Simulation of Traffic at an Intersection

Jun Zhao

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Science at
Concordia University
Montreal, Quebec, Canada

November 1999

0-612-47860-2

ABSTRACT

Simulation of Traffic at an Intersection

Jun Zhao

This simulation is a tool that a user can use to simulate various situations in a traffic intersection. The tool allows a user to change the environment settings while simulating the design of a crossroad. Two kinds of moving objects are considered: vehicles and pedestrians. The multi-threads method is applied to simulate real time system. Certain assumptions are made so that the simulation can focus on interesting scenarios.

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Professor Peter Grogono. He had spent his valuable time, patiently directed and guided me in writing this report even during his sabbatical leave.

I would also like to thank my wife, Linda, who read this report and offered me her best suggestions.

Table of Contents

List of Figures

Chapter 1

Introduction

This report describes a simulation tool, named TSST (Traffic System Simulation Tool). It elaborates the motivation of creating the simulation tool, the design idea, methods of implementation, features of the tool and possible future development.

1.1 Crossroad – An accident-prone area

Nowadays, most people use buses, cars, trucks or trains as their means of transportation. It is not uncommon to find traffic congestion in certain places during rush hours. There are also places labeled as "Accident Black Spots". Serious traffic accidents happen not only on highways, but also on local routes. The cause of many traffic accidents is the poor behavior by certain drivers. Other accidents are the results of pedestrians not following traffic regulations properly when crossing a road. The traffic light system, though useful, might also be a factor. Most accidents happen in places where there are numerous diversions or exits. The busier the road is, the greater the possibility of having an accident.

1.2 Purpose of creating the TSST

Consider the following situations, each of which could cause an accident.

- There are drivers who do not reduce their speed when approaching a yellow light.

- There are inconsiderate drivers who often ignore a red light.
- There are uncooperative pedestrians who do not follow traffic light signals when crossing a road.
- A green light time is improperly or unreasonably set up, too long or too short and could cause traffic-flow bottleneck.

In addition to the above situations, efficient utilization of a traffic system is another area to consider. This includes the balance of current and future utilization of a road. For the safety of a driver or a pedestrian, for saving every road user's time, for increasing the efficiency of road use and for the better development of cities, it is the ultimate goal of a designer to consider various factors together in a design of a traffic network. Simulating a traffic system is an economical and easy way to collect and analyze possible situations that could happen on roads. The purpose of building a good simulation tool is to provide the designers with a vivid picture of the future traffic system to be constructed. At the same time, the designers could recognize in advance possible design flaws.

1.3 Outline of the Simulation Tool

The simulation tool comprises three parts: the user interface, the simulation engine and the animation. The UI allows the user to decide on details of a project such as how many vehicles or pedestrians will be generated per minute; how the traffic light system works, etc. based on the statistics collected from real traffic intersection using sampling methods. These settings come from the research on the statistics. A serious design flaw could arise from a lack of statistical data on traffic and pedestrian flows in certain areas. Such statistics would be extremely useful to a designer in determining how a traffic system should be constructed and how a traffic light system should be set up. By involving the simulation tool, a designer could consider and compare various designs and obtain further statistics data according to different settings.

The simulation engine is responsible for the generation of vehicles and pedestrians; communication between threads; and sending signals to change the color of traffic lights. Finally, the animation part implements the display of all moving objects and the changes in the traffic lights.

1.4 Choice of Platform

The simulation tool, TSST, is designed in Object Oriented language, the UML, and coded in Visual C++ under Windows NT Platform, using Microsoft Foundation Classes. There are several reasons for using Windows Programming and Visual C++. First, with Windows programming being widely used, many development tools are very user friendly, such as Visual C++ 6.0. For instance, Visual C++ can automatically build up a skeleton of the program according to programmer's selection. This function saves time for programmers who can thus concentrate on creating appropriate classes and building up relationships among them.

Second, although Windows programming requires more expensive hardware and could not run on some machines, it is a full-featured, multiprocessing, 32-bit operating system <Ref. [2]>. As the prices of memory and hardware drops continuously, the cost on computer hardware system comparing with that on software system is not significant any more. The enhancement of CPU speed has largely improved the quality of animation.

Third, using multi-thread in Windows programming to simulate a real time system makes the TSST open. Later, different machines could be connected together by network to simulate pedestrian-generating resource, light-signal-generating resource and vehicle-generating resource.

Chapter 2

Implementation of the Simulation Methodologies

This chapter presents the basic concepts of the simulation system, the purpose of building the TSST simulation tool and the simulation model of the TSST.

2.1 Basic Concepts of Simulation

Simulation could be defined as follows:

> The task of running experiments with a mathematical model using numeric technique to imitate the behavior of a system over a given period of time. [5]

Numeric techniques are more appropriate in cases where there are no known mathematical analytic solutions, the solution is too difficult, or it is not practical to solve the set of mathematical expressions. For example, the behavior of a system can be observed when the system is simulated with external input actions, conditions, and the passage of time. A *simulation model* is a mathematical model implemented on a computer, usually in a programming language or in a simulation language, in order to run experiments. [5] Every execution of this experiment with some given set of parameters is a *simulation run.*

2.2 The Purpose of Building a Simulation Tool

Many situations could happen at crossroads because of different human behavior. For instance some pedestrians prefer to wait on the sidewalk when the light is yellow, but some people would go hastily across the road. A pedestrian decides on crossing a road according to the situation of the road and his motivation. It is not possible to model a crossroad system completely with formal mathematical expressions. Simulation may be the only method to solve such informal mathematical models. As José M. Garrido claimed, simulation can be used as an analysis technique and as a design technique, as a general pedagogical tool to complement analytical techniques and to gain better understanding of the behavior of a real system. [5] The purpose of building the TSST simulation tool is to let a designer or an analyst get more information of a crossroad system through simulation runs.

2.3 Problem Formulation

Before proceeding further with the description and analysis of the TSST system, some critical questions are explicitly raised and we hope the simulation model can answer them. It is obvious that the success or failure of an analysis using simulation models rests upon how clearly we state the objectives of the simulation model. This means careful formulation of the problem is essential when modeling a system.

Questions about pedestrian are as follows:

1. What is the maximum rate of generating pedestrians?
2. What is the speed range of a pedestrian?

3. What is the maximum density of pedestrians in the display area?
4. What is the average walking distance of pedestrians?
5. What is the average walking time of pedestrian?

Questions about vehicle are as follows:

1. What is the speed range of a vehicle?
2. What is the average life time (running time) of vehicles?
3. What is the probability distribution invoked to decide the running direction of a vehicle?
4. What is the probability distribution invoked to decide on which lane a vehicle will adopt?
5. What is the maximum rate of generating vehicles?

Questions about traffic light, congestion and accident are as follows:

1. What is the range of the duration of each traffic light?
2. What is the probability of having an accident?
3. What could be applied to measure or predict that congestion has happened?
4. What is the primary factor causing an accident?

Part of the answers to the above questions is found in section 2.4.1. The rest can only be expressed by the results of a simulation run.

2.4 Simulation Modeling

The TSST is built to achieve two goals. One is to get the probabilities of accidents under given conditions. Another is to get certain conditions that will cause congestion.

2.4.1 Brief Analytic Model

To develop a simulation model of the TSST system, we would analyze it first. To reach the first goal, we use a mathematical formula to explain the analysis. Let A_i represents the event of accident caused by the i-th vehicle. Assume each event A_i is independent with other events, where $i = 0, 1, 2, \ldots$. Let A represent the event of accident caused by all vehicles. Then

$$P(A) = P(\cup_{i=0,n} A_i) = 1 - P(\cap_{i=0,n} \bar{A}_i) = 1 - \prod_{i=0,n} P(\bar{A}_i) = 1 - \prod_{i=0,n}(1 - P(A_i)).$$

Where n is the number of vehicles. [14] If we assume that all vehicles have the same probability of accident, say p, then

$$P(A) = 1 - (1 - p)^n \rightarrow 1 \text{ when } n \rightarrow \infty, \text{ where n is the number of vehicles.}$$

From the formula above, we can obtain the following conclusions.

1) Unless every vehicle follows the traffic rules completely, i.e. $p = 0$, accident will happen someday.
2) For given n, the probability $P(A)$ will decrease when the p decreases. That is, a designer of a traffic system could improve the safety of a crossroad area by reducing the average of the probabilities of accidents caused by vehicles.
3) For given p, the probability $P(A)$ increases according to the increase in n, the number of vehicles. If the rate of the flow of vehicles can be controlled, it means that the

number of vehicles running through a traffic intersection per minute is limited under a threshold, the probability P(A) of occurring an accident will reduce.

To achieve the second goal, we need a criterion to measure situations to see if any congestion has occurred. A congested system may broadly be described as a system in which there is a demand for the resources of a system, and when the resources are not available, those requesting the resource wait for them to become available. [7] In the TSST, when a moving object is generated, it is put into the waiting queue. If there is proper space to display even part of the object in the simulation area, (i.e. the resource is available to this object), it changes to a proper list that contains running objects. The level of congestion in the TSST system is measured by the waiting queues of resources requests. Let Q_l represents the waiting queue length and T_q the threshold defined by user. When $Q_l > T_q$ we can say there is congestion.

There is another way to measure if congestion has happened. Let D_p and D_v represent the density of pedestrians and of vehicles respectively; let user-defined T_{dp} and T_{dv} represents the threshold of pedestrian density and of vehicle density respectively. When $D_p > T_{dp}$ or $D_v > T_{dv}$ we could conclude that the road is congested.

The answers to part of the questions listed in section 2.3 are as follows:

1) The maximum rate of generating pedestrians can be obtained from collecting of samples. For example, the maximum rate is defined as 100 pedestrians per minute. This definition is used to limit a user's setting of the rate of generating pedestrians.

2) The walking speed of a pedestrian is equal to the length per step multiplied by the steps per minute. Considering the adult and senior, the length of step ranges between 50cm/step and 135cm/step. Assuming that a person should use a wheelchair and move by someone if one's step length is less than the lower limit. The upper limit, 135cm/step, is given by the medical studies. The steps per minute ranges between 40steps/minute and 200steps/minute.
3) The speed range of a vehicle should neither be less than 0 nor greater than 200km/Hr.
4) We let a user to decide on the running direction of a vehicle. But we use uniform distribution to decide on which lane a vehicle will adopt. We can see that a driver often changes to a lane where there are less vehicles.
5) The answer to the maximum rate of generating vehicles is 1000 vehicles per minute. Assuming there is a maximum of 8 lanes at each direction, and there is a maximum of 4 directions. The average of vehicles passing each lane per minute is 30. Thus, 8 * 4 * 30 = 960 vehicles/minute.
6) It is easy to define the duration of a traffic light. The range can be from 1 second to 5 minutes.

2.4.2 Simulation Model

We have not discussed more about the analytic model in the above section because the analytic model could not describe the TSST system completely. The cause of an accident by an individual vehicle could be complex. The driver's motivation and behavior, the road situation, the vehicle speed, the pedestrian's behavior, or the color change of traffic light, all these may affect a driver's decision. The cause of congestion is also complex. In general, accident and construction are the two major causes. The design flaw of a traffic system could also be an implicit cause. For example, during rush hour, some exits and entrances are always congested.

An advantage of the simulation model is that we do not need to make assumptions and constraints as we did in the analytic model. We will let simulation runs to answer part of user's questions listed in section 2.3.

Variables for pedestrians:

R_p = the rate of generating pedestrians.

$N_p(t)$ = number of life pedestrians at time t.

$N_{pRunover}(t)$ = number of pedestrians who has left the simulation at time t.

T_{pd} = the threshold of pedestrian density, defined by user.

$D_p(t)$ = the density of pedestrians.

S_p = the pedestrian speed.

L_p = the distance that a pedestrian walks from origin to destination.

T_p = the time that a pedestrian spends on walking from origin to destination.

$Q_{pl}(t)$ = the length of pedestrians waiting queue of at time t.

T_{pq} = the threshold of pedestrians waiting queue length, defined by user.

Variables for vehicles:

R_v = the rate of generating vehicles.

S_v = the vehicle speed.

$Q_{vl}(t)$ = the length of vehicles waiting queue at time t.

T_{vq} = the threshold of vehicles waiting queue length, defined by user.

$N_v(t)$ = number of life vehicles at time t.

$N_{vRunover}(t)$ = number of vehicles which has left the simulation at time t.

A_i = the event of an accident caused by the i-th vehicle.

A = the event of an accident.

Conditions:

C(1) : $R_p <$ maximum rate of generating pedestrians, say Rpx.

C(2) : $R_v <$ maximum rate of generating vehicles, say Rvx.

C(3) : minimum pedestrian speed, say $S_{pmim} < S_p <$ maximum pedestrian speed, say S_{pmax}.

C(4) : minimum vehicle speed, say $S_{vmim} < S_v <$ maximum vehicle speed, say S_{vmax}.

Formula for pedestrian:

$L_p = T_p * S_p + \lambda_p$; the distance that a pedestrian walked from origin to destination. The variable λ_p describes random factors such as the interaction of pedestrians and road situations, which would affect the walking speed and the walking route of the pedestrian.

PedGen(t) = $R_p * t$; the number of pedestrians generated from the beginning of a simulation run to time t.

$N_p(t)$ = PedGen(t) – $N_{pRunover}(t) - Q_{pl}(t)$.

$N_{pRunover}(t)$ is influenced by the interaction of pedestrians and road situations.

$D_p(t) = N_p(t)$ / size of sidewalk; where the size is fixed.

If $D_p(t) > T_{pd} \Rightarrow$ define : congestion has taken place.

If $Q_{pl}(t) > T_{pq} \Rightarrow$ define : congestion has taken place.

Formula for vehicles:

$L_v = T_v * S_v + \lambda_v$; the distance run by a vehicle is fixed, but the running time is affected by some random factors. The variable λ_v describes the random factors, such as traffic light change or pedestrian's behavior, that would affect the running time of a vehicle.

$VehGen(t) = R_v * t$; the number of vehicles generated from the beginning of a simulation run to time t.

$N_v(t) = VehGen(t) - N_{vRunover}(t) - Q_{vl}(t) <$ maximum number of vehicles that could be displayed.

If $Q_{vl}(t) > T_{vq} \Rightarrow$ define : congestion has taken place.

We have to point out that the simulation model built in this version is simple. We could always improve the simulation model in future work. For example, the calculation of the probability of an accident could be considered complicatedly. Recall that,

$$P(A) = P(\cup_{i=0,n} A_i) = 1 - P(\cap_{i=0,n} \bar{A}_i) = 1 - \prod_{i=0,n} P(\bar{A}_i) = 1 - \prod_{i=0,n}(1 - P(A_i)).$$

In real system, those $P(A_i)$ could be different because each driver is an individual. Furthermore, we could divide A_i into three parts as follows:

A_{i1} = the event of accident caused by driver's behavior.

A_{i2} = the event of accident caused by driver's motivation.

A_{i3} = the event of accident caused by road situation.

Thus, $A_i = A_{i1} + A_{i2} + A_{i3}$.

Then we could weight items of A_i as below:

$$A_i = \alpha * A_{i1} + \beta * A_{i2} + \delta * A_{i3}.$$

Where $\alpha + \beta + \delta = 1$; $\alpha \geq 0$; $\beta \geq 0$; $\delta \geq 0$. Then,

$$P(A_i) = \alpha * P(A_{i1}) + \beta * P(A_{i2}) + \delta * P(A_{i3}).$$

Then A_{i3} could be finely divided into the events of accidents caused by vehicles' position, by pedestrians' position and by color of traffic light, etc.. However, such division will not largely improve the quality of the simulation tool. We prefer to use a simple model in this version.

2.5 Advantage of a Simulation

Simulation modeling can be used effectively to achieve the following objectives: [5]

- Predict the behavior of an existing system subject to some specified condition;
- Study a system before it is built (i.e., at its design stage).

Some of the most relevant cited advantages of simulation are: [5]

1. A simulation model can be used repeatedly to analyze proposed designs or policies.
2. Simulation data are usually much less costly to obtain than similar data from the real system.
3. Simulation methods are usually much easier to apply than analytical methods.
4. Simulation models do not have restrictive simplifying mathematical assumptions common to analytical methods. Almost any conceivable performance measure can be estimated from data generated with simulation models.

5. In some cases, simulation may be the only approach that can be used to reach a solution to a given problem.

2.6 Simulating Real Time System

The TSST is to simulate a real time system in which behavior changes with time. A user could obtain both visible results and analysis results. When increasing moving objects are walking or waiting in the traffic intersection, we could predict that congestion has risen. When a collision occurred, be it with a vehicle or a pedestrian, user could hear a beep and see the overlap of two moving objects. Alternatively, user could get the analysis results by running the generating-reports functions. In other words, what a user can see from a simulation run are recorded for statistics analysis later.

Let us view the inside of TSST system. There are three independent resources of generating vehicles, pedestrians and traffic light signals respectively. When these data arrive at the main program loop, they follow the first-come-first-serve rule. Any delay in transferring the data could change the situation of road so that an individual occurrence could change. However, user could get a statistical result by running a simulation for a certain time.

Chapter 3

The Design of the Simulation Tool

Designing Object-Oriented software is hard, and designing reusable Object-Oriented software is even harder. The design should be specific to the problem at hand but also general enough to address future problems and requirements. [11]

In chapter 2, we have built the simulation model and in this chapter we will discuss more about the high level design of the TSST system according to the design architecture to avoid (or at least to minimize) redesign.

3.1 Specification of Problem Definitions and Requirements

Tracing requirements in a large project is one of the most basic and, at the same time, the most difficult needs of development. In this section, we will discuss the important part of the problem definitions and the requirements.

3.1.1 Simulation Components

Goal of TSST

Recall that the TSST is a simulation tool to simulate a real time traffic system and to provide traffic system designer a tool to analyze various situations taken place in a crossroad.

User Interface

A user should set up the simulation environment before running a simulation. The TSST system provides default settings for the environment. The environment includes parameters as follows:

1) the kind of moving objects;
2) the maximum number of moving objects generated per minute;
3) the speeds of different kinds of moving objects;
4) the origin and destination of a moving object;
5) the number of lanes associated with directions;
6) the duration of the colors of traffic lights;

A user could run, pause, continue or stop a simulation run.

A user could save or load a simulation project.

A user could play a real-time simulation, replay or reverse a simulation run.

A user could use the TSST system to generate several reports. The formats of the reports are built-in.

Moving Object

The moving objects refer to pedestrians who walk on sidewalks and vehicles that run along lanes. Different kind moving objects could have different moving speeds. The same kinds of moving objects have the same physical sizes and shapes.

Moving objects have their own life cycles, that is, from generation to appearance (walking or running), then to disappearance.

3.1.2 Simulation Management

Traffic Light

A traffic light system comprises red, yellow, green lights and a control center. The control center controls the duration of different colors of traffic lights. The order of changing light colors is from Green to Yellow then Red.

Crossroad Area

Crossroad area is a place where pedestrians walk around, vehicles run through and traffic lights are erected. This area should include lanes, stop lines, sidewalks, crosswalks, boundaries of the crossroad area and traffic lights.

Animation

The TSST should provide not only the function of playing real-time simulation but also the functions of reversing and replaying a simulation run.

Save and Load a simulation run

A simulation run is also called a simulation project. The TSST system should allow a user to save current simulation run in a file and load a saved simulation project.

Report Generation

The TSST should have functions of generating statistical reports. This part will be developed in the future.

3.2 Modeling the TSST System

Analysis, the first step of the OMT methodology, is concerned with devising a precise, concise, understandable, and correct model of the real world. [13] At top level design, we have divided the TSST system into several independent objects based on the analysis. The objects are the Settings, the View, the Document, the Background, the Moving Object, the Traffic Light, the Simulation Engine, the Database, the Animation and the Report Generation. We will discuss these objects in the following sections.

3.2.1 Settings of Simulation Environment

The Settings object acts as a container that holds all user-input parameters of the simulation environment and the name of a simulation project. It also provides users an interface by which the simulation settings could be modified or reset. A user could either save only the settings of a simulation run or the whole simulation run. To save a simulation run, its settings must be saved into a file with other information and data. Each saved simulation project must have a name. Reloading a saved project implies resetting the simulation environment. The Settings object will be invoked by the Simulation Engine for calculation, and by the Document object for saving the settings of a simulation project.

3.2.2 View, Document and Background

The View brings a user an interface of user-computer interaction. It contains certain menu items by that a user can run, pause, continue or stop a simulation; save or load a project; generate various reports.

The View is responsible for drawing moving objects. When the View is to draw, it requests the moving object lists from the Animation object. After drawing all objects from the lists, the View releases the object lists.

When the size of a crossroad area or the number of traffic lanes are changed, the View will redraw the crossroad area by calling the Background object. The Background object contains built-in methods for drawing sidewalks, stop lines, traffic lanes, boundaries of the crossroad, buildings and traffic lights.

The Document takes care of saving and retrieving of the settings of a simulation to or from a file. The Settings are saved or retrieved in a certain format and order.

3.2.3 Moving Object

A moving object could be anything that moves either on a road or on a sidewalk, such as pedestrians, bicycles, or vehicles. A moving object must contain all information relative to its characteristics. The information includes unique name or identifier, object type, moving speed, object size and shape, origin and destination. The same kinds of objects have the same moving speeds, sizes and appearances. Dynamically, each moving object also contains its own current position. The View can display all information of a moving object on the screen.

Moving objects are generated from two independent resources: the pedestrian generation system and the vehicle generation system. Each moving object is unique. A new generated moving object will be put in a buffer. The Animation checks the buffer periodically to fetch the new objects and put them to the proper lists according to each object type. When arriving at its destination, a moving object is removed from the Animation object. The Database saves all steps of a moving object from the origin to the destination until no more space is available.

A moving object's life cycle contains the following status:

Generated $\Rightarrow$ Birth $\Rightarrow$ Moving $\Leftrightarrow$ Stop $\Leftrightarrow$ Moving $\Rightarrow$ RunOver.

Where, the '$\Rightarrow$' represents 'changes to', and the '$\Leftrightarrow$' means 'changes to and back'.

The Moving Object is divided into two types: the Vehicle and the Pedestrian. We do not consider other type objects such as bicycles because they appear very less in a crossroad area and their speeds are not fast. We can see them as those pedestrians at fastest walking speed. To simplify the simulation model, the Vehicle is sub-divided into the Car and the Truck, and the Pedestrian is sub-divided into the Adult and the Senior.

We have mentioned that moving objects are generated from independent resources. These resources consist of the Pedestrian-Generation resource and the Vehicle-Generation resource. They are responsible for generating adults and seniors, cars and trucks, respectively. These moving object generation resources request the proper buffer to append the new generated moving objects.

3.2.4 Traffic Light

The Traffic Light consists of red, yellow and green lights. There are, at maximum, four sets of traffic lights: one facing north (named south light), one facing south (north light), one facing west (east light) and one facing east (west light). The north and south lights are always lit in the same color. The same applies to the east and west lights. When north/south lights are green or yellow, east/west lights must be red. Such rule also applies *vice versa*.

The Traffic Light signal changes in order and lasts certain minutes according to the settings of the traffic light system. The sequence is: green, yellow, red, green, ... The duration is defined by users and kept in the Settings object. The Traffic Light object periodically sends signals to notify the View object to redraw colors of lights.

3.2.5 Simulation Engine

The Simulation Engine is the core of the TSST system: its quality depends on how the engine is designed and implemented. The Simulation Engine contains all algorithms and constraints of the simulation for calculating the next steps for all moving objects and removing the run-over objects from the moving object lists. A programmer could improve the TSST system by modifying the Simulation Engine.

The real world situations and the activities that take place in a crossroad are very complicated. Some assumptions and constraints are necessary to catch the major points of the TSST system and to simplify the coding of the simulation tool. The assumptions and constraints are listed as follows:

About vehicles

- A vehicle is not allowed to turn left or right.
- A vehicle can not change lanes.
- A vehicle can not make U-turns.
- A vehicle has to keep a minimum distance away from vehicles ahead.
- A vehicle can not move onto a crosswalk where a pedestrian is walking on at the same time.

- A vehicle can not move onto the center section where there is a vehicle from left or right side.

About traffic lanes

- All traffic lanes have the same width.
- Any traffic lane is straight.
- The direction of a lane could be one of the followings: from north to south (N2S), from south to north (S2N), from west to east (W2E) and from east to west (E2W).
- For a given direction, the number of lanes cannot be reduced or increased at both sides of the center section. For instance (see figure 3.1), the number of southbound lanes south to CS must be equal to the number of southbound lanes north to CS.

About pedestrians

- A pedestrian could only walk along sidewalks or crosswalks.
- A pedestrian must wait at the sidewalk if there is a vehicle in the crosswalk he is to go across.
- A pedestrian could step on the crosswalk only when the traffic light is not red.
- While walking on a crosswalk, a pedestrian will continue forward to his destination no matter what will happen or had happened.

By these constrains, the collision could occur only when two vehicles entering center section simultaneously from the left or right sides of each other. We, in this version, do not consider the accident occurred between vehicle and pedestrian. This is a shortcoming of the TSST.

3.2.6 Database

To reverse and replay a simulation, a physical database, called TSSTDB, is applied to keep all data of a simulation run. The Database contains data variables and methods. The methods includes saving, retrieving, searching and deleting data. Since the capacity of a database is limited, when the database is full, a message will be prompted to the user. The Database object will connect with the physical database by ODBC.

A user could save a simulation run by retrieving its data from the Database then saving them in a file. The data includes the simulation settings, information of all objects and the sequence of objects' steps. Alternatively, a user could choose not to save those steps.

Before running a simulation, the Database posts a message to remind a user if he wish to save the last simulation run. If his answer is 'No', all data of the last simulation run are cleared from the Database.

3.2.7 Animation

The Animation object contains lists of moving objects and methods for playing, replaying and reversing a simulation. When it is called to play a real-time simulation, the

Animation checks the buffers that hold all newly generated moving objects. If the buffers are not empty, it fetches the new objects and appends them to appropriate lists. It in turn invokes the Simulation Engine to calculate the next steps of moving objects and calls the Database object to save all living objects' positions to the TSSTDB. Finally, the View will redraw all moving objects again.

The Animation allows users to replay or reverse a simulation. If a user selects a saved simulation project, the Animation will reload the project. Reloading a project means to reset up the simulation environment, to put all generated objects on appropriate lists and to store all steps of moving objects into TSSTDB. The Animation fetches an object's next step from TSSTDB instead of calculating the position by calling the Simulation Engine. To reverse a simulation run, the Animation fetches data from the end to the beginning.

3.2.8 Report Generation

The Report Generation is responsible for generating various predefined-format reports. Its implementation is left for future development. We describe its functionality in high level to make the TSST system design complete. The Report Generation either opens saved project files or calls the Database to obtain original data for generating report.

3.3 Design of the User Interface

The TSST system is built on the Document-View architecture. The View provides users an interface to control a simulation run. We tried to make the interface as simple as possible so that a user could use the simulation tool easily. Another reason of choosing the Document-View architecture is that the architecture meets our requirements and has already been accepted by many users.

A user can control a simulation by clicking on items from the menu bar of the View. We arrange those items into four submenus, the 'Project', the 'Build', the 'Animation', and the 'Report'. The 'Project' submenu is responsible for generating a new simulation project, opening an existing one, saving the settings of a currently opened project. Every simulation project must be associated with a title. To save a newly generated project, a dialog box appears to let user type in a project name. If the name already exists, another dialog box pops up to confirm that the user is to replace the previous one.

The 'Project' submenu is also responsible for the settings of a simulation. When a user click on the Settings item to set up the simulation environment, a property-sheet pop up, which contains four pages that allow a user to set up a simulation environment. The first page, named as Lane page, consist of the number of lanes in different directions and the utilities of each lanes. The second page, named as Pedestrian page, allows a user to select the kinds of pedestrians included in a simulation. It also lets a user determine the walking speeds of adults and seniors, the percentages of adults among pedestrians and the rate of generating pedestrians (per minute) and the probabilities of possible origins and destinations that pedestrians move from and to. The third page, named as Vehicle page, allows a user to determine which kinds of vehicles are included in a simulation, to input the percentage of cars among total vehicles and to set the rate of generating vehicles (per minute). The last one, named as Light page, consists of the settings of the duration of traffic light signals and the probabilities that a vehicle might run red or yellow lights.

The 'Build' submenu is responsible for clicking on the 'Run', 'Pause', 'Stop' and 'Continue' items by a user. When one of these items is selected, a relative call back function is invoked to implement certain tasks. The selected item is disable meanwhile. A simulation starts running when item 'Run' is clicked. A running simulation could pause then continue by clicking on the items 'Pause' and 'Continue' respectively. But, clicking on the item 'Stop' means to terminate a simulation.

The 'Animation' submenu contains three subsidiary items: 'Play', 'Replay' and 'Reverse'. These items do not have call back functions. A check mark appears beside the item selected. One and only one item can be marked at a time. The default selected item is the 'Play'.

The 'Report' submenu comprises named items that reflect the report types to be generated. Recall that the Report is left for future developer.

Another user interface is the simulation drawing. Figure 3.1 illustrates the drawing of the area. The area is divided into several sections. Each section is labeled in a unique abbreviation. The labels are used in programming to represent individual sections.

3.4 Using UML as a Design Tool

The Unified Modeling Language (UML) is used to help the design of the TSST system. The UML is a kind of Object-Oriented Language and destined to be the dominant, common modeling language used by the industry. The UML comprises views, diagrams, model elements and general mechanisms.

Views (not the View object) show different aspects of a system that are modeled. A view is not a graph, but an abstraction consisting of a number of diagrams. The views also link the modeling language to the method/process chosen for development. <Ref. [3]>

Diagrams are the graphs that describe the contents in a view. UML has nine different diagram types. Model elements represent common object-oriented concepts such as classes, objects, and messages, and the relationships among these concepts including association, dependency, and generalization. General mechanisms provide extra comments, information, or semantics about a model element.

The following figures illustrate part of our high level design as discussed in section 3.2.

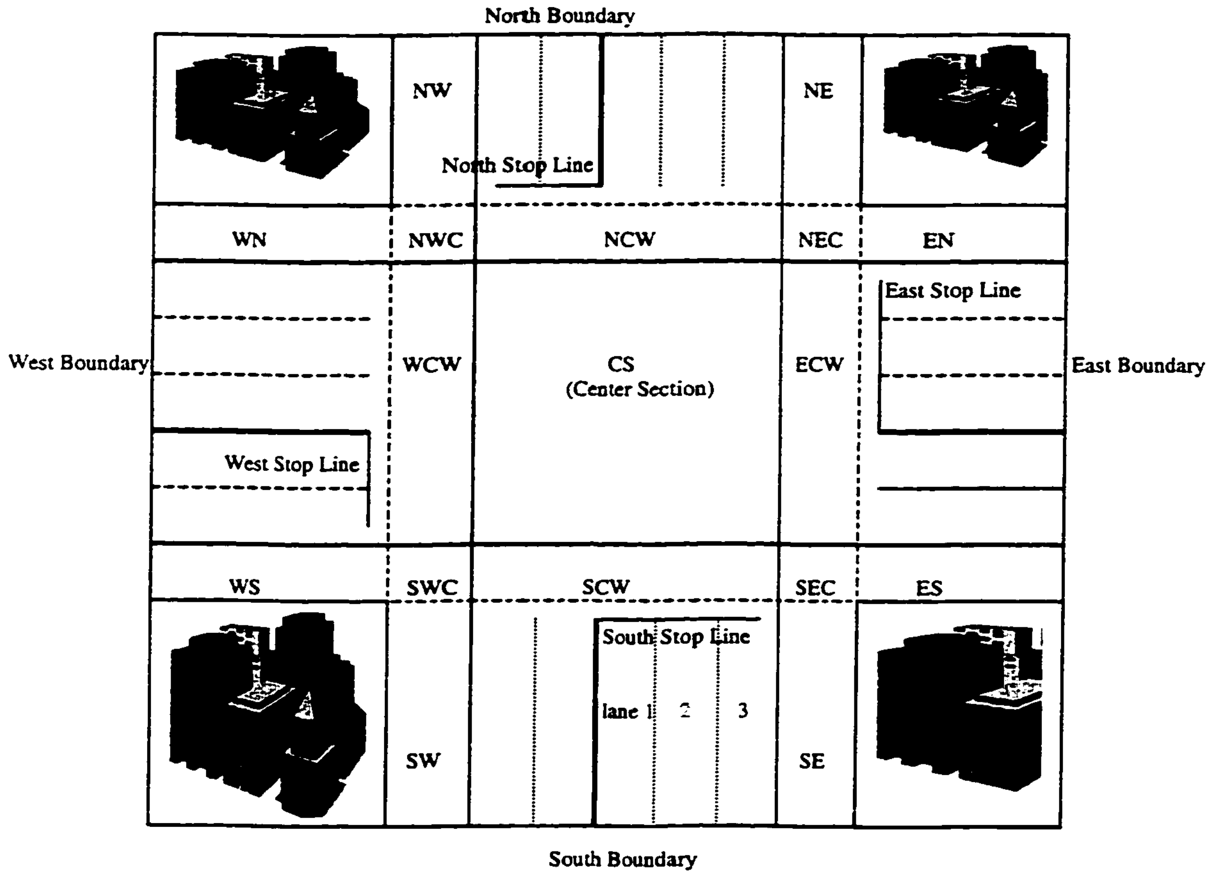


Figure 3.1 The Diagram of Traffic Intersection.
Comments: This figure is a crossroad map of a simulation run. The purpose of this map is to show the sub-areas and their relative names mentioned in this report.
Keys : NCW = north crosswalk; SCW = south crosswalk; WCW = west crosswalk; ECW = east crosswalk. The NW, NE, WN, EN, WS, ES, SW, SE, NWC, NEC, SWC, SEC are different segments of the sidewalks.

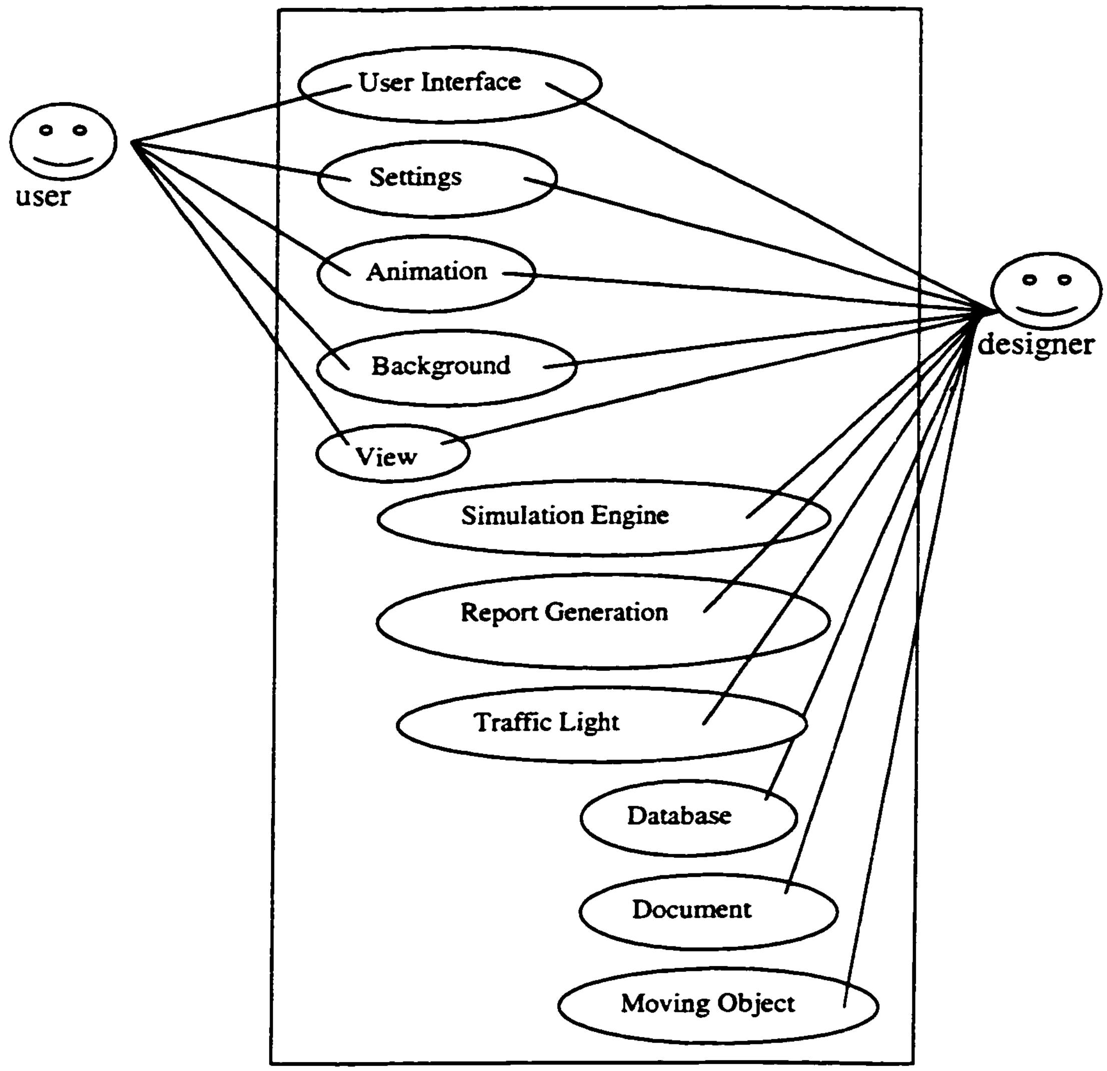


Figure 3.2 This diagram shows the different views from a user and a designer of the simulation tool. A user can only view the User Interface that includes part of the View and of the Settings, the Animation and the Background. A designer can view all of these objects.

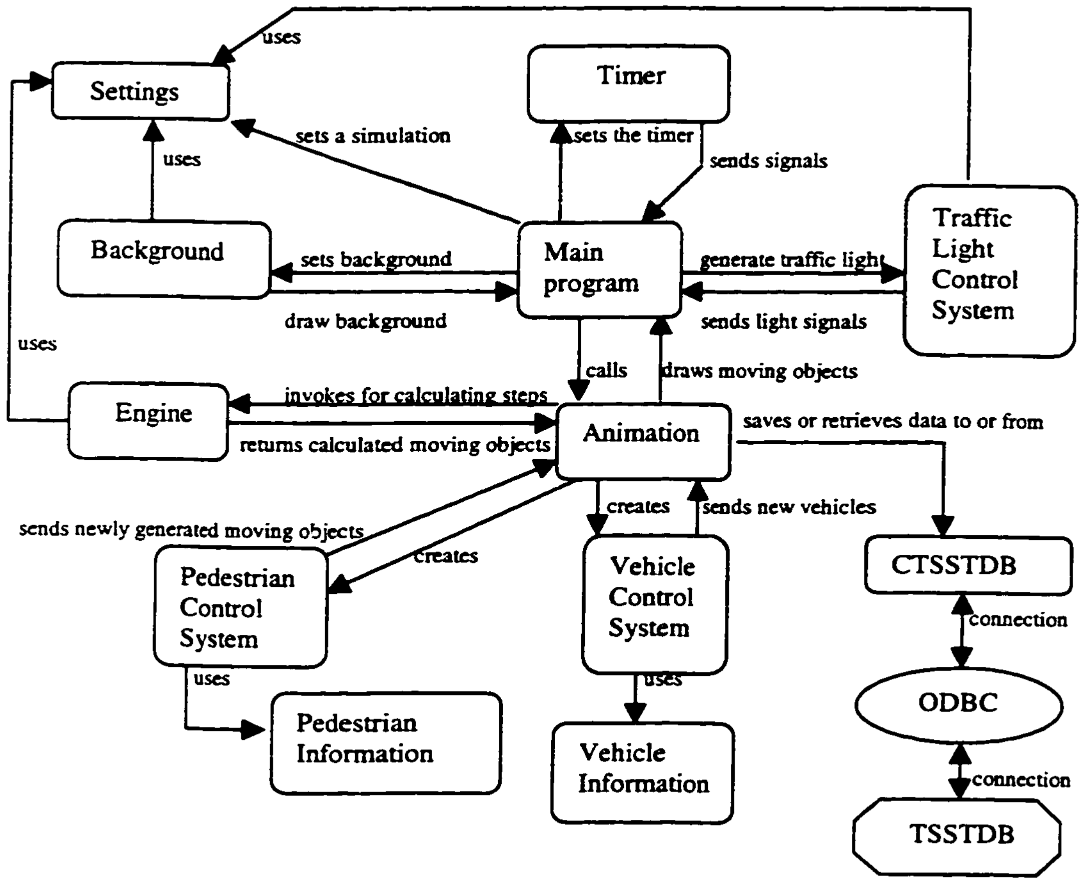


Figure 3.3 Relationship between main program and its three control systems generated.

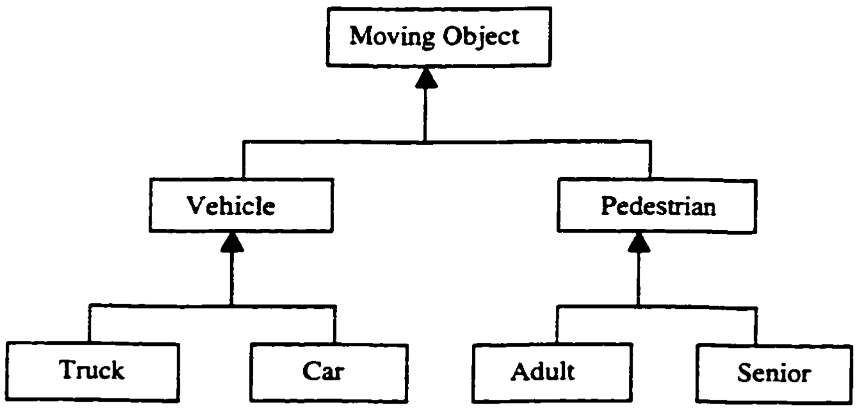


Figure 3.4 A class diagram of the hierarchy of moving objects.

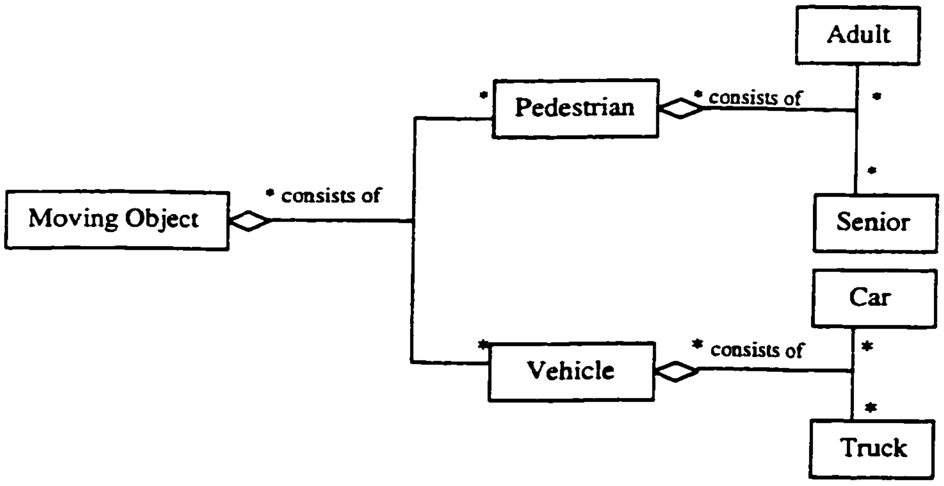
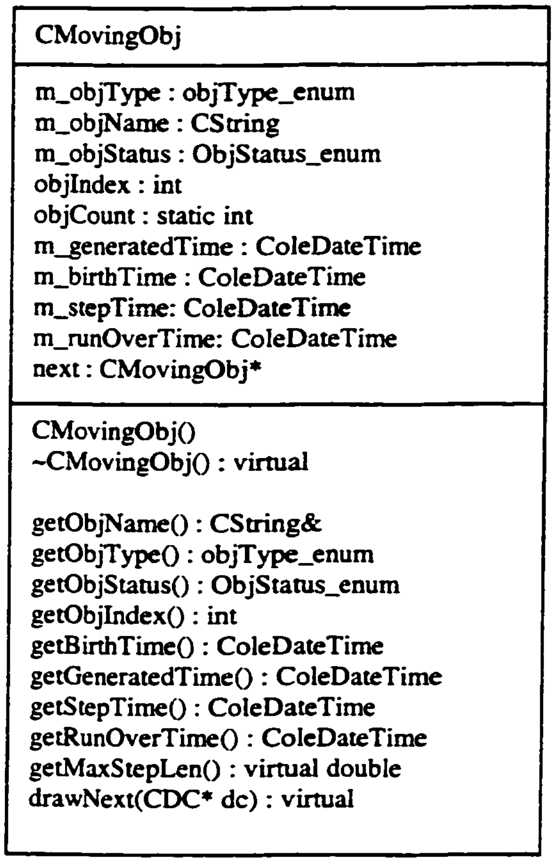


Figure 3.5 This diagram shows the aggregation associations between the Moving Objects and the Pedestrian, Vehicle; between Pedestrian and Adult, Senior; between Vehicle and Car, Truck.



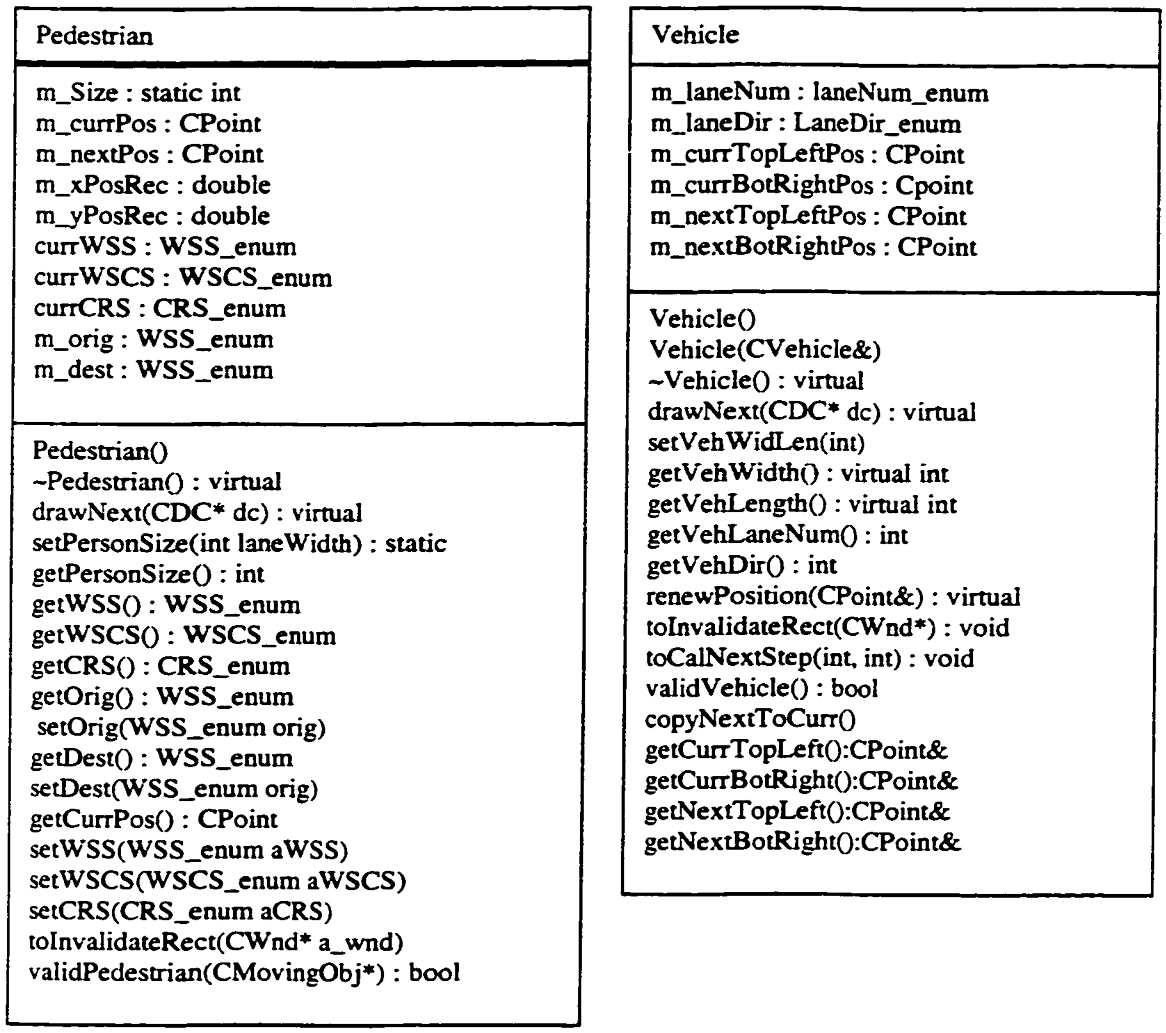
```
CMovingObj

m_objType : objType_enum
m_objName : CString
m_objStatus : ObjStatus_enum
objIndex : int
objCount : static int
m_generatedTime : ColeDateTime
m_birthTime : ColeDateTime
m_stepTime: ColeDateTime
m_runOverTime: ColeDateTime
next : CMovingObj*

CMovingObj()
~CMovingObj() : virtual

getObjName() : CString&
getObjType() : objType_enum
getObjStatus() : ObjStatus_enum
getObjIndex() : int
getBirthTime() : ColeDateTime
getGeneratedTime() : ColeDateTime
getStepTime() : ColeDateTime
getRunOverTime() : ColeDateTime
getMaxStepLen() : virtual double
drawNext(CDC* dc) : virtual
```

Figure 3.6 A class CMovingObj and a class Vehicle with its member attributes and operators. The Vehicle is a subclass of CMovingObj.



```
Pedestrian
----------------------------------------
m_Size : static int
m_currPos : CPoint
m_nextPos : CPoint
m_xPosRec : double
m_yPosRec : double
currWSS : WSS_enum
currWSCS : WSCS_enum
currCRS : CRS_enum
m_orig : WSS_enum
m_dest : WSS_enum
----------------------------------------
Pedestrian()
~Pedestrian() : virtual
drawNext(CDC* dc) : virtual
setPersonSize(int laneWidth) : static
getPersonSize() : int
getWSS() : WSS_enum
getWSCS() : WSCS_enum
getCRS() : CRS_enum
getOrig() : WSS_enum
 setOrig(WSS_enum orig)
getDest() : WSS_enum
setDest(WSS_enum orig)
getCurrPos() : CPoint
setWSS(WSS_enum aWSS)
setWSCS(WSCS_enum aWSCS)
setCRS(CRS_enum aCRS)
toInvalidateRect(CWnd* a_wnd)
validPedestrian(CMovingObj*) : bool
```

```
Vehicle
----------------------------------------
m_laneNum : laneNum_enum
m_laneDir : LaneDir_enum
m_currTopLeftPos : CPoint
m_currBotRightPos : Cpoint
m_nextTopLeftPos : CPoint
m_nextBotRightPos : CPoint
----------------------------------------
Vehicle()
Vehicle(CVehicle&)
~Vehicle() : virtual
drawNext(CDC* dc) : virtual
setVehWidLen(int)
getVehWidth() : virtual int
getVehLength() : virtual int
getVehLaneNum() : int
getVehDir() : int
renewPosition(CPoint&) : virtual
toInvalidateRect(CWnd*) : void
toCalNextStep(int, int) : void
validVehicle() : bool
copyNextToCurr()
getCurrTopLeft():CPoint&
getCurrBotRight():CPoint&
getNextTopLeft():CPoint&
getNextBotRight():CPoint&
```

Figure 3.7 Classes CPedestrian and CVehicle with part of their member attributes and operators. They are subclasses of CMovingObj.

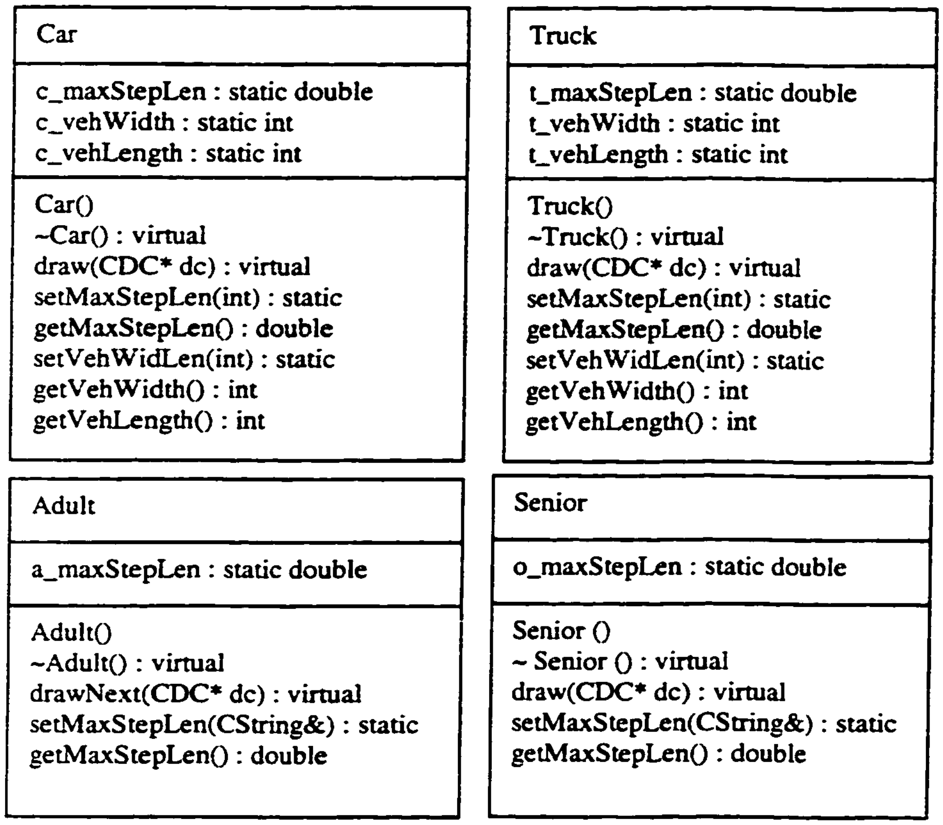


Figure 3.8 A class Car and a class Truck are the subclasses of Vehicle; and a class Adult and a class Senior are the subclasses of Pedestrian.

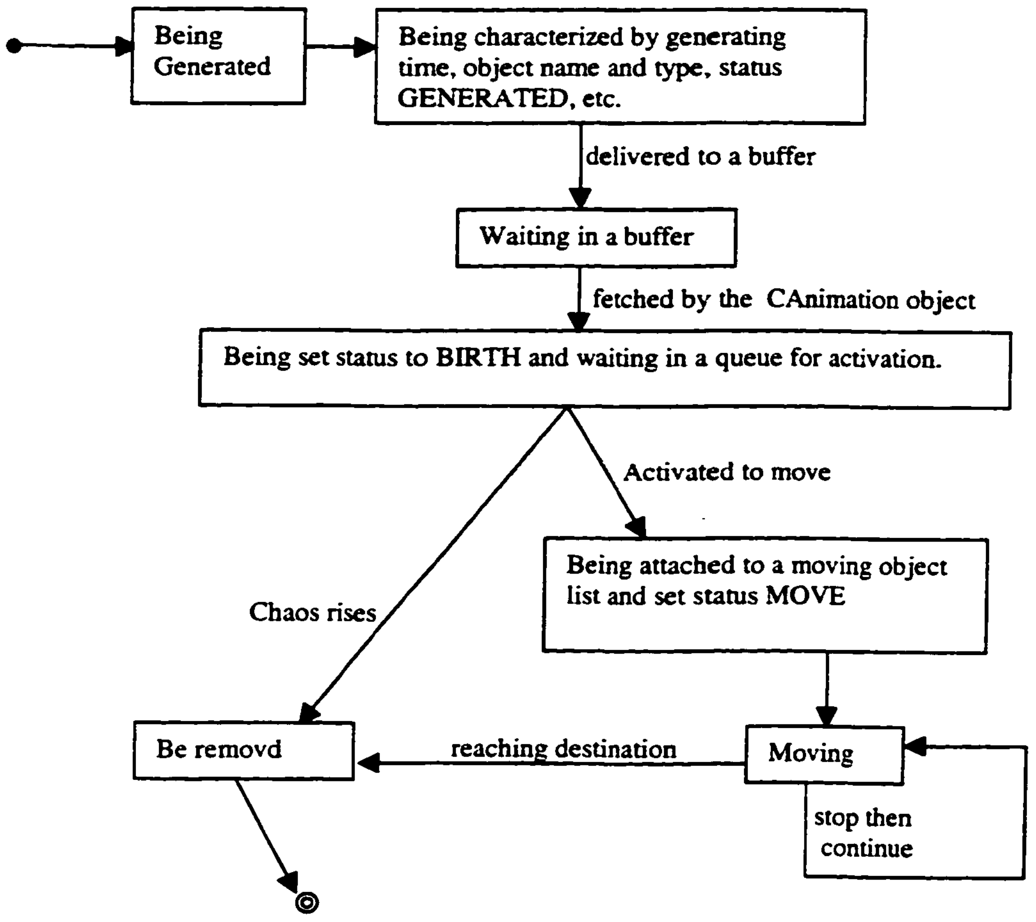

Being Generated
Being characterized by generating time, object name and type, status GENERATED, etc.
delivered to a buffer
Waiting in a buffer
fetched by the CAnimation object
Being set status to BIRTH and waiting in a queue for activation.
Activated to move
Being attached to a moving object list and set status MOVE
Chaos rises
Be removd
reaching destination
Moving
stop then continue

Figure 3.9 A state diagram of life cycle of a moving object. The detail of the state Moving will be further drawn into two other figures. One is of pedestrian moving state. The other is of vehicle moving state.

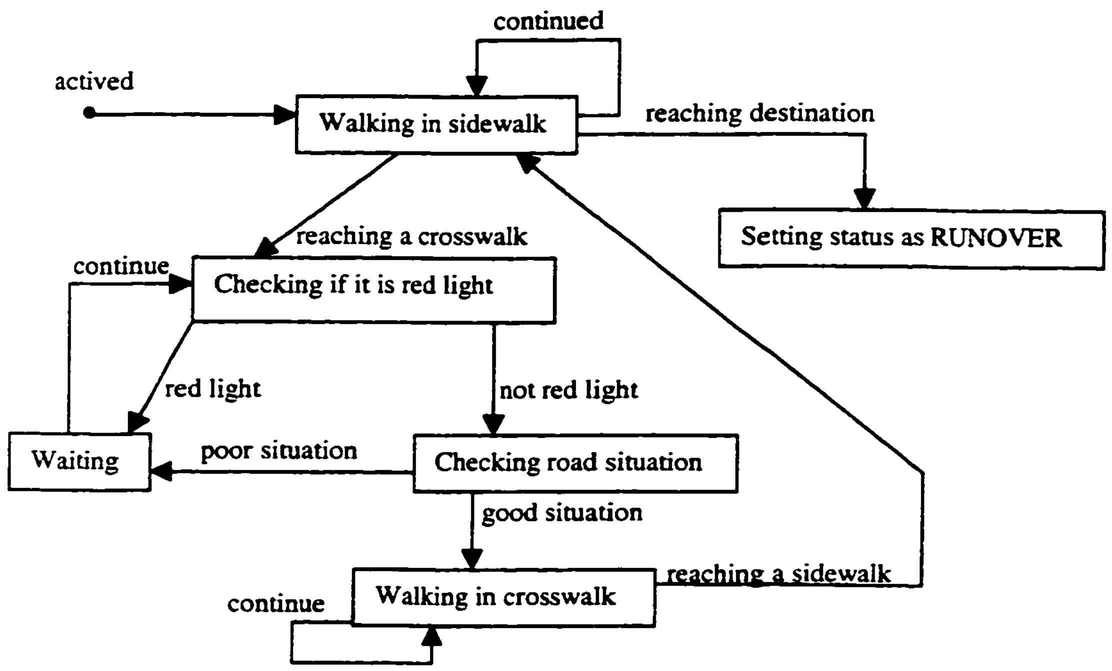


Figure 3.10 A state diagram that explains the detail of the state Moving in figure 3.9 for a pedestrian object.

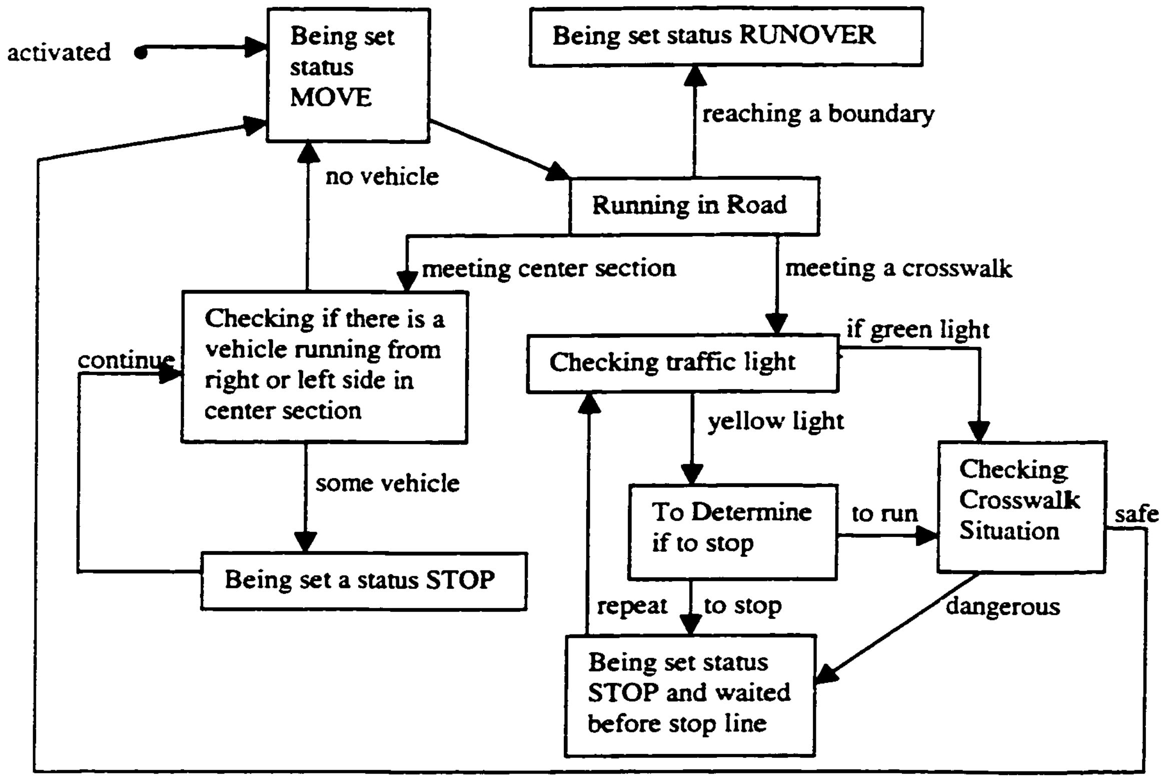


Figure 3.11 A state diagram of the detail of the state Moving in figure 3.9 for a vehicle object.

Chapter 4

Implementation of the Simulation

In this Chapter, we will describe the implementation of the design described in Chapter 2. Concentrations are put only on those important classes and their member functions.

4.1 Introduction

From the viewpoint of programming, the implementation could be divided into four modules. The first module, the *User Interface Group*, concerns the classes CMajorRepView, the CSettings, the CMajorRepDoc, the CAnimation, the CLightCtrlSys (the traffic light controller system) and the CReport.

The second module, the *Animation Group*, includes the classes CAnimation, the CEngine (the simulation engine), the CTSSTDB, the CMovingObj and the CMovObjCtrlSys (the moving object generation controller).

The third module, *Simulation Engine Group*, includes the classes CEngine, the CLightCtrlSys, the CSettings and CAnimation.

The fourth module, the *Database Group*, only includes class CTSSTDB, which will be discussed solely in Chapter 6.

Recall that the traffic light system is an independent control system. A work thread is applied to simulate the traffic light controller. Multithreading method will be discussed in Chapter 5.

4.2 Implementation of the User Interface

As mentioned above, the *User Interface Group* concerns those classes relative to the user commands discussed in section 3.3. This section will illustrate the details of some functions.

4.2.1 The Class CMajorRepView

The class CMajorRepView is derived from the MFC member class CView. The following functions react immediately to the user commands.

- The function OnProjectNew() is to create a new simulation project.
- The function OnProjectOpen() is to open a saved simulation project.
- The function OnProjectSave() is to save the current simulation project. If the project has not been assigned a title, it will invoke the OnProjectSaveAs().
- The function OnProjectSaveAs() is to save a simulation project with a new title.
- The function OnProjectSettings() invokes the CSettings object to set up a simulation environment.
- The method OnBuildRun() will run a simulation.

- The method OnBuildPause() will make a running simulation a pause.
- The method OnBuildStop() will terminate a running simulation.
- The method OnBuildContinue() will resume a paused simulation.
- The operation OnAnimationPlay() sets up the type of a animation as PLAY.
- The operation OnAnimationReplay() sets up the type of a animation as REPLAY.
- The operation OnAnimationReverse() sets up the type of a animation as REVERSE.

4.2.2 The function CMajorRepView::OnBuildRun()

The function OnBuildRun() initializes the execution of a simulation by calling the member function toExeSimulation() of the class CAnimation. Then, it launches the *timerThread* and the *trafficLightThread*. After that, it calls the pDC() and the OnPrepareDC() to prepare the client paint device context.

4.2.3 The Operation CMajorRepView::OnPaint()

Screen flicker is a severe problem in animation. To overcome this problem, we added code to handle WM_PAINT messages directly. Only the CView class in MFC implements the OnDraw(), so we override the function OnPaint() to improve the quality of the simulation through the use of a memory device context. The animation of the TSST system uses GDI (Graphics Device Interface) bitmaps. GDI bitmap objects are represented by the MFC Library version 4.2.1 CBitmap class.

We cannot select a bitmap into a display device context in the program of OnDraw(). Thus, a special memory device context is created for the bitmaps using the CDC::CreateCompatibleDC() function in CMajorRepView::OnInitialUpdate() function. This program generates its own bitmap to support the smooth motion on the screen. The principle is: drawing on a memory device context with a selected bitmap and zapping the bitmap onto the screen <Ref.4>. Then, every time when OnPaint() is called, it prepares the memory device context for drawing, passes function OnDraw() a handle to the memory device context, and copies the resulting bitmap from the memory device context to the display.

The function OnPaint() performs in order the following three steps to prepare the memory device context for drawing:

1. Select the bitmap into the memory device context.
2. Transfer the invalid rectangle from the display context to the memory device context. The functions CDC::SelectClipRgn() and CDC::IntersectClipRect() are invoked to minimize the clipping rectangle otherwise the program may run much slower.
3. Initialize the bitmap to the current window background color. The CDC::PatBlt function fills the specified rectangle with a brush pattern. The brush is constructed first and selected into the memory device context.

After the memory device context is prepared, OnPaint() calls OnDraw() with a memory device context. Then the CDC::BitBlt() function copies the updated rectangle from the memory device context to the display device context.

4.3 The Class CSettings

Two points need to be described briefly here. One is that the class CSettings is defined as a pure static class, i.e. its all data and function members are declared as static. This definition permits other objects, such as the CEngine object, to access the function members by using the fully qualified class syntax.

For example, the data member nsLanes represents the number of lanes southbound. The function member theNSLanes() returns the address of nsLanes. An object could directly call theNSLanes() to access the nsLanes as follows:

CSettings::theNSLanes().

The second point is that the member function toOpenSettingsSheet() of the class Csettings. It pops up a property sheet that is composed of the CLightPage, the CLanePage, the CVehiclePage and the CPedestrianPage objects. A user could modify a simulation environment through these property sheet pages. Default values of a simulation environment are provided.

4.4 Properties Moving Objects

In this section, we will introduce the characteristics of moving objects.

4.4.1 Thc class CMovingObj

The class CMovingObj is a base class which describes common features of moving objects including the object type (m_objType), the object name (m_objName), the object status (m_objStatus), the object index (objIndex) and the generating time (startTime). Note that the terms in the parenthesis are its data members. It also has some virtual functions to be overridden by its subclasses.

To make a link list of moving objects, the CMovingObj has a data member, called the *next*, of type CMovingObj.

4.4.2 The class CPedestrian

The class CPedestrian is the subclass of the CMovingObj, which stores unique information of the pedestrians. Information includes the current position (m_currPos), the next position (m_nextPos), the origin (m_orig), the destination (m_dest) and the body size (m_Size). Terms in the parenthesis are the data members of the CPedestrian.

The operation toPrepareForDraw() is responsible for preparing the drawing of a pedestrian. It first checks the pedestrian's status. If the status is 'Birth', it registers the pedestrian's current and next positions. Then, another operation draw() will draw the registered next position of the pedestrian.

The classes CAdult and CSenior are subdivided from the class CPedestrian. In addition to the inherited features from their parent class, they also carry their unique maximum step lengths and walking speeds.

4.4.3 The class CVehicle

Similarly, the class CVehicle is derived from the class CMovingObj and has its own two subclasses, the CCar and the CTruck.

The class CVehicle contains the data member m_laneNum for the number of lane; the m_Dir for the driving direction; the m_currPos for a vehicle's current position; the m_nextPos for the next position. It also overrides the toPrepareForDraw() of its parent.

4.5 Implementation of the Simulation Engine

The key function of the CEngine is to calculate the next steps for all moving objects based on the assumptions, constraints and calculation algorithms. The assumptions and the constraints have been explained in section 3.2.5. In this chapter, we will describe the algorithms and the most important function, the toCalNextStep(). To depict a clearer picture, we will use examples to describe the relative algorithms. Figure 3.1 illustrates the abbreviated terms invoked in following sections.

4.5.1 Algorithms

a) The algorithm of calculation the next step of a vehicle (using an example).

- Assume that a vehicle, called the GMCV, is running northbound in lane 1.
- Before the GMCV appears, it checks if there is some space to show at least its hood. If space is available, it changes its status to 'Birth'.
- The moving GMCV will check that the distance between it and the rear of the next vehicle in the same lane does not fall below a stipulated minimum.
- When approaching the South Stop Line, the GMCV vehicle checks if there is any pedestrian in the area SCW. Then it checks the traffic light color. It will stop before the South Stop Line when in the following situations: 1) there is at least one pedestrian walking inside the area SCW, 2) it decides to stop when the light is red or yellow.
- Before entering area CS, the GMCV checks if any vehicle in area CS moves westbound or eastbound.
- When approaching area NCW, the GMCV checks if any pedestrian is walking in area NCW.
- When the GMCV has crossed the north boundary, its status changes to RunOver, then it is removed from the moving object list.

b) The algorithm of calculating the next step of a pedestrian (using an example).

- Assume a pedestrian, called the AMAN, is moving from NW to ES.
- His status will change to 'Birth' if there is space for him to enter the NW.

- The AMAN always checks if there is any moving objects blocking his way forward. If he is being blocked, he will check his right to see if he could move one step to the right. If not, he will then check his left. If it is impossible to move, he stays.
- When arriving at area NWC, he chooses either NCW or WCW as his next walking area. Assume he determines to walk across NCW.
- Before he steps in the NCW, he checks the traffic light color first. If the light is green, he checks if there is any vehicle crossing NCW. When the situation is safe, he walks across NCW.
- After arrived at NEC, he checks the traffic light and the road situation again to determine when to cross ECW.
- When arrived at SEC, he walks to his destination ES.
- After crossing the East Boundary, he changes his status to RunOver and is removed from the moving list.

c) Recall that we use a timer to control the time gap of drawing moving objects. The gap is DTIME milliseconds. In other words, the TSST system draws moving objects 1000/DTIME times per second. Thus, the maximum step length of a vehicle per drawing is calculated by the following formula:

$$\text{maxVepLen} = \frac{\text{vehicleSpeed} * \text{DTIME}}{\text{dSecond}} * \text{FACTOR}$$

where dSecond = 3600 milliseconds.

d) Similar to the explanation in above c), the maximum moving length of a pedestrian per drawing is calculated by the following formula:

$$\text{maxPedLen} = \frac{(\text{steps/min}) * (\text{length/step}) * \text{DTIME}}{\text{dSecond}} * \text{FACTOR}$$

where dSecond = 60*1000 milliseconds.

Where, in c) and d), the unit of DTIME is millisecond. The unit of the maxPedLen and the maxVecLen is pixels per DTIME milliseconds. The FACTOR is predefined by the programmer, which maps centimeters to pixels.

4.5.2 The Function toCalNextStep()

In brief, the function toCalNextStep() is to search all pedestrians and vehicles from moving object lists and to calculate their next positions. Before storing a new position to the m_nextPos, the m_nextPos' value is copied to the m_currPos. The toCalNextStep() calls the toCalPedNextStep() and toCalVehNextStep() to calculate the next steps of pedestrians and vehicles respectively.

The toCalPedNextStep()

A moving pedestrian always moves from one area to another or otherwise to his destination. Thus it checks a pedestrian's current area and destination to determine the next area the pedestrian will move to. When the next area is given, it restricts to check the objects' positions in the two 'current' and 'next' areas. This saves the time in searching for object lists.

We would like to use an example to explain further the function toCalPedNextStep(). Assume that a pedestrian called AMAN, is to move from SE to NW. His route could be one of the following two paths:

Path1: SE $\Rightarrow$ SEC $\Rightarrow$ ECW $\Rightarrow$ NEC $\Rightarrow$ NCW $\Rightarrow$ NWC $\Rightarrow$ NW, or

Path2: SE $\Rightarrow$ SEC $\Rightarrow$ SCW $\Rightarrow$ SWC $\Rightarrow$ WCW $\Rightarrow$ NWC $\Rightarrow$ NW.

Assume AMAN chooses the path1.

- When his current area is SE or ECW or NCW, he always invokes the function moveToWSCS() to calculate his next position till he reaches SEC or NEC or NWC respectively.
- When he is in area SEC or NEC, the function moveToCRS() is called for calculation.
- When arrived NWC, he calls the function moveToWSS() to determine his next step.
- While being in NW, his destination is the North Boundary, so the function moveToBoundary() is applied till he disappears.

The toCalVehNextStep()

The primary idea here is to obtain the maximum steps in different situations, then find the min step from them. The list below represents the steps of the toCalVehNextStep().

1. It checks the moving direction of a vehicle because different moving directions determine different procedures for calculating next steps.

2. It gets the rear position of the vehicles in front. This is to keep two vehicles between a minimum distance.
3. When a vehicle has crossed the stop line, it ignores the light commands and keep moving forward. Two situations will happen as follows:
   - When the hood of a vehicle has already in center section, it only checks the front crosswalk. If there are pedestrians, it calculates its maximum step forward but not crossing the crosswalk.
   - If the hood of a vehicle is already in the crosswalk just beyond the stop line, it checks the center section. If some vehicles inside CS moves from right or left side, it calculates its maximum step forward but not crossing the center section

4 When a vehicle has not crossed the stop line, we will consider the following cases:

- If there are pedestrians in the front crosswalk, it calculates its maximum step to the stop line.
- If there is no pedestrian in the front crosswalk, before entering the crosswalk, a vehicle has to consider the traffic light commands.
  - ➢ If GREEN light, the vehicle calculates its maximum step without constraint.
  - ➢ If YELLOW or RED light, the vehicle must determine either to stop before the stop line or to run red/yellow light. Based on its choice, then it calculates its maximum step.

5 Among those "maximum" steps above, a vehicle chooses the min step as its next step and calls the function renewPosition() to save the step to its data member m_nextPos.

4.6 Implementation of the Animation

In the *Animation Group*, the class CAnimation is another interesting point to describe. It has several pointers of type CMovingObj and fetches the moving objects either from some buffers or from the database.

The class CAnimation has a public member function, called the toExeSimulation(), which calls an appropriate method for a simulation run according to the user's selection. The method could be thePlay() or theReplay() or theReverse().

4.6.1 The method thePlay()

The function thePlay() is responsible for the real time simulation. Once invoked, it checks if the current simulation project has started. If not, it empties the moving object lists by invoking the toClearObjList() and launches two work threads to generate pedestrians and vehicles. Then it opens the database TSSTDB and calls the member function toCleanSteps() of CTSSTDB to delete all data from TSSTDB. After that, it returns.

When the thePlay() is called during a simulation run, it first fetches the moving objects from the BufferOne for pedestrians and from the BufferTwo for vehicles. Then it calls the operation toCalNextStep() of the CEngine object to calculate the next steps of all moving objects. Finally, it calls the member function toSaveSteps() of CTSSTDB to save those new calculated next steps to the TSSTDB database and returns.

4.6.2 The method theReplay()

Different from thePlay(), the member function theReplay() does not launch any work threads and does not call the toCalNextStep() to calculate next steps of the moving objects. Instead, it fetches the steps of moving objects in descending order from the TSSTDB database by calling the method theRetrieve() of the CTSSTDB object. Before updating the moving object lists, it calls the method checkLists() to remove all run-over objects from the moving object lists.

4.6 .3 The method theReverse()

The only difference to theReplay() is that the operation theReverse() fetches the next steps of moving object in ascending order from the TSSTDB database.

Chapter 5

Multi-Threads

Win32 and MFC support multiple threads within a process. When an application starts, it has one primary thread. The application may then start up additional threads that execute independently. All threads share the one memory space of the process in which they are created. When an application has multiple threads, the order in which they are executed is a random one.

We used the multithreading method to simulate the real time situations. In this chapter, we will explain why we use threads and how the threads are applied.

5.1 Motivation for Threads in Simulation

According to the requirements in Chapter 3, the work of the traffic light control system, of generating pedestrians and of generating vehicles could be hidden in background. In the next two subsections, we could see that multithreading will satisfy our requirement.

5.1.1 What Is A Thread

Mike Blasaczak in his book "MFC with Visual C++ 5" mentioned the thread as follows:

The term *thread* is shorthand for 'a thread of execution', and it represents the most fundamental information a running program needs while it is executing: a user-mode stack to hold temporary variables and return addresses for subroutines, a kernel-mode stack to hold addresses for interrupt service returns, and a set of processor registers. <Ref.[1]>

A process is a running program that owns its own memory, file handles and system resources. A thread is a separate execution path contained in an individual process. In other words, threads are managed by the operating system and each thread has its own stack.

MFC (Microsoft Foundation Class) implements two types of threads. One is called *user-interface thread.* A user creates such a thread to deal with parts of a user interface. If a user is interested in creating a thread that simply goes off on its own and does something else, such as background calculations, without interfacing with the user, *worker thread* can be invoked. These threads are based on *CWinThread.*

5.1.2 Using Worker Threads to Do Background Work

Worker threads are handy any time when a user wants to do something such as calculations or background printing. To get a worker thread up and running, a user implements a function that will be run in the thread, then creates the thread with *AfxBeginThread()* which returns a pointer to the newly created CWinThread object. The new thread will continue to execute the function specified until that function returns, or until the *AfxEndThread()* is called from within the thread. The thread will also terminate if the process in which it is running terminates. Each thread may have its own priority.

A Windows message is the preferred way for a worker thread to communicate with the main thread and the main thread always has a message loop. This implies, however, that the main thread has a window and that the worker thread has a handle to that window.

5.2 Implementation of Worker Threads in the Simulation Tool

The TSST simulation tool uses worker threads to simulate the real time system. These worker threads are the timer thread, the thread of traffic light control system, the thread of pedestrian generation and the thread of vehicle generation.

5.2.1 The Timer Thread

The timer thread is launched when users click on the Run item from the menu bar. The purpose of using a timer thread is to periodically notify the CMajorRepView object that it is time to redraw the moving objects stored in the CAnimation object. The timer thread is initialized in the function OnBuildRun() which is a member of the CMajorRepView, and pointed by a pointer *timerThread* of type CWinThread. The *timerThread* is the data member of class CMajorRepVeiw. By the *timerThread*, the parent of the timer thread could send it some requests, such as *SuspendThread* or *ResumeThread*.

We provide a controlling function, the timerCtrlSysThread(), to implement the timer thread. The timerCtrlSysThread() simply consists of an inner loop and the function Sleep(). Every DTIME milliseconds it wakes up and posts a message to the CMajorRepView object. Here the DTIME represents the predefined time gap between two times of drawing moving objects.

5.2.2 The Thread of Traffic Light Control System

In real life situation, the durations of traffic lights are predefined and independent of drivers and pedestrians. A worker thread is created for this purpose and named as the light thread pointed by the lightCtrlThread of type CWinThread. Its controlling function is the trafficLightThread().

Similarly to the time thread, the trafficLightThread() has an inner loop. Within the loop, there are three Sleep() functions and three PostMessage() functions. Each time the lightThread wakes up, the global variable, the *lightCol*, will be assigned a different color in order of Green, Yellow and RED, and then the function PostMessage() is invoked to notify the CMajorRepView object that the traffic light color needs to redraw.

5.2.3 The Threads of Moving Objects

There are two worker threads responsible for generating pedestrians and vehicles respectively. One is pointed by the *vehGenThread*, the member of the CPedestrian; and another is pointed by the *pedGenThread*, the member of the CVehicle. These two worker threads are launched when the member function thePlay() of the CAnimation is called for the first time to start a simulation run.

The controlling function vehicleCtrlSysThread() periodically generates vehicles then appends them to the buffer BufferOne. Within its while loop, the function generateNewVehicles(), followed by the Sleep() function, is called to generate a pedestrian. The sleeping time depends on the rate of generating vehicles. The function generateNewVehicles() is a member of the CVehicleCtrlSys class which is inherited from CObject.

The class CVehicleCtrlSys contains all necessary information on generating a vehicle including the numbers of lanes and the probabilities of generating vehicles in different directions, the rate of generating vehicles, etc..

We do not describe the controlling function pedestrianCtrlSysThread() and only brief the class CPedestrianCtrlSys here. The CPedestrianCtrlSys is inherited from CObject and plays the role of pedestrian generation. It includes all information input by the user, decides on what type of a pedestrian to be generated and determines the origin and destination where a pedestrian will follow.

5.3 Message Handling Function

A thread class may implement a message map just as for any other classes derived from *CCmdTarget*. However, a user may also use a special message map macro, ON_THREAD_MESSAGE, to handle messages that are sent directly to the thread rather than to a given window. A user can send messages directly to a thread as follows:

```
        pMyThread-> CwinThread::PostThreadMessage()( , , );
```

This is similar to the ::PostThreadMessage() API call. We have seen this API call above. How is the message handling implemented in this simulation tool?

Recall that the *::PostThreadMessage()* API call is invoked in the threads to post messages. It dispatches a message to the receiver according to the input parameter. The value of the input parameter is predefined. We define the values as follows:

```
        #define LIGHTCTRLMSG            WM_USER + 5
        #define TIMERCTRLMSG        WM_USER + 6
```

The two predefined values connect to message member by a pair of message map macros, the BEGIN_MESSAGE_MAP and the END_MESSAGE_MAP. The message map is a way provided by MFC to associate Windows and user messages with the functions that handle them. Within this pair of macros, we can use several other macros to indicate exactly what messages we will map. The macros take care of the actual definition of the data structures and functions. For instance, in our programming two message maps are defined between the macros as follows:

```
        ON_MESSAGE(LIGHTCTRLMSG, lightCtrlMsg),
```

ON_MESSAGE(TIMERCTRLMSG, timerCtrlMsg).

Here, the *ON_MESSAGE* is used instead of a special message map macro *ON_THREAD_MESSAGE.*

The two parameters *lightCtrlMsg* and *timerCtrlMsg* are two member functions defined in class *CMajorRepView*. Every time when a worker thread posts a message to its main thread, one of these functions is invoked by the main message loop. The function *lightCtrlMsg()* calls the member function *drawLights()* of class Background to redraw the colors of the traffic lights. The function *timerCtrlMsg()* calls the member function *OnPaint()* to redraw all vehicles and pedestrians.

5.4 Synchronization

Synchronization is a very important aspect of thread programming. Synchronization objects are a collection of system-supplied objects that allow threads to communicated with one another. There are four such objects in Windows: critical sections, semaphores, mutexes, and events. All of these objects have different patterns of initialization, activation and use, but they all eventually represent one of two different states: signaled or unsignaled. Except for critical sections, all of these objects are waitable. <Ref. [1]> In this simulation tool, critical sections are used to synchronize different objects.

Two synchronization objects of type CCriticalSection are defined. One is the m_sinLockOne defined in file vehicleCtrSys.cpp. After generating a new vehicle, the

CVehicleCtrSys object invokes its member function AddToBuffer() to append the newly generated moving object to the BufOne. To protect interruption, the AddToBuffer() calls the function Lock() to block other threads to access the BufOne. Then the new generated vehicle objects are attached to the buffer BufOne. When the attachment is done, the function Unlock() is called to release the synchronization object m_sinLockOne.

On the other hand, the CAnimation object is to fetch the moving objects from the BufOne by calling its member function getObjFromBufOne() which in turn calls the m_sinLockOne.Lock(). If the BufOne is currently blocked, the CAnimation object will wait until the resource is released. Otherwise, the CAnimation object accesses the blocked BufOne and fetches the moving objects then releases the synchronization object.

Another synchronization object, defined in file pedestrianCtrSys.cpp, is the m_sinLockTwo. Similar to the description above, the CAnimation object and the CPedestrianCtrSys object will block the BufTwo when they are to access the buffer. Since the action of m_sinTwo.Lock() is very similar to m_sinOne.Lock(), we will not describe it in detail.

Chapter 6

Using Database

6.1 Motivation for Using Database in this simulation tool

We have mentioned several times in previous chapters that the Simulation tool uses a database to realize the replay and the reverse of a simulation run. The reason of using a database is that this simulation tool is a real time system, each step of a moving object is dynamically calculated in the run time. Since multithreading is applied, we could imagine that the time slots might not be distributed to all threads evenly. In other words, the same simulation environment could result in different simulation runs. To exactly repeat a simulation run, we use a database to store all moving objects and all their steps for retrieval later.

6.2 The Tables and the Relationships

We use the database tool Microsoft Access to build our database project named TSSTDB (see section 3.2.6). The MS Access is a small database tool that makes it is easy to generate and to maintain a database project. After building up the TSSTDB, we need a driver to access the data source. The ODBC driver manager is invoked to register the TSSTDB so that our application can access an open database.

We create two tables, named Objects and ObjSteps as follows:

Objects

ObjID	ObjType	ObjName	ObjBirthTime

ObjSteps

StepID	ObjID	CoorX	CoorY

The primary key of the table Objects is the ObjID. Each moving object has a unique ID. A sequence of steps of a moving object is retrieved by checking the object's ObjID. The key of the table ObjSteps is (StepID, ObjID). An object could stay at a place for a few seconds so its CoorX and CoorY could keep unchanged, however its StepID changes every time it is drawn.

6.3 The class CTSSTDB

To implement database under Visual C++ and Windows NT environment, we use the MFC ODBC classes. ODBC defines a standard set of functions for data access and carries a specification for which vendors can write drivers that grant the application access to almost any of the database currently available. ODBC specifies that each driver must implement a standardized set of SQL keywords, setting on a very specific syntax for its SQL queries. [] In short, ODBC is the interface between the database TSSTDB and the TSST system.

By using ODBC, we use objects instead of connection handles and statement handles. The two principal ODBC classes are CDatabase and CRecordset. Objects of class CDatabase represent ODBC connections to data source and objects of class DRecordset represent scrollable rowsets. The class CTSSTDB is derived from CRecordset to match the columns in tables Objects and ObjSteps.

The class CTSSTDB has three major member functions: toSaveSteps(), toCleanSteps() and toRetrieveSteps(). We will describe them in the following sections.

6.3.1 The Functions toSaveSteps() and toCleanSteps()

In section 4.5.1, we have described the operation thePlay(). When thePlay() is invoked the first time, it calls the CTSSTDB::Open() to open the database TSSTDB and then calls the toCleanSteps() to remove all data from the TSSTDB.

The function toCleanSteps() executes a loop, calling Delete() to delete a database record during each iteration. The loop terminates when the database is empty. When thePlay() is called during a simulation run, it calls CTSSTDB::toSaveSteps() to save all newly calculated steps.

In turn, toSaveSteps() calls the CTSSTDB::AddNew() to begin the process of adding a new record to the record set. Then it sets those member variables of CTSSTDB object as appropriate. Finally it calls CTSSTDB::Update() to complete the saving process.

6.3.2 The Function toRetrieveSteps()

Different from the function thePlay(), the function theReplay() and function theReverse() call the member function toRetrieveSteps() of the CTSSTDB to retrieve data from the database TSSTDB.

The toRetrieveSteps() has a parameter which passes the step number to variable StepID. It, with the StepID, fetches the appropriate data from table ObjSteps simply by using an inner loop and calls the appropriate variables of CTSSTDB. The second argument of the toRetrieveSteps() is a pointer of type CMovingObj, which points to the moving object list stored in CAnimation object. The newly fetched steps are inserted into the appropriate moving objects. If a newly fetched ObjID does not appear in the list, the moving object with this ObjID will be fetched from table Objects and be appended to the moving object list. The m_strFilter member of CTSSTDB object is invoked to join tables Objects and ObjSteps.

Finally, we would like to point out that the database TSSTDB and the class CTSSTDB should be greatly improved if the Report part is to be added into the TSST system. This is because different reports may requests different data set. Retrieving sufficient data set from the database TSSTDB can largely save memory space and enhance execution speed. Another reason is that to develop a new report may request to generate a new relational table in the TSSTDB and to create a new function in class CTSSTDB. Since the TSST is an open system, many new features and functions can be developed. However, we cannot implement all of them in limited time. So we leave them for future work.

Chapter 7

Results from Simulation

In this Chapter, we will discuss some results from the simulation runs. Since the replay and reverse of a simulation run do not influence the results, we concentrate on playing simulation runs. The simulation runs include the one without pedestrian, the one without vehicle and the one with both pedestrians and vehicles.

7.1 Simulation without a Pedestrian

Simulation 1. A user can simulate a road situation without a pedestrian. To do this, a user should unmark the Adult and the Senior check boxes in Pedestrian Page. The following is the settings of the simulation:

	North to South	South to North	West to East	East to West
No. of Lanes	3	2	1	0
Probability of having a vehicle	70%	50%	30%	0%

	Car	Truck
Selection	√	√
Speed (KMPH)	70	50

The percentage of cars among vehicles is 50. The maximum number of vehicles generated per minute is 80.

The duration of traffic light is defined below:

Direction	North – South	West – East
Duration of green light	2000	3003
Duration of yellow light	1500	2000

The probabilities of a vehicle running a red light and a yellow light are set 10% and 50% respectively.

During a simulation run, we could see that most vehicles stopped before stop lines when the light was red except for a few vehicles ran a red light. And many vehicles ran yellow light. The reason is that even when a vehicle ignored a red or yellow light, it could not neglect a possible running vehicle at the center section. Thus, most of them were forced to stop before the stop line. A few of collisions occurred.

Simulation 1.2. If the probability of running a red light changed to 50%, we found that more vehicles ran the red light and the number of collisions increased obviously. Some vehicles waited at a crosswalk just before entering the center section where vehicles had already entered from the left or right side.

Simulation 1.3. Finally we changed the probability of running a red light to 0 and that of running a yellow light to 100%, we found that no collision occurred. In this simulation, running a yellow light did not cause serious trouble except increasing traffic jams.

7.2 Simulation without a Vehicle

Simulation 2. To simulate situations without a vehicle, a user should unmark the check boxes of Car and Truck in the Vehicle Page. The settings are as follows:

	North to South	South to North	West to East	East to West
No. of Lanes	3	1	2	0
Generate a vehicle	70%	50%	60%	0%

	Adult	Senior
Selection	Yes	Yes
Speed (MPH)	130	55

The percentage of adults among pedestrians is 51%.

The maximum number of pedestrians generated per minute is 50.

The probabilities of a pedestrian from different origins to different destinations are defined as follows:

	NW	NE	SW	SE	WN	WS	EN	ES
Origin	100	0	0	0	0	0	0	0
Destination	0	0	0	0	0	0	0	100

The duration of traffic lights in different direction are set up as below:

Direction	North & South	West & East
Duration of green light	2000	3003
Duration of yellow light	1500	2000

Based on the settings, all pedestrians originate only from the NW and moved only to the ES. There were two paths that a pedestrian could follow:

1. NW->NWC->NCW->NEC->ECW->SEC->ES.
2. NW->NWC->WCW->SWC->SCW->SEC->ES.

All the abbreviations above are given in figure 3.1. When arriving at NWC, a pedestrian either turned left to NCW or walked straight to the WCW. We found that the pedestrian's decision matched our design:

- If the east light was green, the pedestrian walked across NCW.
- If the south light was green, the pedestrian walked across WCW.

We also found that while a pedestrian is walking at NCW and the east light changed to red, the pedestrian continued ahead to NEC.

Simulation 2.1. Next, we changed the origin-destination settings as follows:

	NE	NE	SW	SE	WN	WS	EN	ES
Origin	10	10	10	10	10	10	20	20
Destination	0	0	0	5	5	10	20	60

We could see that pedestrians appeared from every origin but no pedestrian walked to NW, NE and SW sections. When two pedestrians met, they tried to give way to each other. When they had no way to step aside, they stayed and waited.

7.3 Simulation with Vehicles and Pedestrians

The simulations mentioned above worked well. Now, we proceed with a combination of Simulation 1 and Simulation 2, named Simulation 3.

7.3.1 Chaos from a Poor Design of a Crossroad

Simulation 3 ran well for the first three minutes or so. Then came the traffic congestion. After another five minutes, it seemed that Simulation 3 had run out of all internal resources. The animated display showed a very messy situation. Performance also went down.

One of the reasons was that too many pedestrians were walking on crosswalks and too many vehicles were waiting at the center section or crosswalks. In the last five minutes of the traffic jam, with all the moving objects in their active status, a maximum of 400 vehicles and 250 pedestrians were found on the crossroad. Moreover, the shorter duration of green lights was hindering vehicles from driving away quickly from the center section.

7.3.2 An Improved Design

A user could improve a design of a crossroad in different ways according to real situations otherwise a design could be distorted.

Simulation 3.1 was a modification from Simulation 3, its maximum number of vehicles generated per minute and number of pedestrians were reduced half to the 40 and 25 respectively. In this case, we assume that numbers of lanes in different directions were predefined. After running the simulation five minutes, we saw that the situation has been eased out and there was no more traffic congestion. Thus we concluded that the crossroad design in Simulation 3.1 was better than that in Simulation 3.

If we assumed that the maximum number of vehicles generated per minute and number of pedestrians in Simulation 3 were predefined. Then we modified the number of lanes in certain directions and prolonged the duration of the green light signal. This modification is called as the Simulation 3.2.

Simulation 3.2. This last simulation was modified from Simulation 3 as follows:

	North to South	South to North	West to East	East to West
No. of Lanes	6	4	4	4
Generate a vehicle	70%	50%	50%	50%

Direction	North & South	West & East
Duration of green light	10000	10000
Duration of yellow light	3000	3000

This simulation worked well without serious traffic congestion and the design of the crossroad was fair. There are many criteria for a good design but detailed discussions are beyond the scope of this report.

7.4 Problems

A serious problem had occurred when the traffic condition in a simulation became chaotic. In such case, we could see that more and more vehicles jammed at the center section, crosswalks and all lanes whereas, at the same time, an increasing number of pedestrians were either waiting or walking on the sidewalks. A user could not stop a chaotic simulation by clicking on item STOP from the main menu bar. Thus, we added a detector to check if the waiting queue of the moving objects is too long. If it is too long, a message dialog pops up to let users suspend all work threads and then terminate the chaotic simulation.

Chapter 8

Conclusion

8.1 Advantages of Simulation

The TSST simulation tool provides users an alternative to analyze their crossroad design. With this tool, a user could easily modify the different settings of a construction project. By the functionality of replay and reverse of a simulation run, users could compare different designs and could repeat a simulation in ascending or descending order. The greatest advantage of the TSST system is the property of simulating real time situations. Many unexpected situations could happen based on the settings obtained from the statistical results of the real world.

From the viewpoint of a programmer, the TSST system still has a lot of room for future development because this simulation tool is designed in Object-Oriented Methodology and implemented in C++ language. For example, we might, instead of using multi-threads to simulate a real time system, use a network system to transfer real data obtained directly from radar and the traffic light system. Connection to a real world will be enormously helpful for analyzing a constructed crossroad.

8.2 Disadvantages of Simulation

The TSST system has too many constraints on performance. Restrictedly speaking, the TSST system currently only simulates part of real time system. For instance, in simulation, it is reasonable to permit a vehicle to turn left or to turn right or to change lanes. Keeping the situation as close to reality as possible might largely increase the complexity in programming.

Another disadvantage is that, in the TSST system, we use 'symbols' to represent the different moving objects. This may help in reducing animation flicker but make the animation less vivid. It is possible to use some images, such as a car, a truck, an adult or a senior instead of 'symbols' because nowadays the speed of the CPU is fast enough and the memory is large enough for running a simulation.

8.3 Future Work

A lot of new features could be added in the future. The most interesting one is to add the report generation feature. Although the TSST tool offers a visible simulation result, a statistical report will be more helpful in analyzing the design of a crossroad.

As mentioned in section 8.2, the interface needs to be improved in many fields. We could add a simulation control bar to quickly, for example, stop or pause a simulation run.

To repeat a segment of a simulation run, we could add a field to the database table ObjSteps. The new field will record the running time of the next step. By then, a user could select a time interval and repeat a simulation within that time gap.

In general, to improve the quality of the TSST system, a programmer could always develop some new features into the system.

Appendixes

Part A. The Class CEngine

```
//////////////////////////////////////////////////////////////////////
// Engine.h: interface for the CEngine class.
//////////////////////////////////////////////////////////////////////


#if
!defined(AFX_ENGINE_H__4452DFF9_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED_)
#define AFX_ENGINE_H__4452DFF9_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "MovingObj.h"
#include "Vehicle.h"
#include "Pedestrian.h"

class CEngine {

COleDateTime m_currTime;
CPoint xxRearArr[4][8];

public:

CEngine();
virtual ~CEngine();

void toCalNextStep();
void toCalPedNextStep();
void toCalVehNextStep();


/*
to sort pedestrian list according to their current positions:
sidewalk sections : NWS, NES, SWS, SES, WNS, WSS, ENS, ESS
intersected sidewalk sections : NWC, NEC, SWC, SEC
crossroad sections : NCRS, SCRS, WCRS, ECRS
*/
void sortPedList();
void sortWSSlist(CMovingObj** WSSlist);
void sortWSCSlist(CMovingObj** WSCSlist);
void sortCRSlist(CMovingObj** CRSlist);
void sortUnIssuedPedList(CMovingObj** unIssuedPed);
```

```
/*
called by above sorting functions.
*/
void addToList(CMovingObj** listOne, CMovingObj* listTwo);

/*
to check road situation :
if situation & 1 > 0 : some vehicle from N2S across the center section;
if situation & 2 > 0 : some vehicle from S2N across the center section;
if situation & 4 > 0 : some vehicle from E2W across the center section;
if situation & 8 > 0 : some vehicle from W2E across the center section;
if situation & 16 > 0 : some vehicle's head in north crosswalk;
if situation & 32 > 0 : some vehicle's head in south crosswalk;
if situation & 64 > 0 : some vehicle's head in east crosswalk;
if situation & 128 > 0 : some vehicle's head in west crosswalk;
*/

/*
to check if there is any vehicle in center section;
the return value stores the situation happened in center section about
vehicle;
if situation is 0 no vehicle is crossing the center section.
*/
int anyVehInCenSec();

/*
to check if any vehicle from direction "dir" is crossing center section.
the second input parameter, "total", represents the number of lanes in
direction dir;
return 1 if any; otherwise 0.
*/
int anyVehFromTheDirInCenSec(LaneDir_enum dir, int total);

bool isIntersectedWithCenSec(CVehicle* ptr);


/*
only to check if any hoods of vehicles is in corsswalk;
a vehicle's hood is in second crosswalk is equvilent to that the vehicle
is across the center section.
the return value stores the situation happened in corsswalks about
vehicles;
if situation is 0, no vehicle's hood is in any crosswalk.
*/
int anyVehInCrossWalks();

/*
to check if any vehicle's hood is in the first corsswalks from input
direction.
in the input direction has "total" number of lanes.
return 1 if any; otherwise 0.
*/
int anyVehInTheFirstCroSec(LaneDir_enum dir, int total);

/*
to check if any vehicle's hoods in the first crosswalks.
if any, return 1; otherwise 0.
*/
bool isHoodInFirstCroSec(CVehicle* ptr, LaneDir_enum dir);

bool IsPtInRect(CPoint pt, CRect rt);
bool IsPtInRect(CPoint* pt, CRect rt);

/*calculate next steps*/

//to generate next steps for all living pedestrians
void generateNextStepsForAllPedObjs(int situation);

void toFindNextStep(CPedestrian* curr, int situation);

/*
to check if it is possible to launch an unissued object.
rule : search the start edge of the origin from middle to right to left.
When possible, return true immediately otherwise flase.
*/
bool isPossibleToIssue(CPedestrian* curr);

void copyNextPosToCurrPos(CPedestrian* curr);

/*
when option is true, consider horizontal way : input coor is a y value;
when option is false, consider vertical way : coor is a x value;
*/
bool getStartPos(CPedestrian* curr, int first, int second, int coor,
bool option);

/*
this function will search pedListHead to check if there is any
pedestrian walking in given section.
If not return false immediately.
If found, let p1 points to the first one; then to find the last one
walking in the given sectioin and let p2 point to the one just follow
it.
*/
bool getP1P2ofWSS(CMovingObj** p1, CMovingObj** p2, WSS_enum aWSS);

void posInSection(CPedestrian* curr);


/*
the rules for a person moving are as below:
(1) a pedestrian can move forward, or turn to left, or turn to right, or
stay at the same place without moving;
(2) a pedestrian could not mobe backward;
(3) if there is a vehicle in his front crosswalk or center section, the
pedestrian cannot move forward to that crosswalk;
(4) a pedestrian cannot run red or yellow light;
(5) a pedestrian has to move forward if he is walking in a crosswalk no
matter what color the traffic light is.
*/

/*
if a pedestrian is in WSS :
  if his current section is equal to the origin, move forward to WSCS;
  if it is equal to the destination, move to the end of the section;
if a pedestrian is in WSCS :
  if it is beside destination section, move to his destination;
  if not, move to CRS;
if a pedestrian is in CRS :
  move to WSCS.
*/
```

```
void toGetCurrObjNextStep(CPedestrian* curr, int situation);

/*
curr is the moving object;
aCRS is the name of the section moved to;
aSection is the section moved to;
*/
void moveToCRS(CPedestrian* curr, CRS_enum aCRS, CRect& aSection, int
situation);

/*
if pnt is right to rect, set pt = (-1, 0);
if pnt is left to rect, set pt = (1, 0);
if pnt is top to rect, set pt = (0, 1);
if pnt is bottom to rect, set pt = (0, -1);
*/
CPoint getMovingDir(CPoint pnt, CRect& rect);

/*
if pnt is right to rect, set pt = (-1, 0);
if pnt is left to rect, set pt = (1, 0);
if pnt is top to rect, set pt = (0, 1);
if pnt is bottom to rect, set pt = (0, -1);
*/
CPoint getMovingDir(CPedestrian* curr); //this is for moving to boundary

/*
This function searchs pListHead for objects walking in aWSCS;
if found, p1 will point to the first object who is walking in aWSCS;
otherwise, p1 points to NULL.
p2 will point to the object who just follows the last one walking in
aWSCS or to NULL.
*/
bool getP1P2ofWSCS(CMovingObj** p1, CMovingObj** p2, WSCS_enum aWSCS);

/*
This function searchs the pListHead for objects walking in aCRS
if found, p1 will point to the first object in aCRS;
otherwise, p1 points to NULL.
p2 will point to the object just next to the last object in aCRS
if p1 != NULL;
otherwise, p2 points to NULL.
*/
bool getP1P2ofCRS(CMovingObj** p1, CMovingObj** p2, CRS_enum aCRS);

bool isNextPosAvail(CPedestrian* curr, CPoint& dirPnt, CMovingObj* p1,
CMovingObj* p2, CMovingObj* p3, CMovingObj* p4);

int isPosAvail(CPedestrian* curr, double xM, double yM, CMovingObj* p1,
CMovingObj* p2, CMovingObj* p3, CMovingObj* p4, int blockObjID);

int isPosAvail(double x, double y, CMovingObj* p1, CMovingObj* p2,
CPedestrian* curr, int blockObjID);

/*
if return value is zero, it is OK to get a position;
otherwise return the index of the object blocking the curr move forward.
*/
int isPosAvail(CPoint& pnt, CMovingObj* p1, CMovingObj* p2, CPedestrian*
curr, int blockObjID);
```

```
/*
this function is to correct the calculation of the next step of the curr
object. This is because the calculation of the next step does not
considerthe boundary of the section the moving object is walking on and
the traffic light color.
*/
void constrianNextPos(CPedestrian* curr, CPoint& dirPnt, CRect&
aSection, bool isToCRS, CRS_enum aCRS, int situation);

void moveToBoundary(CPedestrian* curr, CRect& aSection);

void issueDeadPed(CPedestrian* curr);

void moveToWSCS(CPedestrian* curr, WSCS_enum aWSCS, CRect& aSection);


/*
to check if any vehicle from left or right side in center section.
the return value depends on input direction and follows the definition.
*/
int anyLRVehInCenSec(LaneDir_enum key);

/*
to check if any vehicle's head coming from left or right side in their
first crossroad section.
any vehicle's head in second crossroad are ignored because the vehicle
is in center section.
the input direction is the current checking vehicle.
return value depends on input direction and definition if any;
otherwise 0.
*/
int anyLRVehInCroSec(LaneDir_enum key);

//about pedestrian in corssroad section

/*
the return value stores the pedestrian's situation in all corssroad:
if situation = 0 : no pedestrian in any crosswalk;
if situation & 256 > 0 : some pedestrian in north crosswalk;
if situation & 512 > 0 : some pedestrian in south crosswalk;
if situation & 1024 > 0 : some pedestrian in ease crosswalk;
if situation & 2048 > 0 : some pedestrian in west crosswalk;
*/
int anyPedInCroSec();

//check if any pedestrian in the given crosswalk.
//return 1 if any; 0 for none.
int anyPedInTheCroSec(CRS_enum target);

//check if any pedestrian in first crossroad of the given direction.
//return value depends on input direction and definition if any;
//0 for none
int anyPedInFirstCroSec(LaneDir_enum key);

//check if any pedestrian in second crossroad of the given direction.
//return value depends on input direction and definition if any;
//0 for none
int anyPedInSecondCroSec(LaneDir_enum key);


void initRearBound();
```

```
bool generateNextStepForNextVeh(CVehicle* curr, int situation);

bool IsObjDead(CVehicle* curr);

/*
this function is to check if it is possible to issue an unissued object
*/
bool isPossibleToIssue(CVehicle* curr);


// to use bitwise AND "&", only 12 low bits used

//if situation = 0 : no vehicle in center section
//if situation & 1 > 0 : some vehicle from N2S in center section
//if situation & 2 > 0 : some vehicle from S2N in center section
//if situation & 4 > 0 : some vehicle from E2W in center section
//if situation & 8 > 0 : some vehicle from W2E in center section

//if situation = 0 : no vehicle's head in any crossroad
//if situation & 16 > 0 : some vehicle's head in north crossroad
//if situation & 32 > 0 : some vehicle's head in south crossroad
//if situation & 64 > 0 : some vehicle's head in ease crossroad
//if situation & 128 > 0 : some vehicle's head in west crossroad

//if situation = 0 : no pedestrian in any crossroad
//if situation & 256 > 0 : some pedestrian in north crossroad
//if situation & 512 > 0 : some pedestrian in south crossroad
//if situation & 1024 > 0 : some pedestrian in east crossroad
//if situation & 2048 > 0 : some pedestrian in west crossroad
void toFindNextStep(CVehicle* curr, int situation);

};


#endif //
!defined(AFX_ENGINE_H__4452DFF9_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED_)
```

Part B. The Class CAnimation

```
//////////////////////////////////////////////////////////////////////
// Animation.h: interface for the CAnimation class.
//////////////////////////////////////////////////////////////////////

#if
!defined(AFX_ANIMATION_H__4452DFF8_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED
_)
#define AFX_ANIMATION_H__4452DFF8_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "Engine.h"
#include "VehicleCtrlSys.h"
#include "PedestrianCtrlSys.h"
#include "template.h"

//controlling functions
UINT vehicleCtrlSysThread(LPVOID pParam);
UINT pedestrianCtrlSysThread(LPVOID pParam);

class Canimation {

friend class CEngine;

protected:

bool animRun;
bool replayRun;
bool reverseRun;
CEngine* m_Engine;

public:

bool& IsAnimRun() { return animRun; };
void startAnim() { animRun = true; };
void stopAnim() { animRun = false; };
void startReplay() { replayRun = true; };
void stopReplay() { replayRun = false; };
void startReverse() { reverseRun = true; };
void stopReverse() { reverseRun = false; };
void theReverse();
void theReplay();
void thePlay();
void toClearObjList();
void drawMovingObj(CDC*);
void toInvalidateObjects(CWnd*, int);
void toTerminatePlay();

CAnimation();
virtual ~CAnimation();

protected:

void getObjFromBufOne();
void getObjFromBufTwo();
void toInvalidatePedCurr(CWnd* a_wnd);
void toInvalidatePedNext(CWnd* a_wnd);
void toInvalidateVehCurr(int idx, int jdx, CWnd* a_wnd);
void toInvalidateVehNext(int idx, int jdx, CWnd* a_wnd);

/*
pointer to the vehicle thread which is created when user clicks on Run
button.
The vehCtrlSysThread is a connection between the main thread and the
work thread.
*/
CWinThread* vehGenThread;

/*
pointer to the vehicle control system which is passed to the new created
work thread about vehicle system so the control system can access view
window.
*/
CVehicleCtrlSys* vehCtrlSys;

/*pointer to the pedestrian thread which is created when user clicks on
Run button.
```

```
The pedCtrlSysThread is a connection between the main thread and the
work thread.
*/
CWinThread* pedGenThread;

/*
pointer to the pedestrian control system which is passed to the new
created work thread about pedestrian so the control system can access
view window.
*/
CPedestrianCtrlSys* pedCtrlSys;

};

#endif //
!defined(AFX_ANIMATION_H__4452DFF8_78DB_11D3_A7F4_820EEA6AE49B__INCLUDED
_)
```

References

[1] Mike Blaszczak (1997). *Professional MFC with Visual C++ 5*. WROX.

[2] David Bennett, et al.. *Visual C++ 5 Developer's Guide*. SAMS PUBLISHING.

[3] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit*, WILEY COMPUTER PUBLISHING.

[4] David J. Kruglinski, *Inside Visual C++ Fourth Edition*, Microsoft Press.

[5] José M. Garrido. *Practical Process Simulation Using Object-Oriented Techniques and C++*. Artech House, Boston, London

[6] M.M. Woolfson and G.J. Pert. *An Introduction to Computer Simulation*. OXFORD UNIVERSITY PRESS.

[7] Stewart V. Hoover, Ronald F. Perry. *SIMULATION A Problem-Solving Approach*. ADDISON-WESLEY PUBLISHING COMPANY.

[8] Peter Robinson. *Object-oriented Design*. CHAPMAN & HALL.

[9] Mohamed Fayad, Mauri Laitinen. *Transition to Object-oriented software development*. John Wiley & Sons, Inc.

[10] Wolfgang pree. *Design Patterns for Object-Oriented Software Development*. ADDISON-WESLEY PUBLISHING COMPANY.

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Resuable Object-Oriented Software*. ADDISON-WESLEY PUBLISHING COMPANY.

[12] Robert J. Muller. *Database Design for Smarties using UML for Data Modeling*. MORGAM KAUFMANN PUBLISHERS, INC.

[13] James Rumbaugh, et al.. *Object-Oriented Modeling and Design.* PRENTICE HALL.

[14] Bernard Ostle, Linda C. Malone. *Fouth Edition, Statistics in Research, Basic Comcepts and Techniques for Research Workers.* IOWA STATE UNIVERSITY PRESS / AMES.