uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES
ET POSTOCTORALES

FACULTY OF GRADUATE AND
POSDOCTORAL STUDIES

Ke Liu
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

HLA/RTI Based Emergency Preparedness and Response Training Simulation

TITRE DE LA THÈSE / TITLE OF THESIS

Prof. N. Georganas
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Prof. J. Lang

Prof. G. Wainer

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

# HLA/RTI Based Emergency Preparedness and Response Training Simulation

by

**Ke Liu**

A thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical Engineering

The Ottawa-Carleton Institute for

Electrical and Computer Engineering

School of Information Technology and Engineering

University of Ottawa

© Ke Liu, Ottawa, Canada, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 978-0-494-49239-0*
*Our file Notre référence*
*ISBN: 978-0-494-49239-0*

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

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

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

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

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

**To My Beloved Parents**

# Abstract

Many wake-up calls have been received for emergency response, due to natural disasters such as hurricanes, fires or man-made incidents, for example: oil and chemical spills, city bombings, and the terrorist attacks of 9-11. The emergency responders need to work in a coordinated, well-planned manner to best mitigate the impact of an emergency incident. Simulation systems as valuable tools provide a wider range of training at a much lower expense for emergency preparedness and response, and can be used for vulnerability assessment, organizing, educating and decision support. This is identified as the only feasible approach when it is difficult to emulate real-life experiences.

This thesis presents an emergency evacuation training simulation, taking the demonstrative example of the SITE building of the University of Ottawa. The objective of the research is to design a multi-user distributed simulator to conduct the safety training in the scenarios of emergent evacuation. The real-time interaction and collaboration, in the simulation, among the users are achieved over HLA/RTI, the IEEE standard for distributed simulation and modeling.

# Acknowledgement

First of all, I own heartfelt thanks to my supervisor Dr. Nicolas D. Georganas. This work would not have been possible without his perceptive suggestions, continued guidance, encouragement and support. His conversations and encouragements will be always remembered.

Special thanks goes to Dr. Xiaojun Shen, postdoctoral fellow at the DISCOVER lab, who guided me through this project from the beginning, and gave lots of suggestions, corrections and helped in bringing the thesis to completion.

I would like also to thank my colleague Yue Xing and friend Jian Dong, who provided advice on materials and overall knowledge.

Last, but not least, I would like to give my appreciation to my family for their endless love and support during the past two years, without which the completion of this thesis would not be possible.

Ke Liu

Ottawa, September, 2007

# Acronyms

2D – Two Dimensional

3D – Three Dimensional

API – Application Programming Interface

DIS – Distributed Interactive Simulation

DL – Display List

FOM – Federation Object Model

FedExec – Federation Executive Process

FED – Federation Execution Data

GLU – OpenGL Utility Library

GLUT – OpenGL Utility Toolkit

GLUI – OpenGL User Interface Library

GUI – Graphic User Interface

HLA – High Level Architecture

IERF – Integrated Emergency Response Framework

LRC – Local RTI Component

MOM – Management Object Model

M&S – Modeling and Simulation

NIST – National Institute of Standards and Technology

OS – Operating System

OMT – Object Model Template

RTI – Run Time Infrastructure

RIS – Runtime Interface Specification

RtiExec – RTI Executive Process

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

UML – Unified Modeling Language

VR – Virtual Reality

# Table of Contents

**Abstract**..........3

**Acknowledgement**..........4

**Acronyms**..........5

**Table of Contents**..........7

**List of Figures**..........10

**List of Tables**..........13

**Chapter1 Introduction**..........14

1.1 General..........14

1.2 Motivation..........16

1.3 Research Scope..........18

1.4 Contributions..........19

1.5 Structure of the Thesis..........20

**Chapter 2 Background Review**..........21

2.1 Emergency Response Simulation System Development..........21

2.1.1 Integrated Emergency Response Framework..........22

2.1.2 2D Based Simulation System..........27

2.1.3 3D VR Based Simulation System..........30

2.1.4 HLA Based Distributed Simulation..........32

2.2 System Architecture..........39

2.2.1 Research Preliminaries..........39

2.2.2 Application (Simulator) Architecture..................................... 42

**Chapter 3 Simulation Graphics and Associated Audio..............................46**

3.1 Static 3D Graphics Modeling....................................................46

3.1.1 Modeling of 3D Virtual World........................................... 46

3.1.2 3D Avatars Design.........................................................50

3.2 Graphic Control in C++......................................................... 55

3.2.1 Data Structure Modeling...................................................55

3.2.2 3D Transformation and Corresponding Coordinates....................57

3.2.3 3ds Data Structure.........................................................62

3.2.4 Texture Loading (Mapping) ..............................................66

3.2.5 Object / Texture Rendering................................................74

3.2.6 Object Normalization...................................................... 78

3.2.7 Avatar Controlling / Cameral Following Algorithm..................... 83

3.2.8 Collision Detection........................................................ 87

3.2.9 Auto-avatars Moving Algorithm...........................................91

3.2.10 User Interface...........................................................101

3.3 Simulation Audio..............................................................108

3.3.1 OpenAL Implementation..................................................108

3.4 Simulation Scenarios (Use Cases)............................................110

**Chapter 4 Interaction and Collaboration.......................................... 113**

4.1 Develop with RTI 1.3..........................................................113

4.1.1 RTI1.3NG.......................................................................113

4.1.2 FedExec Management Areas......115

4.1.3 Implementation......117

4.2 Requirements of RTI......124

4.2.1 Network Requirement......124

4.2.2 System Requirement......124

4.2.3 RTI Installation......125

4.3 Simulation Results......126

4.3.1 Tainting Cases Analysis......126

4.3.2 Parameter and Configuration Sensitivity Assessment......129

4.3.3 Simulation Demonstration......132

**Chapter 5 Conclusion and Future Work...... 134**

5.1 Conclusions......134

5.2 Future Work......135

**Publication......136**

**References...... 137**

# List of Figures

Figure 2-1: Simulation Systems....21

Figure 2-2: Integrated Emergency Response Framework.... 22

Figure 2-3: 2D Evacuation Simulation System for DISCOVER Lab....27

Figure 2-4: 3D VR Based Simulation with Multi-agents.... 32

Figure 2-5: Functional View of a DIS HLA Federation....35

Figure 2-6: HLA Components.... 38

Figure 2-7: The Framework of Emergency Evacuation Training Simulation.... 40

Figure 2-8: Simulation Architecture.... 41

Figure 2-9: Application Architecture....42

Figure 3-1: 2D SITE Ground Map....47

Figure 3-2: Basic 3D Models of SITE Building with Five Separate Stories.... 48

Figure 3-3: Convert 2D Spline (Wireframe) to 3D Solid Model....48

Figure 3-4: Imported 3D Furniture Models.... 48

Figure 3-5: Texture Materials....49

Figure 3-6: 3D Graphics Modeling of the SITE Building.... 50

Figure 3-7: 3D Avatar Hierarchical Modeling....52

Figure 3-8: 3D Avatar Design....52

Figure 3-9: Avatar's Walking Action.... 53

Figure 3-10: OpenGL Graphic Pipeline....58

Figure 3-11: Modeling Transform.... 59

Figure 3-12: Setting Camera (Eye) Position....................59

Figure 3-13: Setting View Volume (Default)....................60

Figure 3-14: Orthographic Projection (3D Scene Mapping onto 2D Screen).........61

Figure 3-15: Setting View Port on 2D Screen....................62

Figure 3-16: 3ds Structure....................67

Figure 3-17: Texture Mapping Distortion....................70

Figure 3-18: Counterclockwise Texture Mapping....................72

Figure 3-19: Screen Texture Mapping (Two Floor Maps)....................73

Figure 3-20: 3ds Object Rendering Flowchart....................76

Figure 3-21: Polygon Normalization....................79

Figure 3-22: Vertex Normalization....................81

Figure 3-23: Sphere – Plane Collision....................82

Figure 3-24: Camera/Avatar Rotation....................84

Figure 3-25: Camera/Avatar Flat Moving....................85

Figure 3-26: Ground Floor 2D Map....................88

Figure 3-27: Collision Detection....................90

Figure 3-28: Path Matrix....................97

Figure 3-29: Cell Center Calculation....................99

Figure 3-30: Path Choosing....................100

Figure 3-31: GLUT Pop-up Menu....................102

Figure 3-32: System Finite State....................104

Figure 3-33: ***SimSITE*** User Interface....................107

Figure 3-34: Senario1 Automatic Evacuation........................................... 110

Figure 3-35: Senario2 Evacuation Training...............................................112

Figure 4-1: Communication between Ambassadors..................................114

Figure 4-2: Interaction between a Federate and RTI....................................115

Figure 4-3 (a): RTI Initial...............................................................118

Figure 4-3 (b): Object Update...........................................................119

Figure 4-4: Federation Initialization.................................................129

Figure 4-5: Proof-of-Concept............................................................. 132

Figure 4-6: Simulation Result in Different Scenarios..................................133

# List of Tables

Table 3-1: 3ds File Hierarchy.....64

Table 4-1: Auto-avatar Following Evacuation Times.....127

Table 4-2: Evacuation Plan Following Evacuation Times.....127

Table 4-3: Evacuation Leader Following Evacuation Times.....128

Table 4-4: Baseline Case Auto-avatar Evacuation Times.....130

Table 4-5: Effect of Population Density on Evacuation Times.....131

Table 4-6: Effect of System Configuration on Evacuation Times.....131

# Chapter 1
# Introduction

## 1.1 General

Emergency preparedness and response are named for all the activities of identifying, detecting, planning, vulnerability analyzing and preparing to unanticipated events which may cause injury, loss of human lives, damage and destruction of critical infrastructure elements [1]. Simulation is a valuable tool for emergency preparedness and can be used for training, learning and decision-making.

The goal of training simulations is to provide high-fidelity realism to increase the diffusion of innovative and less-invasive procedures while decreasing the trainee's learning curve. This leads to two key issues in the design and implementation of training simulations: verisimilitude – accuracy/realism of simulations and appropriate training modalities to facilitate skills generation and transfer by performing emulated tasks in the testing environments.

3D Virtual Reality (VR) technology provides the simulations with immersion, interactivity and information intensity, which gives us the sense of being mentally immersed in the simulations by delivering interactivity and feedback to one or more modalities. With the innovative technologies in high-fidelity VR, simulations of the

actual emergency environments and user interactions can be achieved to a great extent.

The objective of the presented research is to design a collaborative training simulator, where real-life experiences are emulated. The real-time interaction and collaboration, in the simulation, among the users are achieved over HLA/RTI [2]. High Level Architecture (HLA) is the IEEE standard 1516 to build a large scale distributed modeling and simulation (M&S). With a set of services provided by HLA's implementation, the Run Time Infrastructure (RTI) gives the specification of a common technical framework for reuse, collaboration and interoperation of different simulators.

## 1.2 Motivation

In order to develop the decision-making and problem solving skills in emergency response situations, individuals have to be involved in training situations that mimic real life experiences. For this purpose, there are great demands of preparedness for emergency responses both for natural and man-made disasters or other catastrophic events. Simulations provide experiences that can be used to prepare for future situations, while modeling is an indispensable problem-solving methodology for the real-world problems solutions. Modeling and simulation provide measurements, standards, and technical advice. Moreover, it helps the engineering and construction industries improve the quality of building materials to make them more reliable and long lasting. On the other hand, responders believe that M&S can be a very promising training tool even though there is a concern that it may be perceived as a replacement of hands-on exercises and worse yet, reduction of costs.

Obviously, simulation and modeling play an important role in the domain of emergency response and preparedness simulations. A number of efforts for using simulation to study different kinds of emergency incidents is in progress. Emergency response can benefit a lot from research in the full range of disaster life-cycle areas, including development of the simulation systems. To put all the things into a nutshell, M&S increase reliance on information technology and systems for communications and monitoring, control risk increase due to inadequate security, and enhance threat detection and protection.

Meanwhile, how to develop an accurate realistic simulation so as to provide the immersion effects is one of the most challenging issues. Graphic modeling and rendering techniques, the fundamental techniques, will be introduced in Chapter III of the thesis. A 3D simulation engine has been developed, where sophisticated algorithms such as object movement detection and collision detection are designed to ensure the realism of the simulation. Training modes and scenarios are also considerations in the design of the training system.

## 1.3 Research Scope

The requirements of M&S of the emergency response have a great range, which includes reduction time for model development; reduction research costs through neutral simulation component libraries; permitting customization to meet individual scenario needs; standardized interfaces and enhanced capabilities for data import and export between heterogonous simulation systems, and rapid communication to multiple users. Therefore, the developed simulation system will be immersive, collaborative, real-time and interoperable [3].

The strategies [4] to design an emergency response simulation start with identifying the user needs; having a market scan of existing tools and relevant standards; then bringing together experts from different areas of the nation to help establish a roadmap for moving forward; collecting all of the relative data; finally setting up an initial framework for the designed simulation. For the purpose of the effectiveness, the integration of modeling and simulation tools has to follow interoperability standards. UML can be an advantage specification to define data structures in emergency response simulations.

The actual implementation was completed by going through two main stages, graphic design and collaborative communication setup (HLA/RTI). *3dsMAX* and *Visual Studio .NET 2003* were applied in this project with the former being the static graphic modeling software and the latter the development environment.

## 1.4 Contributions

The main contributions of this thesis are summarized as follows:

1 Identified the requirements and issues of modeling the behavior of a distributed simulation system and presented an analysis of the existing M&S techniques.

2 Provided a feasible approach to develop an Integrated Emergency Response Framework (IERF), which complies with the HLA architecture.

3 Designed the static 3D graphic models for the SITE building of the University of Ottawa in 3ds MAX format.

4 Developed a simulation engine to control and render the 3D scenes of the SITE building. Implemented collision detection and optimal path finding algorithms, which can be adopted in any 3D evacuation simulation system by easily changing the corresponding location maps.

5 Developed a framework to support an interactive and collaborative evacuation simulation using the IEEE standard 1516 (HLA/RTI).

6 Implemented, using the RTI services, a distributed virtual evacuation simulation which provides a practical foundation to enhance interactivity and interoperability.

7 Combined many technologies together which include graphic design, 3D engine development, and distributed communication establishment. A proof-of-concept demonstration had been carried out successfully among three research labs in the University of Ottawa.

## 1.5 Structure of the Thesis

The thesis is organized as follows: Chapter 1 introduces the motivation, research scope, and contributions. Chapter 2 addresses an overview of emergency response simulation systems and the key elements. Project design architecture and related knowledge are also described in this section. Chapter 3 focuses on graphic and associated audio design; from the developer's point of view, the core algorithms and implementations are discussed in details. Chapter 4 depicts some HLA/RTI concepts and approaches for the development of a collaborative virtual environment. In this chapter, the proof-of-concept demonstration is presented as well. Chapter 5 concludes with the summary and potential future research topics.

# Chapter 2
# Background

## 2.1 Emergency Response Simulation System Development

There are significant demand for emergency response and preparedness for both natural and man-made disasters. A great number of research efforts have been targeted at the development of emergency response modeling and simulation tools. These include planning tools such as virtual representations of city landscapes, human body representations for enhanced medical response, tele-haptic [5] simulators that perform remote operations to local objects, war theatre simulation tools which can be used for homeland security applications, and discrete event simulation tools for planning medical resources to attend the affected population [6]. Some of these simulation systems have already been used in several countries, for example as shown in Figure 2-1. On the left is the screen capture of a chemical plant fire simulation system and on the right is a radiological accident modeling for emergency response.

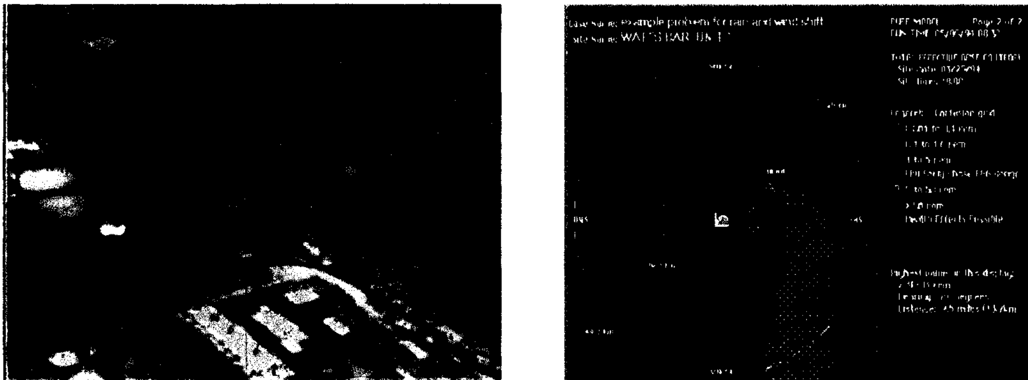


Figure 2-1: (Left) Chemical Plant Fire Simulation System [7]; (Right) Radiological Accident Simulation System [8]

### 2.1.1 Integrated Emergency Response Framework

In order for the simulation to be efficient, the simulation itself should be immersive, collaborative, real-time, and interoperable. Therefore, the development of modeling and simulation systems should be focusing on data standards and protocols to enable information systems to inter-operate between each other. According to NIST (National Institute of Standards and Technology) [9], the emergency response framework is mainly divided into 3 parts: disaster event analysis, entities of interest, and applications study, which is illustrated in Figure 2-2.

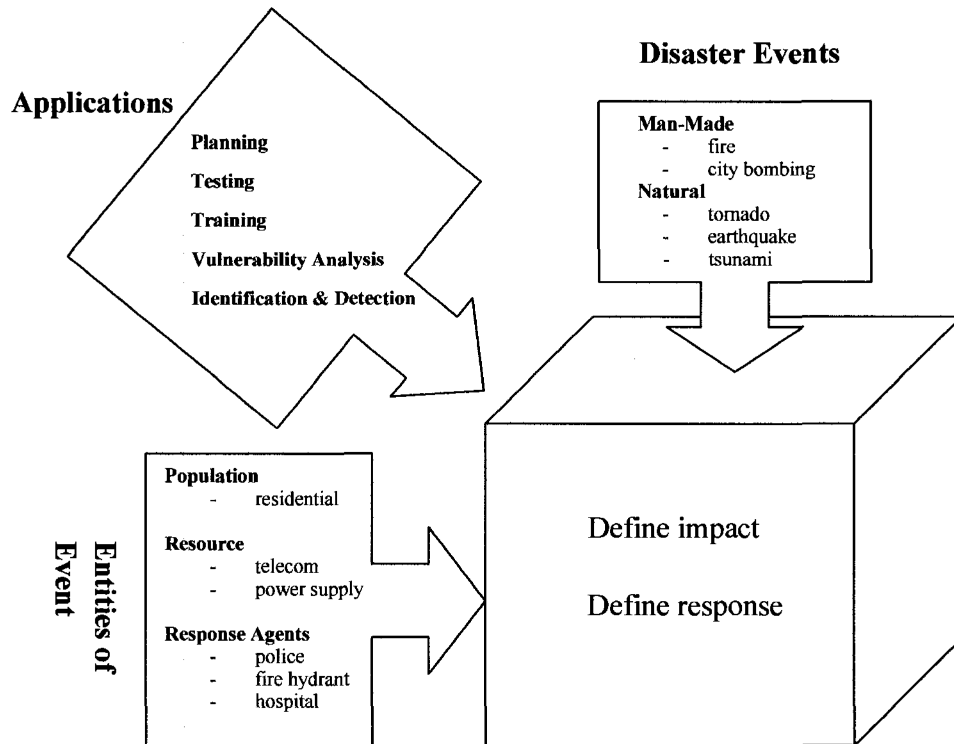



Figure 2-2: Integrated Emergency Response Framework

**Disaster Event**

The emergency response agencies should be able to respond efficiently to man-made and natural disaster events. Depending on the type of disaster event, the requirements on the modeling and simulation capabilities can differ a lot. For instance, in the event of a building exploding, the model is required to have the ability to simulate the impact of explosion, smoke, fire and the occupants. While, in the event where hazardous chemicals and materials are released to the environment, the ability to model the dispersion of release in the atmosphere and surrounding environment will be required.

**Entities of an Event**

Another consideration in emergency simulation design is the *Entities of Interest*, i.e. minimizing the impact of disaster events on entities which are of interest. These include first and foremost the human population. If there were no human beings, there would be no disasters. The impact of the disaster on resources, especial on the infrastructure resources, also needs to be understood and contained. The second major class of *Entities of Interest* is the response agents. The response agents are quite possible to become the affected entities of interest themselves, for instance, fire personnel suffer deaths and injuries while fire fighting. The response agents' actions must be modeled in order to understand how they can contain and mitigate the aftermath of a disaster event. Different strategies, which can reduce the exposure risk for the response agents, can be tested by planners.

**Applications**

The third issue is the *Application* study. Depending on the applications they are designed for, different M&S tools need to have different functions and purposes. An application for vulnerability analysis of a disaster event will have different technical requirements from the one designed for emergency response personnel training. The training applications usually have strict requirements of interaction and interactivity in the cases of dealing with alternate simulated event sequences in accordance with the responses from the trainees. Similarly, in order to determine the possibilities of threats, the identification and detection of threat application must have the capabilities for pattern matching against previous scenarios. Some basic types of applications in the domain of emergency response simulations are briefly described below.

*Planning Applications* require two types of tools, one to determine the impact of a disaster event, and the other to aid the development of the response action plans and strategies. The information sources identification and data requirements are equally important in developing an effective simulation system. Taking an interior building hazard prediction modeling for instance, the impact of wind, temperature, population density, building structure will need to be evaluated together with street data, local police office, fire station and hospital locations. Examples of the *Planning Application* are listed in the following documents:

- Information collecting (building plans, city maps, etc.)

- Setting up a communication infrastructure

The *Identification and Detection Application* helps to determine the possibility of the occurrence of a disaster and its potential threats through intensive study of historical data (especially of similar disaster scenarios) and environmental factors. Some examples of *Identification and Detection Applications* are listed below:

- Identifying the potential of tsunami occurrence given the weather conditions
- Detecting the occurrence possibility of forest fire given the temperature conditions

*Testing Applications* have tools and equipments used for verifying the impact of a disaster event. They are often used in the situations where real life experiments cannot be performed. By analyzing the testing results, people can gain the experience, and many complications could be avoided. Some examples of *Testing Applications* are given as follows:

- Vehicle airbag and security testing system
- Testing the sensor and network communication system

*Training Applications* are the most common applications designed for the purpose of emergency response. Their major goal is to teach responsive agents to handle emergency events. This type of applications includes interactive simulations which are to imitate real life scenarios for trainees to practice emergency response procedures and actions. Evaluation of response actions is usually carried out with the help of the tools and it also can help the trainee to learn what works best under a given circumstance. These tools range from interactive simulations using a simple desktop

monitor to 3D immersive environments. The training procedures are as follows: trainee training, back-story brief, scenarios setup, and exercise run. The trainee's actions are captured during the exercise, and those captured actions are then reviewed, analyzed, and compared against the standard protocols. An evaluation is done for each step of the exercise. The overall assessment will help the emergency responders gain the relative experiences to figure out how to sustain their strengths and how to improve on their weaknesses. Some typical *Training Applications* are:

- School evacuation training system
- Military training for terrorist attacks

The *Vulnerability Analysis Application* is mainly designed to facilitate the evaluation of emergency response plans. M&S tools are used to generate a number of disaster event scenarios and assess the performance of emergency response actions and strategies. Some examples of *Vulnerability Analysis Applications* are listed below:

- Evaluation of inner-building hazards response strategies
- Evaluation of security plans and procedures

There are many other tools that could be used to design a more efficient simulation system. Those tools include but not limited to: data collection tools, analytic tools, scenario generation tools, data migration tools, security tools, modeling, training, testing/validation, planning and preparedness, detection and anticipation, response

and diagnosis, user interface, environmental (static and dynamic), warning representation, recovery and maintenance tools.

### 2.1.2 2D Based Simulation System

Generally, there are two basic approaches for building up an emergency simulation system. One way is to use 2D techniques to develop the system, for example: a simple indoor evacuation simulation system can be developed using 2D technology, as shown in Figure 2-3 , a 2D evacuation simulation for the DISCOVER laboratory in SITE 5077 of the University of Ottawa.

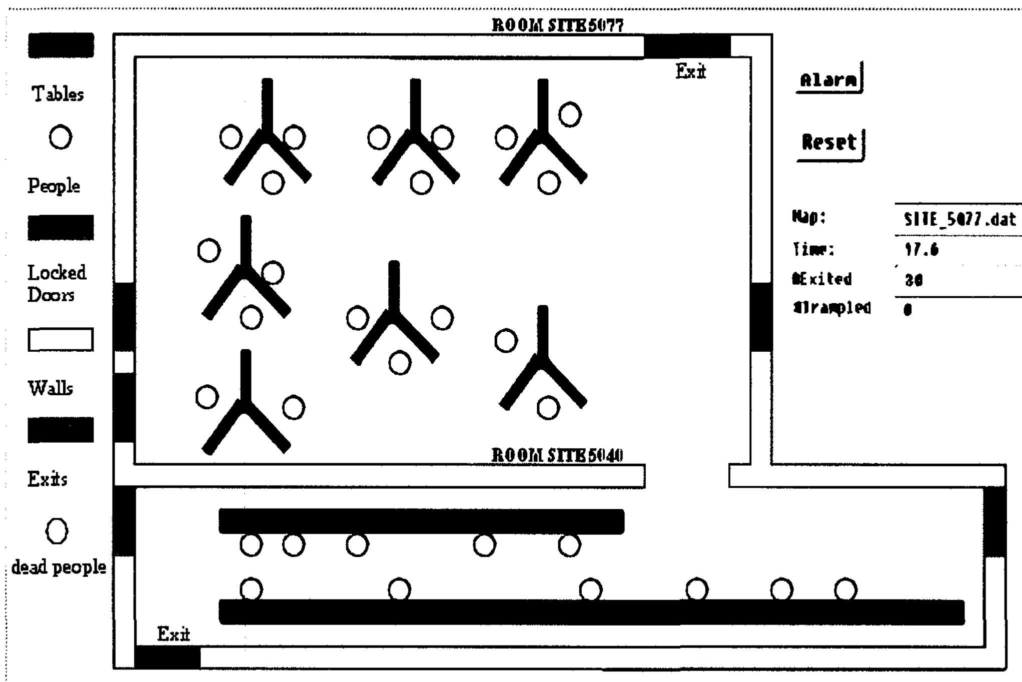



Figure 2-3: 2D Evacuation Simulation System for DISCOVER Lab (SITE University of Ottawa)

The blue bars indicate the tables (obstacles) in the room. The circles represent the students working in the lab, which are termed *avatars* in the rest of the thesis. The green bars and red bars stand for the doors (red bars are the intermediate doors to another room, and green bars are the doors to exits). The system uses a continuous-space model. For each time-step of the simulation, the location of each avatar in the room is recalculated based on the avatar's motion direction, surrounding obstacles, and various human attributes, such as speed, patience and aggression.

The evacuation simulation system is designed for the training purpose of man-made disasters, such as escape in the case of a fire. To design this simulation system, the developer need to collect relative data, including the building maps, the room structures, the population densities, and so on. Also, to model the human behaviors before, during, and after the emergency, the psychology and the physics need to be studied. In the simulation, avatars contain various attributes that represent human behaviors and characteristics, such as size, speed, mass, strength, waiting time, aggression, patience, dexterity, and agility.

Some real life scenarios are introduced in this system to make the simulation more realistic. When an avatar is blocked and unable to move for an extended time interval, it becomes aggressive and will push other avatars on its way. When other avatars have been pushed, their positions will be affected by the force and their masses. The avatars that are pushed beyond their thresholds will fall. If a fallen avatar incurred

further extrusion before it can stand up, it will be eventually trampled. Other avatars will be able to walk over a trampled avatar. The trampled avatars (dead people) are shown as red circles in the design. The user interface will contain functional buttons needed for system control. Important information is recorded on the screen such as the simulation time lapsed, the number of people escaped and trampled. This system was originally developed by Nathan J Kopp [10]. We adopted it in the application for the DISCOVER lab by the inclusion of the new 2D floor map. By introducing this idea, the optimized path finding algorithms developed for the presented research, can be easily adapted to other 3D evacuation systems.

Most of the research theories are based on the assumption that any architectural space can be described by 2D geometries. This needs to ignore the guidance of the individual by the 3D architectural space itself. It leads to the research blank on the relationship between the 3D architecture spaces' forms and the individual simulators' behavior patterns. In many situations, the continuous 3D view changes are simplified and described by 2D view field analysis, and the continuous 3D spaces are divided into independent 2D planes; however, the broken 2D view will definitely sacrifice a lot of spatial information which is useful to the simulation behaviors. Due to the limitations of the 2D design, a vivid 3D world is more preferable in many simulations.

### 2.1.3 3D VR Based Simulation System

With the advances in computer technologies and human computer interfaces, a new technology called Virtual Reality (VR) has emerged, which allows a user to visualize, manipulate, and interact with a computer-simulated environment in a natural manner. The visualization part involves the computer generated imaging, auditory, or other real time sensual outputs to the user of a virtual world. This world may be a 3ds model, a scientific simulation, or a view into a database [11].

A 3D VR system is an interactive computer simulation, which senses the real world stimuli, especially participants' positions, perceptions, and actions, and gives one the feeling of being physically immersed in the virtual environment by delivering feedback to one or more modalities [12]. The feature of allowing users to experience a life-like real time virtual performance makes VR suitable for a wide range of applications such as entertainment, e-commerce, industry training, emergency planning and response, and many others. Specifically, a collaborative virtual environment designed for evacuation simulations will be presented in this thesis.

Sophisticated 3D VR simulation and modeling tools are the most important means for many industries designing components, products, and production processes. 3D simulation tools can be used to create finer-grained process models. For example, VR tools have been used to successfully model the evacuation of a facility, by modeling the behavior and movement of people through a building's hallways, doors and

stairwells. The same technology can be used to model the behavior of the compromised system as well, for example, the sequence of failures of the components, such as valves and pipes, as the effect ripples through the system. These predictions are based on physical simulation of the actual structure of the system. The feasibility of rescue, reconstruction, decontamination and dismantling operations can be planned and simulated collaboratively, taking into account hazardous conditions. These immersive simulations can be used to train and prepare personnel to handle the anticipated difficulties.

One way to develop a 3D VR simulation system is using a game engine featured with multi-agents [13]. Serious games are a crucial part of the future modeling and simulation, not just for training or entertainment purposes. Game-based simulations with real-time interactions will play a key role in the system testing and evaluation, especially in the area of emergency response. Figure 2-4 shows a game engine developed 3D high-rise building emergency evacuation simulation, involving 7 stories and multiple avatars within a vertical fire zone. The simulation provides 3D virtual spaces and establishes communication within the spaces. It enables avatars to interact with each other visually as well as verbally. The overall evacuation response is the cumulative behavior of many individuals with different responses. We can clearly see that the 3D VR simulation provides more information, such as spatial data, than the 2D one. Each person's current status shows on the screen in real-time. Comparing 3D design with 2D simulation described in section 2.1.2, 3D is more attractive and

realistic, but technically more challenging to implement and develop.

Figure 2-4: 3D VR Based Simulation with Multi-agents [14]

### 2.1.4 HLA Based Distributed Interactive Simulation (DIS)

**DIS Development Process**

DIS is an application of distributed systems technology that enables models with alternative scenarios to be connected over computer networks so that they interoperate within the same simulation execution. Initial research on distributed simulation was mainly for defense purposes [15]. It gave an approach for supporting the interoperability between models and the reusability of them [16]. To consummate this approach, many other services, such as execution time and cost reduction,

geographical distribution, integrating simulation models, and fault tolerance had also been added into the system [17]. Other benefits, including existing components reusing, information hiding supporting, and heterogeneous models integrating, had been made [18].

In order to realize distributed M&S, simulation standards and efforts were initiated in the defense community. In 1989, IEEE standard 610.3 issued a glossary of modeling and simulation terminology, which defined the terms in the field of modeling and simulation such as general M&S concepts, M&S types, M&S variables, game and queuing theories. In 1995, IEEE released standard 1278.1, the protocols for DIS simulations, where protocol data units (PDUs) were defined for the data exchange over a network. The same year, IEEE announced the next standard 1278.2 for DIS communication services and profiles to support information exchange between simulations. In 1996, IEEE 1278.3: *Recommended Practice for DIS—Exercise Management and Feedback* appeared. The following year, IEEE 1278.4: *Trial-Use Recommended Practice for DIS - Verification, Validation, and Accreditation* came out [19]. The year 2000 is a milestone in the development of DIS. That was the year High Level Architecture (HLA) first came into place. HLA is the most recent and advanced approach for integrating simulation models [20]. The IEEE 1516 standard specified the HLA framework, rules, federate interface, and Object Model Template (OMT).

## **HLA Functional Overview**

HLA, the framework to achieve the goals of reusability and interoperation, is based on the assumption that no simulation can satisfy for all possible uses. An individual simulation or a set of simulation components, which are developed for one purpose, can be re-used for different purposes based on the HLA concepts. Thus, it reduces the time and cost in the creation of the synthetic environment for a new purpose and provides developers the option of distributed collaborative development of complex simulation applications. Nowadays, HLA is widely used in a large range of simulation and multimedia application areas, such as research, engineering, entertainment, education, and so on [21]. These application areas are hardly related to each other, which indicates that HLA can meet various requirements for distributed simulation development. HLA not only supports the simulation data collection and the simulation activities monitoring, it also supports interfacing with live participants; however, it does not prescribe a specific implementation, nor does it mandate the use of any particular set of software or programming language. As more advanced technologies become available, new options of implementations will be possible within the framework of HLA.

HLA defines an elementary simulation - *federate*. A set of federates constructs a *federation*. Figure 2-5 indicates the composition of the major functional components in an HLA federation. The key component is the federate. A federate can have diversified modalities, such as a computer simulation, a manned simulator, a

supporting utility like a viewer or a data collector, or even an interface to a live player or instrumented facility [22]. The federate includes all object representations. HLA allows the objects in a simulation to exchange messages and interact with each other, which is supported by the Run Time Infrastructure (RTI) services. No constraints will be added on the representation of the federate. The RTI, another functional component, is the core part of HLA, which constitutes an implementation of the HLA interface specifications. It acts like a distributed operating system for the federation, which provides a set of general-purpose services that support federate-to-federate interactions and federation management. All interactions among federates eventually go through the RTI. The third component is the runtime interface. In order to invoke the RTI services and to support runtime interactions among federates, the HLA runtime interface specification (RIS) provides a standard way for federates to interact with the RTI.

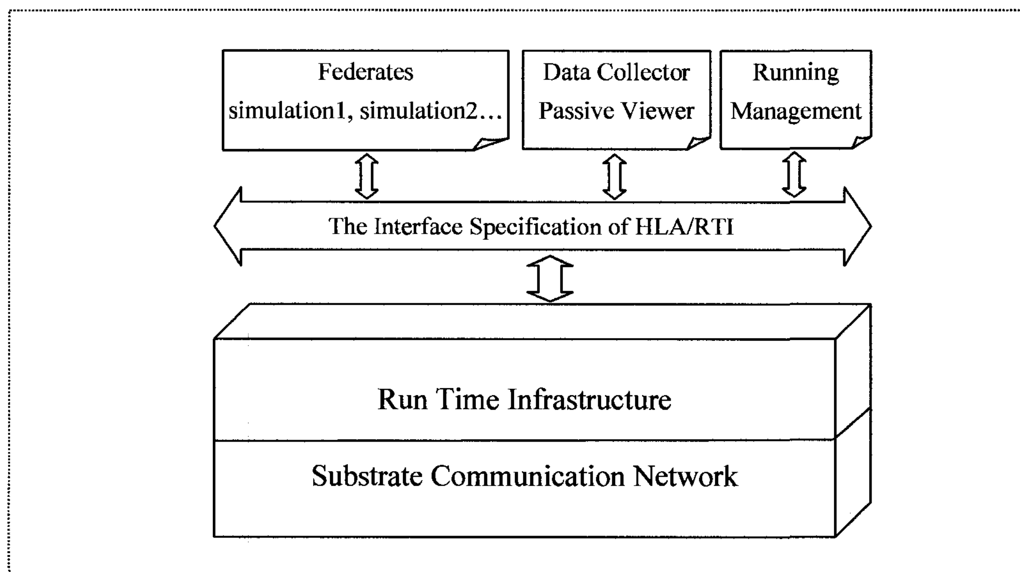



Figure 2-5: Functional View of a DIS HLA Federation

**HLA Components**

Three major components are formally defined in HLA: the Runtime Interface Specification (RIS), the Object Model Template (OMT), and the HLA Compliance Rules [23]. The overall structure of HLA components is illustrated in Figure 2-6.

The HLA RIS [24] describes the runtime services provided to the federates by the RTI. RIS specifies six classes of service: Federation Management Service, which provides some basic functions in order to create, join and operate the federation; Declaration Management includes publication, subscription, and supporting control functions, which supports efficient and reliable data exchange management with the information provided by federates; Object Management, which offers a service at the object level such as object creation, deletion, resignation, destruction and so on; Ownership Management is a service that supports the transformation of the ownership and updates corresponding attributes of objects during the execution; Time Management Services, which coordinate federate logical time advancement and support runtime simulation data exchange synchronization; and the last service - Data Distribution Management Services that provide a flexible and extensive mechanism for further extending the sophistication of the RTI's information routing capabilities [25]. The HLA interface specification defines the mechanisms for accessing these services, both functionally and through an application programmer's interface.

HLA object models are the descriptions of the elements in a simulation system. There

are no constraints of the HLA object models. HLA requires each federate and federation to formalize the object model using a standard Object Model Template (OMT) Specification [26]. All objects and interactions managed by a federate are described according to the OMT. It provides open information sharing across the simulation. Three types of object models are defined in HLA, the Federation Object Model (FOM), the Simulation Object Model (SOM), and the Management Object Model (MOM). The HLA FOM introduces all shared information (e.g., attributes, interactions) and contemplates inter-federate issues. The HLA SOM describes salient characteristics of a federate and presents objects and interactions which can be used externally. The HLA MOM identifies objects and interactions used to manage a federation. The FOM, SOM and MOM all have been documented using the HLA OMT standard form.

Every standard needs a regulation, and HLA is no exception. HLA Compliance Rules summarize the key principles of the HLA, which express all the federations' and federates' rules [27]. For example, the rules allow only one federate to own object attributes at any given time during the execution, and all object representations should happen in the federates. Information needs to be exchanged through the RTI using the HLA RIS. Every federate of the HLA must document their public information in their SOM. The significance of the rules will become more apparent as we apply them to our project deign, as we will see later on.

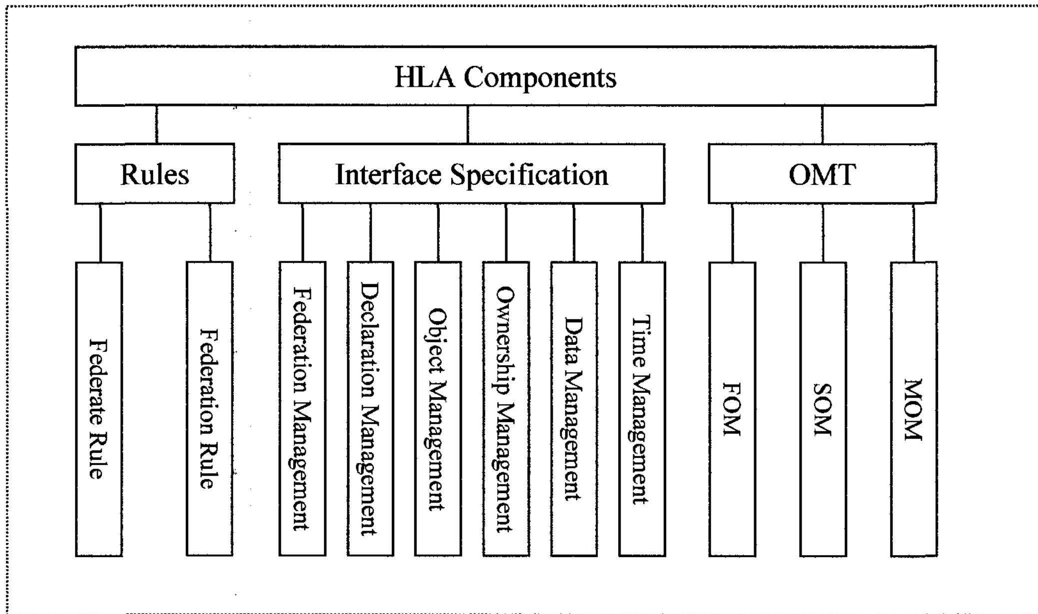



Figure 2-6: HLA Components

In conclusion, HLA has provided a promising framework for DIS simulations. Nowadays, researchers are focusing on the study of the simulation interoperability and the reuse of the components. As the core of HLA, RTI can perform the communication from federate to federate. Moreover, it provides an interface of interactions among federates and enhances the ability of the distributed data management and time management. With the development of the simulation technology, HLA will become more and more widely used in the future.

## 2.2 System Architecture

### 2.2.1 Research Preliminaries

It is first necessary to state the problems that the modeling and simulation study has to address in order to choose the appropriate model type. The presented research has chosen the scenario of an explosion at a university building that will result in fire and human casualties. The purpose of the system is to create a distributed simulation environment where the requirements of an emergency evacuation are addressed through evacuation models and tools, terrain and visualization.

Before modeling of the simulation system can begin, all the relevant information must be collected. This involves the acquisition of geographical/structural data to enhance the ability to respond to threats, emergencies and disasters and to use in the development of emergency operations plans and procedures (e.g., building structure with floor plan details, security provisions, explosion resistance, map of exit locations, availability of response personnel, population density information by time of the day, broadcast and network system). Once those are complete, one should study computational models of building geometry, fuel distribution, smoke and fire flows, and do research on structural and fire damage, as well as human behavior in a panic and high stress situation.

After the identification of the methodology, information sources, and data requirements, the framework of emergency response simulation needs be developed,

which involves the development of the standards for interoperability and integration, the design and demonstration of the distributed simulations by using commercial simulation software and the simulation framework. Then, all the models will be tested and evaluated through a few applications for later use.

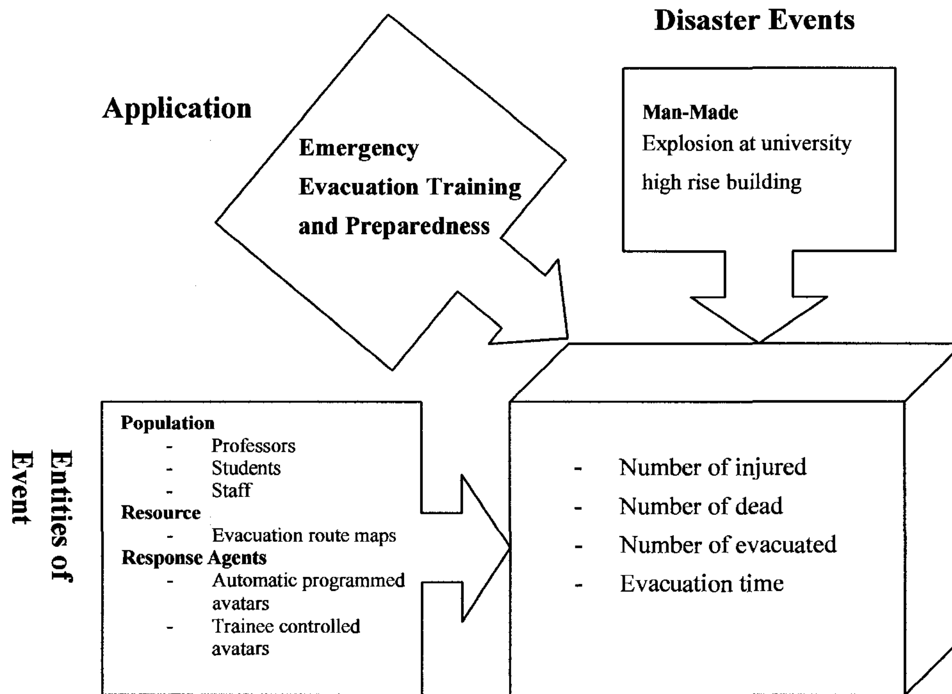



Figure 2-7: The Framework of Emergency Evacuation Training Simulation

The requirements of an overall system for emergency response should be multi-facetted, real-time, and synchronized. No single simulation model or software system is capable of meeting all these requirements. A number of simulation tools have to be integrated to address multiple aspects of a single disaster event. There are a few modeling tools that can be considered, e.g. explosion simulation, fire simulation, emergency response simulation, traffic flow and evacuation modeling, and so on. In

order to apply the designed training simulator on different nodes over the network and to involve other types of simulators later on, the system architecture is to be designed in such ways that there are mechanisms in place to coordinate the initiation, execution, and shutdown of distributed simulations, enabling data transfers from dispersed data sources. As a result, time synchronization is achieved and simulators can be interconnected within distributed environments. The need for reusability and interoperability for multiple simulations (federates) has become the drive for adopting HLA/RTI infrastructure to design the system structure. Figure 2-8 illustrates the conceptual architecture of the simulation.

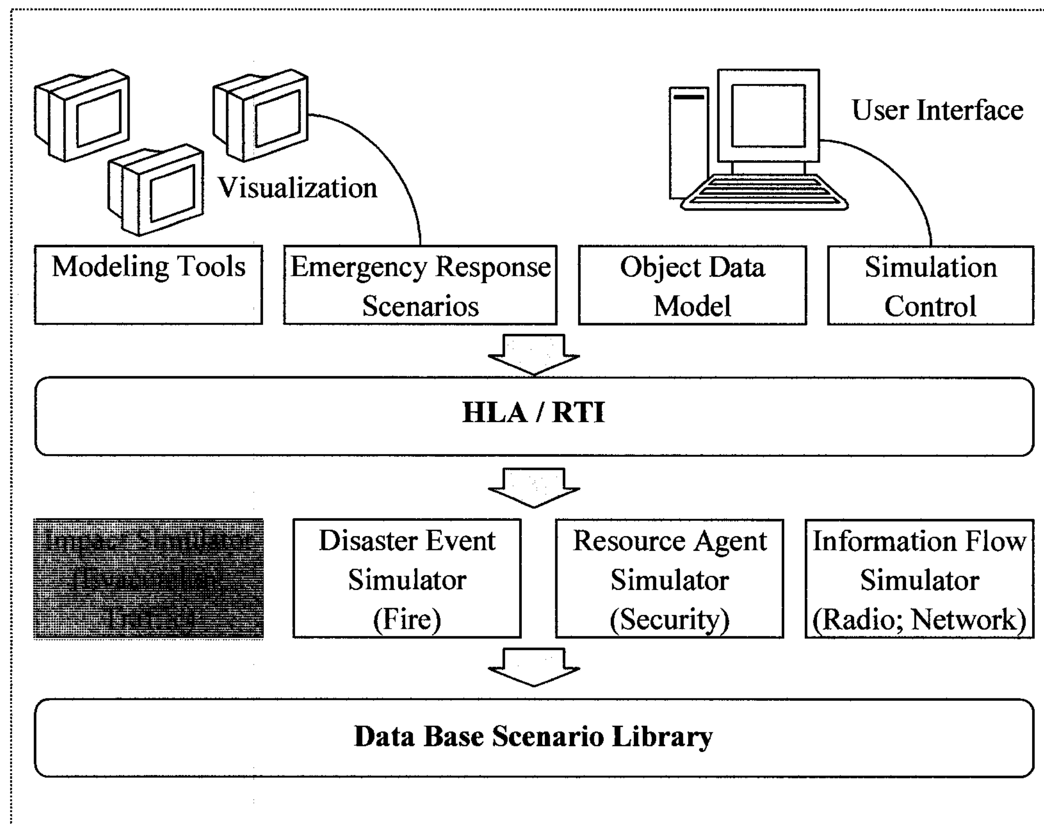



Figure 2-8: Simulation Architecture

### 2.2.2 Application (Simulator) Architecture

Facing the challenges of designing a safety training simulation that is able to emulate real life scenarios, a virtual environment is presented in this research to simulate the events of emergency evacuation in a university building. Figure 2-9 presents the application design architecture that involves various industry standard technologies.

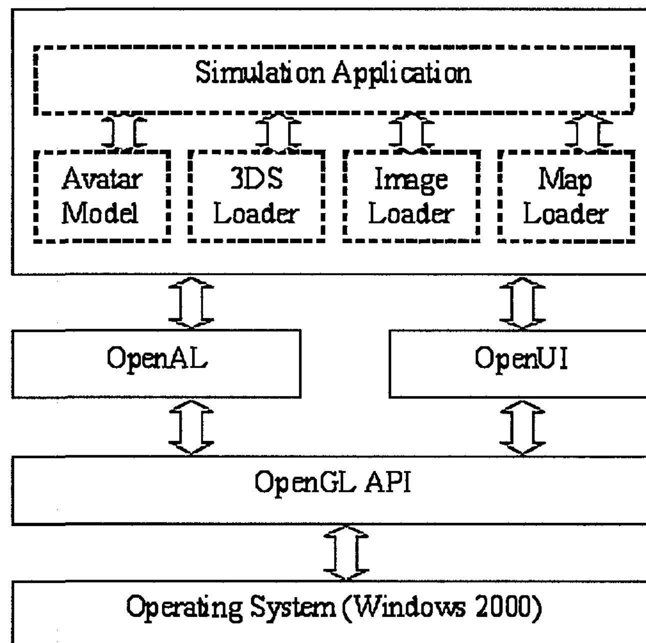



Figure 2-9: Application Architecture

**<u>OpenGL:</u>**

The Open Graphics Library (OpenGL) is a specification which defines a cross-language and cross-platform API for 3D graphics applications [28]. The graphics functionalities provided by OpenGL include, window/Linux standard implementation, PC graphic driver implementation, and OpenGL Architecture Review Board (ARB) specifications.

OpenGL is a rendering engine providing the functions of drawing the geometric and image primitives, such as points, lines, triangles and texture operations. The rasterization-based rendering process is suitable for interactive applications, and this is one of the reasons why OpenGL is chosen in this research project as the development tool. A few other OpenGL related libraries (e.g.: GLUT) are also used. Those extra libraries provide features that are not available in OpenGL [29].

GLUT - OpenGL Utility Toolkit is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Features offered by GLUT include but not limited to: multiple windows control and rendering, callback driven event processing, input devices monitoring, geometric primitives (cubes, spheres) drawing, and support of simple pop-up menus.

GLU is the OpenGL Utility Library. It consists of a number of functions that use the base OpenGL library to provide higher-level drawing utilities. Those features include texture generation, quadric surfaces drawing, and polygonal primitives rendering (NURBS, cylinders), which are applied between screen- and world-coordinates.

**OpenUI**

Any software application needs a friendly user interface to allow the user to interact with the system. OpenUI uses the GLUI – a GLUT-based C++ user interface library, to provide functionality for designing OpenGL based graphical user interfaces (GUI).

GLUI provides GUI widgets such as menus, buttons, checkboxes, and so on. It is OS independent, relying on GLUT to handle all system-dependent issues (system-specific function calls), such as input and output management.

**OpenAL:**

Open Audio Library is a cross-platform audio API, designed for efficient rendering of multi-channel three dimensional positional audio. This technology is applied in our application to create sounds to mimic real life scenario, e.g. the alarm during the simulation execution. The details of implementation will be described in Chapter3.

**Applications:**

In this thesis, the evacuation simulator consists of four main components. The *Avatar/Agent Model* is a representation of human objects that can perform intelligent behaviors and animations. It is developed by using OpenGL technology. The *3ds Model Loader* loads the graphics models - *.3ds* files, generated by 3DsMAX, converts them to scene-graph compliant ones and delivers them to the application for rendering. The *Image Loader* maps the image files (textures in *.tga* format) to OpenGL and displays them later on. The last one is the *Map Loader* which can generate the shortest path to exit, based on each avatar's position. The map loading mechanics is designed based on 2D coordination calculation, which can be easily adapted to other 3D models, as long as the *Map Loader* is modified to adapt to their corresponding structures.

The discussion so far has addressed the ground work needed for the actual implementation of a simulation system. The next few chapters will describe how ***SimSITE***, the HLA/RTI based evacuation training simulation, is developed.

# Chapter 3
# Simulation Graphics and Associated Audio

## 3.1 Static 3D Graphics Modeling

Graphics modeling, which is the first and foremost step to realize verisimilitude in VR-based simulations, involves the modeling of 3D virtual space and human avatars.

### 3.1.1 Modeling of 3D Virtual World

The static virtual world is modeled by 3dsMAX7.0 in light of the SITE building at the University of Ottawa [30] as the prototype. The reason behind choosing the SITE building is its complex floor plans with multiple exits and evacuation routes, where students, professors, and support staff occupy different floors. The complexity of this building makes it possible to create different realistic scenarios. Compared with hand drawing in vertex coordinates by OpenGL or other graphics modeling platforms, the graphics modeled by 3dsMAX allows us to create any arbitrarily complex objects in a more intuitive and human way.

The architecture of the actual SITE building was studied intensively. The designers went to the building many times to take pictures of the layouts, collect the geometry data, sketch the architecture, do the measurement, and compare the model with the real entities from different perspectives. Based on the 2D floor maps (example shown in Figure 3-1) provided by the security office, a primitive 3D model was initially

constructed. It contains 5 different stories shown in Figure 3-2, which was built by mesh, polygon, NURBS, surface modifier and other predefined primitives modeling components provided by 3dsMAX7.0. By using the function of "*EXTRUDE*", 2D schemes can be translated to three-dimensional ones. Example shown in Figure 3-3 illustrates how the 3D wall is built. A few 3D furniture models (see Figure 3-4) were originally imported from some open source websites [31]. They were modified to meet the requirements of this simulation.

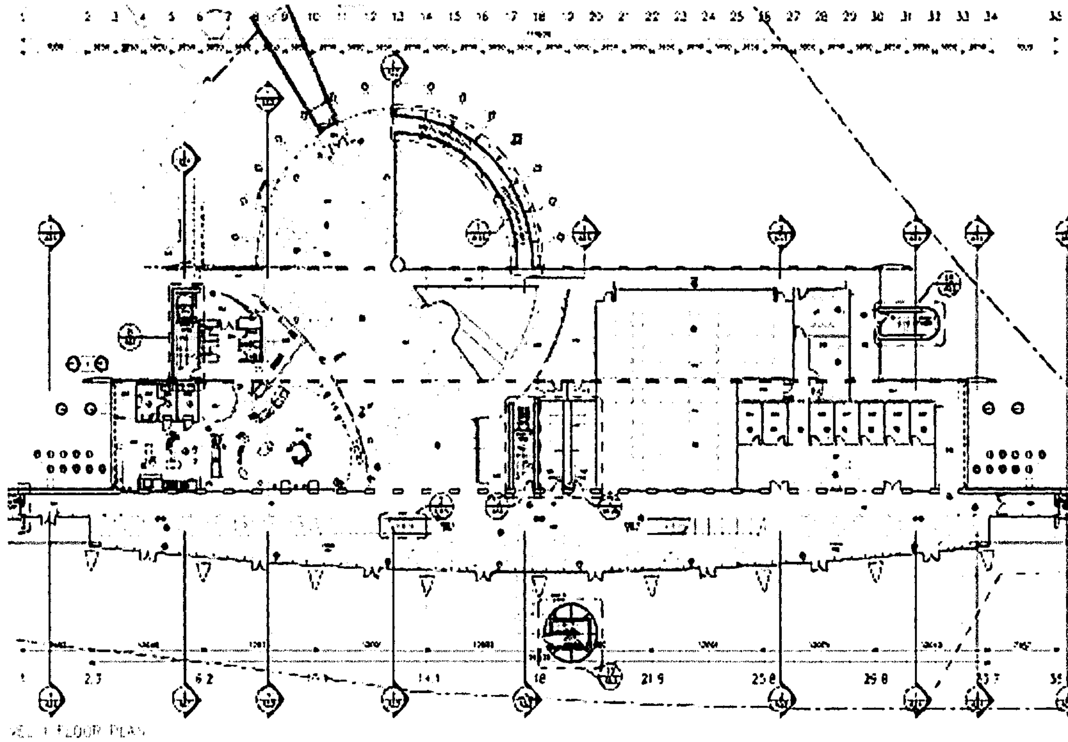


Figure 3-1: 2D SITE Ground Map

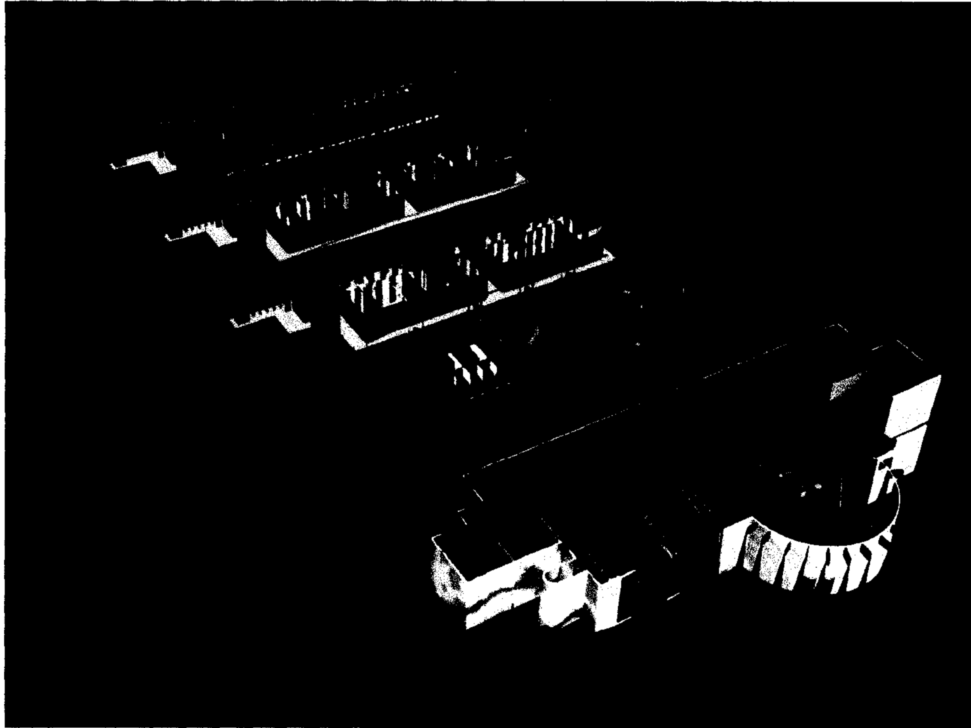
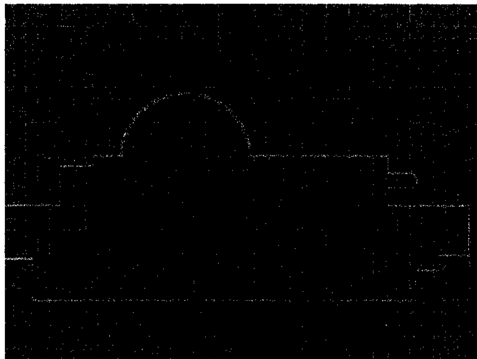
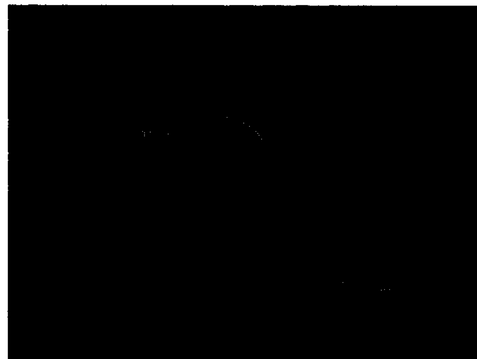


Figure 3-2: Basic 3D Models of the SITE Building with Five Stories



(a) 2D Wall Contour Line



(b) 3D Result after Applying "*EXTRUDE*"

Figure 3-3: Converting 2D Spline (Wireframe) to 3D Solid Model

Figure 3-4: Imported 3D Furniture Models

Texture generally means the details of a surface which includes hatching, patterns, illuminating and material information. Texture mapping is the process of placing a raster graphic image, or wrapping a texture image, on object surfaces during graphic rendering. No matter how elaborate the object model has been created, an object without texture just looks like a human skeleton without skin. Most of the textures used in this project were created by photographing the real entities of the SITE building. Figure 3-5 shows some of the textures that have been applied to the system. The rendering results show that the same scene with the texture binding [32] (Figure 3-6) is more realistic than the one that does not have any texture applied (Basic Model Figure 3-2).

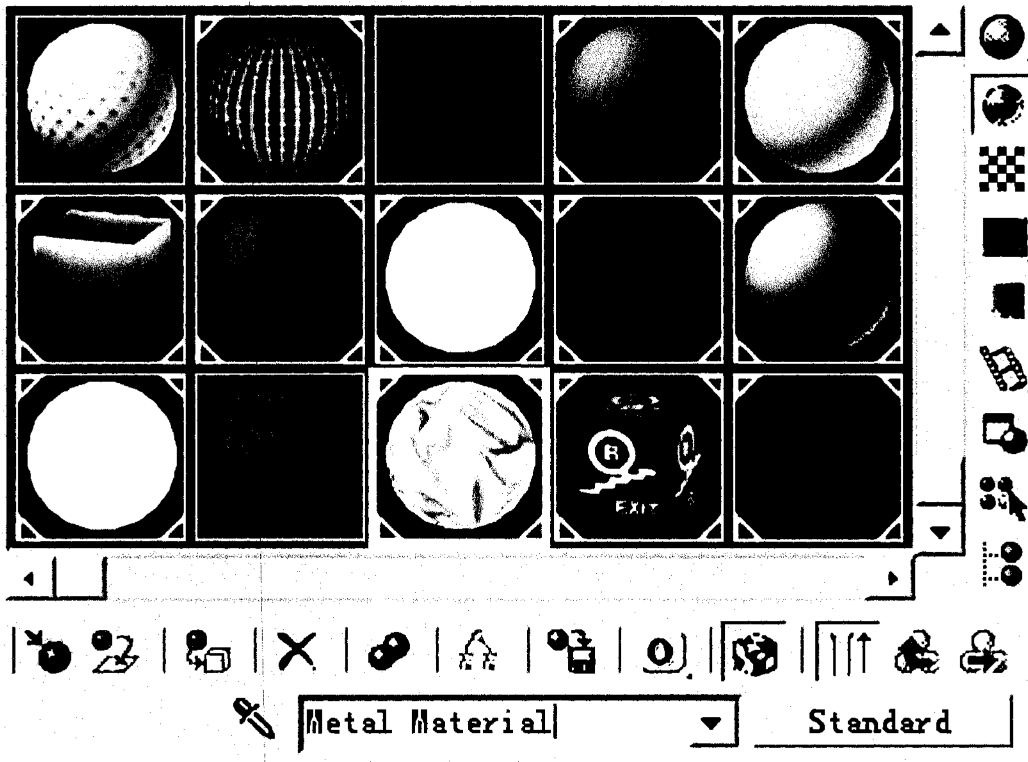
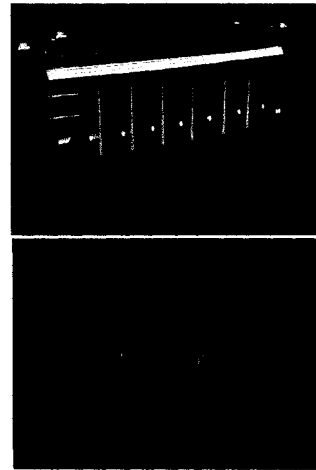
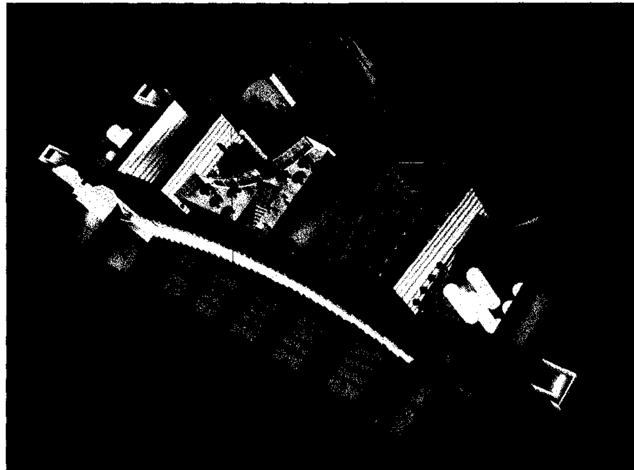



Figure 3-5: Texture Materials

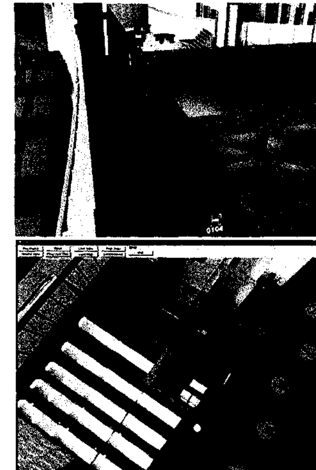
**(a) Front View of the Building**



**(b) Back and Side View**



**(c) Ground View of the Building**



**(d) Other Parts of the Building**

Figure 3-6: 3D Graphics Modeling of the SITE Building:

So far, the static 3D virtual environment has been developed. In the next section we will discuss the main actors – avatars in the virtual world.

### 3.1.2 3D Avatars Design

The 3D avatars were referenced [33] and modified in *Visual Studio .NET 2003* development platform by using the C library - OpenGL. The Avatar's appearance was

created according to the prototype of the famous cartoon character – Homer Simpson [34]. The same avatar will be applied to the different players (student, professor, and staff) in the ***SimSITE*** system with different colours.

The hierarchical modeling in Figure 3-7 shows the different components in the graphic design. The top level component (e.g.: head) forms the base of a model and there are subsidiary components (e.g.: ears, eyes, moth, nose) that fall beneath that base. OpenGL provides the capability to associate different components in a scene for composing complex combinations and transformations. By applying transformation technologies, the shape of a primitive can be scaled, the orientation can be rotated, and the position can be translated; however, the order in which these transformations are performed is important. A different rendering result will be generated if the transformation is swapped. More explanation about transformation will be given in section 3.2.2 – "3D Transformation and Corresponding Coordinates", which is the most fundamental technique in the design of an OpenGL based 3D simulation engine. Besides the transformation functions, function *glClipPlane()*can be used to take plane equations to cut a geometry object in anyway you prefer. Finally, OpenGL uses *Matrix Stacks* functions *glPushMatrix()* and *glPopMatrix()* to facilitate the collection and hierarchical grouping of components. By combining the transformation with clipping functions, a 3D avatar was successfully drawn on the screen as shown in Figure 3-8.

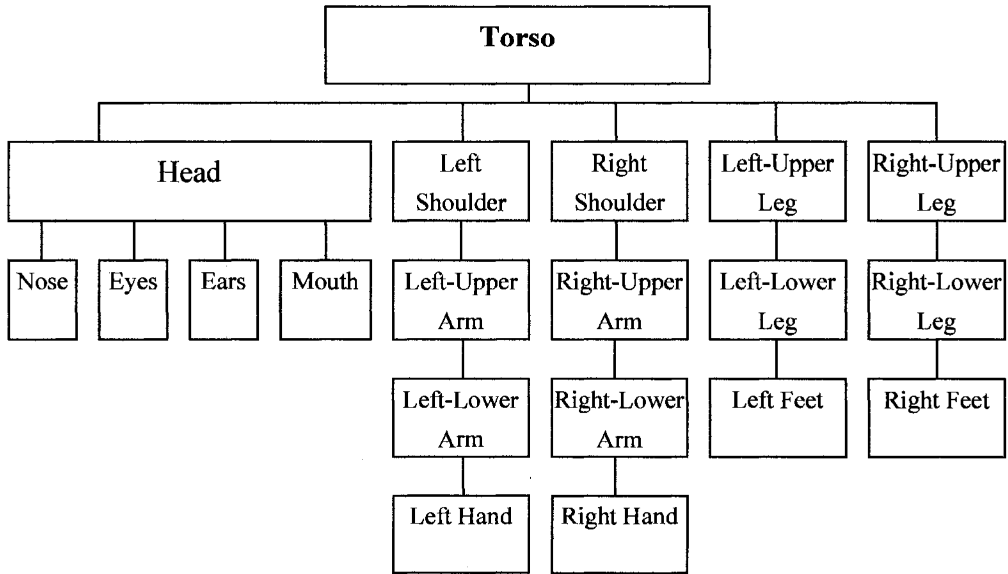
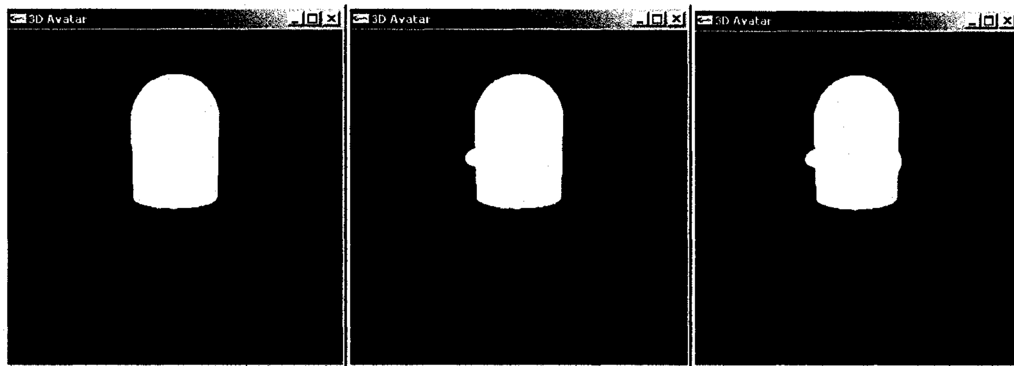
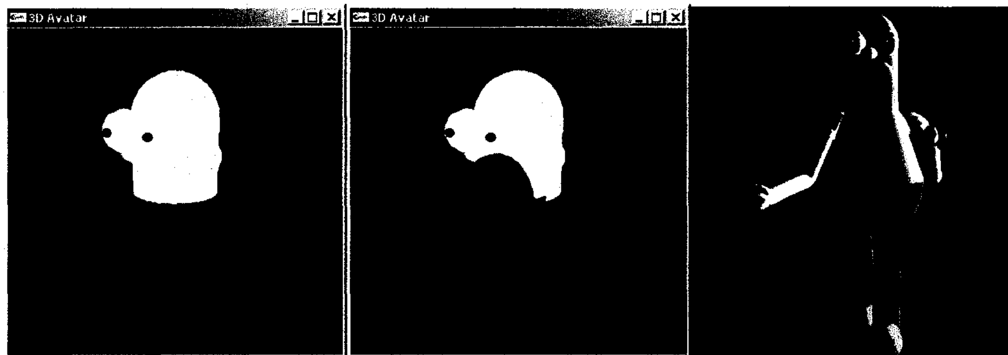



Figure 3-7: 3D Avatar Hierarchical Modeling



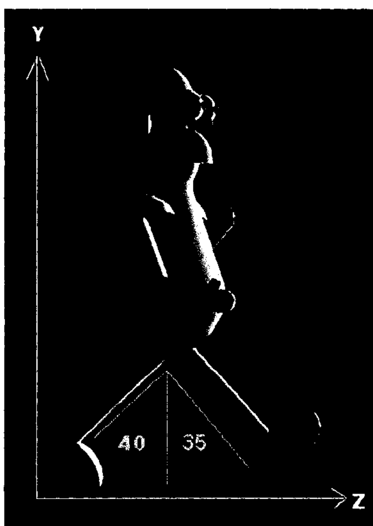

(a) Cobbra (Head) (b) Nose (c) Ears




(d) Eyes (e) Mouth (f) Overall Appearance

Figure 3-8: 3D Avatar Design

Once the static model is completed, human and social features need to be added to the avatars, such as walking on the ground and running up and down the stairs. Walking/running actions of an avatar should be synchronized with the movements of the shoulders, arms, knees, and legs. The synchronization is controlled by different functional variables. To be more realistic, each part of the body's movement should be restricted in a particular range, as can be seen in the example of the avatar's leg movements in Figure 3-9. Once positive limitation of the joint flexion is reached, each leg moves to the opposite direction, and step variables are increased or decreased respectively until reaching the negative limitation. These changes are rendered eventually in the rotation function *glRotatef(variable, x, y, z)*. Similarly, this can be done with the shoulders/arms movements to achieve the walking effects of an avatar. By increasing the scopes of the step variables, a walking action can be eventually translated into a running action.




```
--------------------------------------------------------------
/************* Avatar Leg Movement ************/
if (left_legYZ>=35 && right_legYZ<=35)
            legdirection=1;
if (left_legYZ<=-40 && right_legYZ>=-40)
            legdirection =-1;
if (legdirection>0){
            left_legYZ   -=15;
            right_legYZ +=15;
        }
else{
            left_legYZ   +=15;
            right_legYZ -=15;
        }
--------------------------------------------------------------
```

Figure 3-9: Avatar's Walking Action

So far, the avatars have the capability to move. By altering the rendering positions (x, y, z coordinates), they can travel and walk to different stories by taking the stairs. To simulate the evacuating behaviors of human beings in ***SimSITE*** system, the avatars cannot only move but they are also able to find the optimized path to escape and avoid the collisions simultaneously. The details on the movement of avatars in ***SimSITE*** will be discussed in section 3.2.8 - "Shortest Path Calculation".

## 3.2 Graphic Control in C++

Graphic control and rendering plays an important role in graphic design, which addresses how to construct a 3D simulation engine using OpenGL. A 3D simulation/game engine is a combinational structure where functions and algorithms are used to calculate, visualize 3D objects on a 2D screen after various transformations. The ***SimSITE*** graphic engine performs many functions such as object data structure modeling, positioning objects in an environment by 3D transformations, 3D models and textures loading, guiding the avatars' movements in various scenarios, and finally rendering scenes on the screen. The engine has to also obey the interactive program structure that performs the application functionalities in separate threads generated from user interface. After the setup, the main program executes in a dead loop, and the communication among threads is based on event-driven or message passing scheme.

### 3.2.1 Data Structure Modeling

Before getting into the discussion of the data manipulation, it is helpful to explain the representations of the data structure [35] which contains the model information to present the objects such as a 3ds model in the scene, and load the related material information.

The basic elements of an object are the VERTICES. Every vertex is composed of two (x, y) or three coordinates (x, y, z) depending on where it is located on a plane or in a

space. To achieve the best resolution in the graphics rendering, each coordinate is expressed by a FLOAT/DOUBLE variable.

```
------------------------------------------------------------------------------------------------
class CVector3 // 3D point class used to store the vertices of our model
    { public: float x, y, z;};
------------------------------------------------------------------------------------------------
```

When dealing with the texture mapping, a COORDINATE MAPING structure is introduced that initializes two variables U and V to assign each vertex to a 2D point of the image and cover the object as desired.

```
------------------------------------------------------------------------------------------------
class CVector2
    { public: float u, v; }; //coordinates used to identify a 2D point in a texture
------------------------------------------------------------------------------------------------
```

The FACE structure describes the polygons of the object. Most of polygons are composed of 3 vertices. This structure defines the index of the vertex and texture coordinate arrays, and provides the vertices to their corresponding face along with the correct texture coordinates.

```
------------------------------------------------------------------------------------------------
struct tFace
    { int vertIndex[3];   // forms the triangle
      int coordIndex[3];  // indices for the texture coordinate to texture this face
    };
------------------------------------------------------------------------------------------------
```

The MATERIAL structure holds the information for an object material that contains the name of the material, the color and texture of the material, the texture ID, and the U, V tiling and offset info.

```
------------------------------------------------------------------------------------------------
struct tMaterialInfo
    { char   strName[255];        // texture name
      char   strFile[255];        // texture file name
```

```
    BYTE   color[3];              // color of the object (R, G, B)
    int    texureId;              // texture ID
  };
```

The OBJECT structure that contains all the information for the model/scene, identifies fields such as the number of the vertices, faces, and texture coordinates in a model, the presence of the texture and its ID, info of the object vertices, faces, normals, and UV coordinates.

The last and the most outer structure is the MODEL structure, which holds the 3D model objects and the materials information.

```
struct t3DModel
  {  int numOfObjects;                    // number of objects in the model
     int numOfMaterials;                  // number of materials for the model
     vector<tMaterialInfo> pMaterials;    // material list
     vector<t3DObject> pObject;           // object list
  };
```

Till now the basic structures that are needed to draw a 3D object have been defined.

### 3.2.2 3D Transformation and Corresponding Coordinates

In the previous section, we constructed the first part of the rendering pipeline: how to acquire the object data and store the data in a structure. Before the actual drawing of a 3D scene by using OpenGL, the coordinate system needs to be set up to perform the viewing process; and then, the transformations [36] will be applied to place the objects in the 3D world and render the scene on the 2D screen. All 3D transformations are based on matrices manipulations. Every 3D vertex mainly needs to go through

three kinds of transforms (multiply three transform matrices) like the graphic pipeline shown in Figure 3-10.

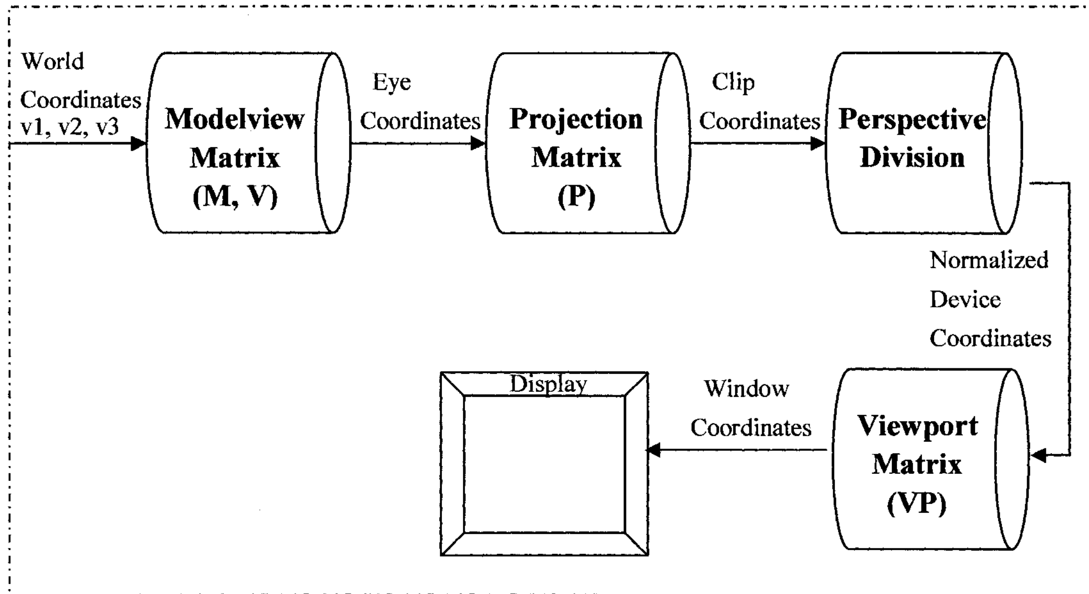



Figure 3-10: OpenGL Graphic Pipeline [37]

As we can see from the figure above, the first applied matrix is called *Modelview Matrix*; as the name indicates, this matrix combines the objects modeling transformation and the camera viewing transformation to position and rotate the camera. After *Modelview* transformation, the current coordinate system has been converted from original *world* coordinate to *camera (eye)* coordinate system where the object lighting and texture generation are applied. The following functions are used to set the modeling transformation in ***SimSITE***:

- glScaled (); // produce a non-uniform scaling
- glTanslated();// move object into 3D space
- glRotated(); // apply rotations of angle degrees around the vector

These three functions have already been used and introduced in section 3.1.2 - "3D Avatars Design". The result of applying the modeling transformations is illustrated in Figure 3-11.

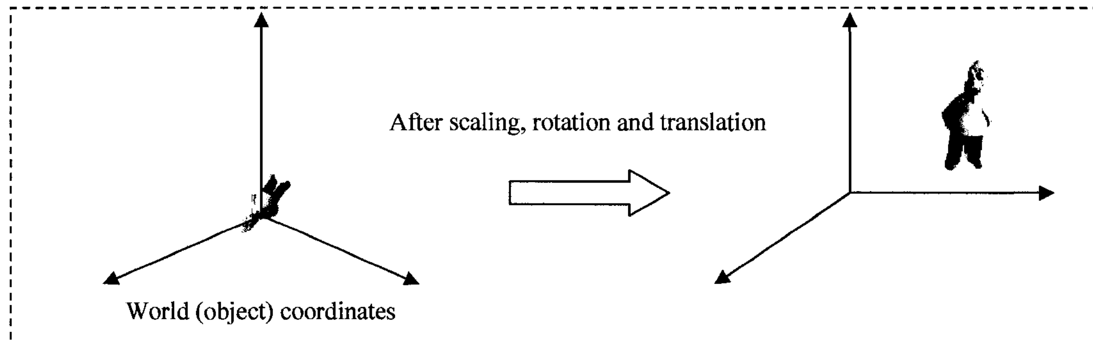



Figure 3-11: Modeling Transform

Functions are used to set the viewing transformation in ***SimSITE***:

- glMatrixMode(GL_MODELVIEW); // switch to Model View mode
- glLoadIdentity();// reset matrix
- gluLookAt (eyex, eyey, eyez,lookatx,lookaty,lookatz, 0.0, 1.0, 0.0);

// eyeX, eyeY, eyeZ specifies the position of the eye point; lookatx, lookaty, lookatz specifies the position of the reference point.

The effect after applying the viewing transformations is presented in Figure 3-12.

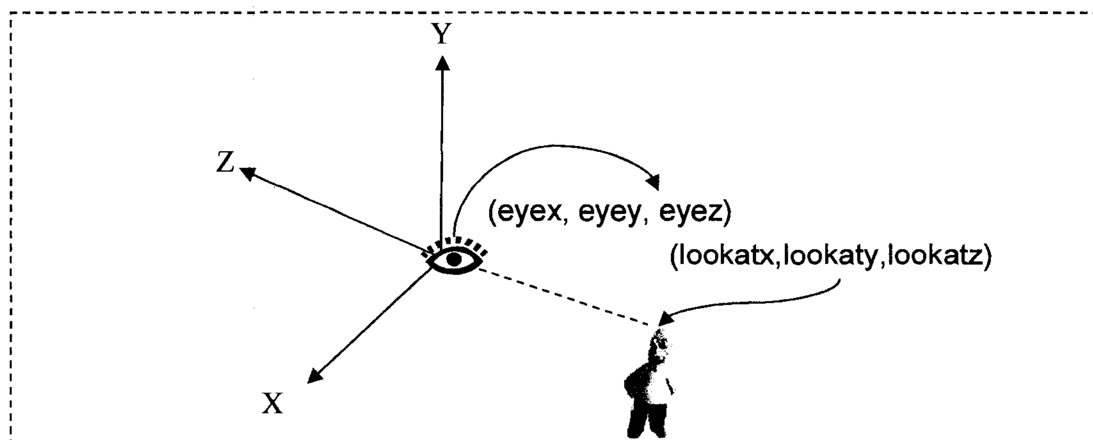



Figure 3-12: Setting Camera (Eye) Position

The *Projection Matrix* transforms the vertices into a normalized coordinate system. By setting up the *perspective transformation*, the camera view volume (zoom) is decoded in projection matrix. Then, by adjusting the *Orthographic projection*, the boundaries of the parallel viewing volume is specified, and the 3D scene is finally mapped onto a 2D screen. Generally, *perspective projection* is referred to as a projection that consists of both the *perspective transform* and the *orthographic projection*, i.e., *perspective projection* = *perspective transform* + *orthographic projection* [38]. After the *perspective projection* transformation, the current coordinate system is converted to the *clip* coordinate system where the objects or portions of objects that are excluded from the view volume will be clipped away. Functions are used to set the *perspective* transformation in ***SimSITE***:

- glMatrixMode (GL_PROJECTION); // switch to projection mode
- glLoadIdentity();
- gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 300.0);

The result after using the *perspective* transformations is shown in Figure 3-13.

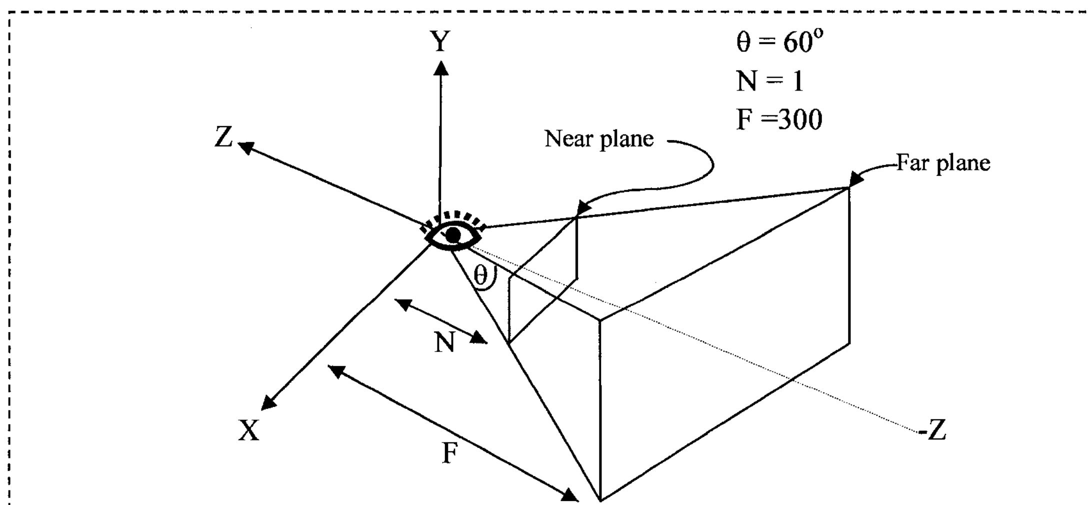



Figure 3-13: Setting View Volume (Default)

The following functions are used to set the *orthogonal projection* transformation in ***SimSITE***:

- glMatrixMode(GL_PROJECTION);
- glLoadIdentity();
- gluOrtho2D(0,txtWidth, 0, txtHeight);// left, right , bottom, top

The result is demonstrated in Figure 3-14.

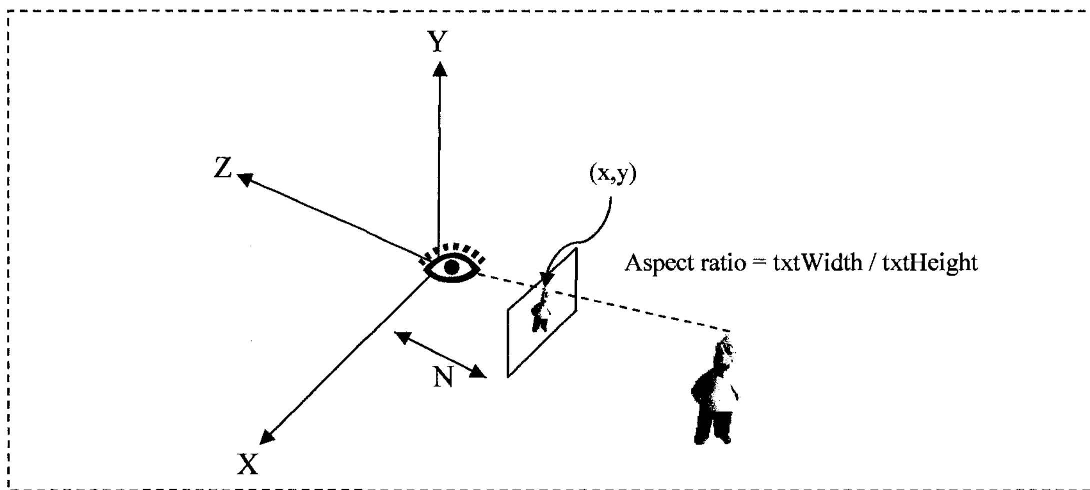



Figure 3-14: Orthographic Projection (3D Scene Mapping onto 2D Screen)

The last transformation after performing the clip manipulation is achieved by applying the multiplication of the *Viewport Matrix*. The *Viewport* transformation maps the surviving portion of the block into a viewport that always matches the aspect ratio of the screen window. It converts all the points that will be used to the current viewport resolution. The function below is used to set the *viewport* transformation in ***SimSITE***:

- glViewport (0, 0, (GLsizei) w, (GLsizei) h);

// lower left corner, width and height of the viewport

And the results are shown in Figure 3-15.

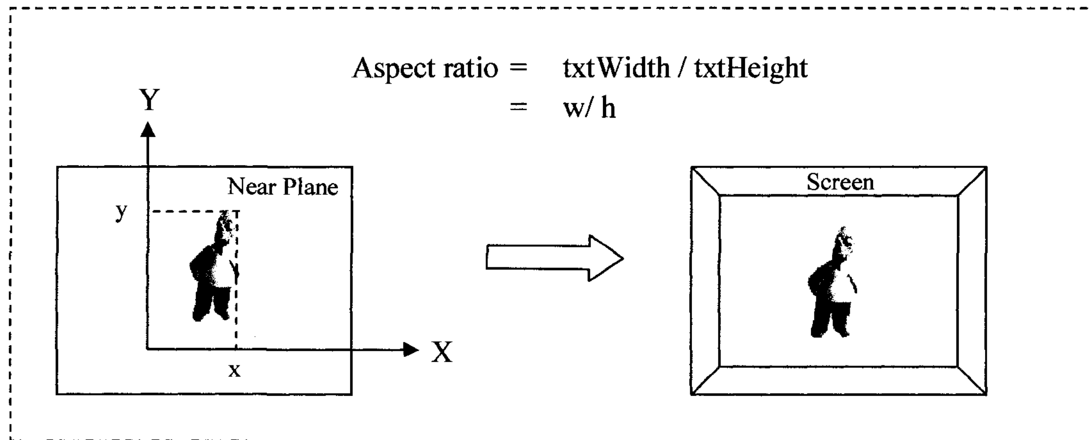



Figure 3-15: Setting View Port on 2D Screen

In a conclusion, each vertex of an object is subject to a sequence of transformations that transform it from the world coordinates (object coordinates) to camera coordinates, and later to clipping normalized coordinates, and finally, into the window coordinate system for screen display. Transformations are widely used in many OpenGL applications. In ***SimSITE,*** they are mainly applied in rendering, display, and windows reshape functions.

### 3.2.3 3ds Data Structure

In section 3.1, we discussed the creation of the static 3D graphic models (the virtual world), which are in .3ds format exported from 3dsMAX. A 3ds file typically consists of data called chunks. Chunks contain the information to describe the details of a 3D scene, which includes objects, materials and key frame information, such as object name, face description, lighting and camera information, color and texture, and mesh block. The chunks are arranged in a hierarchical structure so that the sub-chunks can

only be accessed relative to their parent chunks have been achieved first. Every chunk is composed of three fields: identifier, length, and data. The *chunk identifier* is a unique code to identify the type of data in the block. The *chunk length* shows the sum of the data length including the attached subordinate chunks. To parse out sub-chunks, the format of their parent truck should be known. According to the *chunk identifier*, if a chunk is useless, the parsing program can skip the entire chunk and its subordinate chunks. And, the *trunk data* contains all the data for the scene with variable length. Below is a table of the order that shows the distribution of most important chunks and sub-trucks in a 3ds file.

```
PRIMARY (0x4D4D)
    OBJECTINFO (0x3D3D)
        EDIT_MATERIAL   (0xAFFF)
            MAT_NAME            (0xA000)
            MAT_DIFFUSE         (0xA020)
            MAT_MAP 1           (0xA200)
                MAT_FILENAME            (0xA300)
        EDIT_OBJECT         (0x4000)
            OBJ_MESH            (0x4100)
                OBJ_VERTEX              (0x4110)
                OBJ_UVCOORS             (0x4140)
                OBJ_FACE                (0x4120)
                    OBJ_SMOOTH                  (0x4150)
                    OBJ_MATERIAL                (0x4130)
                OBJ_LOCAL               (0x4160)
                OBJ_VISIBLE             (0x4165)
            OBJ_LIGHT           (0x4600)
                LIT_OFF                 (0x4620)
                LIT_SPOT                (0x4610)
            OBJ_CAMERA          (0x4700)
    EDITKEYFRAM (0xB000)
        KEYF_OBJDES         (0xB002)
    VERSION (0x0002)
        :
        :
        :
```

Table 3-1: 3ds File Hierarchy [39]

Once we understand how to read chunks by searching for the ID, we will be able to retrieve the information stored in chunks. As the hierarchy shows, the PRIMARY chunk (4D4D) indicates that the file is a valid file in 3ds format. As aforementioned, to read a particular element, its parents must be accessed first. For instance, to reach the chunk MAT_NAME (0xA000), the chunks have to be accessed in the following order: the PRIMARY (0x4D4D) trunk, then the OBJECTINFO (0x3D3D) trunk, and finally the EDIT_MATERIAL (0xAFFF) trunk.

A 3ds object is loaded and saved in the format defined by the engine in the following order:

1) A 3ds file is imported and opened. After the first trunk is read, the trunk ID and the chunk length are obtained. To make sure this is a valid 3ds file, checking must be done by examining the first chunk to verify whether it is equal to PRIMARY (4D4D).

2) A "while" loop is implemented in every main trunk reading procedure, which continues its execution until the end of the trunk. The trunk info is read in each iteration of the loop, and the chunk ID is analyzed via a switch. The meaningless trunk is then skipped by updating the trunk length.

3) If the chunk reading allows us to reach another chunk (sub-trunk), for example, MATERIAL or OBJECT trunk, we can move to that trunk and read all the data. The following code handles the sub-trunk reading situation when next trunk is an OBJECT trunk (0x4000).

---

```
//************* Handle all the information about the objects *******************
void CLoad3DS::ProcessNextObjectChunk (t3DModel *pModel, t3DObject
*pObject, tChunk *pPreviousChunk)
{
    while (pPreviousChunk->bytesRead < pPreviousChunk->length)
    {
        ReadChunk(m_CurrentChunk);
        switch (m_CurrentChunk->ID)
        {
        case OBJECT_MESH:            //(0x4100)
            // indicate next trunk will be a new OBJECT trunk
            // read information using recursion
            ProcessNextObjectChunk(pModel, pObject, m_CurrentChunk);
```

```
        :
    case OBJECT_VERTICES:        //(0x4110)
        ReadVertices(...);       // read objects vertices info
        :
    case OBJECT_FACES:           //(0x4120)
        ReadVertexIndices(...);  // objects faces data
        :
    case OBJECT_MATERIAL:        //(0x4130)
        // object owned material name could be a color or a texture map.
        ReadObjectMaterial(...);
        :
    case OBJECT_UV:    // (0x4140) object UV texture coordinates
        ReadUVCoordinates(...);
        :
    }
}
```

---

4) All the materials info will be read from MATERIAL trunk (0xAFFF) with the same idea as the OBJECT trunk reading. Details will be covered in the next section.

### 3.2.4 Texture Loading (Mapping)

In ***SimSITE*** project, two texture loading techniques have been implemented. They differ in where the texture coordinates are computed. Both of them use Targa (TGA) format images as the original texture recourses due to their high resolution and transparency supporting capabilities.

The first texture mapping mechanism is embedded in the 3ds loader program, which was discussed in the previous section. By using the object ID, the corresponding texture information is retrieved from the 3ds module and loaded into OpenGL. At this stage, the texture is not detached from the object, but is treated as a part of the object

and rendered with the object at the same time. Before we talk about texture mapping, let's review the 3ds data structure first.

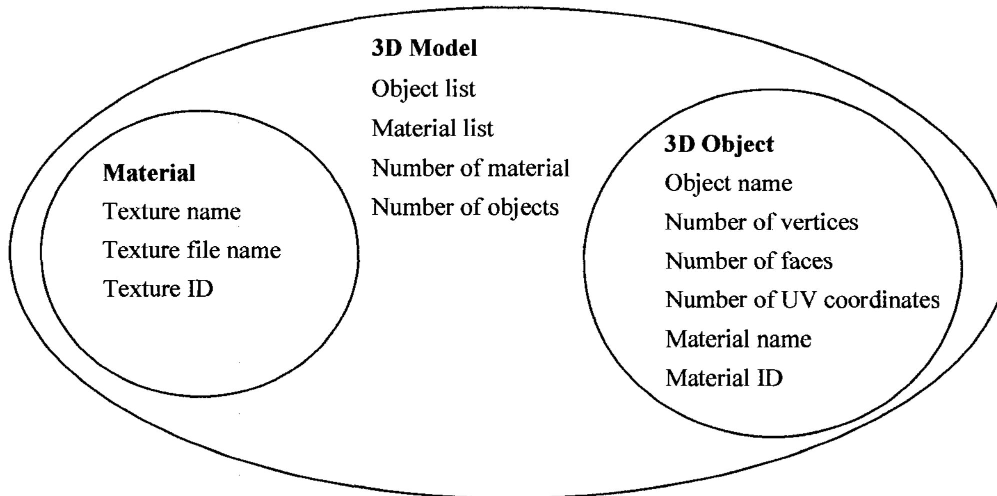



Figure 3-16: 3ds Structure

As shown in Figure 3-16, the 3ds model contains two critical components - the *material* and the *object*. If the object is designed by 3dsMAX with a relative material, no matter whether it is monochrome or actual texture map (image texture), when the loader reads the MATERIAL chunk (AFFF), the *material name* (A000) will be obtained. We call it *texture name* in the following sections to distinguish it from the *material name* (4130) that is obtained from OBJECT trunk. The texture name will be stored into the material list in *3D model* structure. Later on, when the loader reads the OBJECT trunk (4000), the sub-trunk named OBJ_MATERIAL (4130) will be accessed. By reading this sub-trunk, the loader retrieves the *material name* bound by this object. Then, the loader needs to go through all the textures stored in the material list and checks which texture matches the *material name* that the loader has just

obtained from the OBJ_MATERIAL trunk. Finally, the *materialID* of the object is assigned to the material index (preparation for rendering).

```
//********************** Read MATERIAL trunk (AFFF) ****************************

void CLoad3DS::ProcessNextMaterialChunk(t3DModel *pModel, tChunk
*pPreviousChunk)
{
    while (pPreviousChunk->bytesRead < pPreviousChunk->length)
    {
        ReadChunk(m_CurrentChunk);
        switch (m_CurrentChunk->ID)
        {
        case MATNAME: // (A000) This chunk holds the name of the texture
            // read in the texture name, and store it into material list
            fread(pModel->pMaterials[pModel->numOfMaterials - 1].strName,
            1, m_CurrentChunk->length - m_CurrentChunk->bytesRead,
            m_FilePointer);
        :
}
```

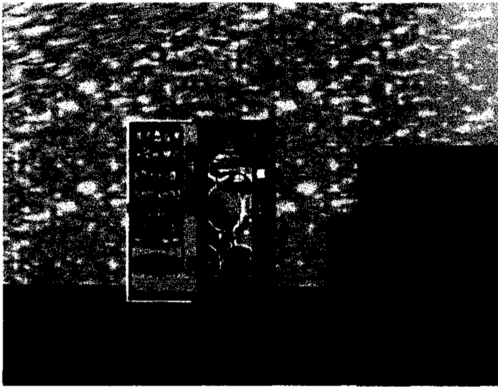
```
//*********************** Read Object Material chunk (4130) *********************

//Read in material name assigned to the object, and set the materialID
ReadObjectMaterial (t3DModel *pModel, t3DObject *pObject, tChunk
*pPreviousChunk)
{   //get the material name
    pPreviousChunk->bytesRead += GetString(strMaterial);
    // Go through all of the textures stored in material list
    for(int i = 0; i < pModel->numOfMaterials; i++)
    {   // If the material we just read in matches the current texture name
        if(strcmp(strMaterial, pModel->pMaterials[i].strName) == 0) {
            pObject->materialID = i;   // Set the material ID
            // we found the material, check if it's a texture map.
            if(strlen(pModel->pMaterials[i].strFile) > 0)
                { pObject->bHasTexture = true; } // object has a texture map
        }
    }
}
```

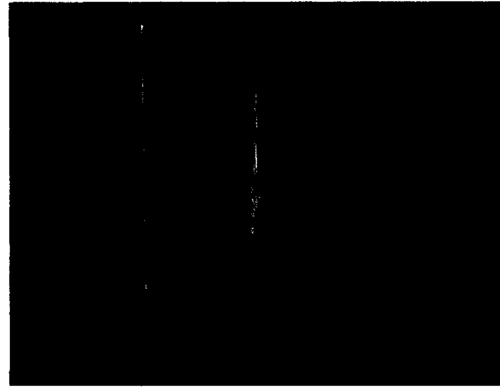
After reading the material and object information from the 3ds file, before the actual rendering process starts, it should be clear as to where the material comes from, whether it is obtained from the image files or the 3dsMAX material library. If the material is obtained from the library (usually it is just monochrome), the *bHasTexture* flag will show false. The object *color* information will be simply passed into OpenGL and rendered afterwards (see section 3.2.5 in detail). However, if the material is from TGA files, we need to firstly build the textures from those source images; then, set the *texture ID* for this material; and finally at the rendering stage, use a 2D texture mapping mechanism to physically bind the texture to the target object and display them together on the screen.

The second approach is to perform textures loading at the refining stage. It contains all visual effects including distortions with respect to the projective transformations. For instance, the 3ds loader designed by OpenGL1.4 only maps the images with the size (width and height) of power of 2; while, 3dsMAX does not have this restriction, i.e., any format of images can be used to the 3ds models for the purpose of texture mapping. However, when these models are loaded into OpenGL, some textures which do not fit the criteria can not be displayed properly, which is illustrated in Figure 3-17. In this case, instead of modifying and completely reloading the entire 3ds models, we can simply overlay those models with adjusted textures after they have been loaded into OpenGL. Or we can even add some new objects directly sketched in OpenGL and apply the textures on them. The operation provides one-to-one mapping between

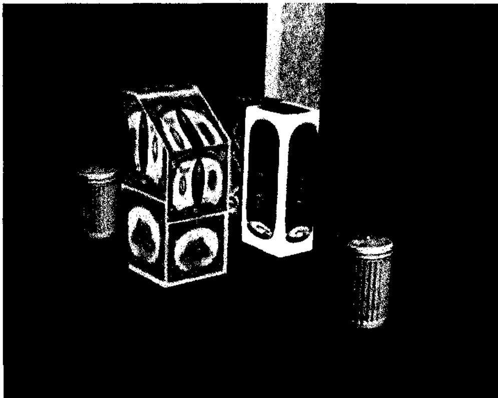
objects and textures.



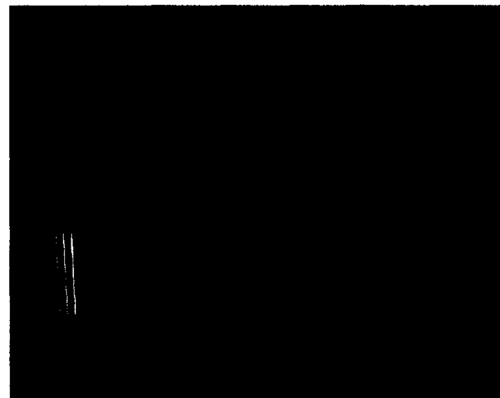
(a) Original Texture Mapping



(b) Texture Mapping Upside-down

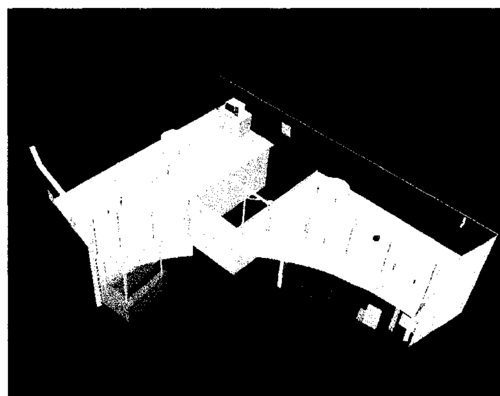


(c) Original Texture Mapping



(d) Texture Mapping Distortion

(e) Original Texture Mapping



(f) Texture Mapping Losing

Figure 3-17: Texture Mapping Distortion

Furthermore, in the ***SimSITE*** application, it is very helpful if exit routine maps are offered in the evacuation simulation in the case that the trainees lose their directions in the building. By applying the second technique, a static image (floor map) can be displayed on the user interface. This can be done by first generating some texture objects; and then applying the routine *bineTexture()* to read all TGA image information into the raster image buffer. Inside the routine, image is loaded first to logically bind the texture onto its target; and the texture environment is set up; finally, a 2D texture is created by *glTexImage2D()* method that retrieves the image data from the memory buffer, and passes this texture back to OpenGL to render. In the *display()* routine (located in *main()* program), after the 2D texture is created, we can physically bind the texture onto the target object, and render them together. In the following examples, the target is a quadrangular polygon. Different textures (e.g.: maps) will be mapped onto the polygon, depending on the scenario (e.g.: a different floor). The counter-clockwise fashion [40] (see Figure 3-18) is applied to map the texture coordinates onto each vertex. The results are shown in Figure 3-19.

---

```
//*********************************** 2D Texture Mapping ***************************
Initial()
{
    mapItems[0]="map1.tga"; mapItems[1]="map2.tga" //2 map objects
    glGenTextures( 2, mapObjects); //generate 2 textures
    bindTexture( mapItems[mapNo],mapObjects, mapNo); // texture loading
}
```

---

```
//************************** 2D texture rendering stage *****************************
Display()
{
    glEnable(GL_TEXTURE_2D); //enable texture mapping
    //active the texture and bind it to object
    glBindTexture (GL_TEXTURE_2D, mapObjects[mapNo] );
    // counterclockwise texture coordinates mapping to vertices
    glBegin(GL_QUADS);
        glTexCoord2i(0, 1);
        glVertex2i(80, 40 );
        glTexCoord2i(1, 1);
        glVertex2i( 740,40 );
        glTexCoord2i(1, 0);
        glVertex2i( 740, 500);
        glTexCoord2i(0, 0);
        glVertex2i( 80, 500);
    glEnd();
}
-------------------------------------------------------------------------------------------
```

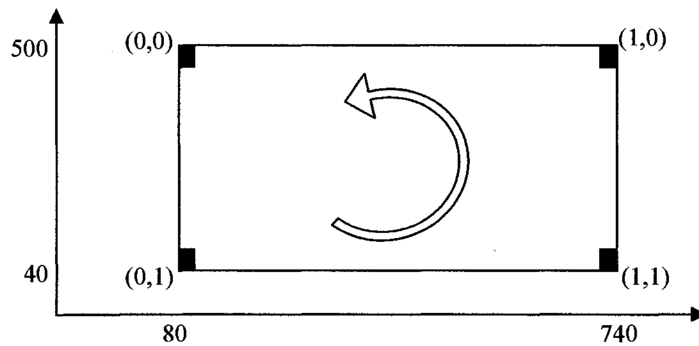



Figure 3-18: Counterclockwise Texture Mapping

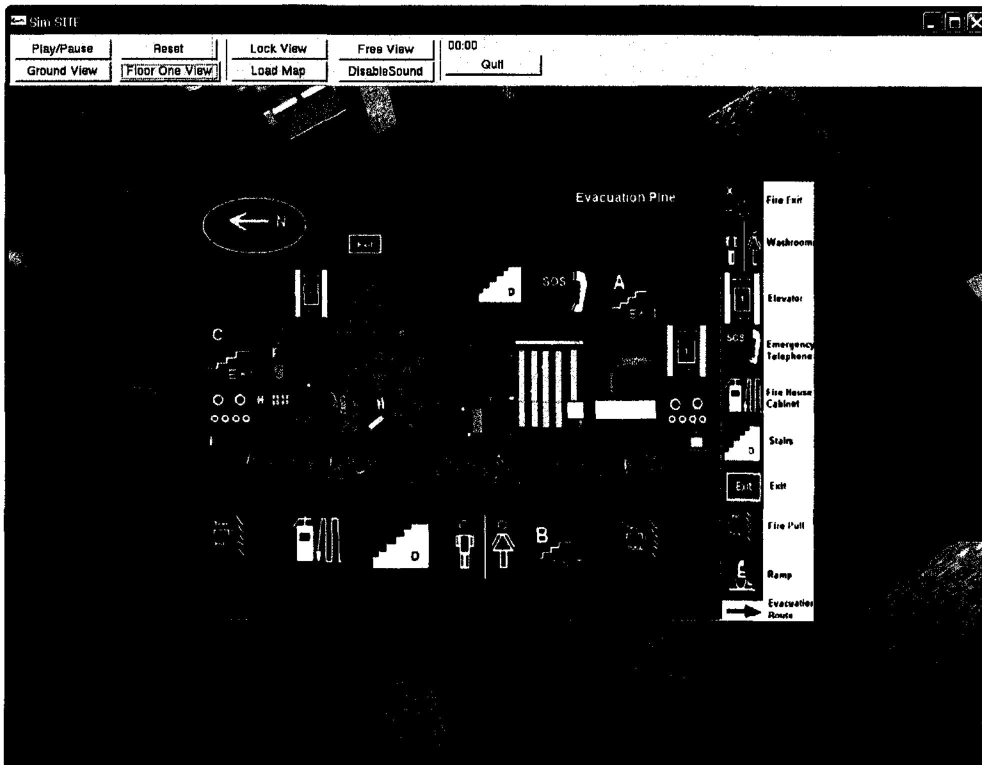



Figure 3-19: Screen Texture Mapping (Two Floor Maps)

So far, the loading of an object in 3ds format has been completed. To summarize, there are four steps to the loading process: opening a 3ds file, processing main trunk, reading object information, and mapping material/texture. All the graphic information that the 3ds files contain will be retrieved and loaded into the graphic controller. The next phase for the engine design is to render scenes graphically on the displays.

### 3.2.5 Object/Texture Rendering

As the graphics pipeline (see Figure 3-10) shows, display and rendering is the last major step. It is the process of generating an image from a three dimensional model, which contains geometry, viewpoint, texture and lighting information, and gives the final appearance of the model on the screen [41].

The class called *Object3DS* is implemented in the ***SimSITE*** to deal with the 3ds object rendering. Before rendering the models in OpenGL, the rendering mechanism is initialized by the parsing of the information (Object info, and Material info). The overall process starts with the following function: *Model_Load.Import3DS(&Model_Info, filename)*. Here *Model_Load* is an instance of the *CLoad3DS* class. By calling the function *import3ds()*, the address of *filename.3ds* is passed to the pre-defined *t3DModel* structure described in section 3.2.1. Then, the techniques discussed in the previous section are performed to read each trunk's information into *Model_info*. The textures if applicable will also need to be prepared for those objects. Finally, all the information is sent to OpenGL for the graphic display.

At the rendering stage, all the objects contained in the 3ds module will be translated into a coordinate space at the specified position, and the rotation transformations are performed according to the adjustment. Texture rendering precedes every polygon pixel by pixel. For each vertex, it needs to determine the corresponding texture coordinate (u, v), access the texture, and then set the vertex to the proper texture attribute. If no texture is mapped to this polygon, the color of the material will be rendered instead. The flowchart of the rending process is shown in Figure 3-20.

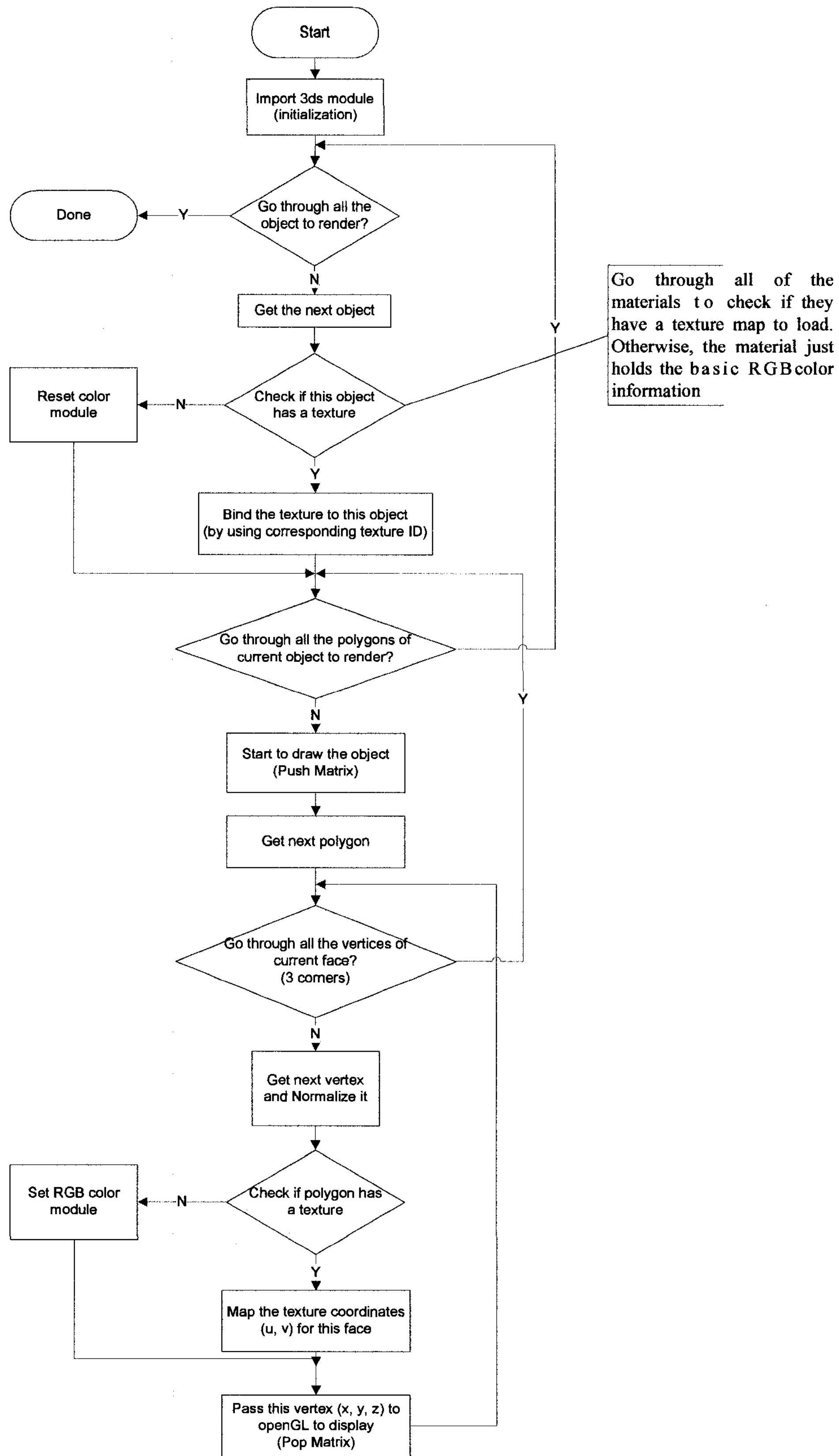



Figure 3-20: 3ds Object Rendering Flowchart

The last step is to display the models graphically. OpenGL supports two rendering approaches: immediate rendering mode and retained rendering mode [42] where per-vertex and per-pixel data are stored in a Display List (DL) in order to achieve high performance [43]. A DL stores a bunch of OpenGL commands which can be repeatedly called. These commands can be optimized by the graphic driver, namely, some geometric transformations can be pre-computed. All the DLs are stored in the memory of the graphics card, which significantly speeds up the process of rendering and displaying. In the proposed ***SimSITE***, there are a few 3D avatars and thousands of objects sitting in twenty 3ds models to render. To enhance the graphic display performance, the avatars are rendered in immediate mode (see section 3.1.2), and all static 3D objects (loaded from 3ds files) are put in retained rendering mode.

The following code illustrates the major steps during the display of the objects using DLs.

```
------------------------------------------------------------------------------------------
//************************* Creating Display Lists **********************************
glNewList(1, GL_COMPILE);// (list ID, mode)
    glPushMatrix();
        glColor3f(1.0,1.0,1.0); // object color
        //object material attributes
        glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
        glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
        //rendering the model, steps shown in Figure 3-20
        DrawModel1->Render();
    glPopMatrix();
glEndList();
        :
//we get 20 3ds models, then we need 20 DLs
------------------------------------------------------------------------------------------
```

In *Display()* callback function, those just created lists will be called instead of executing the commands directly (immediate mode).

```
//**************************** Calling the Display Lists ****************************
void Display(void)
{
    glCallList(1); //Displaying 3ds models in display list
    glCallList(2);
        :
    glCallList(20);
}
```

Finally in the *main()* function, the self-defined callback function (*Display()*) needs to be registered in order to actually display objects on the screen.

```
glutDisplayFunc(Display); //callback function registration
```

So far, the completed 3D loader has been developed. The simulation engine achieves the ability to obtain any 3ds object.

### 3.2.6 Object Normalization:

As described in Figure 3-20 of section 3.2.5 (also Figure 3-10, OpenGL Graphic Pipeline), one important rendering step is the normalization. Why is it so important to normalize the objects? The reason is for the purposes of display and lighting, system performance enhancement, and also collision detection. OpenGL provides two types of normalization techniques. The *polygon normalization* proves the face reflection ability of lighting effects. It is dependent on the inclination degree of polygons with

the lighting sources, which is the angle ($\theta$) between the polygon normal vector and light vector. See Figure 3-21 below:

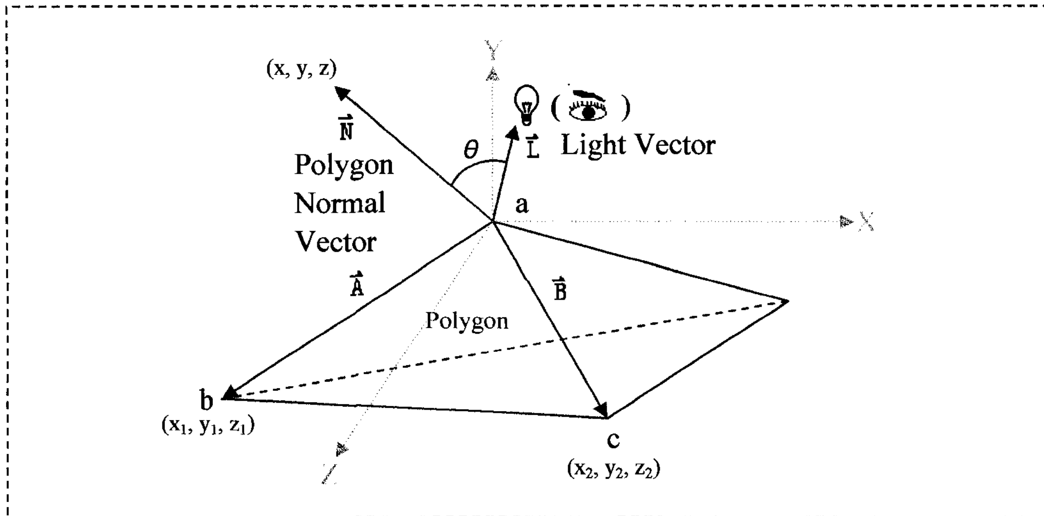



Figure 3-21: Polygon Normalization

The Polygon Vector $\vec{N}$ is the vector orthogonal to the polygon plane, which can be calculated by the right hand rule used in cross product of two vectors ( $\vec{A}$ and $\vec{B}$ ) that are co-planar to the polygon. Then: $\vec{N} = \vec{A} \times \vec{B}$ →

$$x' = (y_1 * z_2) - (z_1 * y_2)$$

$$y' = (z_1 * x_2) - (x_1 * z_2)$$

$$z' = (x_1 * y_2) - (y_1 * x_2)$$

Where x', y', z' are the coordinates of polygon vector

The normal vector can be acquired by normalizing the polygon vector $\vec{N}$.

Then:

$$x = x' / (\sqrt{(x'^2 + y'^2 + z'^2)})$$

$$y = y' / (\sqrt{(x'^2 + y'^2 + z'^2)})$$

$$z = z' / (\sqrt{(x'^2 + y'^2 + z'^2)})$$

Note: √ is square root
Where x, y, z are the coordinates of polygon normal vector

Therefore, each face (polygon) normal of the object can be calculated this way. The Light Vector $\vec{L}$'s origin is attached to the origin of the normal vector, and the end points to the light source. After normalizing this vector, finally, the illumination degree can be simply achieved by: $\boldsymbol{\theta = \arccos (N \bullet L) / (|N|\ |L|)}$

Since the angle $\theta$ denotes the inclination degree of polygons, it also indicates whether this polygon is facing the viewer or not. Figure 3-21 indicates that if the viewer's position (eye/camera's position) is put at the same position as the light, the $\theta$ can be calculated the same way. If the angle is between $90^{\circ}$ and $270^{\circ}$, the polygon is facing the viewer. Otherwise, it is a hidden surface that needs to be removed in rendering stage to enhance the system performance, which is called *Back-face Culling* [44].

Another normalization supported by OpenGL is called *vertex normalization.* It is the technique that creates the smooth lighting look - Gouraud shading [45]. It shows a more realistic appearance (covering up blocky looking objects) when rendering happens. The vertex normalization can be obtained by first calculating the face normals for all polygons, and then taking the average of all normals around each vertex. Therefore, this technique is also called "Average of the normals of the polygons adjacent to the vertex"[46]. The diagram shown below illustrates the method:

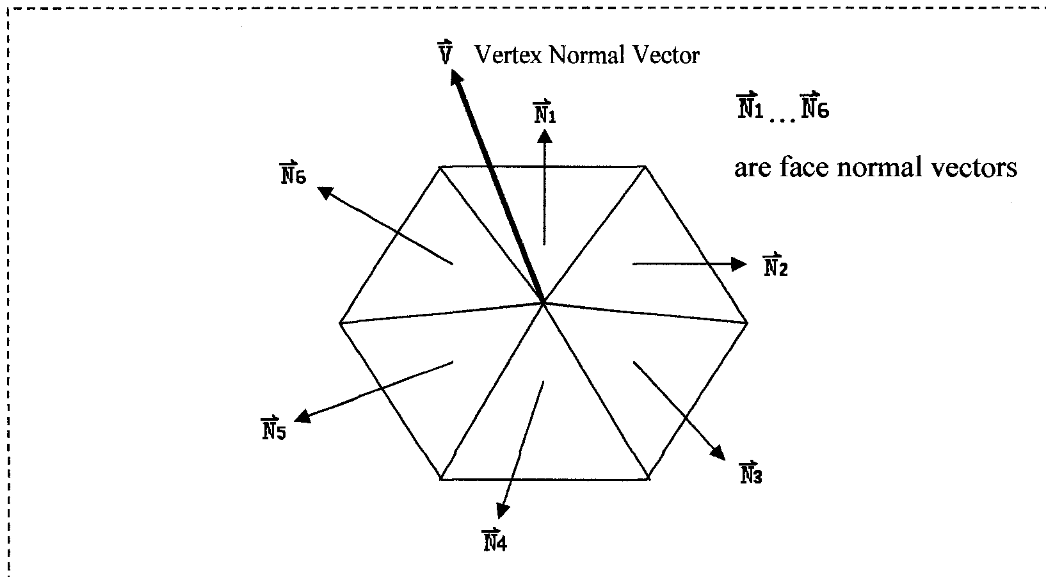



Figure 3-22: Vertex Normalization

$$\overline{V} = \left\| \frac{\sum_{i=1}^{k} \overline{N}_i}{k} \right\|$$ , (where k=6) →

$$\mathbf{x' = (x_1+x_2+x_3+x_4+x_5+x_6) / 6}$$

$$\mathbf{y' = (y_1+y_2+y_3+y_4+y_5+y_6) / 6}$$

$$\mathbf{z' = (z_1+z_2+z_3+z_4+z_5+z_6) / 6}$$

The normal vector can be acquired by normalizing the Vertex vector $\vec{V}$ .

From the above discussions, it is clear as to how to calculate the polygon and vertex normal. By applying both normalization to all faces and vertices, the objects loaded into the developing environment enter an illuminated world.

Now let's look at how to apply vector normalization for the purpose of collision detection. To decide if collision detection is necessary, *Bounding Sphere* method is a basic approach [47]. In this approach, every object is surrounded by a sphere. Since

each point on the sphere surface has the same distance to the center, it is easy to determine whether an object has collided with others. If the distance (*d*) between two sphere's center is less than or equal to the sum of two sphere's radius (*r*+*R*), then a collision has occurred. Similarly for the sphere-plane based collision detection, the distance between the sphere's center point and the plane is calculated by the dot product of the plane's normal and the sphere's position:

**distance = plane.normal • sphere.position [48]**

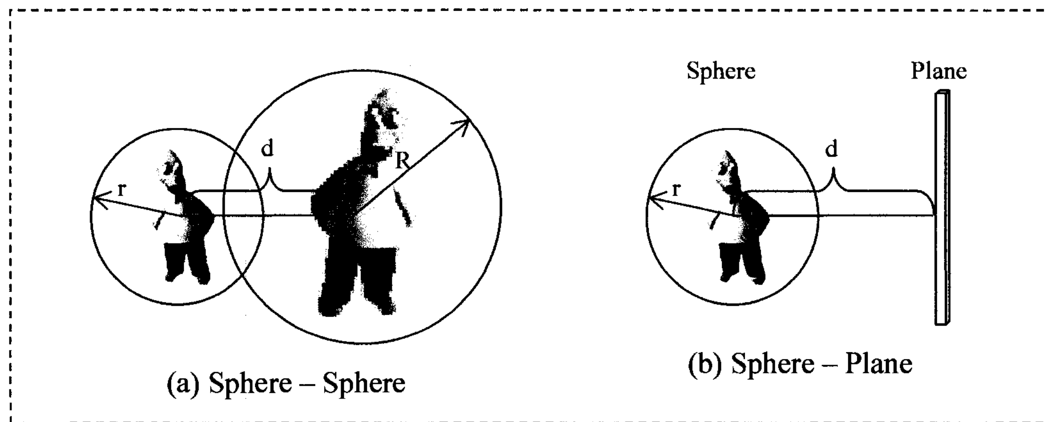



Figure 3-23: Sphere – Plane Collision

The numeric sign of the distance has positive or negative value (+ or -) depending on which side of the plane the sphere is located. If the value has changed from one sign to the opposite sign, or the numeric value of the distance changes to zero, it indicates that the collision has occurred.

### 3.2.7 Avatar Controlling / Camera Following Algorithm

The previous sections introduced the methods to load static objects. The next critical task for engine development is to manipulate the avatars in the virtual environment, such as to make avatars move forward, backward, or change directions. GLUT allows us to build applications that detect keyboard inputs using either the *normal* keys, or the *special* keys like Up/Down. When these keys are pressed, the corresponding events are triggered.

The two events (*orientMe* and *moveMeFlat*) are triggered by keyboard inputs. The keys are called direction keys. Cameras are set up to simulate the functions of eyes in order to view an object in the system. Direction keys are used to move the camera (eyes) in the virtual space. For example, in the *XZ* plane, the Left and Right keys rotate the camera around the *Y* axis, whereas the Up and Down keys move the camera forwards and backwards along the current direction. The PgUp and PgDn change the position of the camera when bird view of system is needed. In ***SimSITE***, there is only one avatar that can be controlled by the user/trainee through the keyboard. If the system is to work in the first person view, the camera should be located on the avatar's head to make the movement of the avatar in sync with the camera. Thus, the user's movement of the camera can manipulate the avatar.

The rotation of the camera, implemented in function *orientMe(),* is illustrated in Figure3-24, where, $\alpha$ is the camera shooting angle between the camera position and its focused object relative to $X$ axis; $\theta$ is the rotation step angle, which depends on the

camera rotation (each step is 0.05 degree in our implementation); *d* is the distance from the camera (eyes) to the controlled avatar on *XZ* projection plane (the 3D problem is simplified into 2D one); *d* is a fixed value when dealing with the rotation (*d* equals to zero when first person view is applied); *x, z* are the coordinates of the avatar's previous position; and *x', z'* are the coordinates of the rotated avatar's location. We will find these two coordinates and update them in the camera setting (*gluLookAt(eyex, eyey, eyez ,**lookatx** , lookaty, **lookatz**, 0.0, 1.0, 0.0)*) and avatar rendering functions to display in the next time period. As can be seen, the controlled avatar's position is determined by the camera (eyes)'s position and rotation. This way, the camera is "following" the avatar's movements simultaneously.

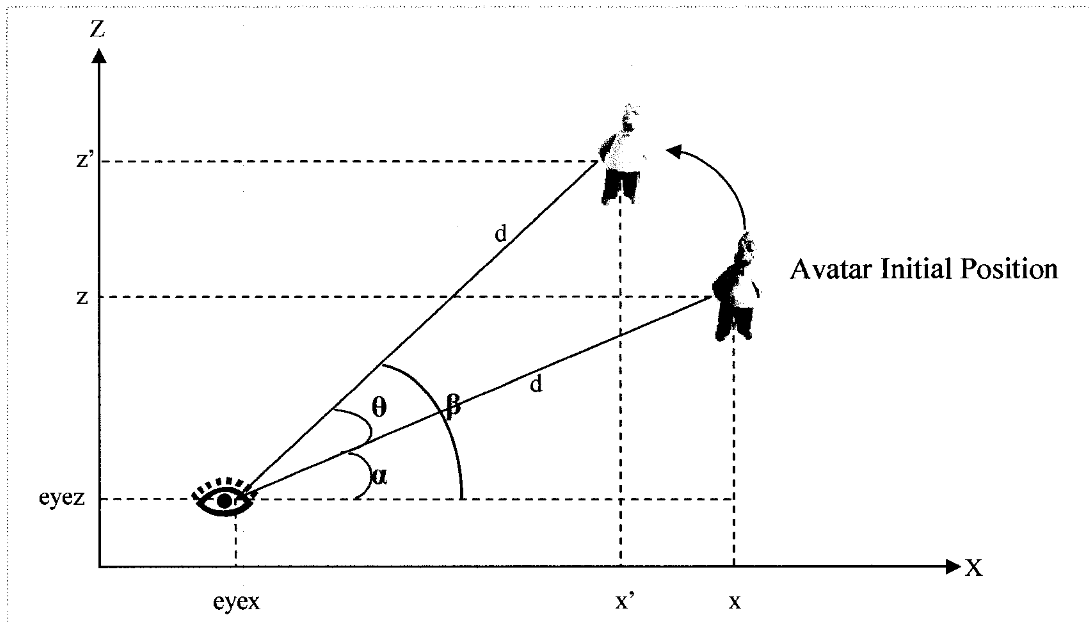



Figure 3-24: Camera/Avatar Rotation

$\alpha$ = -atan((z-eyez)/(x-eyex))

$\beta = \alpha + \theta$

x' = eyex+ d * cos ($\beta$)

z' = eyez + d* sin ($\beta$)

The "flat" motion of the camera, implemented in the method *moveMeFlat(),* is illustrated in Figure 3-25. By pressing the Up or Down key, the camera moves forward and backward along the current line of sight. The avatar's position changes with camera (eye) position. After the camera is flatly moved, the new position becomes (*eyex', eyez'*), and the new look-at position is (*x', z'*). These values need to be updated in camera setting (*gluLookAt(**eyex**, eyey, **eyez** , **lookatx** , lookaty, **lookatz**, 0.0, 1.0, 0.0))* and avatar rendering functions.

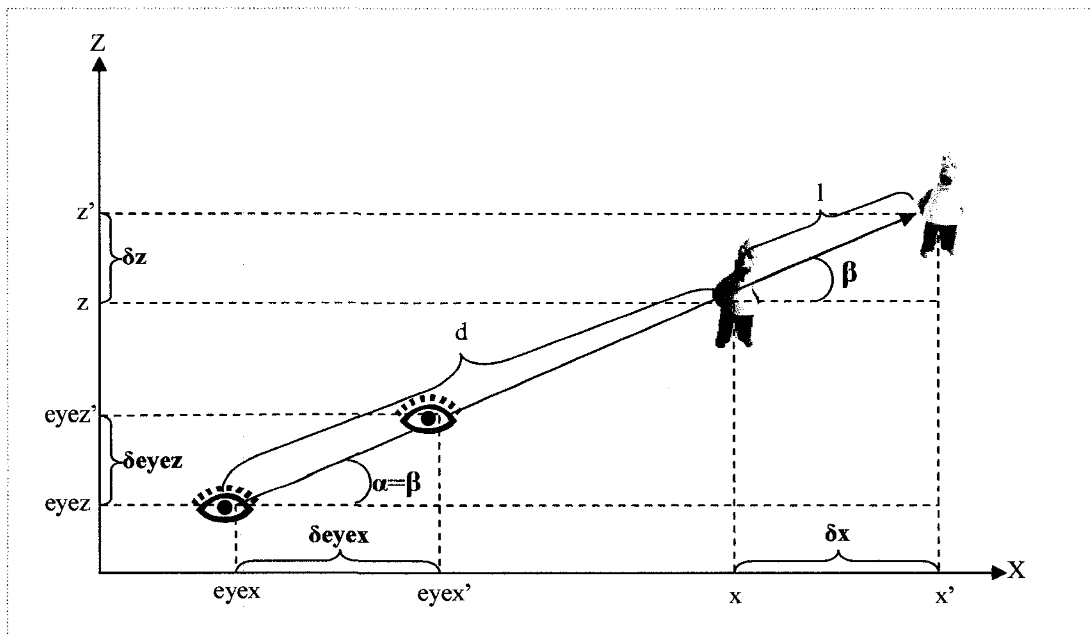



Figure 3-25: Camera/Avatar Flat Moving

**$\alpha = \beta$ = -atan((z-eyez)/(x-eyex))**

**$\theta = 0$: step angle**

**d: the distance from eye to object on Projection XZ plane. Fixed value**

**l: flat movement step (slow: 1; fast: 9)**

**$\delta$eyex = l*sin($\beta$) = $\delta$x**

**$\delta$eyez = l*cos($\beta$) = $\delta$z**

**eyex' = eyex + $\delta$eyex**

**eyez' = eyez + $\delta$eyez**

**x' = eyex'+ d * cos ($\beta$)**

**z' = eyez' + d* sin ($\beta$)**

The last step is registering the self-defined keyboard callback functions into GLUT. In *main()* function, write:

- glutKeyboardFunc(keyboard);
- glutSpecialFunc(specialkeys);

And then, the user can manipulate the camera and move the avatar anywhere within the virtual place through the keyboard inputs.

### 3.2.8 Collision Detection

Collision detection is a key issue in physically-based modeling, geometric modeling, and computer animation. Our 3D virtual environment creates a computer-generated environment filled with virtual objects. Such an environment should give the users a sense of presence, which should enable them to feel both the surrounding objects and the moving avatars. For example, the avatars should not pass through each other and other static objects.

In ***SimSITE***, interactions between moving objects are modeled by dynamic constraints and contact analysis. The avatars' motions are constrained by various interactions, including collisions. To make the system more realistic, more than thousands of objects in the virtual space need to be checked for possible collisions as the user moves, which requires an extremely complex algorithm that could consume lots of time and memory. It is not practical for running a big simulation in real time over the network. Therefore, a fast and simple interactive collision detection algorithm is designed for our simulation.

Since the avatars always walk on the ground (no matter on which storey), and the main obstacles (walls) are always located at fixed positions, a 2D design can be considered instead of the 3D design. Ignoring the height (*Y* axis), 2D floor maps are defined for avatars. The number of levels in the building determines how many of

these 2D maps are needed. Even if the structure of the virtual environment is changed later on, the algorithms such as path choosing and collision detection will still be applicable. Developers only need to change the 2D maps for corresponding structures.

Here, the SITE ground and first floor are taken as example. Based on floor construction, since the layout of each floor is fixed, two floors information have been saved into two text files respectively. These two files correspond to the 3D virtual space projected on the *XZ* plane. Figure 3-26 shows the *Map_Ground.txt* file that holds the actual ground floor data: symbol *'d'* stands for *door* to the goal or exit; *'f'* indicates *free space* which the avatars can go through; '*w*' shows where the *walls* (obstacles) are located; (Symbol *'l'* marked in *Map_first.txt* refers to the location of the stairs that connect the first floor and the ground.) A two-dimensional matrix called *Map Matrix* is set in the program which contains 18x39 cells. It covers the whole area of the map. Each cell has the size of 10, corresponding to the avatar's moving step; each avatar needs around 10 steps to walk across one cell (each step measured as 0.9). Similar as the Cell-DEVS technique for modeling complex cell spaces [49], we need to read the data from the text files and put them into this matrix for manipulation later on.

```
Map_ground - Notepad
File  Edit  Format  Help
wwwwdwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwfffwwwwwww
wwwwfwwwfffwwwwwww
wwwwfwwwfffwwwwwww
wwwwffffffffwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwffffffffwwwwww
wwwwfwwfffffwwwwww
wwwwfwwffffffffwww
wwwwfwwwffwffffwww
wwwwfwwwwwwfffwwww
wwwwfwwwwwwffffwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwffdwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwfwwwwwwfwwwwww
wwwwffffffffwwwwww
wwwwffwwwwdwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwfwwwwwwwwwwwww
wwwwwwwwwwwwwwwwww
```

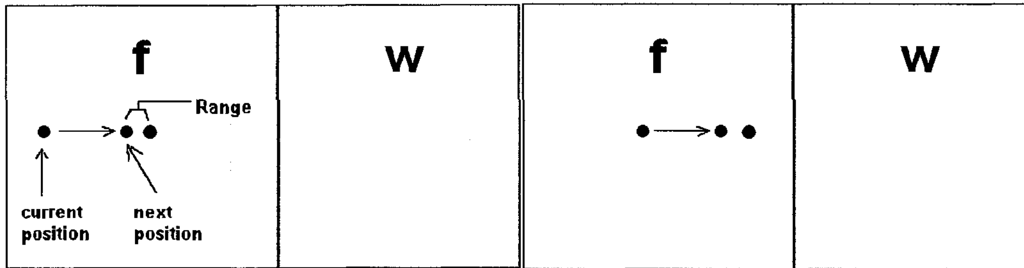
Figure 3-26: Ground Floor 2D Map

```
----------------------------------------------------------------------------------------------------
/**************************** Creating Map Matrix **********************************
//fstream provides an interface to read and write data from files as input/output
streams
fstream MapData;
MapData.open(mapfile,ios::in); //open .txt file
    if (MapData.fail()){ //if fail to open
        cout<<"opening file is failure"; //display the message
    }
    //create Map Matrix according to the map text files
    while(!MapData.eof()&&j<Map_RowSize){
        getline(MapData,sLine,'\n');
        while(i<Map_ColSize&&sLine[i]!='\0'){
            MapMatrix [j][i]=sLine[i]; //saving the data
            if((sLine[i]=='d') ||(sLine[i]=='l')) //exits and stairs
                setExit(j,i); //set this cell as the goal
            i++;
        }
        j++;
        i=0;
    }
----------------------------------------------------------------------------------------------------
```

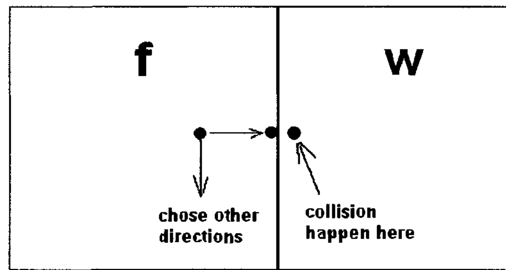
The collision will be checked before the user navigates the avatar. The next possible position (*nextx, nextz*) of the avatar will be sent with a small distance range to check if the collision will happen. The small distance range (0.55 used in program) should be smaller than one avatar step (0.9). If the next cell has been checked as a wall ('*w*'), the collision is detected. The avatars can not be moved, and the user needs to choose other paths to go.

From Figure 3-27, we can see that after taking the next step, the avatar will not move outside the cell; by adding the small distance range, no collision will happen; therefore, it will proceed to the next position; step2 is the same as step1, the next

position will be approached; in step3, even another step won't cause any collision (will be on the edge), but, by adding the small distance range, the collision will happen after step3; therefore, there is no point to continue moving in this direction. The avatar should be guided to a new direction from the current position.




(a) Avatar Motion: Step1 (b) Avatar Motion: Step2




(c) Avatar Motion: Step3

Figure 3-27: Collision Detection

```
--------------------------------------------------------------------------------------------------
/************************************ Collision Detection *****************************
checkCollision(double nextX,double nextZ,double range){
    int ncX,ncZ;

    //check collision for four neighbor cells located in 4 different directions
    getMaxtrixCoodi(ncX,ncZ,nextX+range,nextZ); //towards North
    if(MapMatrix[ncX][ncZ]=='w') //next cell is wall
        return -1; //stop moving in this direction

    getMaxtrixCoodi(ncX,ncZ,nextX-range,nextZ); //towards South
    if(MapMatrix[ncX][ncZ]=='w')
        return -1;
```

```
    getMaxtrixCoodi(ncX,ncZ,nextX,nextZ+range); //towards East
    if(MapMatrix[ncX][ncZ]=='w')
        return -1;

    getMaxtrixCoodi(ncX,ncZ,nextX,nextZ-range); //towards West
    if(MapMatrix[ncX][ncZ]=='w')
        return -1;

    return 1;
}
```

---

In the end, the collision checking function needs to be embedded into the avatar controlling methods such as *orientMe()* and *moveMeFlat()* mentioned in the previous section. Thus, when the keyboard receives the commands to manipulate the avatar, the collision detection can be processed in advance.

### 3.2.9 Auto-avatars Moving Algorithm

The ***SimSITE*** is a multi-avatar evacuation training simulation. Almost all the avatars should have the "intelligence" to find the optimal ways to escape form the building when an emergency happens, except one that can only be controlled by the user (trainee). These programmed avatars are called "auto-avatars", since they can calculate the shortest path from where they are randomly located to one of the safest exits when emergency occurs; also they can choose an optimal path and automatically travel there with collision avoidance.

## Shortest Path Calculation

Our optimal path finding algorithm is essentially isomorphic to the "Bumble Strategy" in Donald's 1987 algorithm [50], which indicated as follows: "a search node N on a c-space grid is dequeued, and its c-space grid neighbors are generated. The reachable, unexplored, free-space neighbors are put on the end of the queue. Each new neighbor M contains a backpointer to N, and N's direction from M [51]". The Bumble Strategy is simply a breadth-first-search [52] from the goal. By referring to the Bumble Strategy, we implement the dynamic programming algorithm for the ***SimSITE***.

The *Map Matrix* represents the 2D map described in the previous section. Now we want to find the shortest path between any free space (*f*) and the goal (*d*). Since there are many obstacles randomly sitting in the map, the axiom of "the shortest distance between two points is a straight line" is not true any more. Here, a "shortest path" is defined to be one passing through a minimal number of vertices (cells).We assume the moving paths can not be diagonals. The shortest path calculation algorithm is implemented, using a recursive method starting from the goal.

- **The problem is:** Find the shortest distance and path from start point $N$ to destination $G$ (goal/exit) across $n$ different hops in emergency evacuation training scenarios.
- **The base case is:** If $N$ is the same as $G$, then the shortest distance is 0; nothing needs to be done.

- **Reduction (a smaller case):** Now suppose the number of hops between *N* and *G* is at least one. Ignore the very last step - the path between $(n\text{-}1)^{th}$ to $(n)^{th}$, there are $n - 1$ hops between *N* and *G*.
- **Recursion:** The smaller case can be solved by a recursive performance to reach the base case eventually.
- **Building up the solution:** If the shortest path from the destination to the stage *n-1*, in addition to the known path between $(n\text{-}1)^{th}$ to $(n)^{th}$ (It is easy to find the shortest path between two neighboring stages/hops) is found, the whole path between vertex *N* to the destination *G* can be obtained. Similarly, how to find the path between the destination and vertex *n-1*? As long as we can find the path between the destination to vertex *n-2*, and add the path between $(n\text{-}2)^{th}$ to $(n\text{-}1)^{th}$; after that, we need to find the path between the destination to stage *n-3*. Calculating like this, we finally reach the stage just before the base - the destination to the first vertex, which is the neighborhood of the destination, unit *(x+1, z)*, unit *(x-1, z)*, unit *(x, z+1)*, and unit *(x, z-1)*; the distance between the destination to these four units is the shortest path from the goal *G* to the stage1. Eventually, by traversing the path back, the shortest path between *N* and *G* is found.

Same as *Map Matrix*, the *Path Matrix* used in this program contains 18x39 units (called *cell* in the program). The *Path Matrix* keeps the calculation result of the shortest path between each vertex to the exit. For example, as shown in Figure 3-28,

to find the shortest path between Point $N1$ and Exit $G1$, starting from the exit, we first calculate the shortest path from stage1 to the exit. The exit positioned at $(x, z)$, the four neighboring units $(x+1, z)$, $(x-1, z)$, $(x, z+1)$, and $(x, z-1)$ in direction North, South, West and East form the first stage. Unit $(x, z+1)$ is the wall and the other three are the free space. They have the shortest distance to the exit, which is 1. Each passed node in the map as a base will keep calculating four directions; the traversing nodes will only stop under three conditions: reach the goals, hit the wall, or meet the cell with the smaller distance value that was calculated by itself or other nodes from other branches. The next step is to find the shortest path between stage 2 and stage1. 8 units $(x+2, z)$, $(x+1, z-1)$, $(x, z-2)$, $(x-1, z-1)$, $(x-2, z)$, $(x-1, z+1)$, $(x, z+2)$, $(x+1, z+1)$ form stage2. Except units $(x, z+2)$, $(x, z-2)$, and $(x-1, z+1)$ which are walls, all other unites have the shortest distance to the goal as 2. Same idea, the third stage contains 13 units, except wall units, $(x+3, z)$, $(x+2, z-1)$, $(x+1, z-2)$, $(x-1, z-2)$, $(x-2, z-1)$, $(x-3, z)$, and $(x+2, z+1)$ have the shortest distance to the goal as 3. We continue the calculation like this and ignore any wall units and the units that are out of the range; finally the shortest distance will be obtained between each cell to the goal. By putting these distances values into the *Path Matrix*, the shortest path can be traversed by following the smallest descendent distance number marked on the map. For instance, from $N1$ to $G1$, two alternative paths can be chosen by following the descendent number 8, 7, 6, 5, 4, 3, 2, 1, 0. (Note: in the program, to make sure to go through all the vertices, eight neighbor cells (instead of four) surrounding the exit are taken to proceed with the

calculation. This may cause some overlapping calculation, but the smallest distance value will be accepted.)

---

```
/***************** The method to calculate the shortest path ******************/
//x, z: cells coordinate; value: minimum distance from previous cell to the goal
calShortPath(int x,int z,int value){

    //three bases to stop the recursion
    if(x==GoalX&&z==GoalZ)   return 1; // reach the goal
    if(pathMatrix(x,z)==-1)      return 1; //hit the wall
    //the cell had been traversed by other branch, has smaller minpath value
    if(pathMatrix(x,z)<=value+1&&pathMatrix(x,z)!=0&&value!=0) return 0;

    minPath =-1; //initial

    if(x+1<GetMapRowSize()){ // as long as not reach the right range limit
        if(x+1!=GoalX||z!=GoalZ){ // not reach the goal
            if(pathMatrix(x+1,z)>0) //not wall (-1), not goal (0)
                minPath = pathMatrix(x+1,z);
            } //update minPath
        else     minPath =0;
    }

    if(x-1>=0){ // not reach the left range limit of the Matrix
        if(x-1!=GoalX||z!=GoalZ){
            if(pathMatrix(x-1,z)>0){
                //previous value greater than current value, then replace
                if(minPath>0&&minPath>pathMatrix(x-1,z))
                    minPath = pathMatrix(x-1,z);
                else if(minPath<0) //no path value yet
                    minPath = pathMatrix(x-1,z); //assign the value
            }
        }
        else     minPath =0;
    }

    if(z+1<GetMapColSize()){ // as long as not reach the upper/top range limit
        if(x!=GoalX||z+1!=GoalZ){
            if(pathMatrix(x,z+1)>0){
                if(minPath>0&&minPath>pathMatrix(x,z+1))
                    minPath = pathMatrix(x,z+1);
```

```
				else if(minPath<0) minPath = pathMatrix(x,z+1);
			}
		}
		else	minPath = 0;
	}

	if(z-1>=0){ ...} // as long as not reach the bottom range limit

	pathMatrix(x,z)= minPath+1; //update the pathMatrix, by adding 1

	//********************** proceed recursion to 8 directions *******************

	if(z-1>=0){ //not reach the bottom cell
		calShortPath(x,z-1,minPath+1); //recursively calculate next cell below
	}

	if(x-1>=0){ //not reach the left first cell
		calShortPath(x-1,z,minPath+1); //recursively calculate next left cell
	}

	if(z+1<GetMapColSize()){ //not reach the top cell
		calShortPath(x,z+1,minPath+1); //recursively calculate next upper cell
	}

	if(x+1<GetMapRowSize()){//not reach the most right cell
		calShortPath(x+1,z,minPath+1); //recursively calculate next right cell
	}

	if((z-1>=0)&&(x-1>=0)){ //calculate the left lower cell
		calShortPath(x-1,z-1,minPath+1);
	}

	if((x+1<GetMapRowSize())&&(z-1>=0)){ //calculate the right lower cell
		calShortPath(x+1,z-1,minPath+1);
	}

	if(z+1<GetMapColSize()&&x+1<GetMapRowSize()){
			calShortPath(x+1,z+1,minPath+1); //calculate the right upper cell
	}

	if((z+1<GetMapColSize())&&(x-1>=0)){
		calShortPath(x-1,z+1,minPath+1); //calculate the left upper cell
	}
}
```

18 rows

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|---|
| 1 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | |
| 2 | W -1 | W -1 | W -1 | W -1 | W -1 | F (x+2, z+4) 6 | F (x+3, z+4) 7 | |
| 3 | W -1 | W -1 | W -1 | W -1 | F (x+1, z+3) 6 | F (x+2, z+3) 5 | F (x+3, z+3) 6 | |
| 4 | W -1 | W -1 | W -1 | W -1 | W -1 | F (x+2, z+2) 4 | F (x+3, z+2) 5 | |
| 5 | W -1 | W -1 | W (x-1, z+1) -1 | W (x, z+1) -1 | F (x+1, z+1) 2 | F (x+2, z+1) 3 | F (x+3, z+1) 4 | |
| 6 | F (x-3, z) 3 | F (x-2, z) 2 | F (x-1, z) 1 | G1 (x,z) 0 | F (x+1, z) 1 | F (x+2, z) 2 | F (x+3, z) 3 | |
| 7 | F (x-3, z-1) 4 | F (x-2, z-1) 3 | F (x-1, z-1) 2 | F (x, z-1) 1 | F (x+1, z-1) 2 | F (x+2, z-1) 3 | F (x+3, z-1) 4 | |
| 8 | F (x-3, z-2) 5 | F (x-2, z-2) 4 | F (x-1, z-2) 3 | W -1 | F (x+1, z-2) 3 | W -1 | W -1 | |
| 9 | F (x-3, z-3) 6 | W -1 | F (x-1, z-3) 4 | F (x, z-3) 5 | F (x+1, z-3) 4 | F (x+2, z-3) 5 | W -1 | |
| 10 | F (x-3, z-4) 7 | F (x-2, z-4) 6 | F (x-1, z-4) 5 | F (x, z-4) 6 | F (x+1, z-4) 5 | F (x+2, z-4) 6 | W -1 | |
| 11 | N1 8 | F (x-2, z-5) 7 | F (x-1, z-5) 6 | F (x, z-5) 7 | F (x+1, z-5) 6 | F (x+2, z-5) 7 | W -1 | |
| 12 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | |
| 13 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | |
| 14 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | W -1 | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| | | | | | | | | |

39 columns

Figure 3-28: Path Matrix

So far, the *Path Matrix* has been calculated. Each cell contains the shortest distance value to its nearest exit.

## Multi-Goal Selection

Comparing with the Dorst path generation algorithm [53] which applied on the map with only one goal exist, and each cell is visited exactly once, there is more than one exit in the ***SimSITE*** system. The same path calculation algorithm will be applied to each exit. The cell value of the *Path Matrix* will be updated when the distance to the current exit is smaller than the previous one. Therefore, by picking up the smallest distance number on the map, the avatars that are in free space can evacuate from any location through the nearest exit.

```
-----------------------------------------------------------------------------------------------
/****************************** As long as not going through all the exits, calculate
the path matrix. Starts from the exit, recursively calculate its 8 neighbor nodes
one by one, and keep going until all the cells have been considered. Update
the matrix if any smaller value has been obtained ********************************/

while(tmpPnt<=exitPointer && tmpPnt<MAX_EXITS){ //go through all exits

        pathMatrix(GoalX,GoalZ) = 0; //value 0 means exit in pathMatrix

        if(GoalX+1<GetMapRowSize())
            isExit = calShortPath(GoalX+1,GoalZ,0);     //right cell
        if(GoalX-1>=0)
            isExit = calShortPath(GoalX-1,GoalZ,0);     //left cell
        if(GoalZ+1<GetMapColSize())
            isExit = calShortPath(GoalX,GoalZ+1,0);     //upper cell
        if(GoalZ-1>=0)
            isExit = calShortPath(GoalX,GoalZ-1,0);     //lower cell
        if(GoalZ+1<GetMapColSize()&&GoalX+1<GetMapRowSize())
            isExit = calShortPath(GoalX+1,GoalZ+1,0); //right upper
        if((GoalZ-1>=0)&&(GoalX-1>=0))
            isExit = calShortPath(GoalX-1,GoalZ-1,0);  //left lower
        if((GoalZ+1<GetMapColSize())&&(GoalX-1>=0))
            isExit = calShortPath(GoalX-1,GoalZ+1,0); //left upper
        if(GoalZ+1<GetMapColSize()&&GoalZ-1>=0)
            isExit = calShortPath(GoalX+1,GoalZ-1,0); //right lower
        tmpPnt++;
    }
```

## Path Choosing

After all the values in *Path Matrix* are updated, each avatar is assigned to a random position. The avatar then can decide which way it should go when it moves to the center of the cell (see Figure 3-29). This is important, since the avatar's step is much smaller than the size of each cell. Once the avatar starts to move, it will only change directions when it reaches the center of the cell. As shown in Figure 3-30, the initial position is marked value 8, which shows that the distance from this cell to the nearest goal is 8 cells. The algorithm shows that the avatar can only go towards the next cell that neither contains a *'w'* nor holds the value greater than its current value (to avoid going backward). Therefore, the only direction it can go is to the east. It will continue going towards east until it reaches cell 6 that holds the wall as the next position to the east; and then, it will turn north. By following the descendent distance number marked in the map, the avatar can finally reach the goal *G1*.

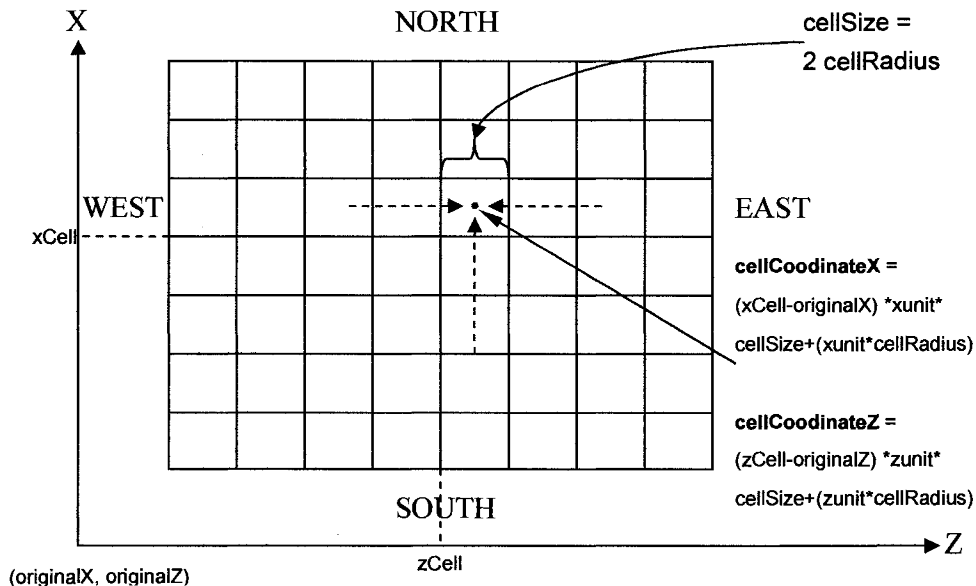



Figure 3-29: Cell Center Calculation

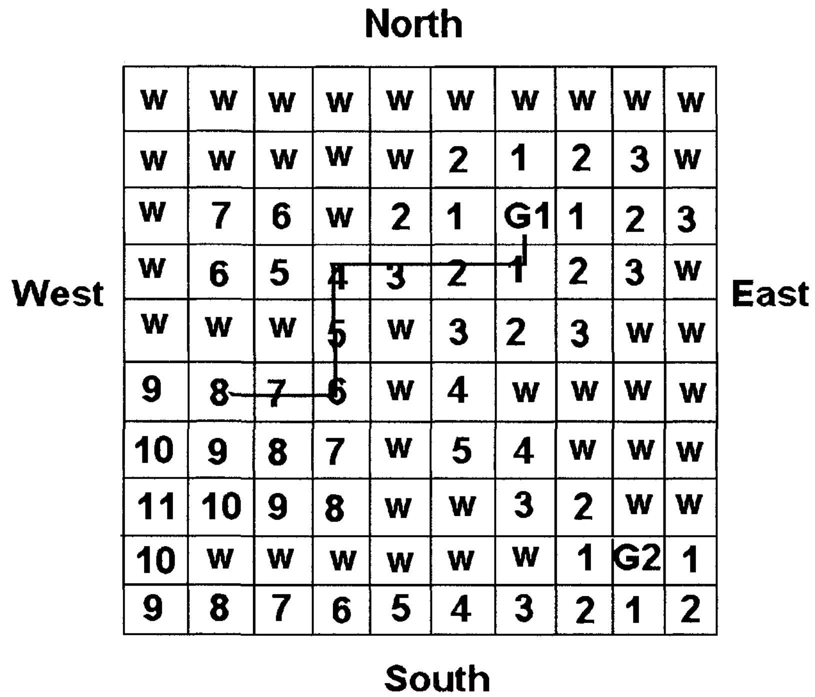



Figure 3-30: Path Choosing

So far, the auto-avatars located on different floors have the ability to find the best way to evacuate. Considering that our virtual world is a high rise building, how to enable the avatars to travel across different stories becomes the next task for designers. This topic will not be covered any further in this thesis.

### 3.2.10 User Interface

**Pop-up Menu Design**

Like any other software system, a user-friendly interface is very important in ***SimSITE.*** This is where this projects benefits from GLUT, which not only provides the utility for primitive object drawing and stroke fonts display, but also has nested pop-up menus. With the help of GLUT tool, a pop-up user menu was developed to handle the interactions between the system and input devices (e.g.: mouse).

In ***SimSITE***, depending on different scenarios, the user can change the system state by manipulating the pop-up menu. For instance, different floor view can be switched (Ground to First Floor, or vise versa) at the user's will and the system camera can be changed from the free view to tracking view when the avatar is manipulated. All these functionalities can be achieved by creating two submenus first and then adding menu entities to these submenus respectively.

```
/*************** Create a submenu called "Change View" to hold two
entities "Ground View" and "Floor One View" *********************************/

viewSubmenu = glutCreateMenu(processMenuEvents);
        glutAddMenuEntry("Ground View",2);
        glutAddMenuEntry("Floor One View",3);

/*************** Create a submenu called "Follow Up" to hold two entities
"Lock Person" and "Free Lock" **************************************************/

lockSubmenu = glutCreateMenu(processMenuEvents);
        glutAddMenuEntry("Lock Person",4);
        glutAddMenuEntry("Free Lock",5); //release to free view
```

Where *processMenuEvents()* is the function to handle the events. For example, when processing menu event – *Lock Person*, the eye/camera will be switched from free view to tracking view. This has been implemented by the Camera Flowing Algorithm stated in section 3.2.7. After all submenus are created, a main menu are added which contains all submenus:

```
/****************** Submenus: "ChangeView" and "Follow Up" as entities
added into main menu – "menu" *************************************************/

menu = glutCreateMenu(processMenuEvents);
    glutAddSubMenu("ChangeView", viewSubmenu);
    glutAddSubMenu("Follow Up", lockSubmenu);
```

Finally, the pop-up menu (in Figure 3-31) is attached to the right button of the mouse to be triggered.

```
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

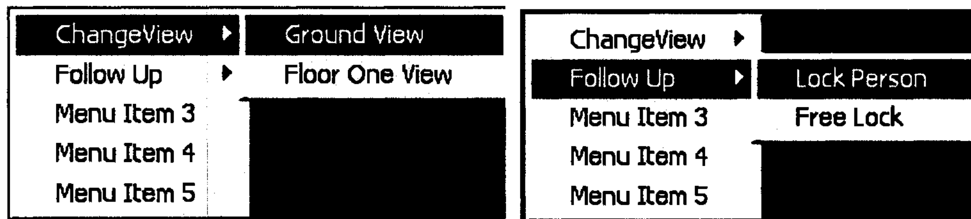



Figure 3-31: GLUT Pop-up Menu

**GLUI**

Although simple OpenGL applications can be built by only using the standard GLUT input methods - the mouse, keyboard, and pop-up menus, these methods tend to be overloaded and unserviceable when system expands and the number of features or options increases [54]. The GLUI, GLUT supported user interface library, addresses

this problem by providing standard user interface elements such as buttons, checkboxes, editable text boxes, spinners, separators and so on. Therefore, by using the platform-independent interface library – GLUI, we are designing a GUI that allows users to interact with the system by manipulating buttons.

To meet the system requirements, more functions have been added into ***SimSITE,*** as the finite state diagram shown below (see Figure 3-32):

- After running the simulation, the system enters the INITIAL state. *Play* function starts or continues the simulation when the *Play/Pause* button is clicked during system INITIAL state or PAUSED state. The system then enters the PLAYING state.
- During the PLAYING state, the auto-avatars will try to evacuate by taking the optimal paths; if the user clicks *Play/Pause* button again, the system enters the PAUSED state.
- The *Pause* function breaks off the simulation time running, auto-avatars moving (they will stay in current positions), and alarm ringing. The difference between the *Reset* function and *Pause* function is that when the *Reset* button is clicked during the PLAYING or PAUSED state, the whole system will be terminated. All avatars will be placed at the initial positions, and the simulation timer instead of being paused will be set back to zero. The whole system goes back to the INITIAL state.

Next, the *Load Map* function is added to load and display the 2D floor maps on the screen. For the functions that are used on a regular basis, functional buttons are designed, e.g. *Lock View*, *Free View*, *Ground View*, and *Floor One View* are implemented to help users change their perspectives from free view to tracking view, from ground floor view to first floor view and vice versa. These buttons work as shortcuts for the pop-menu. In the future, if the system needs to be expanded, those buttons can be replaced from the user interface. Furthermore, the *Disable/Enable Sound* button and the *Quit* button are designed to mute/restart the alarm and exit from the simulation system.

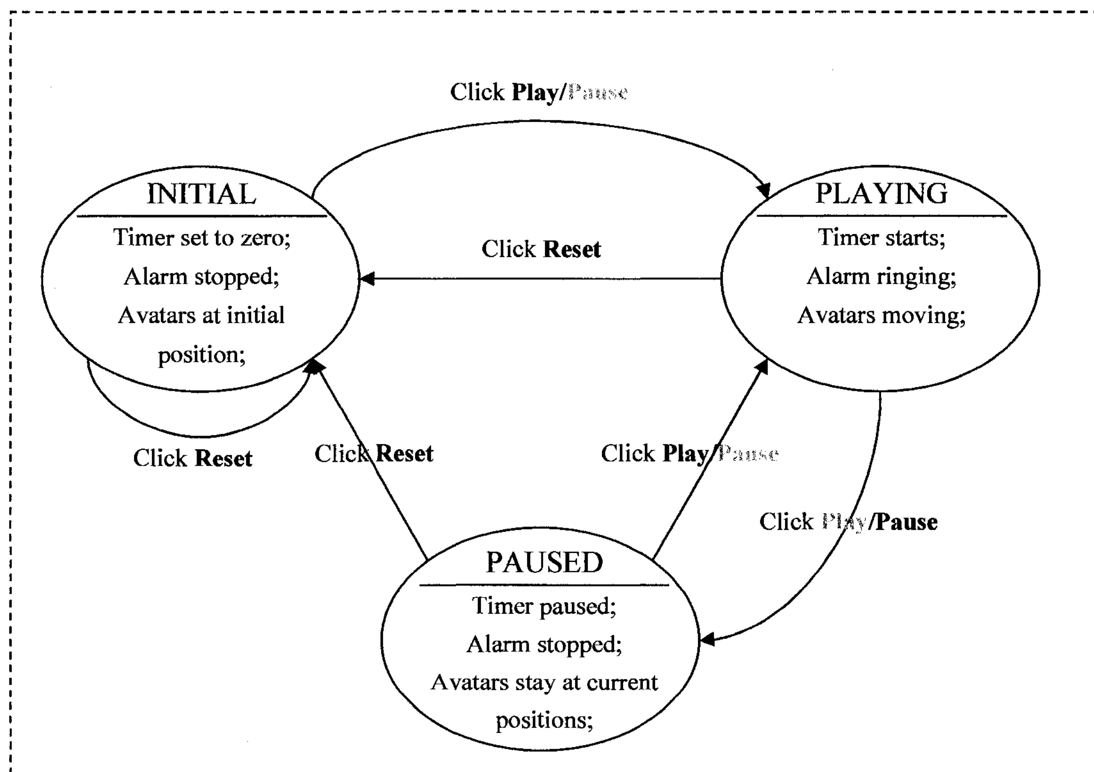



Figure 3-32: System Finite State

To implement the GUI in OpenGL, all core libraries should be included at the beginning: #include <GL/glut.h> #include <GL/glui.h>

Then the regular GLUT windows need to be created and the window ID of the main graphics window needs to be stored. The main window has been created already that supports the redisplay events sent by GLUI sub-windows.

```
main_window=glutCreateWindow ("SimSITE"); //create a main window to display main scene
```

GLUI sub window needs to be created to hold the buttons as entities. Some separators such as the columns have been added to adjust the GUI format. Later on, more functional elements such as editable box will be added as well.

In the end, we register the GLUT idle callback with GLUI to enable GLUI sub window to take advantage of idle events without interfering with the application's operation. In other words, the system will be executing idle tasks with no input from the GUI. The idle function describes the behaviors of the alarm system and the movements of avatars.

```
----------------------------------------------------------------------------------------------------
/*********** Register the idle callback with GLUI, not with GLUT *************/
GLUI_Master.set_glutIdleFunc( idle );
glutMainLoop(); //give the control to GLUT
----------------------------------------------------------------------------------------------------
```

**Text Display**

In ***SimSITE***, a timer is required to record the simulation time. It starts to count when *play/pause* button is clicked during system INITIAL state. To show the timer on the screen, GLUT text rendering techniques are applied. There are two broad categories of methods for drawing texts in OpenGL: image-based and geometry based. To enhance system performance, the first option is adopted. All texts displayed in the GUI are presented in pre-rasterized bitmap font which is basically a 2D font without thickness and will always be facing the viewer, like a billboard [55]. Compared with other fonts (e.g.: outline fonts), bitmap is our first choice because of its fast rendering process [56].

The purpose here is to display the timer as a text label in a GLUI window; therefore, we will perform the static text control function in GLUI. During the initiation, the initial timer characterized by "00:00" is added in the user interface. The static text controller points to this text.

```
-----------------------------------------------------------------------------------------------------

GLUI_StaticText *gluiTime; //initialization a text controller

// set the controller pointing to the text, and add the text in GLUI sub-window

gluiTime=glui->add_statictext("00:00");

-----------------------------------------------------------------------------------------------------
```

After the application starts running, the elapsed time value will be momentarily calculated by function *openGLTimer()* as long as system is in the PLAYING state.

The timer function recursively counts the simulation time based on simulation frame rate (FPS). The new numerical values (number of minutes and seconds) will be sent as characters into a buffer; and the static text controller *gluiTime* will point to that buffer; the GLUT timer function will be called in the main function to register the callback in order to render the text on GUI. It is to be noted that the static text information should always stay in the same position on the screen even when the user moves the camera around. This can be achieved by first drawing the environment as we used to - with a perspective projection, and then switching to the orthographic projection to draw the text.

The final ***SimSITE*** GUI is shown in Figure 3-33.

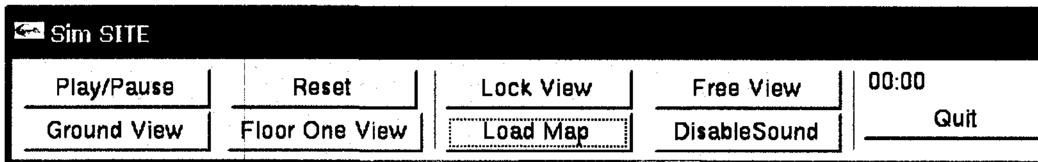



Figure 3-33: ***SimSITE*** User Interface

## 3.3 Simulation Audio

To further enhance the performance of the simulation engine, the audio capabilities are added into the system. In ***SimSITE***, the sound of the alarm is simulated in the event of emergency. The alarm will be triggered when user first clicks the *Play/Pause* button on GUI, and it will be disabled anytime when the *Disable Sound* button is pressed (then the button will be switched to *Enable Sound* to restart the alarm). A different audio API can be used for each operating system. To better cooperate with OpenGL, ***SimSITE*** adopts OpenAL technology for the implementation [57].

### 3.3.1 OpenAL Implementation

To set up OpenAL program for windows OS, several standard libraries are included.

```
#include <windows.h>        #include <stdio.h>
#include "al.h"             #include "alc.h"
#include "alu.h"            #include "alut.h"
```

There are three essential elements in OpenAL design: *buffer*, source, and listener. A buffer stores the sound data, which describes all the information about how to play a sound. A *source* is a point in 3D space to emit the sound. And, a *listener* represents the user who receives the sound. Note: A source itself is not a sample of audio clip. The source only plays back sound from the data stored in the buffer.

Therefore, to play a sound in the application, a buffer is to be generated in the *main()* function first. Then the sample wav file is loaded into the buffer, and a source is created to attach the buffer.

```
-----------------------------------------------------------------------------------------------------
/*************************** Initial the sound elements ***************************/
main(){
:
    StreamInit(); //initial the sound stream and hardware
    alGenBuffers(1, &g_Buffer); //create a buffer
    LoadWave("alarm3.wav", g_Buffer); //load the source file into the buffer
    alGenSources(1,&source); //generate a source
:
}
-----------------------------------------------------------------------------------------------------
/******************************** Load the .wav file ********************************/
ALboolean LoadWave(char *szWaveFile, ALuint BufferID)
{
    :
    if (!szWaveFile) return AL_FALSE;    //if not a wav file

    //load the file
    alutLoadWAVFile (szWaveFile,&format,&data,&size,&freq,&loop);
    : //check the error
    alBufferData(BufferID,format,data,size,freq); // Copy data into ALBuffer
    alutUnloadWAV (format,data,size,freq); // Unload wave file
    :
}
-----------------------------------------------------------------------------------------------------
```

The alarm will be played in the *idle()* function. We assign the source to the buffer that contains the loaded wav file: *alSourcei(source,AL_BUFFER, g_Buffer);*

Then, the sound volume is adjusted: *alSourcef(source,AL_GAIN,0.2f);*

Finally, the sound is played: *alSourcePlay(source);*

Also, to stop the sound, the following function can be called: *alSourceStop(source);*

(Note: The entire AL library can be downloaded from the official website.)

## 3.4 Simulation Scenarios (Use Cases)

The previous sections described how the graphic design is implemented. In order to design a multi-user distributed simulator to conduct the safety training, a virtual environment is developed where multiple avatars are created. With collaborating network setup, the simulation represents different scenarios of emergency evacuation in a high-rise building.

**Senario1: Automatic Evacuation**

1. Emergency starts. The alarm is triggered and the timer starts to count the simulation time.

2. People (auto-avatars) located on the ground floor inside of the building automatically choose the directions of their movement based on the optimal path calculation algorithms.

3. People (auto-avatars) on the first floor look for the entrance to the ground floor, since, except the ground, all the doors, exits, and elevators on other floors of the building will stop operating once the alarm is triggered; people move down to the ground floor and will follow step2.

4. Those people who successfully find the path and avoid all obstacles reach the exit finally to escape.

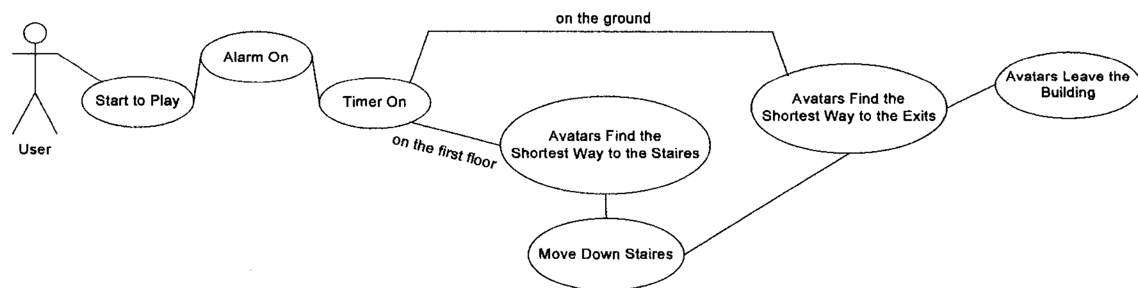



Figure 3-34: Senario1 Automatic Evacuation

**Senario2: Emergency Evacuation Training**

1. Emergency starts. The alarm is triggered and the timer starts to count the simulation time.

2. A trainee will be asked to select the floor he/she is currently located on. This will lead the trainee into either exploring state or following state.

3. If the trainee chooses the first floor, he/she enters the First Floor Exploring State to bird view this shared virtual space.

4. Otherwise the trainee enters the Ground Floor Follow State. The trainee can manipulate one avatar. The camera view perspective can be set to the first person view, the third person view, or global view to see more of the environment. His/her purpose is to manipulate the avatar to leave the building. He/she can simply walk out of the building by following other auto-avatars.

5. If he/she gets lost in the building, which means there are no auto-avatars that can be followed, the trainee has two options:

   - Look at the building plan, by pressing the action key "*L*" to load the building map on the screen (or click the Load Map button on GUI). The requested floor map will be displayed on the screen depending on which floor the trainee is located on. The map will tell the trainee his/her evacuation routes.

   - Wait to be guided by the leader who is managed by another player in the network.

6. The trainee will escape from the building and finish his/her training task, either by following avatars or by map checking or leader guiding.

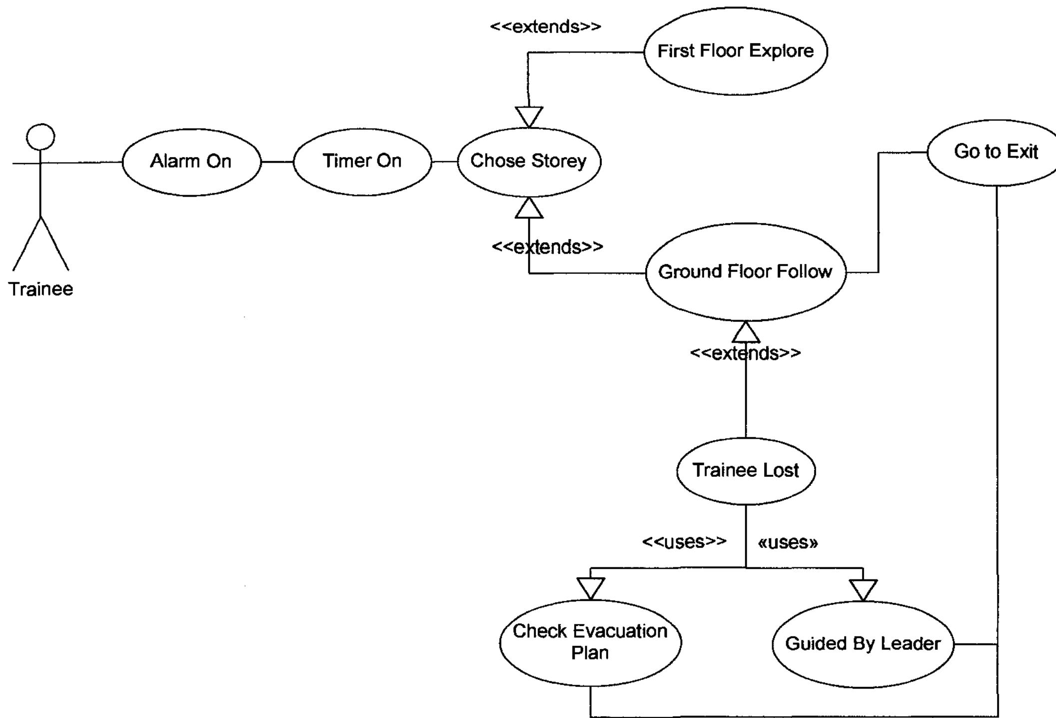



Figure 3-35: Senario2 Evacuation Training

# Chapter 4
# Interaction and Collaboration

## 4.1 Develop with RTI

### 4.1.1 RTI 1.3NG

The Run-Time Infrastructure (RTI) implements the interface specification and represents one of the most fundamental components of the HLA. It provides common services to simulation systems during a federation execution. Federates interact with the RTI in accordance with the HLA interface specification. All exchange of data among federates should occur via the RTI. RTI 1.3-NG [25] implements Version 1.3 of the HLA Interface Specification, which is an architectural foundation encouraging reusability and interoperability. It allows the simulation and the communication to be developed separately. An ordinary simulation system is developed in the previous chapter. Now the RTI concepts will be incorporated into this system to facilitate the interaction and collaboration.

There are three major components embodied in RTI1.3 software. They are the RTI Executive process (RtiExec), the Federation Executive process (FedExec), and the libRTI library. The RtiExec manages the creation and destruction of any federation execution. It initializes RTI components and directs joining federates to the appropriate federation execution. The FedExec supervises a federation. It controls federates joining and resigning the federation, and manages the data exchange

between the participating federates. Federates can be grouped into one or more processes or exist as independent processes. The libRTI library provides RTI services which support the communications among libRTI, RtiExec, and the appropriate FedExec. As a part of the libRTI, class *RTIambassador* provides the RTI services. It plays a very important role in RTI components. Class *FederateAmbassador* supports all the callback functions for each federate. It is an abstract class whose functionalities will be implemented by developers. The communication among the federate objects and two ambassadors is illustrated in Figure 4-1.

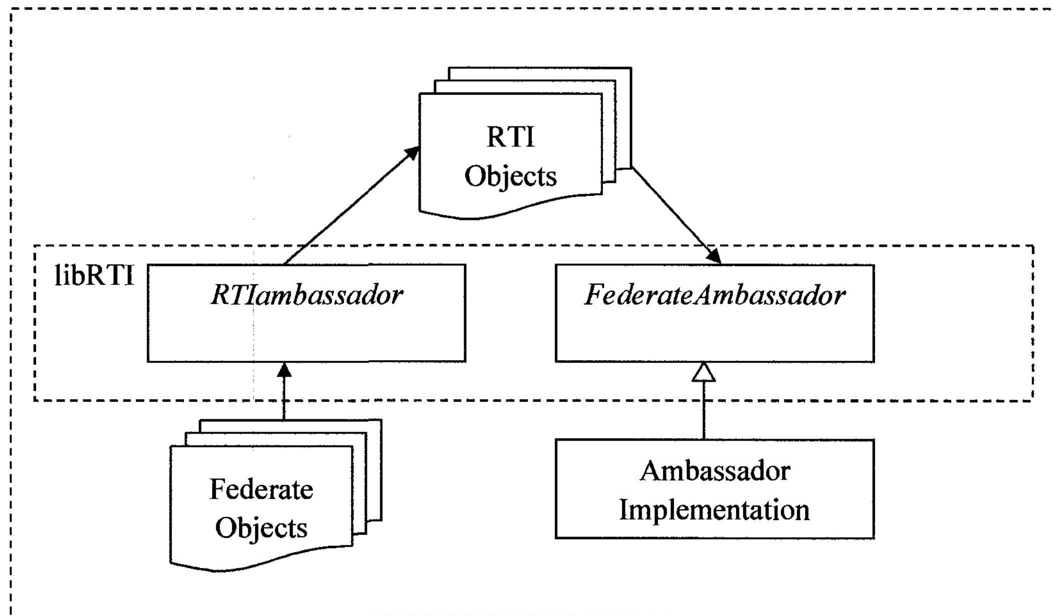



Figure 4-1: Communication between Ambassadors

It is important that a federate communicates with the RTI through the RTIambassador, and that the RTI or the object communicates back to the federate through the FederateAmbassador. The interaction between a federate and the RTI is shown in Figure 4-2.

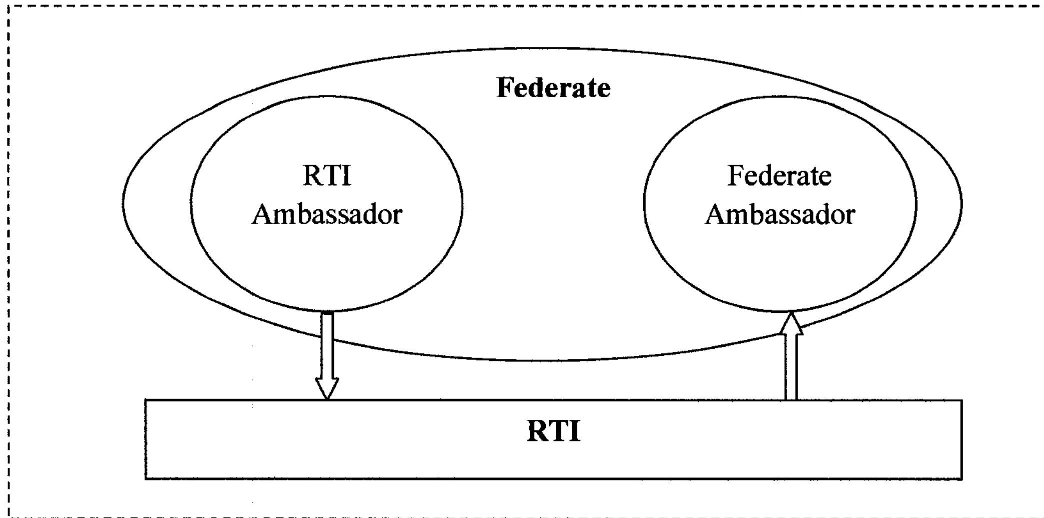



Figure 4-2: Interaction between a Federate and RTI

### 4.1.2 FedExec Management Areas

The communication between federates and federation will be managed in six different areas of the FedExec life cycle. They are Federation Management, Declaration Management, Object Management, Ownership Management, Time Management, and Data Distribution Management. Each of the management areas will provide its own activity coordination. This thesis will focus on the first three management levels, since they are crucial to our RTI communication.

**Federation Management**

The main tasks for Federation Management include the *creation* and *destruction* of federations, the *joining*, *resigning*, *saving* and *restoring* of federates, and the *synchronization* activated by federates. RTIambassador provides a lot of useful methods for the completion of these tasks.

**Declaration Management**

Declaration Management includes *publication*, *subscription*, and other supporting control functions. Federates that act as producers will declare the object attributes or interactions that they want to publish. Federates acting as consumers should declare their subscription interests for the object attributes or interactions that they want to receive. Declaration Management is supported by both RTIambassador services and FederateAmbassador callback methods.

**Object Management**

Object Management provides instance *registration* and instance *update* services to object producers, and it supports instance *discovery* and *reflection* to object consumers. Object Management manipulates the RTIambassador services and FederateAmbassador callback methods to manage objects. For example, when one registered object needs to be deleted from a federate, the RTIambassador method *deleteObjectInstance()* is to be called upon; then, the FederateAmbassador *removeObjectInstance()* callback informs all federates of an "inexistence" for a previously discovered object; finally, the RTIambassador method *localDeleteObjectInstance()* effectively *undiscovers* this object instance.

### 4.1.3 Implementation

By incorporating the concepts mentioned in the previous sections, two RTI classes are implemented in ***SimSITE*** to hold both the RTI and the Federate Ambassador services to deal with the local and remote messages. The class *HumanFed* implements the major functions of RTI API, instances of which can update their states on the local machine. Class *HwFederateAmbassador* is a subclass of FederateAmbassador, which contains methods to enable RTI to call functions in a federate. The basic communication setup is illustrated in Figure 4-3 (a) (b).

**<u>RTI Initial Procedure with Two Federates:</u>**

1) RTI server starts (rtiexec).
2) A thread creates Federate1, then creates Federation through RTIAmbassador.
3) Federate1 joins the Federation through RTIAmbassador.
4) Another thread creates Federate2.
5) Federate2 joins the Federation through RTIAmbassador.
6) Federation informs Federate1 that Federate2 joined.
7) Federate1 publishes the object attributes/instances that it wants to produce through RTIAmbassador.
8) Federate2 subscribes to the object attributes/instances that it is interested in.
9) Vice versa, Federate1 subscribes to the object attributes/instances, and Federate2 can publish the object attributes/instances as well.

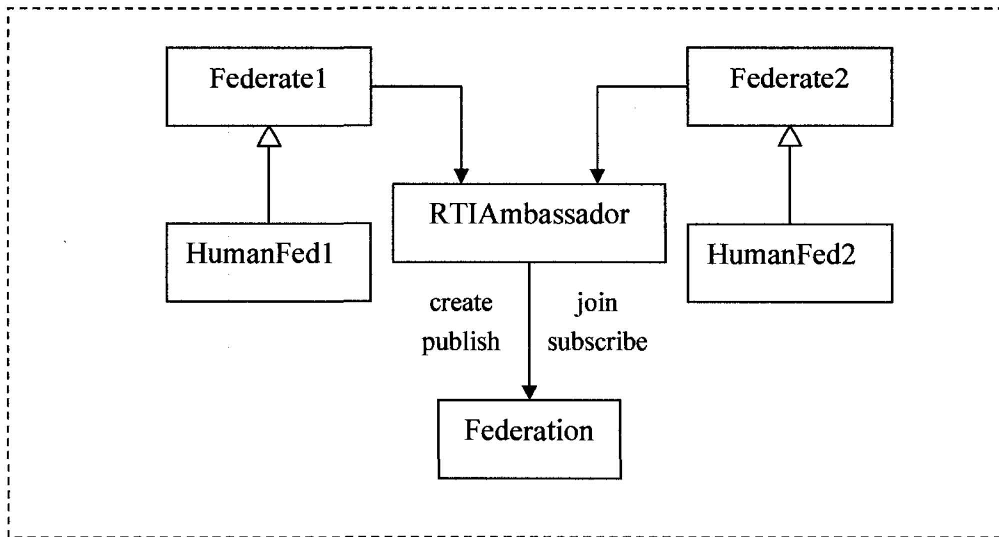



Figure 4-3 (a): RTI Initial

**Object Update**

10) If Federate1 has a new object instance needed to be registered, it will register to the Federation through RTIAmbassador method - *registerObjectInstance()*.

11) The Federation will inform Federate2 through local FederateAmbassador, by calling the method - *discoverObjectInstance()*.

12) If any object attribute/instance (e.g.: position) in Federate1 (HumanFed1) gets changed, Federate1 informs the Federation by calling RTIAmbassador method - *updateAttributeValues()* to update.

13) The Federation conveys the change to Federate2.

14) Federate2 calls the callback function - *reflectAttributeValues()* implemented in FederateAmbassador subclass - HwFederateAmbassador to confirm and to locally update this change.

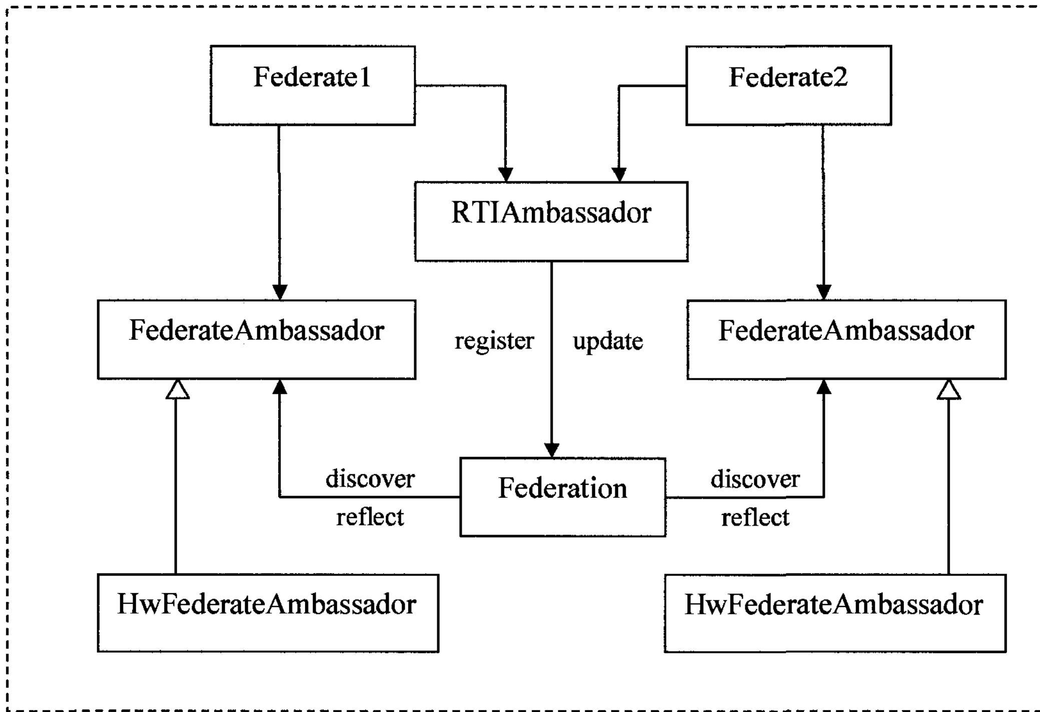



Figure 4-3 (b): Object Update

**RTI Thread**

The overall ***SimSITE*** system has to be programmed in multiple threads. One thread will execute the OpenGL graphic rendering. Another thread that is performing the RTI services will be activated in OpenGL *main()* function. This new thread will first create a federation, and then set up a federate to demonstrate the proper usage of the RTI C++ API. Also, it will yield and control the time for each major activity within the Local RTI Component (LRC) by calling the method - *tick()*. The following procedure describes the main functions of the thread.

**Step 1: Federation Creation** (Federation Management)

By calling the RTIambassador method *createFederationExecution()*, the LRC communicates with the RtiExec process. A parameter contains a FED file that must

exist in the RTI configuration directory. This file specifies the FOM objects and interaction class structures as well as default/initial transport and ordering information for object attributes and interaction classes. Also, User-defined object attributes will be specified in this FED file. If the specified federation does not exist, a new federation with a specified name will be created. Otherwise, a *Federation Execution Already Exists* exception is raised.

**Step 2: Joining the Federation** (Federation Management)

The *joinFederationExecution()* method is called to associate a federate with the federation execution created previously. If the federation execution does not exist, or the specified federate already exists, an exception will be thrown. This method also registers the FederateAmbassador's callbacks by sending its instance as a parameter.

**Step 3: Federate Initiation** (Declaration Management)

All shared attributes such as the position of the avatar have to be translated into the LRC Handles by calling the methods *getObjectClassHandle()* and *getAttributeHandle()*. The object class needed to be handled is *HumanFed.* The attributes that will be handled in the ***SimSITE*** application include the object name, the position (x, y, z coordinates), the colour, and the degree of moving direction.

**Step 4 Publication and Subscription** (Declaration Management)

The RTI will be informed of the data types that a federate can produce, and the object interests and interaction data types that a federate wants to receive are to be declared. The objects (or object attributes) and interactions will be published and subscribed by calling methods: *publishObjectClass()*, *publishInteractionClass()*, and *subscribeObjectClassAttributes()*, *subscribeInteractionClass()*. Also, the publication and subscription methods both require a *RTI:: ObjectClassHandle* and a *RTI:: AttributeHandleSet*.

**Step 5 Object Registration** (Object Management)

An HLA object will be created and the object instance will be registered with the federation execution by calling the RTIambassador method *registerObjectInstance()*. After this step, the object can be noticed by other federates.

**Implementation of FederateAmbassador**

As mentioned previously, the abstract class FederateAmbassador identifies the callback functions that federates need to provide for the federation to communicate back. Each federate must implement the functionalities declared in FederateAmbassador. In the ***SimSITE*** application, these functions covered in five service areas are implemented in class *HwFederateAmbassador*. (Source code has been referenced and modified from RTI1.3NG-V6 Software Package [58].)

Functions implemented for Object Management:

- reflectAttributeValues ()
- receiveInteraction ()
- discoverObjectInstance ()
- removeObjectInstance ()
- attributesInScope ()
- attributesOutOfScope ()
- provideAttributeValueUpdate ()
- turnUpdatesOnForObjectInstance ()
- turnUpdatesOffForObjectInstance ()

As an example, the methods *reflectAttributeValues()* and *receiveInteraction()* will be programmed as following:

```
------------------------------------------------------------------------------------------------
/**********************Implementation of FederateAmbassador Callbacks
**********************Method: reflectAttributeValues () *************************/

void HwFederateAmbassador::reflectAttributeValues ( //response updating
    RTI:: ObjectHandle    theObject,
    const RTI::AttributeHandleValuePairSet& theAttributes,
    const char *theTag) //parameters: handled object and attributes

throw (...) //errors
{
// Find the HumanFed instance for this update (see class HumanFed in detail)
HumanFed *pHumanFed = HumanFed:: FindToUpdate( theObject );
   if ( pHumanFed )
   {
       // Set the new attribute values in this HumanFed instance
       pHumanFed->Reflect( theAttributes ); //see class HumanFed in detail
```

```
        isUpateAttribute= RTI::RTI_TRUE;
    }
}
```

---

```
/********************Implementation of FederateAmbassador Callback
********************Method: receiveInteraction () ****************************/

void HwFederateAmbassador::receiveInteraction (
    RTI::InteractionClassHandle          theInteraction,
    const RTI::ParameterHandleValuePairSet& theParameters,
    const char   *theTag)

throw (...) //errors
{
// update the parameters.
// Attribute updates are provided for an object instance via the RTIambassador
// method: updateAttributeValues(), see class HumanFed in detail
    HumanFed::Update( theInteraction, theParameters );
}
```

---

Service callback functions implemented for other management areas such as Federation Management, Declaration Management, Ownership Management and Time Management are also developed and presented in class *HwFederateAmbassador*. All the callback functions will be registered in the Federation through the RTIAmbassador method *joinFederationExecution()* when the RTI thread is executed.

## 4.2 Requirements of RTI

The implementation of RTI communication is completed in the previous section. To run the simulation in the network, some essential setups have to be done.

### 4.2.1 Network Requirement

The distributed communication protocols used for the RTI1.3NG implementation are based on IP protocol, specifically relying on protocol TCP and UDP. A multiple LAN configuration is supported based on IP communication, which provides an environment to connect federates collaborating with a federation execution. There is a RID file included in the RTI software package, which contains parameters to support different network configurations.

### 4.2.2 System Requirement

The RTI1.3NG implementation supports Windows 98 ($2^{nd}$ Edition), NT 4.0, and Windows 2000 operating systems using the Microsoft Visual C++ compiler version 6.0 running on the Intel x86 architecture [59]. Practically, our application is set up and run on Windows 2000 platform, with Intel Pentium IV processor, and 1GMB of RAM.

### 4.2.3 RTI Installation

After downloading and installing the RTI distribution from the DMSO Software Distribution Center [60], the windows environment configuration needs to be deployed to run the RTI application. The configured environment variables are:

- **RTI_HOME** – set to the RTI installation directory (e.g., J:\LiuKe\RTI\RTI1.3NG-V6)
- **RTI_BUILD_TYPE** – set to the particular build type for the distribution (e.g., Win2000-VC6)
- **PATH** – set to %RTI_HOME%/%RTI_BUILD_TYPE%/bin (e.g., J:\LiuKe\RTI\RTI1.3NG-V6\Win2000-VC6\bin)

Two RTI configuration files (*.fed* and *.rid* file) used to integrate and test the federation will be included in federation execution directory. For further installation details, please refer to RTI-NG 1.3v3.2 Installation Guide.

## 4.3 Simulation Results

The major characteristics of the ***SimSITE*** evacuation training system have been presented in the previous chapters. The aim is to set up a simulation environment for emergency preparedness and evacuation training purposes. To achieve the goal, this multi-agent simulation is designed and implemented using HLA/RTI. HLA, the middleware, handles the information exchange that occurs among different simulated federate members. In ***SimSITE***, the federation consists of three different independent federates. Each federate can run by itself under stand-alone conditions.

### 4.3.1 Training Cases Analysis

The training behaviors are simulated according to the aforementioned scenarios, where the system is configured as the description in the section 4.2.3. Three training cases have been assessed in the experiments; each case is executed five times with the same avatar population randomly relocated for each trial. The training result of the model is assessed based on the average completion time of the five trials.

**Case 1: Auto-avatar following**

In this training scenario, the trainee tries to catch an auto-programmed avatar once the simulation starts, and is led by the avatar to escape from the nearest exit. The completion time (in seconds) for the trainee to exit the building for each trial and the average evacuation time for the group of five trials are listed in the table below:

| Run Time | Trainee Evacuation Time | Average Time |
|---|---|---|
| 1 | 27 | 23.8 |
| 2 | 22 | |
| 3 | 35 | |
| 4 | 22 | |
| 5 | 13 | |

Table 4-1: Auto-avatar Following Evacuation Times

**Case 2: Evacuation plan following**

In this case, the trainee escapes from the building by looking up the evacuation plan momentarily. The map can be loaded on the screen by pressing the action key "*L*". The evacuation results are shown in the table below:

| Run Time | Trainee Evacuation Time | Average Time |
|---|---|---|
| 1 | 273 | 220 |
| 2 | 230 | |
| 3 | 200 | |
| 4 | 212 | |
| 5 | 185 | |

Table 4-2: Evacuation Plan Following Evacuation Times

**Case 3: Evacuation leader following**

In this scenario, three individual dispersed participants join the simulation. One participant plays as the commander role and the others act as trainees. The communication has been set up on the local network among the three research labs. A tele-conferencing system is used to transmit the commands. According to the federation joining status shown in the Figure 4-4, the commander waits until all participants join the simulation. Then the commander broadcasts his/her instructions to every participant on the network, and controls one avatar as the leader to meet the other two in this shared 3D environment. Finally, all the trainees will be led out of the building. The approximate time for all trainees evacuating from the building are shown as below (time starts to count when all federates are connected):

| **Run Time** | **Trainee Evacuation Time (approximate)** | **Average Time** |
|---|---|---|
| 1 | 75 | 77.2 |
| 2 | 86 | |
| 3 | 73 | |
| 4 | 79 | |
| 5 | 73 | |

Table 4-3: Evacuation Leader Following Evacuation Times

```
J:\LiuKe\RTI\RTI1.3NG-V6\Win2000-VC6\bin\rtiexec.exe
was designed to improve usability and add the ability to be run
at multiple locations.  Please consult the RTI Installation Guide
for further information concerning the RTI Console application.


 Hit Ctrl-C to make rtiexec exit

Advertising launcher as RtiLauncher;137.122.91.218;

rtiexec, process id = 4076, endpoint = 137.122.91.218:1417,
multicast discovery endpoint = 224.9.9.2:22605, initialization complete.

"C:\Viewstore\rti1.3ngv6\rting\rti\launch\src\RtiLauncherDistributedObject_i.cpp
", line 59: Environment variable RTI_HOME has not been set.

federation Human finished initialization with process id 916 and endpoint 137.12
2.91.218:1432

Federate DISCOVER is JOINING federation Human at Wed Apr 11 15:35:27 2007

Federate PARADISE is JOINING federation Human at Wed Apr 11 15:36:04 2007

Federate MCR is JOINING federation Human at Wed Apr 11 15:41:15 2007
```

Figure 4-4: Federation Initialization

After comparing these three experimental results, we conclude that the auto-avatar following method is the most efficient way to evacuate from the building. This confirms the advantage of the social interaction rules. And the map looking up is the least efficient way due to the strange and indirect guidance. The leader guiding approach improves the performance of the trainee group by speeding up the initial movement. The network delay has not been noticed obviously.

### 4.3.2 Parameter and Configuration Sensitivity Assessment

The purpose of the assessment is to evaluate the impact of environment settings on the system performance. The testing begins with the establishment of a baseline case and then adjusts specific parameters and system configurations to determine the impacts.

**Case 1: Baseline case**

In the baseline case, the tests are performed on Pentium IV class PCs with 1G RAM running Windows 2000, where 22 3D avatars are loaded. The sensitivity of the model is assessed based on the average completion time of the five trials. The time (in seconds) for the first and last auto-avatar to exit the building for each trial and the average evacuation time for the group of five trials are summarized in the table below:

| Run Time | First Avatar Out | Last Avatar Out | Avg. Evacuation Time |
|---|---|---|---|
| 1 | 4 | 71 | 74.2 |
| 2 | 3 | 65 | |
| 3 | 5 | 78 | |
| 4 | 3 | 80 | |
| 5 | 3 | 77 | |

Table 4-4: Baseline Case Auto-avatar Evacuation Times

**Case2: Effect of population density**

In this experiment, the application is set up with the same configuration as the baseline case while 50 3D avatars are loaded. The time for the auto-avatars to escape from the building is shown in the table below:

| Run Time | First Avatar Out | Last Avatar Out | Avg. Evacuation Time |
|---|---|---|---|
| 1 | 3 | 82 | 85.6 |
| 2 | 4 | 79 | |
| 3 | 4 | 71 | |
| 4 | 3 | 108 | |
| 5 | 5 | 88 | |

Table 4-5: Effect of Population Density on Evacuation Times

**Case3: Effect of system configuration**

In this case, the tests are performed on the machines with a 2G RAM running, where 50 3D avatars are loaded. The time for the auto-avatars to exit the building is listed in the table below:

| Run Time | First Avatar Out | Last Avatar Out | Avg. Evacuation Time |
|---|---|---|---|
| 1 | 3 | 55 | 51.8 |
| 2 | 2 | 48 | |
| 3 | 2 | 45 | |
| 4 | 2 | 52 | |
| 5 | 3 | 59 | |

Table 4-6: Effect of System Configuration on Evacuation Times

As a conclusion, there is no significant difference between the case 1 and case 2, and the overall performance improved dramatically (40%) by updating the application

testing environment. The reason is that, in the ***SimSITE*** application, compared with the static graphic model (the SITE building), the 3D avatars designed by OpenGL contribute only a small portion of the data load. Therefore, the performance will not be decreased significantly with more avatars loaded in the system. However, since the self-collision avoidance has not yet been implemented in the current system, more avatars running in the simulation will cause the collision problem graphically.

### 4.3.3 Simulation Demonstration

The ***SimSITE*** is developed in Visual C++ and OpenGL. As a proof of concept, an evacuation simulation is implemented to demonstrate emergency training among students or employees who work in a multi-storey building with complex floor plans. The overall demo is tested among the three research labs at the University of Ottawa: DISCOVER, MCRLab and PARADISE lab (see Figure 4-4). Figure 4-5 listed below shows typical simulation scenarios of the integrated simulation system.

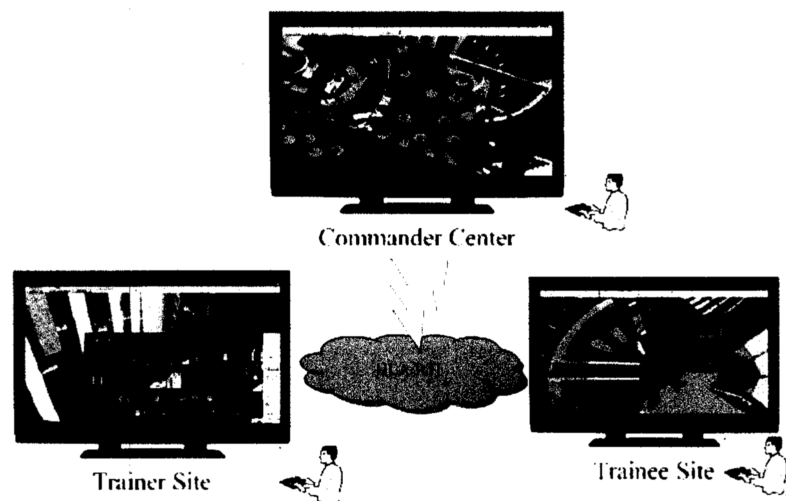
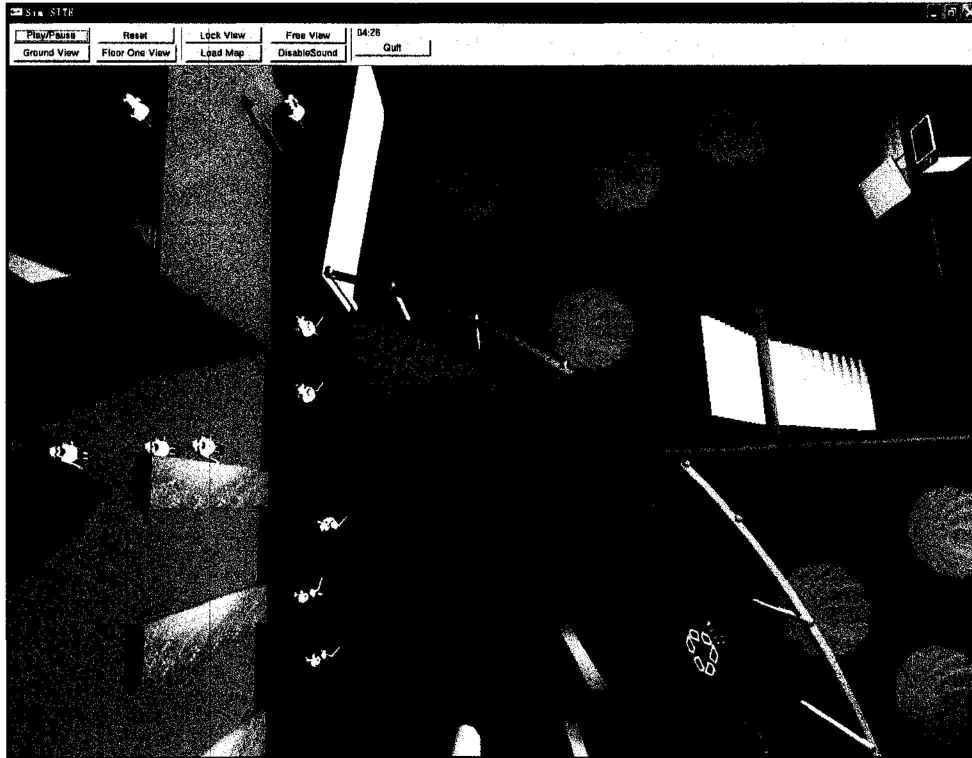



Figure 4-4: Proof-of-Concept




(a) Automatic Evacuation

(b) Emergency Evacuation Training

Figure 4-5: Simulation Result in Different Scenarios

# Chapter 5
# Conclusion and Future Work

## 5.1 Conclusions

The main objective of this research is to design a collaborative training simulator for emergency preparedness and response. The collaboration of diverse personnel and applications will provide improved opportunities for team training. The capability to train responders and commanders together on a wide range of scenarios will enable the development of effective incident management teams. These teams can build on their shared sets of experiences developed through incident management training. The shared experiences will develop the understanding of capabilities and command decisions in the team resulting in increased cohesiveness and effectiveness.

The ***SimSITE*** captures various usages of tools such as planning, vulnerability analysis, identification and detection, training and real time response support. There will be more issues involved in the actual implementation of this application. The thesis addresses some of the fundamental design issues related to the manner in which independent applications can be made to interact with each other. The system architecture and functions of the simulation based on the HLA/RTI are proposed, followed by a detailed explanation about the development process of each federate member (graphic design). Finally, the overall application is tested successfully on the local area network.

## 5.2 Future Work

The simulation industry is always evolving, which continually pushes the applications to a wider, better, and faster level. Future additions to ***SimSITE*** include, but are not limited to, the ability to record the simulation in a database, better collision detection algorithm (self-collision avoidance), human behaviors study, more realistic emergency scenarios development, icon based GUI and mini-map navigation design, involving more varieties of simulators, cooperation with other federates, and more efficient techniques applying to improve the graphics calculation and display. The new tools and improved architecture built later on will make it easier for developers and domain experts across the country to collaborate online to build software and to tailor simulations, better, quicker, and cheaper.

# Publication

K. Liu, X. Shen, A. El Saddik, A. Boukerche, N. D. Georganas, "SimSITE: The HLA/RTI Based Emergency Preparedness and Response Training Simulation", The 11-th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT), Chania, Crete Island, Greece, October 2007

# References


[1] S. Jain and C. R. McLean, “Modeling and Simulation for Emergency Response: Workshop Report, Standards and Tools”, December 2003. Available: http://www.mel.nist.gov/msidlibrary/doc/nistir7071.pdf

[2] DMSO, “High Level Architecture for Simulations Interface Specification”, Version 1.3, Defense Modeling and Simulation Office. Available: https://www.dmso.mil/public/transition/hla/

[3] D. McGrath and C. Carella, “Synthetic Environments for Emergency Response Simulation”, Institute for Security Technology Studies, Dartmouth College, 2005. Available: http://www.ists.dartmouth.edu/library/118.pdf

[4] D. Hall, “Modeling and Simulation for Emergency Response”, Manufacturing Engineering Laboratory, March 4, 2003. Available: http://www.mel.nist.gov/div826/msid/sima/simconf/proc/ftp/hall.pdf

[5] X. J. Shen, J. L. Zhou, A. E. Saddik, and N. D. Georganas, “Architecture and Evaluation of Tele-Haptic Environments”, Proc. 8th IEEE International Symposium on Distributed Simulation and Real Time Applications (IEEE DS-RT 2004), October 2004, Budapest, Hungary.

[6] S. Jain and C. R. McLean, “An Integrating Framework for Modeling and Simulation for Emergency Response”. Available: http://www.mel.nist.gov/msidlibrary/doc/JHSEM.pdf

[7] C. R. McLean and S. Jain, "Vision for an Integrated Emergency Response Framework", NIST. Available: http://www.mel.nist.gov/div826/msid/sima/simconf/proc/ftp/mclean_jain.pdf

[8] C. McLean and S. Jain, "Vision for an Integrated Emergency Response Framework", Manufacturing Simulation and Visualization Program, NIST. Available: http://www.mel.nist.gov/div826/msid/sima/simconf/proc/ftp/mclean_jain.pdf

[9] S. Jain and C. R. McLean, "A Framework for Modeling & Simulation for Emergency Response", 2003. Available: http://www.informs-cs.org/wsc03papers/132.pdf

[10] N. J Kopp, "2D Evacuation Simulation Model". [Online]. Available: http://www.nathan.kopp.com/EvacSim/evacsim_acm.html

[11] J. Isdale, "Introduction to Virtual Reality", A Web-Based Introduction, Version 4 – Draft 1, September 1998. [Online]. Available: http://vr.isdale.com/WhatIsVR/frames/WhatIsVR4.1.html

[12] W. R. Sherman and A. B. Craig, "Understanding Virtual Reality: Interactive, Application, and Design", Morgan Kaufmann Publishers, 2003.

[13] Y. Murakami, K. Minami, T. Kawasoe and T. Ishida, "Multi-Agent Simulation for Crisis Management", JST CREST Digital City Project, Department of Social Informatics, Kyoto University. Available: http://www.lab7.kuis.kyoto-u.ac.jp/publications/02/yohei-kmn2002.pdf

[14] G. Santos and B. E. Aguirre, "Critical Review of Emergency Evacuation Simulation Models", January 2005. [Online]. Available: http://fire.nist.gov/bfrlpubs/fire05/art012.html

Fire Safety Engineering Group, "Exodus Animations", School of Computing and Mathematical Sciences, University of Greenwich. [Online]. Available: http://fseg.gre.ac.uk/fire/EXODUS_animations.asp

[15] S. Singhal and M. Zyda, "Networked Virtual Environments. Design and Implementation", USA: ACM Press, Addison Wesley, 1999, New York.

[16] R. M Fujimoto, "Parallel and Distributed Simulation Systems", USA: John Wiley and Sons, Inc, 2000, New York.

[17] C. A. Boer, A. D. Bruin, and A. Verbraeck, "Distributed Simulation in Industry – A Survey Part 1 – The COTS Vendors", Proceedings of the 2006 Winter Simulation Conference, 2006.

[18] C. A. Boer, "Distributed Simulation in Industry", Ph.D. Thesis, Erasmus University Rotterdam, Netherlands.

[19] S. Jain and C. R. McLean, "Modeling and Simulation for Emergency Response: Workshop Report, Standards and Tools", December, 2003.

[20] DMSO, "High Level Architecture Object Rules v1.3", Technical Report, Defense Modeling and Simulation Office, 1998a.

[21] X. Shi and Q. Zhong, "The Introduction on High Level Architecture (HLA) and Run-Time Infrastructure (RTI)".

[22] J. S. Dahmann and K. L. Morse, "High Level Architecture for Simulation: An Update".

[23] McLeod Institute of Simulation Sciences, "HLA MODULE 1 Part 1- Introduction to the High Level Architecture", University Outreach Program. [Online]. Available:
http://www.ecst.csuchico.edu/~hla/

[24] DMSO, "High Level Architecture Interface Specification", v1.3, Defense Modeling and Simulation Office, 5 February, 1998. Available:
http://hla.dmso.mil/hla/tech/ifspec/if1-3d9b.doc
CHICO, "Introduction to a Fundamental Set of RTI Interfaces", lecture notes, California State University. Available:
http://www.ecst.csuchico.edu/~hla/LectureNotes/HLA_1516_M1_P2.doc

[25] DMSO, "High Level Architecture Run-Time Infrastructure RTI 1.3-Next Generation Programmer's Guide Version 3.2", Department of Defense Modeling and Simulation Office.

[26] DMSO, "High Level Architecture Object Model Template", v1.3, Defense Modeling and Simulation Office, 5 February, 1998. Available:
http://hla.dmso.mil/hla/tech/omtspec/omt1-3d4.doc
CHICO, "The Object Model Template", lecture notes, California State University. Available:
http://www.ecst.csuchico.edu/~hla/LectureNotes/HLA_1516_M1_P3.doc

[27] DMSO, "High Level Architecture Rules", v1.3, Defense Modeling and Simulation Office, 5 February, 1998. Available:

http://hla.dmso.mil/hla/tech/rules/rules1-3d2b.doc

CHICO, "Management of Time in HLA Simulations", lecture notes, California State University. Available:

http://www.ecst.csuchico.edu/~hla/LectureNotes/HLA_1516_M1_P6.doc

[28] Wikipedia, "Open Graphics Library", Definition of OpenGL. [Online]. Available:

http://en.wikipedia.org/wiki/OpenGL

[29] OpenGL Official Website. [Online]. Available:

www.opengl.org

[30] School of Information Technology and Engineering, University of Ottawa. [Online]. Available:

http://www.site.uottawa.ca/eng/

[31] 3ds Model Source. [Online]. Available:

http://www.digitaldreamdesigns.com/3DModels.htm

[32] CGIndia, 3dsMax Tutorials. [Online]. Available:

http://cg-india.com/tutorials/3dsmax_tutorials_uvmapping.html

[33] "3D Simpson Design", class assignment for course CSI4130, SITE, University of Ottawa, Winter 2005

[34] Wikipedia, "Homer Simpson". [Online]. Available:

http://en.wikipedia.org/wiki/Homer_Simpson

[35] Game Tutorial, Partial Source Code. [Online]. Available: www.gametutorials.com

[36] J. JOUVIE, "Moving in a 3D world – Camera", 2004-2007. [Online]. Available: http://jerome.jouvie.free.fr/OpenGl/Tutorials/Tutorial26_Advanced.php

OpenGL, "Transformation". [Online]. Available: http://www.opengl.org/resources/faq/technical/transformations.htm

Fallout Software, "Introduction to 3D". [Online] Available: http://www.falloutsoftware.com/tutorials/gl/gl0.htm

[37] T. M. Reynolds and D. Blythe, "Advanced Graphic Programming Using OpenGL", p. 20.

[38] J. F. S. Hill, "Computer Graphics Using OpenGL", 2nd edition, p. 383.

[39] J. Lewis, "The Unofficial 3DStudio 3DS File Format", December 1998. [Online]. Available: http://www.the-labs.com/Blender/3DS-details.html

D. Vitulli, "3DS Loader", 2003. [Online]. Available: http://www.spacesimulator.net/tut4_3dsloader.html

[40] Pige Master Thesis, p. 26. Available: http://www.edenwaith.com/products/pige/thesis/Thesis%20-%20Part%202.pdf

[41] Wikipedia, "Graphic Rendering". [Online]. Available: http://en.wikipedia.org/wiki/Rendering_(computer_graphics)

[42] Wikipedia, "Retained Rendering Mode". [Online]. Available: http://en.wikipedia.org/wiki/Retained_mode

[43] A. R. Fernandes, "Display List Tutorial". [Online]. Available: http://www.lighthouse3d.com/opengl/displaylists/

[44] H. Bungartz, M. Griebel, and C. Zenger, "Introduction to Computer Graphics", second edition, p. 116.

[45] B. T. Phong, "Illumination for Computer Generated Pictures", Communications of the ACM, p. 311 – 317, 1975.

Gouraud Shading Technology. [Online]. Available: http://freespace.virgin.net/hugo.elias/graphics/x_polygo.htm

[46] D. Vitulli, "Vectors, Normals and OpenGL Lighting". [Online]. Available: http://www.spacesimulator.net/tut5_vectors_and_lighting.html

[47] S. Quinlan, "Efficient Distance Computation between Non-Convex Objects", Robotics Laboratory, Stanford University. Available: http://robotics.stanford.edu/~latombe/cs326/2002/quinlan.pdf

[48] Pige Master Thesis, p. 14. Available: http://www.edenwaith.com/products/pige/thesis/Thesis%20-%20Part%202.pdf

[49] G. Wainer and J. Ameghino, "Using Cell-DEVS for Modeling Complex Cell Spaces", Carleton University, Universidad de Buenos Aires. Available: http://www.sce.carleton.ca/faculty/wainer/papers/AIS04.pdf

[50] B. R. Donald, "A Search Algorithm for Motion Planning with Six Degrees of Freedom," Artificial Intelligence, 31, 1987, pages 295–353.

[51] J. Lengyel, M. Reichert, B. R. Donald, and D. P. Greenberg, "Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware". Available:

http://www.cs.dartmouth.edu/~brd/papers/siggraph/animation.pdf

[52] P. G. Franciosa, D. Frigioni, and R. Giaccio, "Semi-Dynamic Shortest Paths and Breadth-First Search in Digraphs", March 1996. Available: ftp://www.dis.uniroma1.it/pub/Theory/alcom-it/AlcomArchive/WP3/tr-004.ps.gz

[53] L. Dorst and K. Trovato. "Optimal path planning by cost wave propagation in metric configuration space," Proceedings of SPIE-The International Society for Optical Engineering, 1007, November 1988, pages 186–197.

[54] P. Rademacher, "GLUI 2.0 Manuel: A GLUT-Based User Interface Library", June 10, 1999. Available: http://www.cs.unc.edu/~rademach/glui/src/release/glui_manual_v2_beta.pdf

[55] M. Boelter, "Billboarding How To". [Online]. Available: http://nehe.gamedev.net/data/articles/article.asp?article=19
A. R. Fernandes, "Billboarding Tutorial", Lighthouse3d. [Online]. Available: http://www.lighthouse3d.com/opengl/billboarding/

[56] OpenGL Fonts. [Online]. Available: http://www.opengl.org/resources/features/fontsurvey/

[57] J. Maurais, "OpenAL Lesson 1: Simple Static Sound", June 2003. [Online]. Available: http://www.devmaster.net/articles/openal-tutorials/lesson1.php
OpenAL Technology Official Website. [Online]. Available: http://www.openal.org/

[58] Application Hello World Implementation, RTI1.3NG-V6 Software Package. [Online]. Available: https://www.dmso.mil/public/transition/hla/

[59] DMSO, "RTI-NG 1.3v3.2 Release Notes". [Online]. Available: http://hla.dmso.mil/sdc-cgi/hla-cat.pl.

[60] Software Distribution Center. [Online]. Available: https://sdc.dmso.mil/