here.av, here.st,tmp_info->kind,
          tmp_info->buy_price,
```

```
            tmp_info->sell_price ,tmp_info->stock );
    else ctl->get_info(tmp_info->kind,
            tmp_info->stock,
            tmp_info->buy_price,
            tmp_info->sell_price );
    if( previous_buy_price != tmp_info->buy_price ||
      previous_sell_price !=tmp_info->sell_price ) {
     //calculate the change time period,
     tmp_info->last_time_check=curr_time;

     //average the two change time, the current change is calculate by
     //current time - last time check;
     tmp_info->change_period += curr_time - previous_check_time;
     tmp_info->change_period /=2;
     if( tmp_info->change_period < TIME_SLOT )
      tmp_info->change_period = TIME_SLOT ;

     tmp_info->changed_rate=
      double ( abs( previous_buy_price - tmp_info->buy_price ) +
      abs( previous_sell_price - tmp_info->sell_price) ) /
      ( (previous_buy_price + previous_sell_price) /2 ) ;
    }
  }
}
```