5-10-2003

The Simulation System for Propagation of Fire and Smoke

Dmitry N. Shulga

THE SIMULATION SYSTEM FOR PROPAGATION OF FIRE AND SMOKE

By
Dmitry Shulga

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi
May 2003

THE SIMULATION SYSTEM FOR PROPAGATION OF FIRE AND SMOKE

By

Dmitry Shulga

Approved:

Tomasz A. Haupt Research Engineer Engineering Research Center (adviser and committee member)	Edward A. Luke Assistant Professor of Computer Science and Engineering (committee member)
David A. Dampier Assistant Professor of Computer Science and Engineering (committee member)	Susan M. Bridges Professor of Computer Science and Engineering Graduate Coordinator of the Department of Computer Science and Engineering
A. Wayne Bennett Dean of Colledge of Engineering	

Name: Dmitry Shulga

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Tomasz A. Haupt

Title of Study: THE SIMULATION SYSTEM FOR PROPAGATION OF FIRE AND SMOKE

Pages in Study: 92

Candidate for Degree of Master of Science

This work presents a solution for a real-time fire suppression control system. It also serves as a support tool that allows creation of virtual ship models and testing them against a range of representative fire scenarios. Model testing includes generating predictions faster than real time, using the simulation network model developed by Hughes Associates, Inc., their visualization, as well as interactive modification of the model settings through the user interface.

In the example, the ship geometry represents ex-USS Shadwell, test area 688, imitating a submarine. Applying the designed visualization techniques to the example model revealed the ability of the system to process, store and render data much faster than the real time (in average , 40 times faster).

DEDICATION

To my mama, my papa and my lovely sister.

ACKNOWLEDGEMENTS

I would like to express my gratitude to a lot of people for their support and assistance in developing this work, but first of all, I want to say that there is one man without whom this would not be possible at all and who urged me on by way of his incredible support and belief in me. Thank you, Dr. Tomasz Haupt.

There are also many that from behind the scenes have encouraged and supported my work, and I wish to thank them. Thank you Patricia Tatem from NRL, Brad Havlovick from Havlovick Engineering Services, and all ex-USS Shadwell crew.

Thank you to my parents, who miss me enormously, to my sister and to all my friends, who always over-evaluate me and my abilities.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vii

CHAPTER

I. INTRODUCTION 1

1.1 Role of Study of Fire and Smoke Propagation 1
1.2 Hypothesis 4

II. LITERATURE AND TECHNOLOGY REVIEW 6

2.1 Software 6
2.1.1 Fire Protection ASCOS 6
2.1.2 CFAST 6
2.1.3 Berkeley Firewalk 7
2.1.4 Multi Room Fire Code 8
2.1.5 Fire Dynamics Simulator 9
2.1.6 STAR-CD 11
2.1.7 Summary 13

2.2 Technology 13

III. SYSTEM DESIGN 16

3.1 Requirements 16
3.1.1 General Description 16
3.1.2 Simulated Parameters 18
3.1.3 Coding 18
3.1.4 Use Cases 19
3.1.5 System Components 21

CHAPTER Page

3.2 Class Hierarchies 22
3.2.1 Class Identification........ 22
3.2.2 Geometry Classes........ 24
3.2.3 Ventilation and Firemain Systems........ 24
3.2.4 Scene Classes 27
3.2.5 Main Scene 28

3.3 Database Structure........ 29
3.3.1 Geometry 29
3.3.2 Ventilation System 30
3.3.3 Firemain System 32
3.3.4 Simulation Data........ 33

3.4 Database Buffering........ 34
3.4.1 Motivation........ 34
3.4.2 Classes and Structures 36
3.4.3 Scene Classes Dependence........ 38

IV. IMPLEMENTATION DETAILS 39

4.1 Fire Simulation 38
4.1.1 The Model Input File 38
4.1.2 Simulation Multithreading 40
4.1.3 Simulation Modes 41
4.1.3.1 New Simulation........ 42
4.1.3.2 Simulation Replay........ 47
4.1.3.3 Comparison of Two Simulations 52

4.2 Visualization 53
4.2.1 Geometry 53
4.2.1.1 Compartments 53
4.2.1.2 Openings 55
4.2.1.3 Ducts........ 56
4.2.2 Parameters Representation........ 59
4.2.2.1 Options Dialog Tab: Species Color Mapping........ 60
4.2.2.2 Legend Dialog 65
4.2.3 Temperature, Toxicity and Oxygen 68
4.2.4 Smoke........ 68
4.2.5 Multiple Species Visualization 71

4.3 Auxiliary Classes........ 73
4.3.1 Command Line Parameters Manager........ 73
4.3.2 Data Bridging........ 76

CHAPTER Page

V. RESULTS.......... 78

5.1 Designed Software.......... 78
5.2 Hypothesis Validation.......... 86

VI. FUTURE WORK.......... 87

REFERENCES.......... 90

LIST OF FIGURES

FIGURE Page

1: Examples from Fire Dynamics Simulator work - testing area on the left, temperature distribution of the right. 10

2: Examples of STAR-CD simulations. 12

3: The Use-case diagram.......... 19

4: System component diagram.......... 20

5: Geometry class associations. 24

6: Openings class hierarchy.......... 25

7: Ducts class hierarchy. 26

8: Complete scene class hierarchy. 29

9: Geometry tables.......... 30

10: Ventilation duct tables.......... 31

11: Openings table. 32

12: Firemain tables.......... 32

13: Tables for storing simulation data.......... 34

14: Generic class for representing a complex entity (for example, side is composed of vertices). 35

15: Geometry data storage classes.......... 36

16: The BodyStructure class - a container for all ship data. 37

FIGURE Page

17: Relationships between scene and data buffering classes. 38

18: The Object State Edit dialog window. 42

19: The ObjState class. 44

20: The Fire Simulation dialog window. 45

21: The Replay tab of the Fire Simulation dialog window.......... 47

22: The Available Simulations dialog window. 46

23: Sampling in space algorithm. 50

24: Sampling in time algorithm. 51

25: The Compare tab of the Fire Simulation dialog window. 52

26: Compartment interior with enabled polygon offset. 53

27: Polygon offset is disabled. 54

28: Ventilation duct with fans (blue) and dampers (brown)........ 58

29: Color mapping with two critical levels and constant gradient. 60

30: Color mapping with two critical levels and two gradients in the end of each critical value range. 62

31: The Options dialog window. 64

32: The Legend dialog window: temperature color map and critical levels on the left; smoke gradient and critical levels on the right. 66

33: Smoke visualization: the top compartments and the bottom left are partially smoked; the compartment in the middle is free of smoke; a bottom front compartment has high concentration of smoke than any. 69

34: Charles Minard's multidimensional diagram. 71

FIGURE Page

35: Different levels of smoke transparency and aggregation of compartment wall color. 72

36: The Singleton class prototype. 75

37: The CmdParams class. 75

38: Data interfaces. 77

39: The main application window. 78

40: Changed level of details: no compartment walls. 79

41: Fire simulation: visualization of temperature. 80

42: Fire simulation: visualization of smoke. 81

43: Fire simulation: visualization of temperature and smoke..... 82

44: Fire simulation: smoke. 83

45: Fire simulations: comparison of two simulations.......... 84

CHAPTER I

INTRODUCTION

1.1 Role of Study of Fire and Smoke Propagation

In both peacetime and war, fire represents a significant threat to any ship. A fire, whether started by a mechanical failure or damage by from a weapon hit, threatens the ship in a number of ways. The crew's health and ability to operate the ship are affected by direct exposure to the fire or by the spread of smoke and toxic gases through the ship by either natural or mechanical ventilation. Electrical systems can be degraded by thermal exposure, exposure acid gases in the combustion products, or by electrical failure resulting from soot deposition, which might include hampered cooling or dielectric breakdown from the electrical conductivity of the soot. Mechanical systems can suffer thermal damage. Lastly, on a vessel carrying munitions, fire can potentially ignite explosive materials, rocket motors, aviation fuel or other highly flammable substances, which could possibly result in temperatures or overpressures high enough to affect the ship structurally.

There currently exist a number of analytical tools for examining the effects of a fire that can be applied on a ship and to its crew. One could use hand calculations for examining simple scenarios in single compartments. Simple rules can be used to extend this approach to multiple compartments. Zone models are suitable for examining more complex, time-dependent scenarios involving multiple compartments and levels, but numerical stability can be a problem for multi-level scenarios, scenarios with Heating, Ventilation and Air Conditioning (HVAC) systems and for post-flashover conditions. Computational fluid dynamics (CFD) models can yield detailed information about temperatures, heat fluxes, and species concentrations; however, the time penalty of this approach currently makes using CFD unfeasible for long periods of real time or for large computational domains. There exist a variety of network models to mode ventilation systems in buildings or fluid flow in piping networks, but they lack the physical mechanisms needed for fire modeling. Given an increased desire for performance-based examination of a response to a fire, there is the need for a new class of fire model. What is needed is a model that can handle very large, complex structures with ventilation and suppression systems, such as naval vessels.

Modeling of fire and shipboard fire suppression systems is an instrument that may be used to eliminate unsuitable designs prior to real testing and can provide useful optimization insight, namely the structure of the ventilation, firemain and other onboard systems. It can also reduce the number of tests required to identify a suitable design thus lowering the overall system design cost.

The primary input for fire analysis is empirical test data. Due to the large scale of this data, post-processing tools need to be used. Study of this data can reveal design flaws and non-optimality. Consequently, rebuilding and additional testing is required. The next step is an ability to overcome such overhead and to be able to build fire suppression systems that are close to the optimal structure from the very beginning. Logically, ship designers want to have tools that allow them to create virtual models of ships and test these models against a range of representative fire scenarios. These tests, if designed properly, identify the limits of performance of the system against a realistic range of conditions, including worst-case scenarios, and establish agreed-upon and measurable performance objectives [4].

The conclusion is that the development of new ships, real-time control of fire and fire suppression systems, as well as preliminary training of personnel, is an integral part of the modern ship design process. Computerized modeling and testing allows manufacturers to benefit by identifying the variables that have the greatest effects on the system performance and would aid in the development of an optimized design.

This work presents a solution that ultimately is aimed at providing real-time fire suppression control system, serving as a design tool for ship modelers as well as a crew training tool. In the considered example, ship geometry was produced by Havlovic Engineering Services, and it represents ex-USS Shadwell, test area 688, imitating a submarine. The simulation network model was developed by Hughes Associates, Inc. (HAI).

1.2 Hypothesis

With the given condition that there are limitations from the available hardware, namely a single processor PC with Pentium 4 class CPU running Windows NT/XP, and that a fire model must run near to or faster than real time, it is possible to develop a simulation system that can be used as a tactical tool to support onboard fire control and suppression. This system will:

- Generate predictions faster than real time.
- Allow interactive modification of model settings to accommodate the actual conditions of the ship through intuitive and simple to operate graphical user interface (GUI).
- Provide accurate, easy-to-read, real-time visualizations of the model output.

The two major challenges are minimization of the CPU load imposed by visualizations, allowing the model to run as fast as possible, and design GUI and output visualizations that add to the fire suppression process in extreme operational conditions during an onboard fire, with immediate threat of the lives of crew members or the ship altogether.

CHAPTER II

LITERATURE AND TECHNOLOGY REVIEW

2.1 Software

1.2.1 Fire Protection ASCOS

Analysis of Smoke Control Systems (ASCOS) is a software package written to predict the impact of a smoke management system on building airflows [7]. ASCOS solves the steady-state airflow through a connected series of compartments, which are defined at a fixed pressure and temperature. Various correlations are used to account for flow losses in shafts, stairwells, and other form loss types. As a steady-state code, its overall solver is not applicable to a fire model; however, the specialized correlations for flow losses in shafts, hallways, and stairwells would be useful to incorporate into a fire model.

1.2.2 CFAST

Consolidated Model of Fire Growth and Smoke Transport (CFAST) is a zone model fire code written by the Building and Fire Research

Laboratory at the National Institute of Standards and Technology (NIST/BFRL) [4]. CFAST solves a zone-model set of flow equations for a multi-compartment, multi-level structure with a ventilation network. CFAST includes simple ignition, radiation heat transfer, wall/ceiling jet, flame spread, and intercompartment conduction models. CFAST also has GUI for preprocessing, execution monitoring and post processing. The GUI is based on pre-Windows®, MS-DOS® technology and as such is awkward and dated looking. However, some of the overall concepts with regard to the overall setup of the input processor and the ability to monitor parameters during runtime are not valuable.

CFAST, when executing properly, is fast-running and capable of real-time computational speeds. However, the CFAST solver does not contain a sufficient degree of intercompartment coupling for the pressure solution or the ventilation network submodels. As a result CFAST is often unstable and can be overly sensitive to small changes in input conditions. Therefore, while it is a fast solver with many of the phenomena needed for a real-time shipboard model, it is not reliable enough to be considered as a source for the primary solution algorithm.

1.2.3 Berkeley Firewalk

Berkeley Firewalk [25] is an offshoot of the Berkeley Walkthrough program, whose original intent was to interactively model architectural environments from floor plans. Research into integrated simulations in 3D virtual environments combined the CFAST zone model with this basic visualization system to form Firewalk. The system allows a CFAST server, which can run on the user's machine or a separate machine to distribute the computing load, to connect to a Walkthrough client program and 3D visualizer. From the client, the user can walk through the building interactively and initiate, control, and view the impact of CFAST runs in the building being visualized. A VCR-style panel controls the playback of the events being simulated, and a number of viewing modes simulate what the environment would look like and what physical conditions are in the various rooms. Quantitative displays are available to graph or list numerical quantities.

The system can automatically export building geometry to CFAST from the Walkthrough model, allowing the user to model with the Walkthrough tools, providing for an easier and more visual entry of new buildings. The system is designed to provide rapid prototyping, easily understandable visualizations, and greater ease of comparative modeling for the user. The system is part of an ongoing research program into richly interactive virtual environment systems.

1.2.4 Multi Room Fire Code

Multi Room Fire Code (MFRC) [34] is multi-room fire model used as a simulation model for calculation of smoke movement and temperature load on structures. It is capable to calculate the evolving distribution of smoke, fire gases and heat throughout a constructed facility during a fire. The size of the fire is variable during simulation. The model also incorporates the evolution of the species, such as carbon monoxide, which is important to the safety of individuals subjected to a fire environment.

Version 2.7.3 models up to 40 compartments, 100 openings, fan or duct systems, several individual fires, up to one flame-spread object, multiple plumes, ceiling jets, multiple sprinklers and the seven species considered most important in toxicity of fires. The geometry includes variable area/height relations, thermo-physical and pyrolysis databases, multi-layered walls, wind, the stack effect, building leakage and flow through holes in floor/ceiling connections. The distribution includes text report generators, even for graphics with common plotting packages, and a system for comparing many runs done for parameters estimation.

1.2.5 Fire Dynamics Simulator

Fire Dynamics Simulator (FDS) [27] is a large-eddy simulation CFD code written by the NIST/BFRL [11]. FDS solves a low-mach number form of the Navier-Stokes equations using a fast Fourier transform solver for pressure, a Smagorinski subgrid scale model for turbulence and a mixture fraction combustion model. FDS can run in a direct numerical simulation (DNS) mode if certain node resolution conditions are met. In DNS mode, FDS can also use a single-step, finite-rate kinetics model for combustion. FDS has submodels for radiative heat transfer, ID conductive heat transfer, sprinkler nozzles, droplet transport and evaporation, simple pyrolysis, fuel sprays, liquid fuel pools and multi-grid operation. FDS has a companion program called Smoke view, which is an OpenGL application for viewing FDS results with high resolution, and 3D animations.

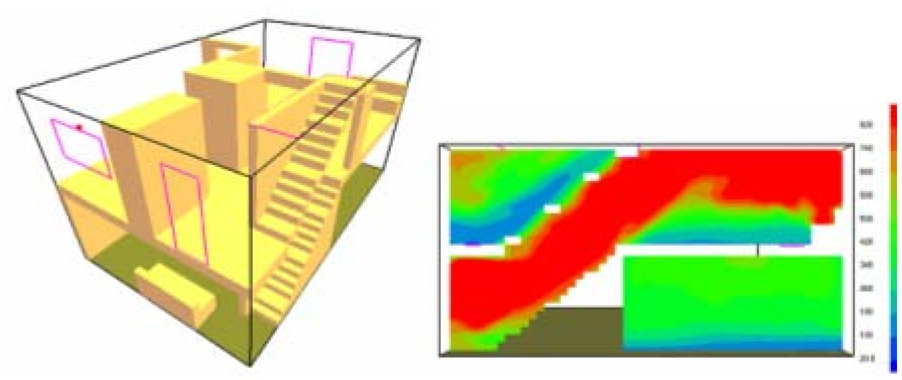


Figure 1: Examples from Fire Dynamics Simulator work – testing area on the left, temperature distribution of the right.

The FDS source is publicly available and the lead authors of FDS and Smoke view are highly responsive to user feedback of the model.

FDS has undergone some verification and validation and compares well with test data in pre-flashover compartments. However, while a CFD tool such as FDS has the greatest potential for accuracy and precision in modeling fires and the associated heat and mass transfer, CFD has a significant drawback. CFD is very computationally intensive, both in terms of time and memory requirements. Real-time CFD for large-scale, complex structures is only possible with large, massively parallel supercomputers. These machines consume space, electric power and cooling resources, and they are costly to acquire, maintain and operate. Furthermore, a successful CFD simulation of fire growth and spread requires detailed knowledge of the fundamental behavior of real life materials exposed to a time-varying heat flux. This is knowledge that does not exist for most solid phase combustibles. Adding the capability to simulate suppression systems only complicates the issue. CFD is not a realistic option [22].

In Figure 1, two levels of a townhouse are modeled by a 10.0 m (32.8 ft) x 6.0 m (19.7 ft) x 5.1m (16.8 ft) tall rectangular volume. For the FDS simulation this volume was divided into 76,500 computational cells. Each cell had dimensions 0.2 m (7.9 in) x 0.2 m (7.9 in) x 0.1 m (3.9 in).

The problem of this package is that the maximum case size for 512 MB of main memory will be around 600,000 cells. If we consider a normal-sized ship, then it might even be impossible for this tool just to allocate enough resources. In addition, it was not designed to work in real-time.

1.2.6 STAR-CD

Star-CD [24] is a general purpose, unstructured CFD code that contains industry-standard models for modeling fire and smoke movement. It is a powerful general purpose CFD code that benefits from an easy-to-use GUI, which allows complex scenarios to be developed, simulated and analyzed without difficulty. Star-CD is widely used by building and transport service companies to investigate fire and smoke movement in different types of buildings. The scenario of interest can be generated via the GUI or imported from many popular CAD packages, allowing for simple geometries and the extremely complex scenarios often encountered in many industrial situations to be simulated.

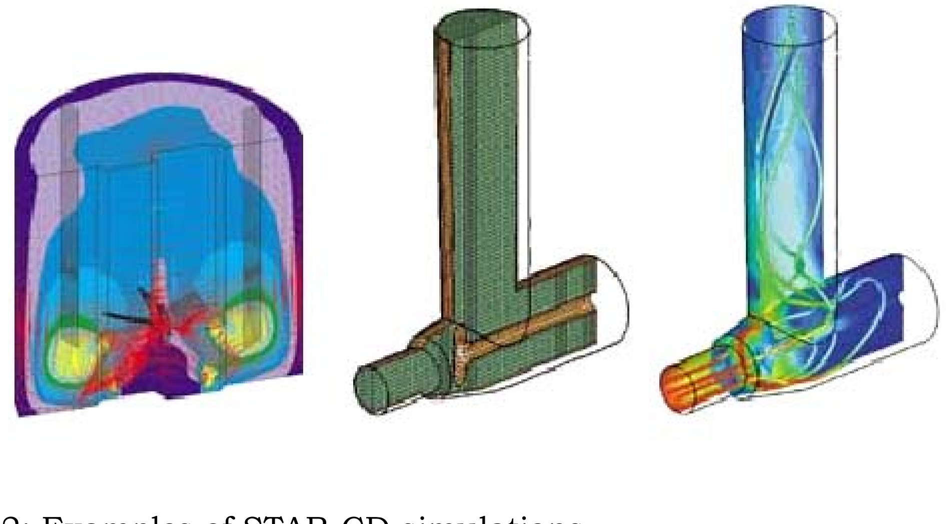


Figure 2: Examples of STAR-CD simulations.

Analysis of the simulation can be carried out using the GUI. Powerful post-processing allows the users to investigate the simulation results in great detail. The information gained can be used to easily manipulate or refine the mesh for further simulations and parametric studies. Two- and three-dimensional plots can be exported to well-known image formats, such as GIF and postscript.

1.2.7 Summary

There are also many other software packages, but none of them satisfy the system requirements. Visualization must be much faster than real time, but CFD-based models are unable to achieve such performance. Simulations must be interactive, i.e. the user must be able to change the states of objects during simulation runs, but none of the presented packages has such GUI capabilities. Finally, a system must accept output from the Network model by HAI, which is obviously impossible for any of them due to specificity of the output format. Thus, it is necessary to develop a system from scratch, though considering existing software as references.

2.2 Technology

Generally, modern visualization software relies on powerful graphics hardware. In fact, graphics hardware plays the leading role in the whole visualization process, dramatically decreasing the load on the main CPU and leaving more room for increasingly complex computational tasks. Nowadays leaders in graphics hardware (NVIDIA [30], ATI [23] and others) offer unprecedented hardware power for regular PC workstations. In addition, they continue developing standards and libraries to access and use the capabilities of their hardware.

The OpenGL [32] graphics system is a software interface to graphics hardware. It allows creation of interactive programs that produce color images of moving three-dimensional objects. With OpenGL, it is possible to control computer graphics technology to produce realistic pictures or pictures that depart from reality in imaginative ways [21].

Microsoft Corporation proposes DirectX [29], a proprietary graphics hardware access interface that is very similar to OpenGL but more naturally integrated with the Windows operating system.

These two interfaces are widely acceptable and supported by the major, if not all, hardware vendors.

Sun Microsystems has also designed platform-independent Java3D, which is an OpenGL-like standard [22]. Although Sun is still working on performance issues, it is another big step towards expansion of availability of graphics and visualization to the public. Java3D provides functionality to present 3-D objects and scenes to a user in a regular browser, therefore globalizing access to local visualization resources.

As hardware evolves, more and more algorithms may be moved to the hardware unloading the main CPU. Exploiting these features is very perspective and performance beneficial, so in the future, the system may incorporate some of such shifts, for example order-independent transparency [31].

The graphics interface is one of the numerous parts that comprises a software product and assists in its development. It represents a lower algorithmic level and must be always supported with a great deal of code written in a high-level programming language. C++ and Java are natural choices for such tasks due to their extreme popularity based on sophisticated language structures, e.g. C++ templates, which allow creation of highly customizable and performance code [18], or exceptional error handling and a wide range of freely accessible support libraries. In addition to simple math routings, there are many sophisticated packages that help to develop user interface, access file system, database, etc. They greatly reduce the time necessary to create similar code with native language tools. Microsoft Foundation Classes (MFC) [14] and MOTIF [33] are examples of such libraries.

CHAPTER III

SYSTEM DESIGN

3.1 Requirements

1.3.1 General Description

The lifetime management of future naval vessels dictates its own requirements in addition to the phenomenological ones. It is necessary that during the ship's exploitation, it is accompanied by all information related to its design, construction, operation, crew training and maintenance. This data must be stored in a digital library or database and carried with the ship. The ability to perform fire modeling is needed throughout the ship life cycle.

Fire modeling is needed to evaluate ship designs and design philosophies in order to quickly arrive at an overall concept to meet required performance goals. As the ship concept is refined, fire modeling is continued to evaluate ship vulnerability and to begin the process of defining ship operations.

During operation and crew training, the requirements imposed upon a fire model change drastically as fire models will be used to aid in damage control and recoverability efforts. In the design phases, the computational time of the various modeling techniques are not critical, but in the operational phases of a ship's lifetime, the model must provide information faster than fire-related events happen on board the ship, while maximally preserving accuracy of prediction.

The collaborative work of HAI, The Naval Research Laboratory (NRL) and a group at MSU made a decision to develop a system with GUI and visualization capabilities. HAI provided a one-zone based network model: it assumes that the modeled environment in each compartment can be represented by one set of physical variables, as opposed to multiple set zone or CFD models. As such, a network model will be capable of modeling an entire ship and its ventilation system. Since the number of variables being solved for is kept to a minimum – one per compartment – a network model also has the potential for the fastest computations. [26].

The system is not intended to be pre- and post-processing software. The ultimate goal is to achieve a level when a model is used on the ship so the methods how a crew will control the system must be carefully considered. The GUI must be relatively simple and straightforward to use – there will be no time for clicking buttons and making out complex schemes and diagrams when a fire is near ship's control room. Moreover, a crew does not need the exact data values, only a general picture.

The next requirement is that crew members do not have to possess any knowledge of fire physics or fire protection engineering. In general, none of the future users of the systems will be experts in fire protection or fire science.

Finally, the system must not be overwhelming. That is, only a minimum of information that is highly relevant or recommended for fire suppression activity must be shown, e.g. if a fire is occurring on deck 1, frame1, there is no need to show deck 4, frame 200.

1.3.2 Simulated Parameters

Fire simulations must produce enough information for making adequate conclusions about environment and ship object states. This information must include the physics and chemistry of the fire, namely temperature, pressure, visibility (smoke) and species concentrations (toxins and oxygen).

In addition, the network must simulate detection systems and determine from available shipboard sensors the current physical status of the ship.

1.3.3 Coding

The design platform is Windows NT/XP. The general system design will be developed with help of Unified Modeling Language (UML) [2]. The graphical user interface will be written with ANSI C++ programming language [18] and Microsoft Foundation Classes (MFC) library [14]. Windows API will provide functionality for multithreading [15]. Visualization algorithms will use OpenGL. Database structure design and integrity checking will exploit SQL and PL/SQL [20].

1.3.4 Use Cases

The developed system is intended to serve as a design tool, a tactical tool, or a training tool depending on the configuration. This dictates what kind of operations must be provided and how they will be used.

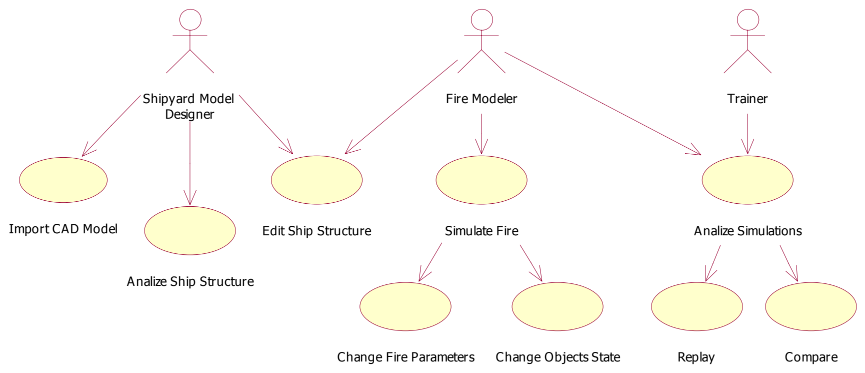


Figure 3: The Use-case diagram.

Firstly, a design tool for shipyard designers must provide a solid feedback on geometry, ducts and other elements, coming from CAD systems. A designer must be able to analyze ship structure and possibly edit it.

Secondly, after a model is ready, the fire modeler must be able to extensively test and validate it. In addition, a modeler must also be able to change ship structure to study the effects, and possibly optimize it for more effective fire suppression.

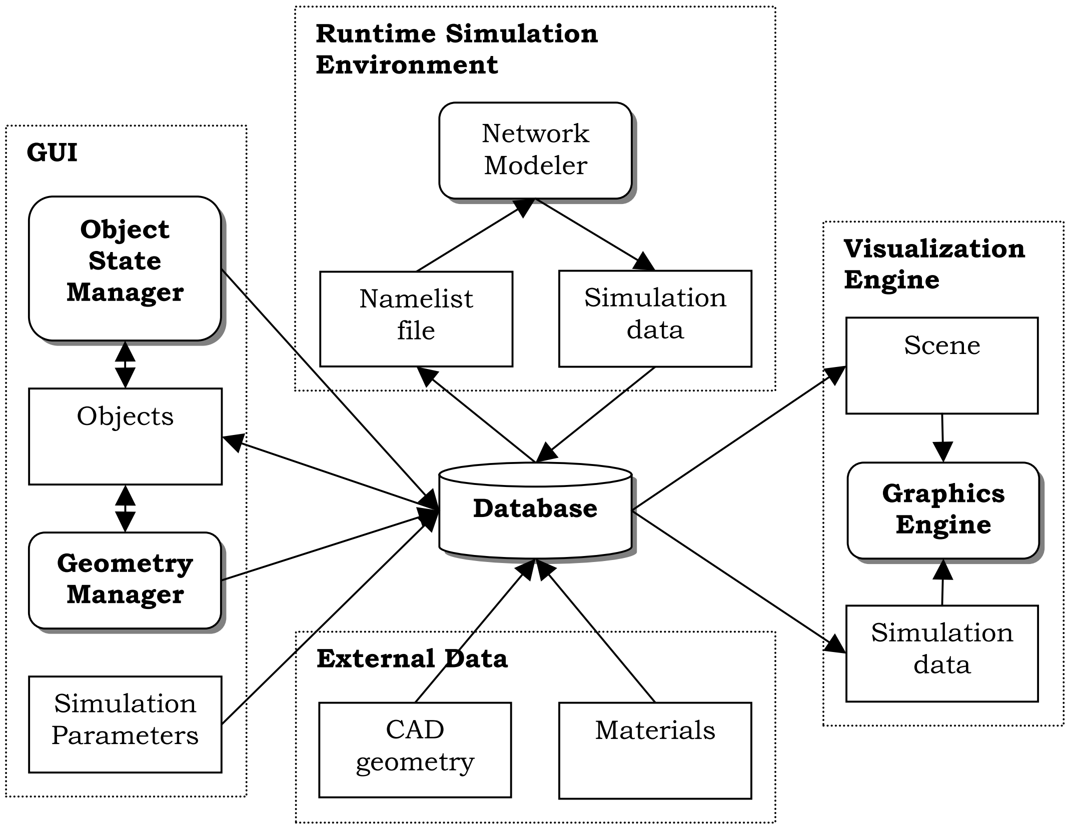


Figure 4: System component diagram.

Finally, the system will be used for training the ship personnel. Thus, a qualified expert must be able to access simulations stored in the database and discuss them with trainees.

1.3.5 System Components

The component diagram below represents a possible system structure. It contains important system elements as well as logical links. It helps gain understanding of the important system parts and collaboration between them (Figure 4).

There are four logical elements comprising system's structure. The first is the database, the storage of all data including geometry, material properties and simulations. The second is the GUI, a mechanism that allows the user to interact with the model (set the states of ship's objects and fire parameters) and run simulations. The third is the Runtime Simulation Environment (a standalone application) that includes generation of the model input file from data stored in the database and provided by the user through the GUI. The fourth part is the Visualization Engine, a machinery designated for displaying a model geometry and generated simulations.

3.2 Class Hierarchies

2.3.1 Class Identification

The object-oriented approach in design is an obvious choice today. It is more difficult to use than the function-based approach, but it produces a solid understanding of core processes that occur in the system and their initiators and participants. The system is represented by a complicated network of objects and their collaborations. A simple but powerful approach of noun extraction was applied to identify them [16].

First of all, the problem should be defined in a concise manner:

A ship is built of compartments. It also includes active elements like doors, hatches, scuttles, a ventilation system with fans and dampers and a firemain system with plugs and valves.

Now additional constrains and more details should be added:

A *ship* is built of *compartments*. A *compartment* is composed of *sides*, each of which belongs to a *wall*. Each *side* is a set of *vertices*. A *wall* is built of two *sides*. It also includes *active elements* like *doors*, *hatches*, *scuttles*, a *ventilation system* with *fans* and *dampers* and a *firemain system* with *plugs* and *valves*. The *ventilation system* is built of *ventilation duct sections*, which are composed of a pair of *ventilation nodes*. A *ventilation node* can be a *simple node*, a *fan* or a *damper*. The f*iremain system* consists of *firemain sections*, which are built of pairs of *firemain nodes*. In addition, a *firemain node* can be a *valve* or a *plug*. The activity of an object suggests its ability to be in multiple *states*. A *duct section* can have the following *states*: fake (virtual object not affecting simulation equations), disabled or enabled. A *door* can be fake, disabled, closed, opened or be a joiner. A *hatch*, *scuttle*, *fan*, *damper*, *valve* or *plug* can be fake, disabled, closed or opened.

The nouns are in italics, and they define candidates for real classes. “Ship” is a general definition of the model, so it should be ignored. Also “state” is not actually a real entity; it is a property of the object, so it is also ignored as being represented as a class.

The denoted general idea about the system allows start of developing possible classes and its hierarchies. All classes can be divided into two groups: scene classes and general classes. Scene classes can be split into two groups: geometry and systems (ventilation and firemain). Geometry is defined by compartments, walls and sides, whereas systems include the ventilation system, which is composed of doors, hatches, scuttles and a duct, and the firemain system.

In following sections, the design of classes of each group is considered separately in details.

2.3.2 Geometry Classes

A compartment represents a volume in space bounded by sides belonging to it. A wall is also defined by sides. Therefore, a side is a main visual element that will define geometric representation of the model.

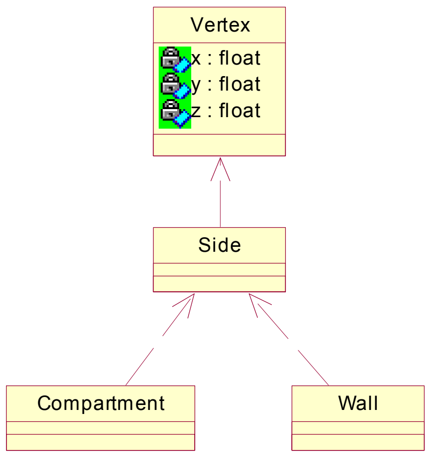


Figure 5: Geometry class associations.

A definition of a side is straightforward. It is a flat (in sense of projection on one of three main planes) polygon represented by a list of vertices, number of which is fixed to four. Similarly, a compartment and a wall are lists of sides.

2.3.3 Ventilation and Firemain Systems

The next observation is that a door, a hatch and a scuttle have the same semantic meaning – they are openings in a wall, floor or ceiling. As any opening does, they all have a position and size (Figure 6).

Generally speaking, a ship may have not only mentioned openings, but potentially any arbitrary located object that can be represented as one and that appears anywhere due to a ship structure modification or outside impact-caused structural damage. The described structure satisfies such cases by adding a new class derived from the *Opening* class.

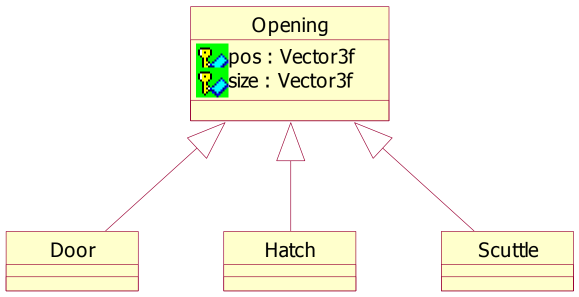


Figure 6: Openings class hierarchy.

The next structural elements are ducts, namely a ventilation duct and a firemain duct. A duct is represented as a network of duct sections. Each section in turn is a pair of nodes, or points in 3-D space. Thus, a network is a collection of interconnected nodes. Connectivity of a given network is represented by duct sections.

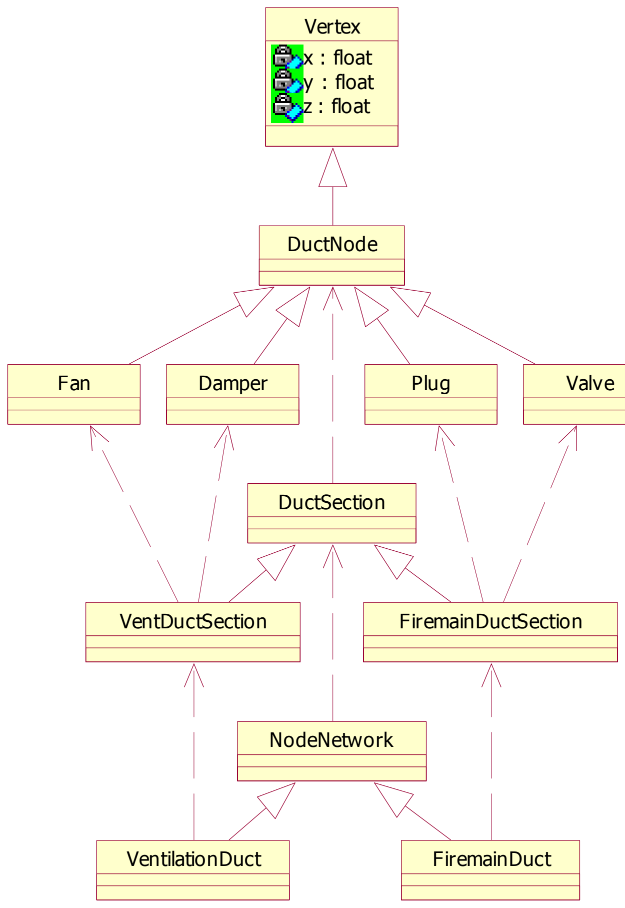


Figure 7: Ducts class hierarchy.

A duct node may carry a meaning that is wider than just a point. It may possess some characteristics or behavioral attributes that may affect a network it belongs to. Nodes of a ventilation duct can be fans or dampers; nodes of a firemain system can be plugs and valves (Figure 7).

Further extension of this class family is also easy. It can be done by deriving from class bases, namely the *DuctNode*, the *DuctSection* and the *NodeNetwork*.

2.3.4 Scene Classes

The previously considered classes are not complete and ready for explicit rendering. They should be turned into classes that may be shown on the screen, i.e. scene classes. A classical approach suggests using an abstract scene class that will represent a base for all other elements and encapsulate all necessary behavioral attributes like ability to draw itself. This is the *SceneObjectBase*.

The *SceneObjectBase* class allows the generalization of objects' representation and unification of the drawing process. Most often each object should have a position and a color as attributes, as well as a drawing routing for calling by the render.

Not all of the ship elements are suited for rendering, i.e. drawing is meaningful only for ship geometry and systems. Also, some objects must be able to interact with a user and accept requests for changing their state. The object state may affect its appearance, physical or behavioral properties. Doors, hatches and scuttles can be examples of such objects. Thus, another hierarchy layer should be introduced. It will define the ability of an object to change its state dynamically.

2.3.5 Main Scene

At this moment, it is possible to proceed with developing a manager of scene objects that will be responsible for maintaining their creation, manipulation and finally release.

The previous section describes a range of scene classes. Among them, at least two groups should be distinguished: static and active objects. Thus, classes that do not have any dynamically updated properties will be children of the general scene base class, whereas for classes with active features (state, for one) a common parent will be created, which will provide its descendants a functionality necessary for maintaining their dynamically updatable properties.

The system should also provide access to instances of the same class so that they can be treated differently from others. The *SceneObjectMngr* class, a manager of scene objects, serves for these needs. It stores instances of the same class as a separate list, thus, always allowing identification and use of them independently. It extensively exploits C++ Standard Template Library [7], which provides unprecedented flexibility, controllability and speed.

The *SceneBase* class is the hierarchy top most class. It manages the whole rendering process and uses the *SceneObjectMngr* for scene objects management. The *SceneBase* provides a set of virtual functions for scene initialization, manipulation and drawing. The functions of that set can be overridden to enhance or change predefined behavior.

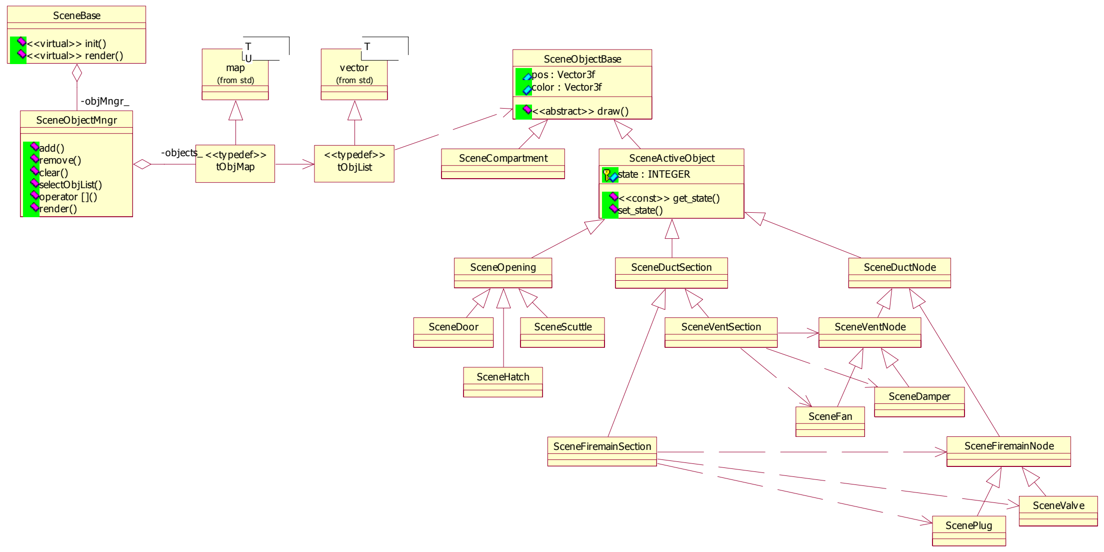


Figure 8: Complete scene class hierarchy.

3.3 Database Structure

3.3.1 Geometry

The *Compartment* represents a rather simple class that does not carry much information so far. Actually, the only additional field besides id is a description field. As mentioned before, each compartment is composed of sides, which in turn is represented as a set of four vertices or points in 3-D space. Each side also belongs to a wall. Generally, a wall consists of two sides and separates two compartments (Figure 9).

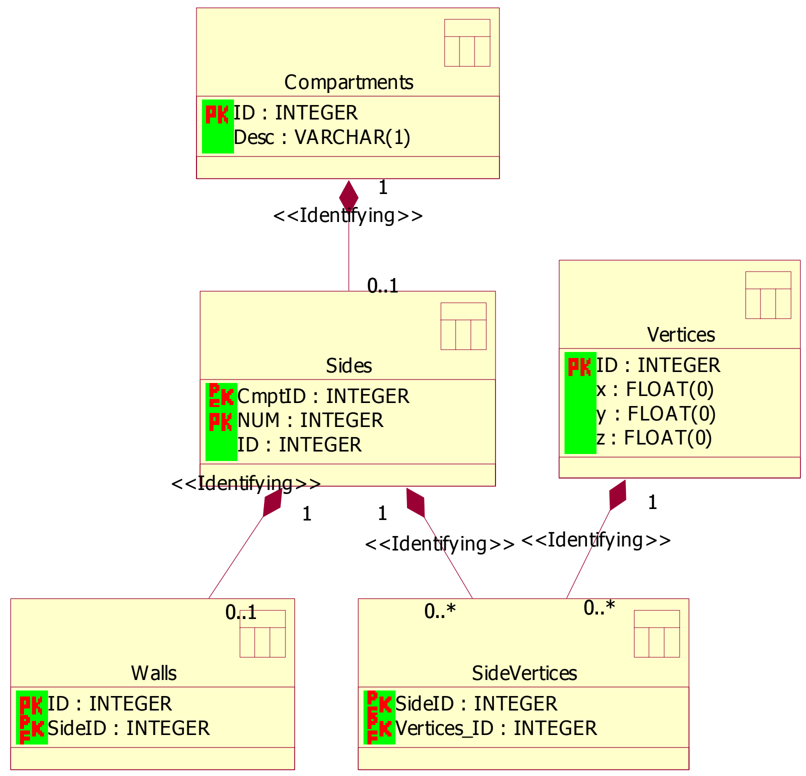


Figure 9: Geometry tables.

From the database standpoint all given counts are not important. Moreover, having the ability to compose, for example, a side from more than four vertices, adds greater flexibility for future system evolution. All following tables are developed to avoid the mentioned rigidity.

3.3.2 Ventilation System

The ventilation system is represented by a ventilation duct and openings. A ventilation duct is a network of ventilation sections, each of which is a pair of ventilation nodes, i.e. points in 3-D space. Theoretically, a section may consist of more than two nodes.

A node can be simple or complex. A complex node is a node that actively participates in the ventilation process. Currently, the only complex nodes are fans and dampers. The following structure reflects the described relationship:

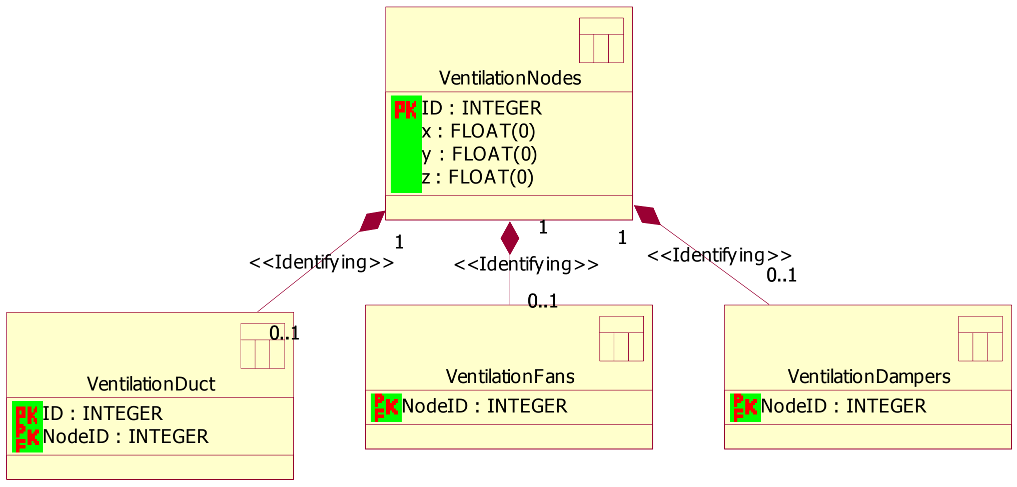


Figure 10: Ventilation duct tables.

Doors, hatches and scuttles have the same physical meaning so it is possible to store the information about them in one table. But for more convenience, three separate tables may be created. Each opening should possess knowledge of what wall it belongs to.

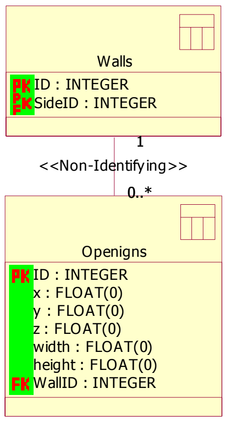


Figure 11: Openings table.

3.3.3 Firemain System

The firemain system is very similar to the ventilation duct. It is also a network of node sections. The current active nodes of a firemain system are plugs and valves. The database table structure looks as follows:

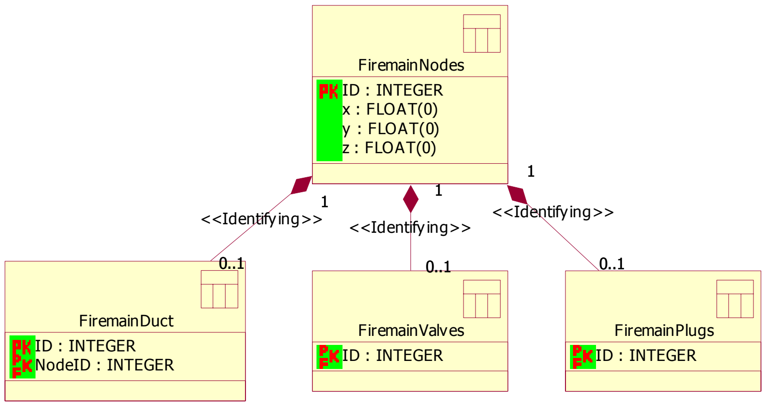


Figure 12: Firemain tables.

3.3.4 Simulation Data

The results of the work of the Network simulator need to be saved for future analysis and replays. The simulator produces a data block that contains the following scalar parameters (not exactly in the same order):

- Compartment temperatures.
- Compartment pressures.
- Compartment O2.
- Compartment CO.
- Compartment soot.
- Compartment heat release.
- Duct node temperatures (not used).
- Duct node pressures (not used).
- Front surface temperatures (not used).
- Back surface temperatures (not used).
- Fire size (not used).

Compartment related data is combined into one table. Similarly, duct node data is placed into another table.

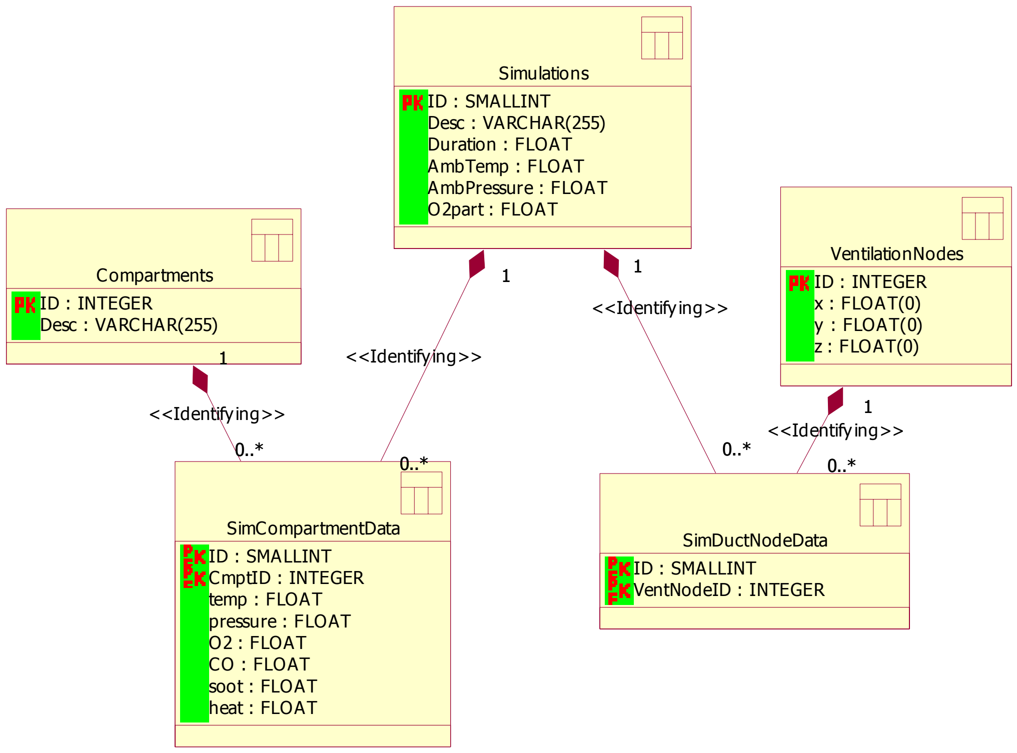


Figure 13: Tables for storing simulation data.

Using these tables, it is very easy to access and study the simulation results in the scope of compartments (all or single), ducts, time or space.

3.4 Database Buffering

4.3.1 Motivation

Scene classes described above are able to render themselves, but they still need to know where they should do it on the screen, i.e. they need to know their coordinates. This information comes solely from the database. Database access is fast but still incomparably slower than access to data stored in the main computer memory. As far as rendering is an extremely demanding process, the best performance of which highly depends on amount of information to render and access information speed, critical information should read, or pre-buffered, from the database into the main memory.

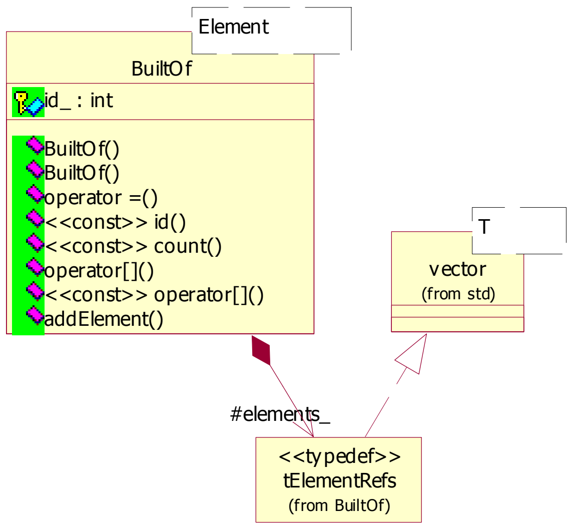


Figure 14: Generic class for representing a complex entity (for example, side is composed of vertices).

The most significant data is ship geometry. Due to complexity, a very quick access to all ducts, e.g. a ventilation duct, must also be provided. The next section presents the data structures for storing and manipulating mentioned types of the data.

4.3.2 Classes and Structures

As referred before, some elements of the geometry are composed of smaller units, e.g. vertices comprise a side, and sides comprise a wall or a compartment. This general approach helps to create a class, customization of which easily allows us to reflect described relationships.

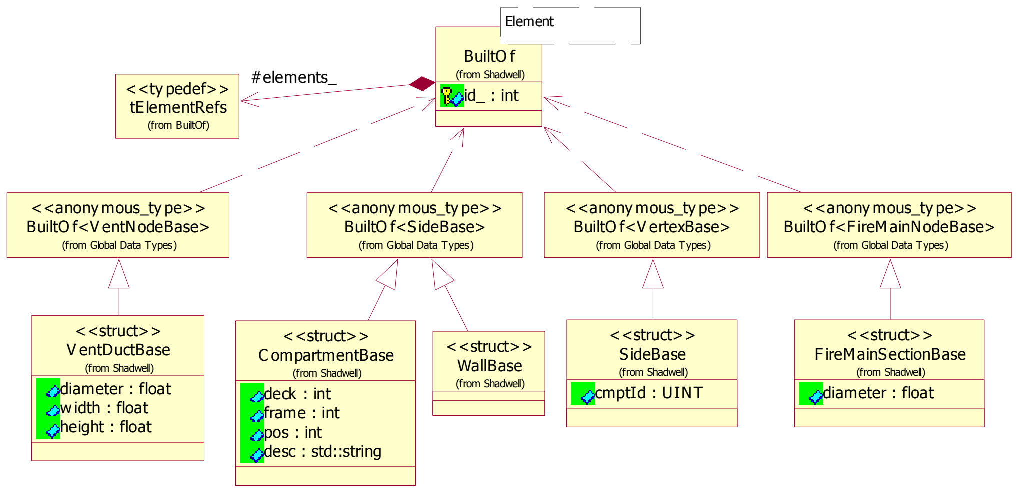


Figure 15: Geometry data storage classes.

The class represents a wrapper around an array of pointers to instances of arbitrary classes. It provides functionality to access interesting elements of the array. Using this generic representation, it is possible to build a data structure or class that will accommodate the information from the database (Figure 14). This class serves as a base class for all complex geometry elements (Figure 15).

A centralized class management increases code accuracy, efficiency and maintainability. The *BodyStructure* class serves as a depository of all geometric data (Figure 16).

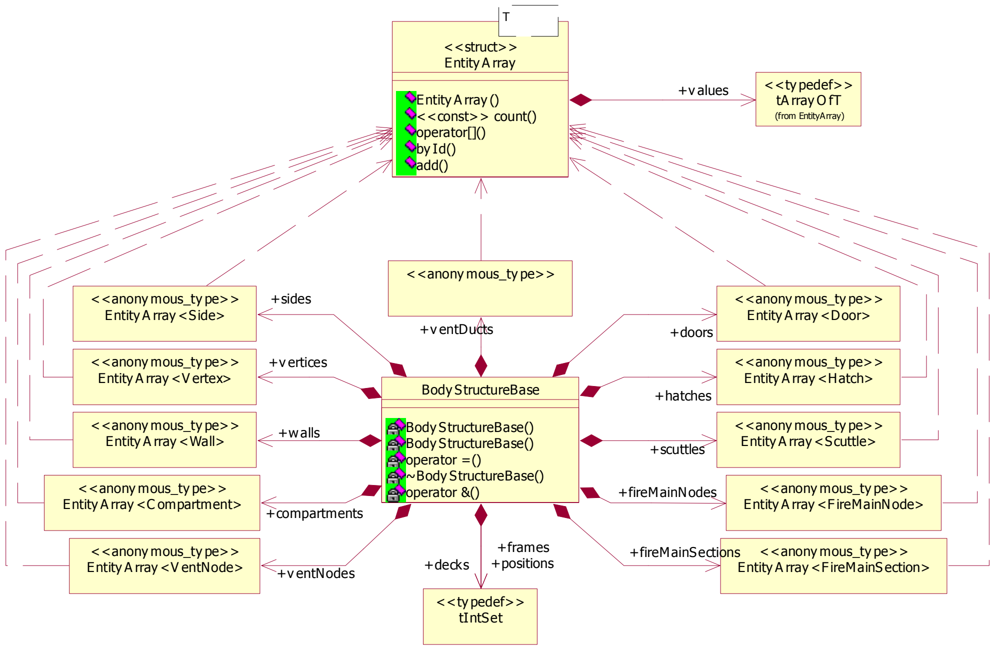


Figure 16: The *BodyStructure* class – a container for all ship data.

The most important detail about the design is that it does not duplicate any data. The simplest unit of the geometry is a vertex. Vertices are read from the database as they are. A set of vertices defines a side, so a side has knowledge of them by creating an array of references to already created and loaded vertices. In turn, a wall or compartment contains references to sides. This is a very flexible and memory efficient scheme that also is extensible and easily evolvable.

The described hierarchy also needs a very sophisticated loader. It can be designed in the manner that it will provide a generic functionality capable of reading different parts of the geometry data with just a few customization details due to the fact that it is built with templates. A source of data is transparent for a loader, i.e. it uses a bridged connection, or interface, to access information (data bridging is described in further sections). After data reading, a loader creates necessary data interconnections by means of references.

4.3.3 Scene Classes Dependence

In the motivation for data buffering, it was mentioned that scene objects must possess information on how to draw themselves. The classes described in the previous section are intended to provide such information. The dependence between them is straight – a scene class is associated with a corresponding data buffering class {Figure 17}.

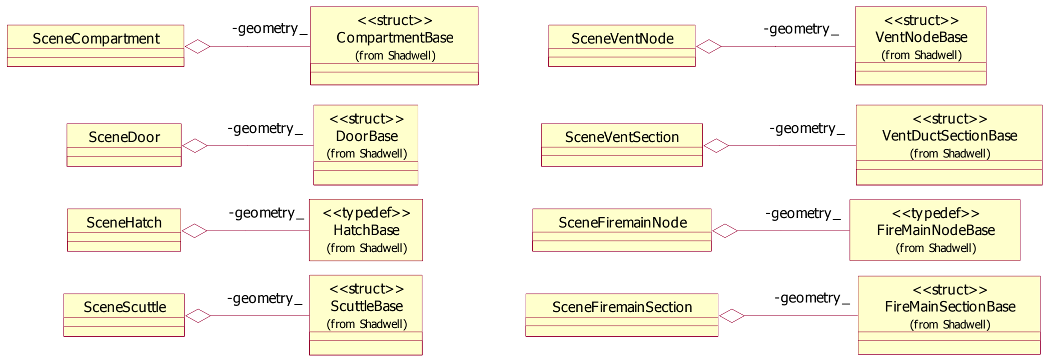


Figure 17: Relationships between scene and data buffering classes.

CHAPTER IV

IMPLEMENTATION DETAILS

4.1 Fire Simulation

1.4.1 The Model Input File

The Network simulator is a standalone application written in FORTRAN 95 that accepts input in the form of a text file a namelist file and produces formatted text output. A namelist file, a standard FORTRAN language feature, comprises lines of formatted text data. Simplifying the FORTRAN standard, definition of the format of the namelist file is as follows:

```
NAMELIST /namelist-group-name/ [attribute=value[, attribute=value…]]
```

Each namelist-group-name defines its own set of attributes. For example, junctions – objects connecting two others (openings and duct sections – in the Network model are defined like this:

```
&JUNC id=5,kloss=2.04,height=-2.66,-2.66,span=1.88,1.88,
area=0.95,location= 3,4,orientation=4,
bidirectional=.TRUE./Door 1' Control-NAV
```

The modeler accepts the following tags:

- EXEC – general simulation parameters.
- FIRE – fire source parameters.
- JUNC – junction parameters.
- CTRL – control element parameters.
- COMP – compartment data.
- SURF – surface data.
- MTRL – compartment walls material data.
- CMPN – component of a material.
- RDCT – ventilation system.
- RNOD – ventilation system nodes.
- RFAN – ventilation fan parameters.
- CURV – an item of tabular data.

Generation of a namelist file is considered in section 4.1.3.1.

1.4.2 Simulation Multithreading

The Network simulator should be run as a separate thread for several reasons. First, the rendering functionality must be available for redrawing a ship model after each time step. Secondly, interaction with the user, which includes pausing, resuming and stopping simulation execution, is still necessary.

Multithreading under the Windows operating system can be achieved in different ways. I use Windows native functions:

- *CreateThread* for a simulation thread properties initialization and its start.
- *CloseHandle* for releasing system resources allocated for a thread.
- *SuspendThread* for pausing or suspending a running simulation thread.
- *ResumeThread* for resuming execution of a suspended simulation thread.
- *TerminateThread* for exiting from or forced termination of a running simulation thread.

A simulation thread can stop normally or forcedly. A normal completion of its execution happens only in case when given simulation time is achieved. In all other cases, that is, preliminary termination by the user or exiting from the program during a simulation, the system imposes a forced thread termination.

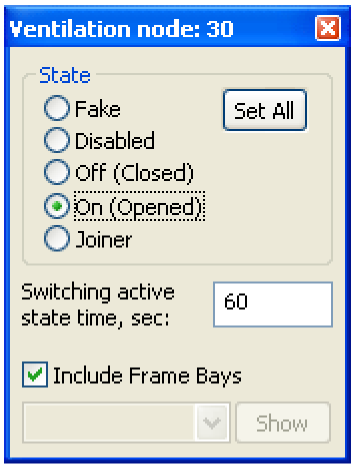
1.4.3 Simulation Modes

There are three modes of fire simulation:

1. Creation of new simulations using the Network simulator.
2. Replay of previously created simulations.
3. Comparison of two previously created simulations.

4.1.3.1 New Simulation

Creation of a new simulation begins with the definition of states of the ship's active objects (doors, fans and etc.), setting simulation (duration of the simulation and ambient environment parameters) and fire propagation parameters (number of fire sources, their strength and etc.) though the GUI.


Ventilation node: 30

State

Fake

Disabled

Off (Closed)

[✓ On (Opened)]

Joiner

Set All

Switching active state time, sec: 60

[✓] Include Frame Bays

Show

Figure 18: The *Object State Edit* dialog window.

Each active object may be in several different states depending on the type. Any object can be *fake* (not related to any physical object included in the simulation area) or *disabled* (existing but invalidated for changing its state). In addition, an opening can be in an *off* (or *closed*) state and an *on* (or *opened*) state. A door also can be in a *joiner* state (ability to vary its area due to construction) (Figure 18).

There are additional features on the dialog window above. One is the ability to set all active objects to the same state by selecting a sought state and clicking on the *Set All* button. Another is the switching time, the time when a selected element changes its state to the opposite, e.g. if a current state is *opened* and switching time is 60 seconds, then the state of the element will be changed to *closed* after the 60th second during a simulation run. The final feature is the ability to include or exclude the *Frame Bays* in the simulation. *Frame Bays* is a submarine specific ventilation system feature. In reality they are model specific, so they may not be present on other ship models. They represent additional ventilation channels between selected compartments or decks. Frame Bays are displayed as vertical flat ventilation sections (Figure 28).

A ship model can have hundreds of active elements. It is impractical to oblige a user to set all of them for each new simulation. To overcome this problem, several default modes may be provided, each of which will define a unique set of states for all active elements. For the currently used ship model, there are three predefined modes: *Recirculation*, *Snorkel* and *Pierside*. By default, the system is set to the most frequently used one.

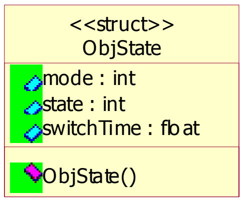
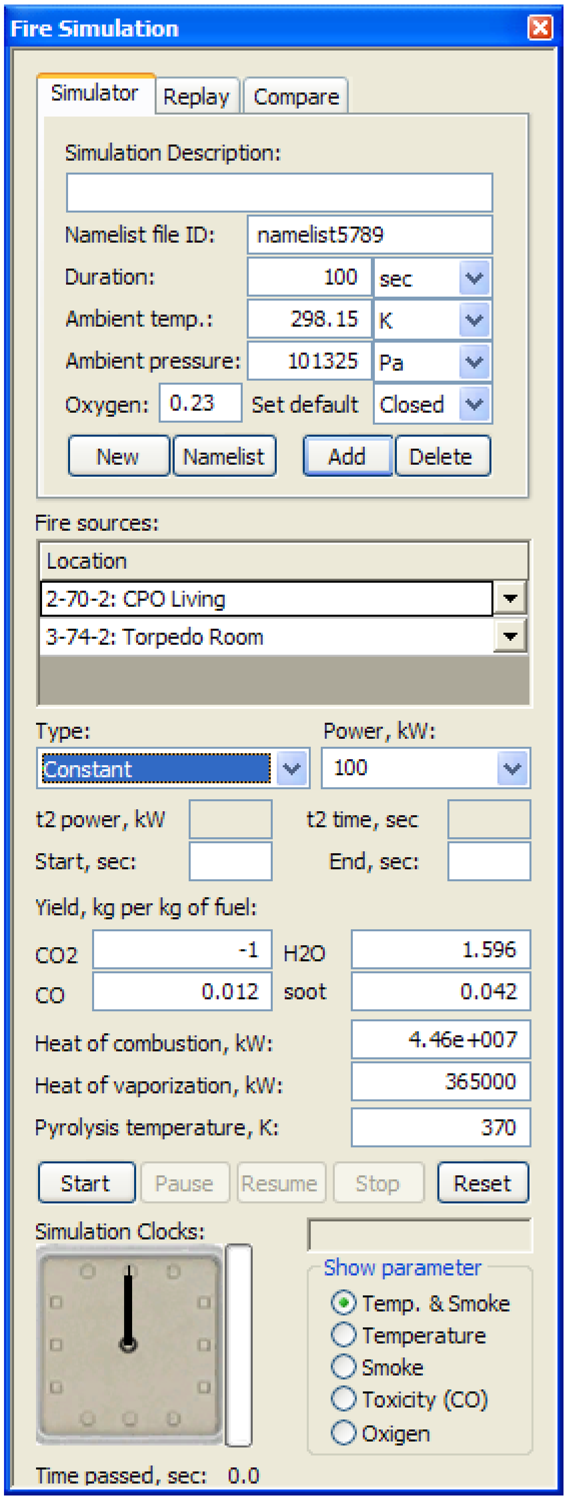


Figure 19: The *ObjState* class.

According to the given description, the implementation of an active object state property concludes in the definition of a class, the UML representation of which is as follows:


Fire Simulation

Simulator | Replay | Compare

Simulation Description:

Namelist file ID: namelist5789
Duration: 100 sec
Ambient temp.: 298.15 K
Ambient pressure: 101325 Pa
Oxygen: 0.23 Set default Closed

New | Namelist | Add | Delete

Fire sources:

Location
2-70-2: CPO Living
3-74-2: Torpedo Room

Type: Constant
Power, kW: 100
t2 power, kW
t2 time, sec
Start, sec:
End, sec:

Yield, kg per kg of fuel:
CO2 -1
H2O 1.596
CO 0.012
soot 0.042

Heat of combustion, kW: 4.46e+007
Heat of vaporization, kW: 365000
Pyrolysis temperature, K: 370

Start | Pause | Resume | Stop | Reset

Simulation Clocks:

Show parameter
[✓ Temp. & Smoke]
Temperature
Smoke
Toxicity (CO)
Oxigen

Time passed, sec: 0.0

Figure 20: The *Fire Simulation* dialog window.

Setting the parameters of a simulation run is the next step. In the top of Figure 20, the user should provide a description of the simulation, necessary for further simulation identification in the list of simulations available for replays, a physical name of the input namelist file for the Network simulator, the duration of the simulation, environment ambient parameters and a species concentration. Each of these data fields has a default value, including a description, which will be set into a name of the input namelist file in case no description is given.

After that, the user should define fire sources. Each fire source has a unique set of parameters that includes fire type (constant, t^2 fire and tabular), power, starting and ending times, fuel parameters (middle part of Figure 20) and others. By default, a fire source is constant in time with a power of 100 Watts. A simulation can have several fire sources, each of which may have different settings.

Assuming validity of all user inputs, an input file for the Network model (1.4.1) is created, in addition using data stored in the database. This process includes processing notation of the ship's geometry, definition of openings and other junctions, wall structure, and behavioral functions for some active elements.

The step occurs when the user clicks on the *Start* button. If it succeeds, the system starts the Network model in the separate thread and begins processing its output step by step. The user may pause or suspend this process, resume execution of a suspended simulation or stop it by clicking on the corresponding button (Figure 20).

The window also offers assistance to the user, providing clocks that show the time passed after the simulation start and a set of controls that allow selection of a parameter to visualize as the simulation is running.

4.1.3.2 Simulation Replay

The same dialog window provides the user with the ability to replay previously run simulations (Figure 21). A user may see the parameters of a selected simulation as well as the states of the ship's active objects, but it is impossible to change them.

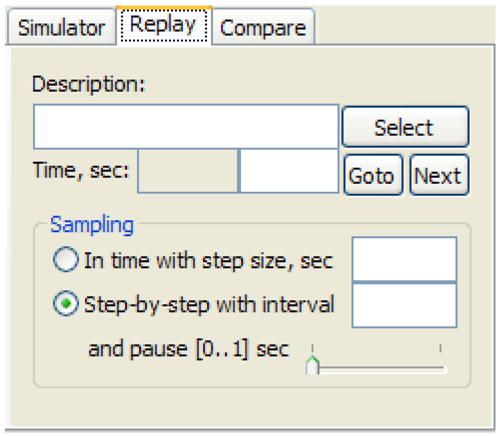
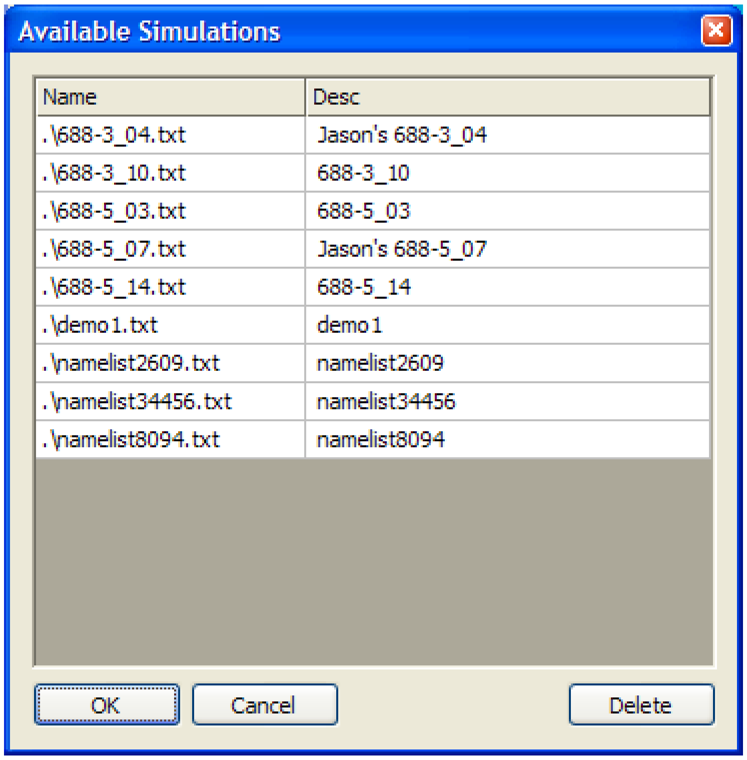


Figure 21: The *Replay* tab of the *Fire Simulation* dialog window.

To replay a simulation, the user must select it from the list of available simulations, which contains names of namelist files and descriptions of simulations. After that a selected namelist file is parsed, and the system sets the ship's objects into the appropriate states. To start the replay, the user must press the *Start* button. The functionality of the rest of the buttons is the same as described before.



Name	Desc
.\688-3_04.txt	Jason's 688-3_04
.\688-3_10.txt	688-3_10
.\688-5_03.txt	688-5_03
.\688-5_07.txt	Jason's 688-5_07
.\688-5_14.txt	688-5_14
.\demo1.txt	demo1
.\namelist2609.txt	namelist2609
.\namelist34456.txt	namelist34456
.\namelist8094.txt	namelist8094

Figure 22: The *Available Simulations* dialog window.

A user must have an ability to jump instantly to a particular time step. Theoretically, this operation can be performed in *O*(1) due to the practically invariant amount of time required to perform the same SQL query to the database. However, it does not hold true in the case of storing data in the text files because they provide only sequential access.

Navigating through the text files may be guided by tags. This technique is sufficient for stepping forward, but it may have significant performance issues while explicitly going backwards. To retrieve the next time step, the system simply finds the next tag identifying a beginning of the step, but while moving backwards, a current position must be reset and all steps preceding a requested must be skipped. Stepping should be used carefully in case of rather big ship models and long simulations due to the significant size of time steps generated by the Network model. The big size of steps causes a considerable performance overhead during step skipping.

The user may sample through the data using sampling in space or step-by-step sampling. During sampled-in-space replays, the system reads each time step and skips it if its id is not divisible by the size of the step (Figure 23).

The output is not uniform in time. That is, frequency of steps is higher, or the size of a time step is less or varies during particularly important changes of environment that have a greater effect on subsequent fire distribution. Thus, plain step-by-step execution will fail for a study of the results in real time because it will run slower when density of steps per time unit is higher. To achieve true real time replays, or more generally arbitrary replay speed constant in time, is what sampling in time was designed for.

Additional controlling functionality is a pause between steps. By default, the system visualizes the data with maximum possible speed, reading data step-by-step and instantly showing it on the screen. To better understand the process of fire, one may decrease this speed by introducing a short delay after each step. The system allows use of a delay in the interval from 0 to 1 second inclusive.

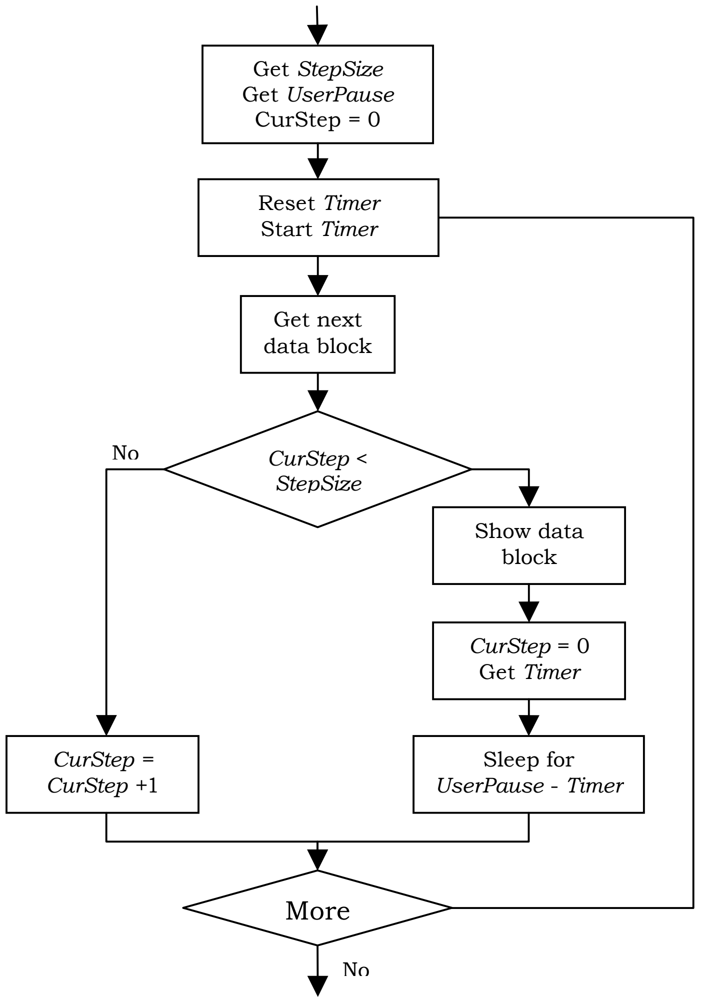


Figure 23: Sampling in space algorithm.

The sampling in time algorithm has a catch, which is pausing between steps. It is incorrect to only wait for a selected amount of time after each step because time required for reading and showing each data block is not zero. This pause must be included into the overall delay as its fraction. In fact, reading time may comprise a significant part of a user defined delay.

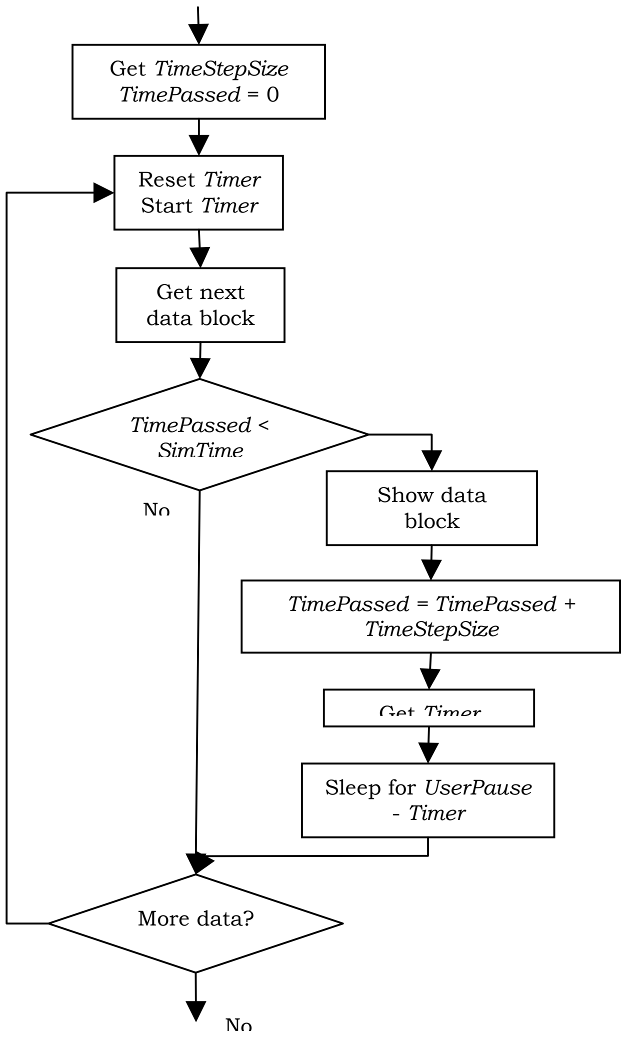


Figure 24: Sampling in time algorithm.

To achieve the best match of a real delay with a requested, time measurements should be performed at the right moments, namely before reading a data block when the timer is started and after displaying it is stopped (Figure 24).

4.1.3.3 Comparison of Two Simulations

Comparison of two simulations represents an extension of the replay mode, only in this case two data sources and two ship models are needed. The *Compare* tab looks very much like the *Replay* tab. Additional overhead is processing and showing two data inputs instead of one, though in the same manner.

The functionality of the *Select* buttons is the same – they offer to select one of the existing simulations. To preview fire source parameters, the user must select an edit box with a name of an interesting simulation. Sampling in time and a variable pause between each step are available; step sampling will be added in the future.

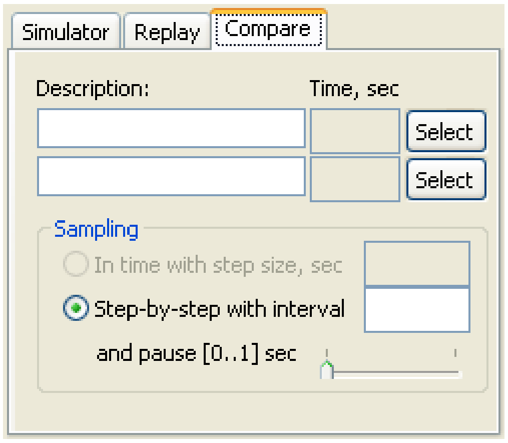


Figure 25: The *Compare* tab of the *Fire Simulation* dialog window.

4.2 Visualization

2.4.1 Geometry

4.2.1.1 Compartments

The compartment is the most significant visualization object. It defines what the whole scene looks like, and making compartments visually attractive is the important issue. Solving this problem will greatly contribute to the overall visualization success.

Each compartment is a set of flat planes or patches. Physically every patch may be different due to various materials that may be used when it is built. For now, the differences are not distinguishable on the screen.

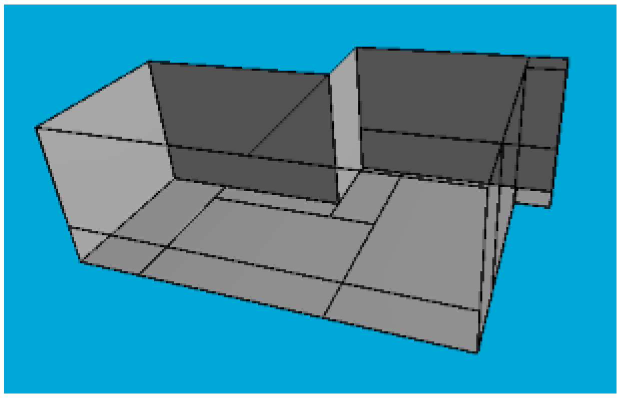


Figure 26: Compartment interior with enabled polygon offset.

The compartment walls closest to a view point should be removed revealing the compartment interior. The OpenGL culling mechanism delivers this functionality. By convention, polygons whose vertices appear in counterclockwise order on the screen are called front-facing. The surface of any reasonable solid can be constructed from polygons of consistent orientation [20]. Drawing the compartment walls so that each patch's face is inside of related compartment, and setting OpenGL to remove back-facing polygons will produce the desired effect of looking inside (Figure 26).

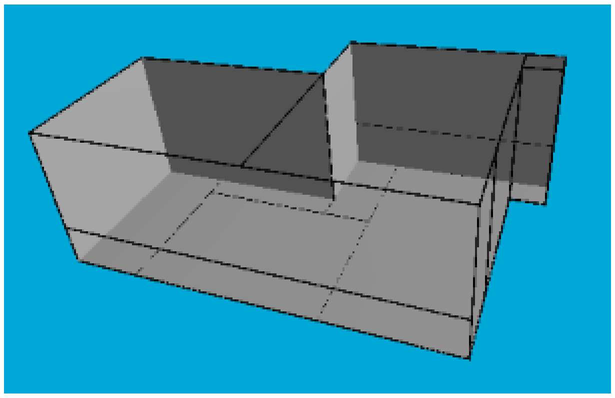


Figure 27: Polygon offset is disabled.

Figure 26 also shows a compartment wireframe defined by wall patches. A wireframe is usually produced by setting a polygon drawing style to *GL_LINES* as an argument for *glPolygonMode* function call. In the considered case the situation is different since both compartments walls and patch contours should be shown simultaneously. Sequential rendering of the compartment walls and then the patch contours does not produce quality results because the edges of polygons in both cases coincide, thus OpenGL cannot perform adequate depth resolution (Figure 27).

Using *glPolygonOffset* provides a solution. It is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges. *glPolygonOffset* sets the scale and units OpenGL uses to calculate depth values. When it is enabled, each fragment's depth value will be offset after it is interpolated from the depth values of the appropriate vertices. The value of the offset is:

$$offset = factor \cdot \Delta z + r \cdot units \,,$$

where Δz is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer [20]. The results of applying *glPolygonOffset*(1.0, 1.0) are clearly seen on Figure 26.

4.2.1.2 Openings

Currently, openings are represented by doors, hatches and scuttles. A door and a hatch are similar – a rectangular object with a predefined width and height. A scuttle is a round object with a predefined diameter. Described physical characteristics define the way each object is shown.

Each opening has a position and physical dimension, which are linearly mapped to pixel size. In addition, each opening possesses knowledge of what wall it belongs to. This information is used to obtain a normal vector of a given wall and detect orientation of a given opening.

The algorithm of rendering of an opening consists of the following steps:

- Calculate orientation (done once).
- Translate to a given position using *glTranslate*.
- Draw geometry.

In some cases spatial positions of the hatch and scuttle match, i.e. they have the same coordinates in 3-D, as well as orientation. Again there is a problem of depth resolution. It can be solved in two ways: drawing objects with different thickness (implemented now) or using *glPolygonOffset* as described in the previous section.

4.2.1.3 Ducts

Generally, a duct is a network of connected tubes, probably of different characteristics. Each tube is represented by a pair of nodes plus some parameters like physical dimension. A duct can be shown as a graph, edges of which are lines in the case of simplified visualization, or like in most cases, cylinders. Generally, a duct can also be interconnected objects with arbitrary shaped cross-section, but then how to define them is a problem that should be solved.

Representing a duct section as a line is simple, and that was a method used in the very beginning. Obviously, it is not visually attractive. More importantly, it does not provide a good understanding of a duct structure due to a lack of correct physical depicturing. This leads to user inability to predict the simulation results. The better way is to have close to real shaped objects, and using cylinders as building blocks for duct sections satisfies this demand.

GLE is a library package of C functions that draw extruded surfaces, including surfaces of revolution, sweeps, tubes, polycones, polycylinders and helicoids. Generically, the extruded surface is specified with a 2D polyline that is extruded along a 3D path. A local coordinate system allows for additional flexibility in the primitives drawn. Extrusions may be texture mapped in a variety of ways. The GLE library generates 3D triangle coordinates, lighting normal vectors and texture coordinates as output. GLE uses the OpenGL API's to perform the actual rendering [20].

At this point, such powerful elements of the GLE library like polyobjects, namely polycylinders, which are intended to provide the capability to draw a cylinder network with different section diameters. The currently used approach has advantages and drawbacks. First, it greatly simplifies dynamic manipulation with each duct section – hiding, showing, changing section's state, etc. – since they are represented as independent class instances. On the other hand, additional objects like spheres have to be used as section joiners to avoid discontinuity.

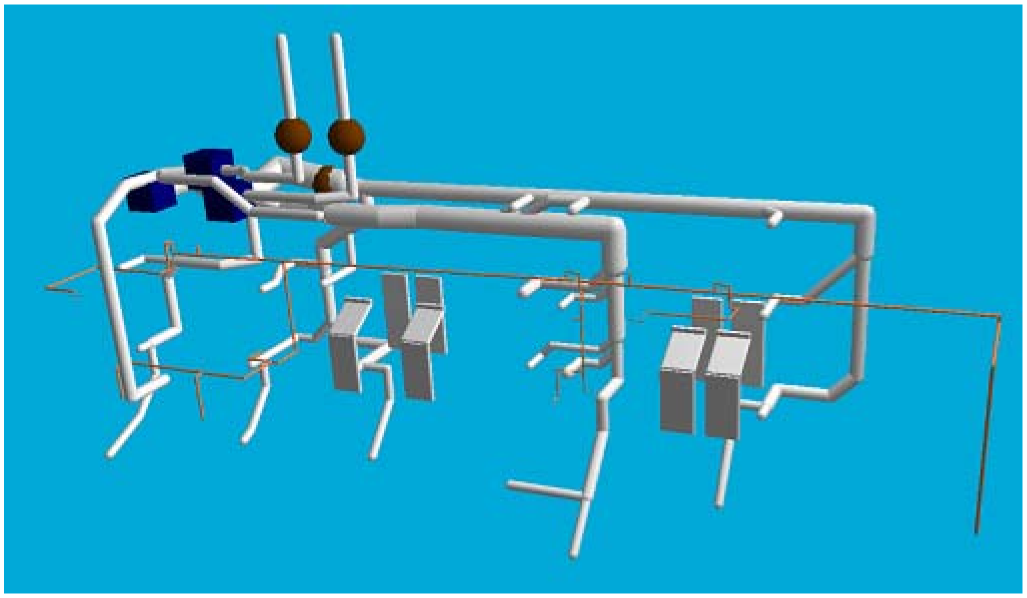


Figure 28: Ventilation duct with fans (blue) and dampers (brown).

The next part of the duct is the nodes. As mentioned before, each node of the duct can be simple or complex. A simple node is a point in 3-D space. A complex node can represent an active element of the duct, e.g. a pump or damper. In the former case, the node is shown as a sphere and in the latter as a custom drawn object.

The overhead of using cylinders and spheres is obvious – each is a set of many triangles (e.g. in average each sphere is 8x8x2=64 triangles), and having rather sophisticated duct networks may have a significant impact on the overall system performance. Therefore, simplification of representation of ducts is an issue that should be considered in the future.

2.4.2 Parameters Representation

The currently used Network modeler provides one-zone representation of output data. That is, one value for each parameter per bounded volume, e.g. compartment. This way of data representation does not provide enough information for creating a quality value gradient in the scope of that volume. For example, if there were more than a single value for the smoke then it would be possible to visualize it as a non-homogeneous instance inside of compartments.

Nevertheless we can still get a picture that will decently reflect the processes taking place during the fire by using color maps, which is one of the best ways to represent physical values changing in time or space. Moreover, such data granularity is satisfying for real-time ship control and making appropriate decisions in case of emergency.

4.2.2.1 Options Dialog Tab: Species Color Mapping

The simulator produces a data block that contains several scalar output values – temperature, density of smoke (soot) or visibility, concentration of oxygen and, finally, concentration of toxic materials (e.g. CO).

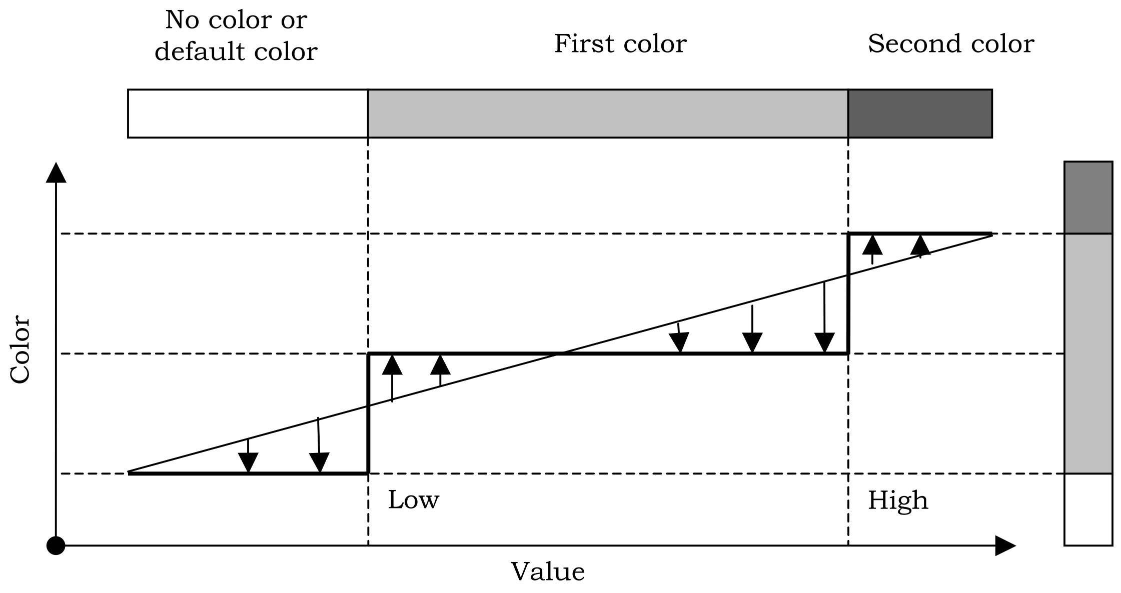


Figure 29: Color mapping with two critical levels and constant gradient.

A good representation of such a type of data is a color. According to studies in cognitive science color saturation should be used to represent a magnitude of scalars [12]. Indeed, changing from light gray to dark gray indicates that a displayed parameter either gained or lost in its magnitude, whereas changing from yellow to red supposes qualitative parameter transitions. The exception can be a desire to show some critical levels. For such cases, dramatic change in color hue vividly notifies the user about passing some important points (Figure 29).

The scalar parameter, e.g. temperature, has several ranges:

- Normal value range.
- Value range safe for protected staff.
- Hazardous value range, when any human presence is life threatening.

Thus, there should be at least three colors used, and the algorithm is simple:

- Get value.
- Get range it hits.
- Select corresponding color.
- Draw related object.

This technique is not sufficient. A person who makes simulations or who controls a ship in real-time also wants to know when a value is close to critical points in order to be prepared to take appropriate actions (for example, to give a command to put on protective suits). Therefore, the selection function should be modified so that in the end of each range (except for the last one) there will be a region showing a transition from a current value range to the next.

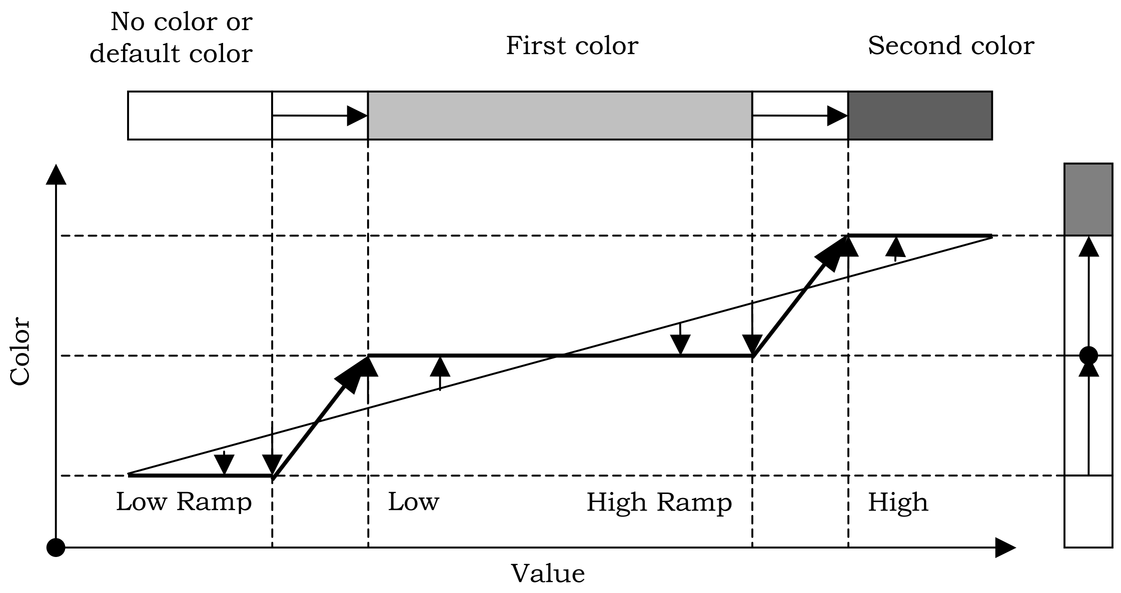


Figure 30: Color mapping with two critical levels and two gradients in the end of each critical value range.

Thus, additional two values must be given, namely, the beginnings of transition regions, which will define when a color must start representing a mixture of adjacent ones. They are called ramp values since they represent ascending parts of the curve (
Figure 30).

The algorithm for color selection gets more complicated – the problem of a correct changing of color channels has to be considered. The logic behind a transition from one color to another is a gradual changing of proper channels. For example, yellow is an RGB triplet with values (1, 1, 0); red is (1, 0, 0). Thus, reducing the green channel from 1 to 0 will produce a desired color set. To provide the ability to set up custom color map is the next step for future work

The currently existing color selection algorithm uses three color channels, known as an RGB triplet, and the following transitions between them:

$$C_0 \to C_1 \to C_2$$

or:

$$\begin{aligned} C_0^R &\to C_1^R \to C_2^R \\ C_0^G &\to C_1^G \to C_2^G \\ C_0^B &\to C_1^B \to C_2^B \end{aligned}$$

The user provides a pair of colors to use by a scalar value of the given parameter. The first step is normalization in scope of a given ramp region, e.g. for a lower ramp region it is:

$$\overline{v} = \frac{v - LowRamp}{Low - LowRamp}$$

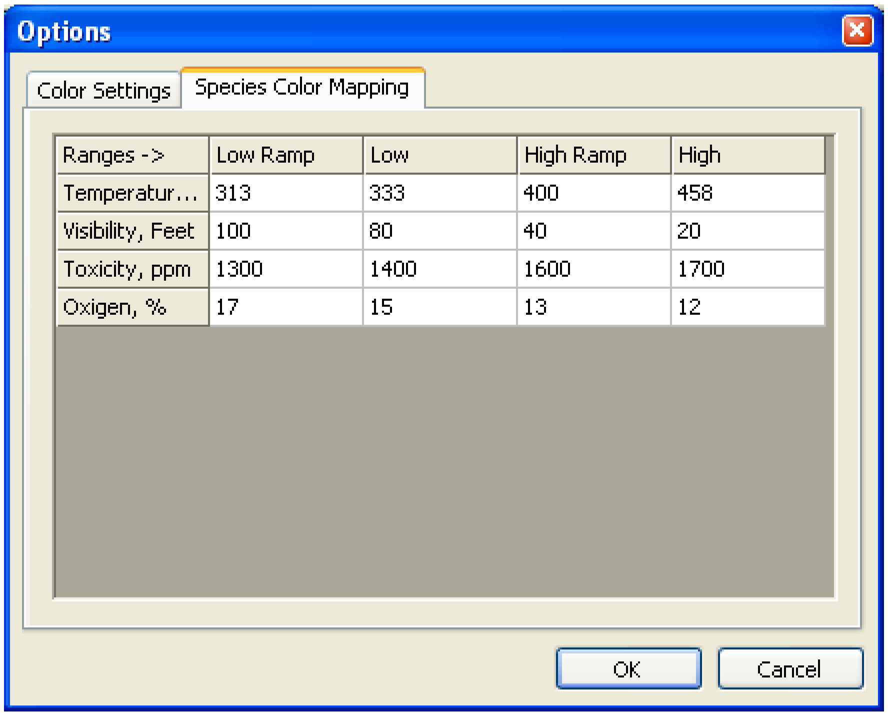
or if *k*-th value range is represented as a pair of numbers:

$$\overline{v} = \frac{v - R_0^k}{R_1^k - R_0^k}$$

Next, this value is appled to a color channel variation:

$$C^i = C_0^i + \overline{v} \cdot \left(C_{k+1}^i - C_k^i \right)$$

Notice that a channel value can as easily increase as decrease depending on the sign of a channel variation, defined as a difference of corresponding adjacent RGB channel pairs.



Ranges ->	Low Ramp	Low	High Ramp	High
Temperatur...	313	333	400	458
Visibility, Feet	100	80	40	20
Toxicity, ppm	1300	1400	1600	1700
Oxigen, %	17	15	13	12

Figure 31: The *Options* dialog window.

The biggest advantage of the described method of color selection is a relative flexibility in changing a number of critical points, or, in other words, transition regions. Indeed, the only thing to be done is to define these regions and provide a proper range identification mechanism to be able correctly normalize a parameter value in scope of that range.

There are three scalar value parameters simulated by the Network model – temperature, oxygen and toxicity (CO). Critical levels for each of them were recommended by Hughes Associates, Inc., but the user also has ability to change them through the *Species Color Mapping* tab in the *Options* dialog window (Figure 31).

4.2.2.2 Legend Dialog

Legend dialog is a helper window, which contains thresholds for the currently selected parameter. It modifies color mapping and text labels, depending on threshold values (Figure 32).

Implementation of the dialog above uses the OpenGL ability to interpolate between two colors, i.e. it is only necessary to set the colors of four points to achieve presented gradients between 313K and 333K temperature values, two for each horizontal line, and OpenGL takes care to create a smooth transition from grey to yellow. The same logic applies to levels of transparency, with the exception that in this case RGB channels are constant, and only the alpha channel varies. The ends of the bar are open to stress that everything above or below given extremes is not important.

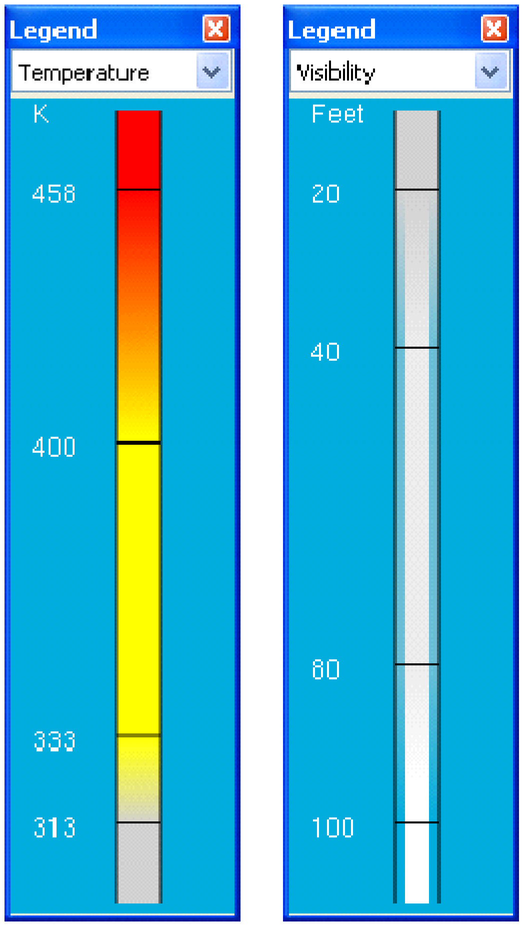


Figure 32: The *Legend* dialog window: temperature color map and critical levels on the left; smoke gradient and critical levels on the right.

The main rendering algorithm for this scene consists of several parts executed in the following order:

- Render background object (for visibility only).
- Render color map and labels.
- Render boundaries.

The background object is used to reproduce the degrading of visibility with an increasing amount of smoke, i.e. alpha channel value. A white rectangle is intended to reproduce such objects of the main scene as a ventilation duct section or an open door.

The color map is essentially a set of rectangles stacked on top of each other. The pattern of their colors reflects the threshold curve described above, namely, that every odd rectangle has a constant color and every even has a transition from one color to another. Despite this fact, each of them can be rendered uniformly, only adjusting color parameters on their edges accordingly.

Another important characteristic of the legend is a realistic representation of a distribution of displayed simulation parameter values. Indeed, only a value range, in which a current measurement falls, represents the user interest, whereas the precise value is unimportant. Consequently, division of the bar into equal parts is not enough. The solution is to apply value ranges inside of the area of the bar defined by two extremes. Such an approach offers to the user a good understanding of the system state and appropriate actions that should be taken.

The example of rendering of the temperature color map is considered below. First of all, we have a range of temperatures from 313K to 458K, which gives 145K difference. Second, a size of the working region of the legend bar is, say, 300 pixels. Thus, each pixel is Δ = 0.48(3) K. The procedure of getting the Y value is as follows:

- Get a temperature value.
- Subtract a lower temperature extreme (313K).
- Multiply it by Δ.
- Add Y_0 - a pixel position of the lower extreme.

2.4.3 Temperature, Toxicity and Oxygen

These species are represented with color maps described in the previous section with color applied to compartments' walls. The user gets a full picture of species propagation and predicts its consequent development by analyzing current environment conditions and settings. It is possible to switch currently visualized species with controls in the bottom of the simulation dialog (Figure 20).

2.4.4 Smoke

Finally, the last of the simulated parameters is visibility or smoke. The smoke is perceived as a loss of clearness of details of objects that it covers. There can be different ways of achieving this effect.

First, using particle systems or volumetric smoke can produce the most realistic smoke. Even though results are very persuasive, the degree of rendering complexity is very high. Each particle is represented as an individual object, so for very dense smoke the number of particles must be rather high. Taking into account additional overhead on physics, smoke takes significant processing time. As long as the project's destination platform is a standalone PC or laptop, this fact starts playing an important role.

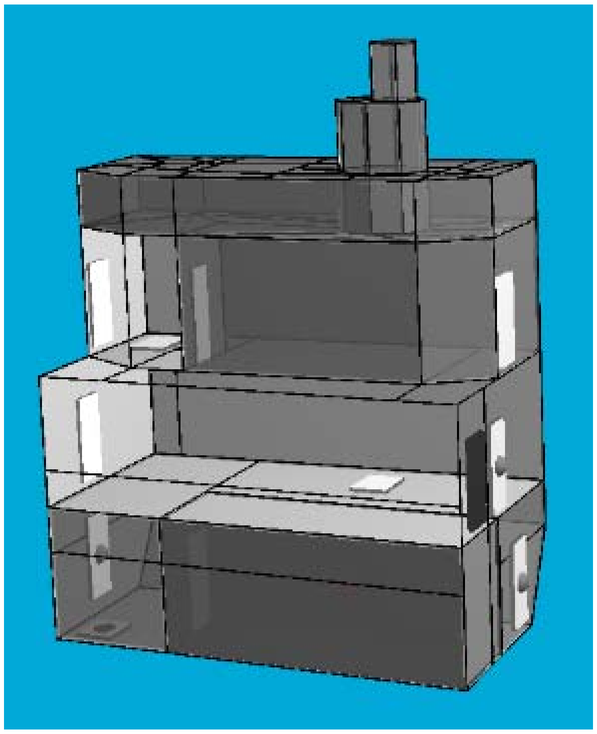


Figure 33: Smoke visualization: the top compartments and the bottom left are partially smoked; the compartment in the middle is free of smoke; a bottom front compartment has high concentration of smoke than any.

The second approach that can be used is imposing another semi-transparent object in front of obstructed model details. In this case, the controlling amount of smoke degrades to manipulating with the color's alpha channel; there is less smoke when an alpha value is lower, and vice versa. There is a good reason for using this method – a data block produced by the simulator is very sparse, that is, it has just a single data value for each compartment. Such conditions prevent a quality smoke analysis inside each compartment, so making complex smoke representations with particle system is hardly possible and even redundant. Nevertheless, volumetric smoke is considered for future work.

The compartment is represented as a set of sides, or quadrangles. Thus, an effect of smoky room can achieved by drawing the same compartment over again with a side color different from the original in its alpha channel value; an alpha blending will produce desired results. The class representing the compartment interior is called the *SceneCompartmentInterior*.

To render a transparent object properly into a scene requires sorting. First, opaque objects are rendered, and then the transparent objects are blended on top of them in back-to-front order. Blending in arbitrary order can produce serious artifacts, because the blending equation is order dependent [12]. By virtue of the 3-D nature of interior objects, two steps sorting is necessary: compartment side sorting and compartment sorting.

2.4.5 Multiple Species Visualization

There is a great example from history that shows multidimensional scalar values visualization. It is Charles Minard French engineer's diagram, which shows the terrible fate of Napoleon's army in Russia [10].

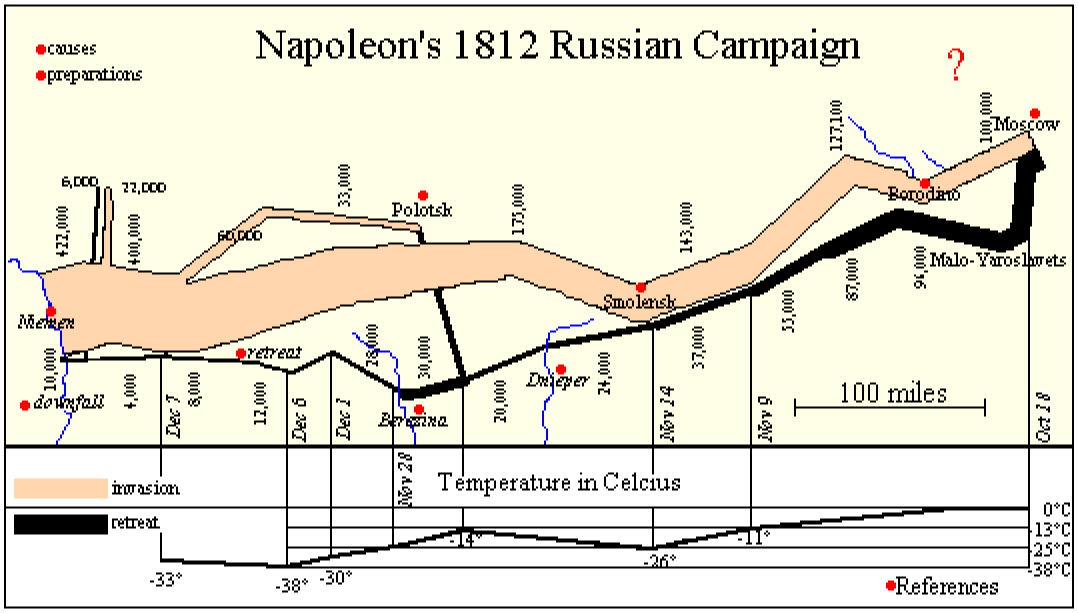
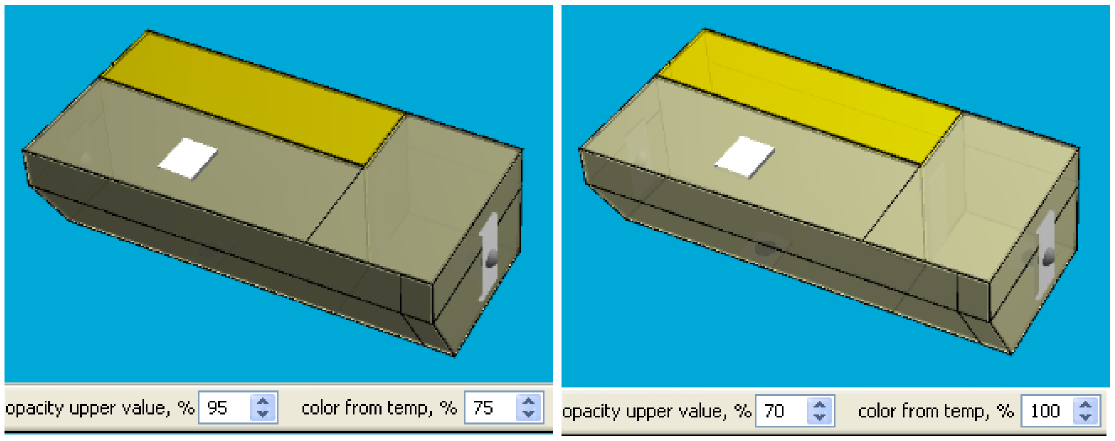


Figure 34: Charles Minard's multidimensional diagram.

Six variables are plotted: the size of the army, its location on a two-dimensional surface, direction of the army's movement and temperature on various dates during the retreat from Moscow. It may be the best statistical graphic ever drawn [15].

To display such an amount of information in a very easy way to perceive and understand is the most important goal of any visualization. Unfortunately, in our case it is hardy possible to show more than two variable without making things confusing. Representing parameters with color does not leave much space for variation. Good training is required if two parameters are shown as a mixture of two colors because it looks absolutely different from the originals.



opacity upper value, % 95 color from temp, % 75

opacity upper value, % 70 color from temp, % 100

Figure 35: Different levels of smoke transparency and aggregation of compartment wall color.

Two variables are still possible to have at once – any parameter with scalar scale and smoke. Smoke is represented as an object that repeats the geometry of the compartment but with transparent walls, the level of transparency of which is defined by smoke concentration. There can be two directions of approaching the problem:

- Show scalar value as colored compartment walls and smoke as an object with transparent gray walls.
- Show scalar value as colored compartment walls and smoke as an object with transparent walls the color of which is borrowed from a scalar value.

Experiments showed that the first approach does not work well because with a high concentration of smoke, the information about a scalar value is lost due to inability to observe compartment walls through the almost solid smoke object's walls. On the other hand, too much color may stress a scalar value parameter, whereas both variables are important in making a correct decision. Two cutoffs help to define the maximum level of wall opacity and the percentage of wall color effect on the smoke (Figure 35).

4.3 Auxiliary Classes

3.4.1 Command Line Parameters Manager

Most programs accept additional parameters that come from a command line. It is often necessary to have access to this information in many places of the code. Generally, the C++ compiler allows a programmer to obtain a list of user-given parameters, which are fed through the main function as its parameters. As a result, a user has a pointer to a variable size array of null-terminated strings and a number of elements in this array. Just having this information may be inconvenient – it is not necessary that every parameter is a string, or, in other words, an array contains homogeneous data. Moreover, it is often the case that a parameter may be preceded by a symbol like a hyphen or slash. For such cases, manipulating these parameters becomes extremely complicated and error-prone.

The *CmdParams* class is designed for handling described problems. Its main purpose is to store user-given command-line parameters and yield their values according to a requested type.

An important characteristic of the considered class is its logical singularity in the scope of the application. Indeed, why should one need more than one instance of the *CmdParams* class if, once given, parameters are never change? Therefore, *CmdParams* should be a singleton.

The idea that lies behind a singleton is relatively simple, but implementation issues are rather complicated. The very first attempts to create such a class were made in 1995 by the famous Gang of Four. In their book, they described the *Singleton* design pattern as a way to "ensure a class only has one instance, and provide a global point of access to it" [4].

A singleton is an improved global variable. The improvement that singleton brings is that it is impossible to create a secondary object of the singleton's type. Thus, the *Singleton* pattern should be used to model types that conceptually have a unique instance in the application. Being able to instantiate these types more than once is unnatural at best and often dangerous [1].

The main principle that lies behind the *Singleton* pattern is a use of static class members. Several things must be taken into account while developing a singleton. First of all, constructors must be private to ensure that the user cannot create any instances of a singleton. This constrain enforces its uniqueness at compile time. Following the same logic, all auto-generated class members, namely, a copy constructor, an equal operator and destructor, must be made private (Figure 36).

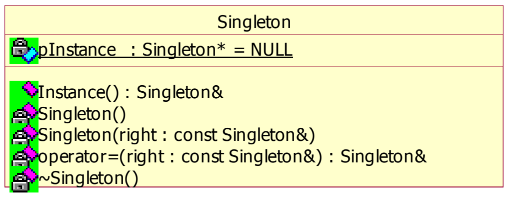

Singleton
pInstance : Singleton* = NULL
Instance() : Singleton&
Singleton()
Singleton(right : const Singleton&)
operator=(right : const Singleton&) : Singleton&
~Singleton()

Figure 36: The Singleton class prototype.

The system's code uses a very solid and sophisticated implementation of the *Singleton* pattern offered by Loki library, which also includes a set of other templates. Loki extends the idea and provides a holder for singleton classes that is very flexible and easy to use.

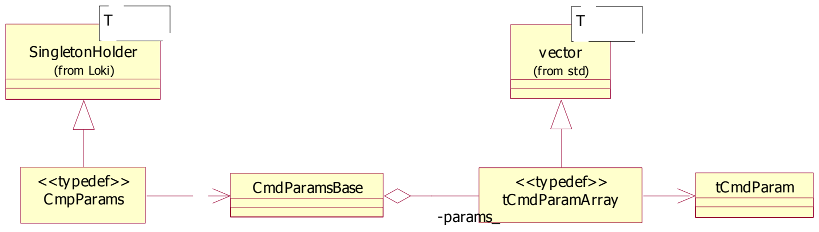


Figure 37: The *CmdParams* class.

3.4.2 Data Bridging

Currently, the model data is represented in two ways. Initially it was a set of text files of a particular structure, and during the time of the project evolution, it transformed into SQL database format, a more natural and highly acceptable way of data managing. Nevertheless, the former data format did not lose its attractiveness – its beauty is in its simplicity. Having text files as a mechanism for data storing allows trouble-free application distribution. Indeed, native file processing routings of C++ help to avoid purchasing and installation of an SQL database server and communication interface like ODBC. On the other hand, text files are good for a relatively small database size, and what is more important, it delegates all data integrity controlling functionality solely to the application. Moreover, text files are static information, and they may require significantly more processing time in case of dynamic data accumulation, happening during simulations. In addition, storing historic data is extremely complicated and error-prone.

Both ways of data acquisition and submittion for further saving and reuse are developed. Logically, the application should not know what kind of data source – text files or a database – it uses. Concealing this knowledge behind an interface is a widely used technique. The *IDataBridge* interface provides desired functionality.

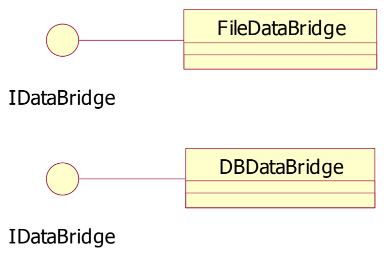


Figure 38: Data interfaces.

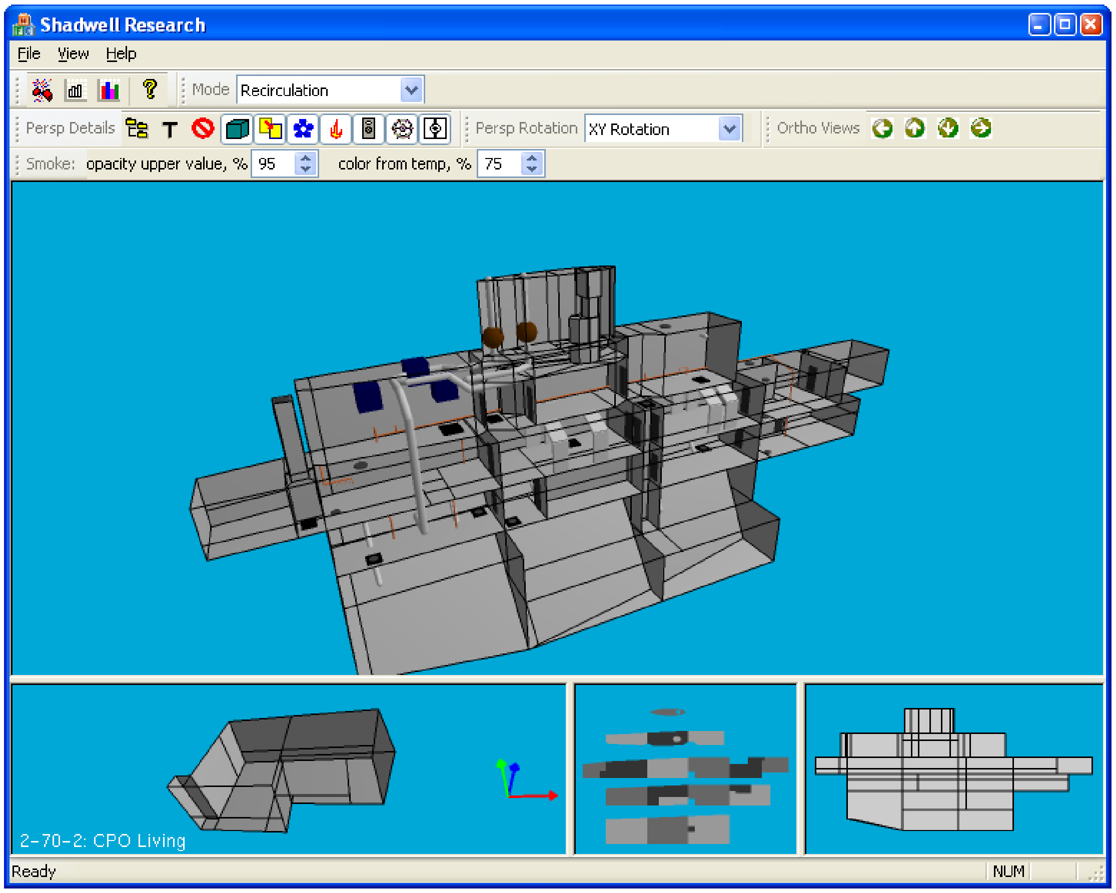
An interface is an abstract entity that provides only behavioral properties for its children. It means that an interface does not have any class data members; it is a set of abstract class member functions. Moreover, an interface does not even provide particular predefined behavior – it is just a declaration of possible function calls.

Interfaces are often used to describe the peripheral abilities of a class, not the central identity, e.g. the Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

CHAPTER V

RESULTS

5.1 Designed Software



Mode: Recirculation

Persp Rotation: XY Rotation

Smoke: opacity upper value, % 95 color from temp, % 75

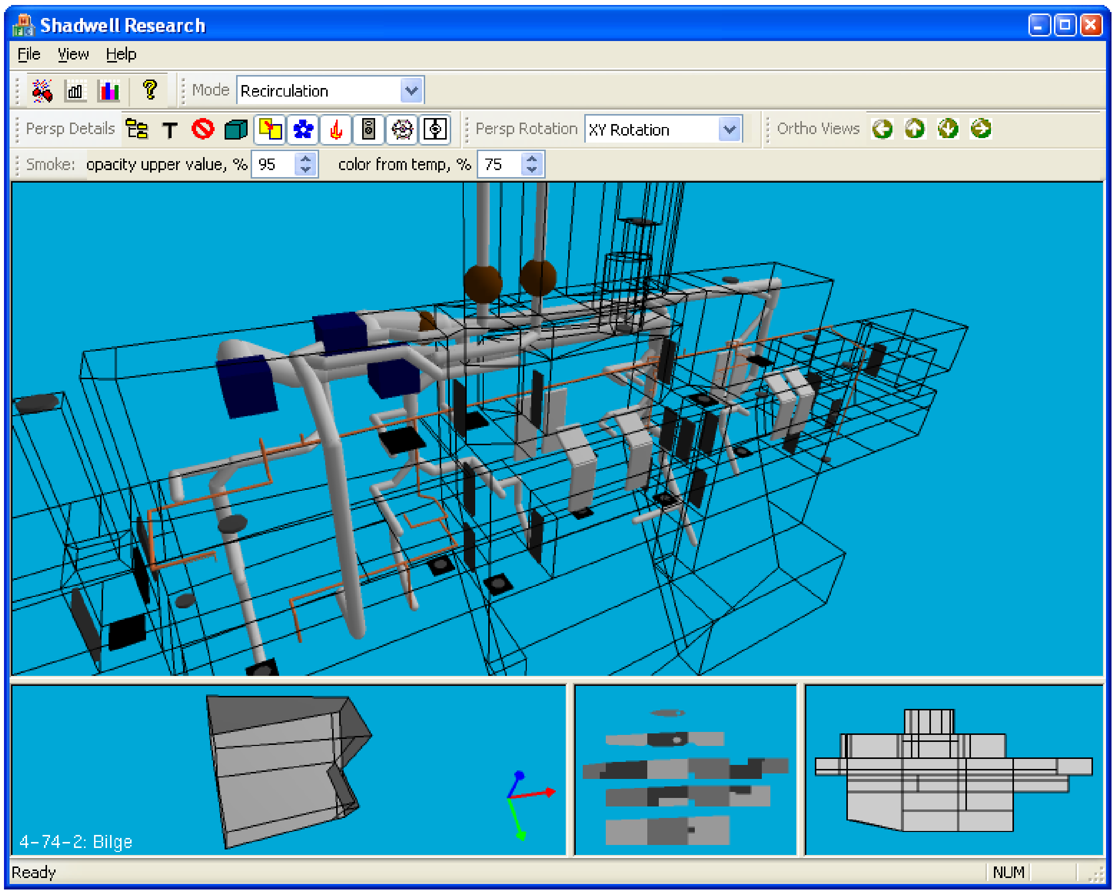
2-70-2: CPO Living

Ready

Figure 39: The main application window.

This work resulted in a designed and working prototype of the onboard fire and smoke propagation simulation system. It consists of several parts: the GUI, the Network model and the database.

Figure 39 shows the display right after the start of the program. 3-D geometry and objects are represented in the main middle area. A user has the ability to manipulate the scene and apply such actions as rotation, translation and zooming using a mouse. The backbone of the geometry is the compartments depicted as front-side opened gray boxes. Doors, hatches and scuttles are visualized as dark (or light, depending on its state) gray rectangles or disks.



Mode Recirculation

Persp Rotation XY Rotation

Smoke: opacity upper value, % 95 color from temp, % 75

Figure 40: Changed level of details: no compartment walls.

It is possible to change the levels of detail of the view using the *Perspective Details* menu bar. It contains buttons that allows showing or hiding of different kinds of objects like compartments, wireframe and others (Figure 40). The results on Figure 28 also were produced using this menu bar.

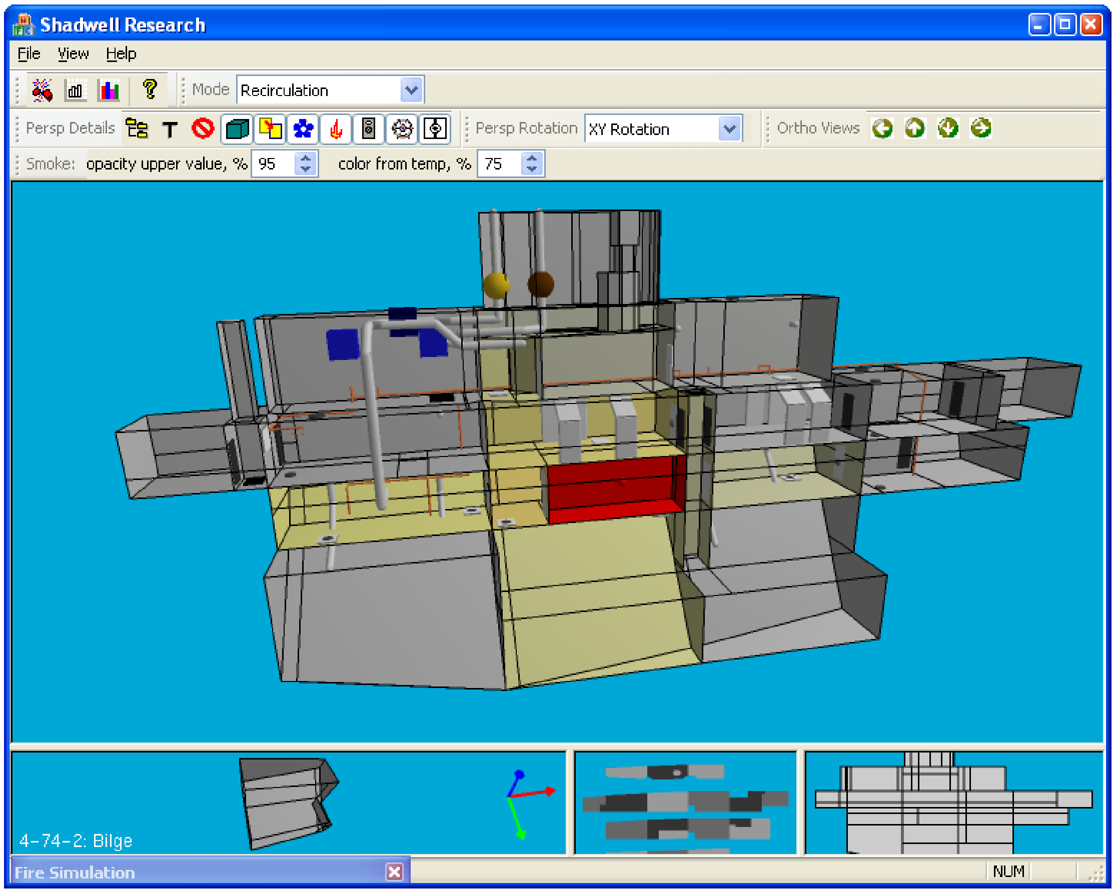
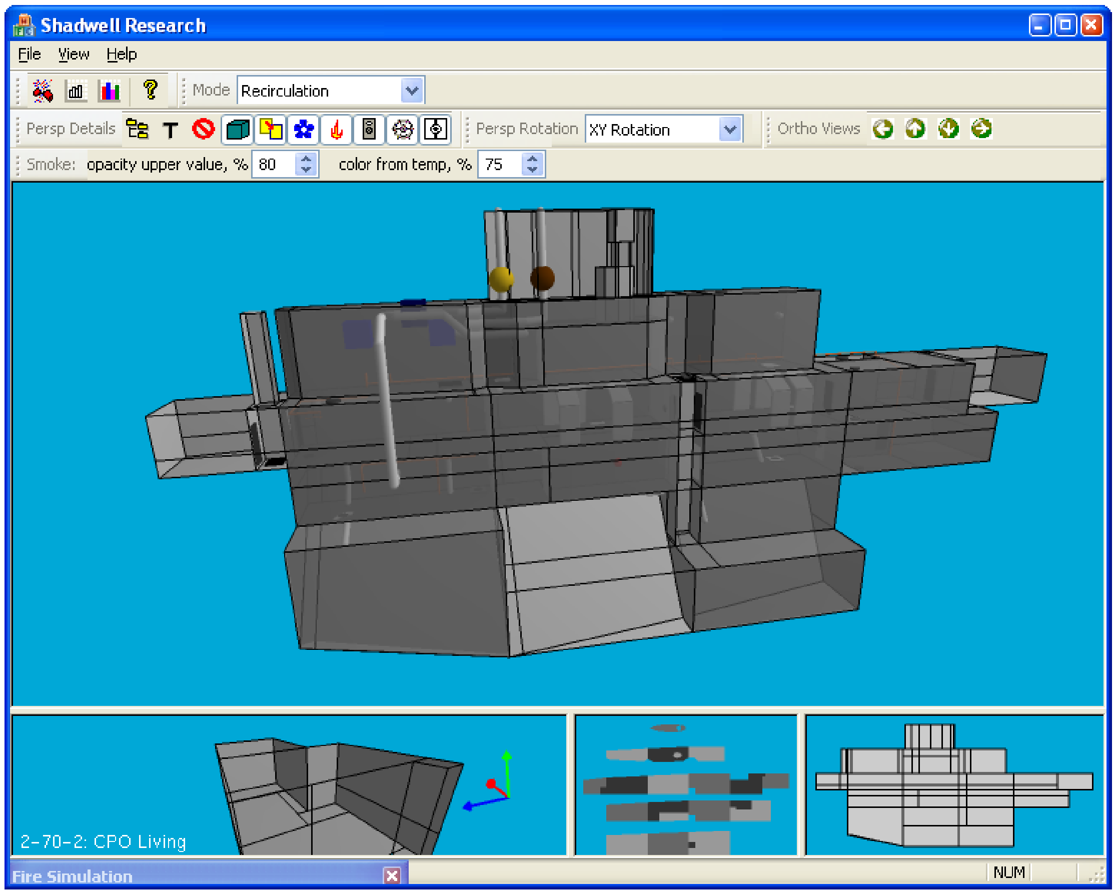


Figure 41: Fire simulation: visualization of temperature.

3-D geometry details of an individual compartment may be studied in the lower part of the window. The most left subwindow presents compartment geometry, whereas the other two allow selection of a compartment by clicking on it on the *Desk View* in the middle and the *Orthographic View* on the right.



Mode Recirculation

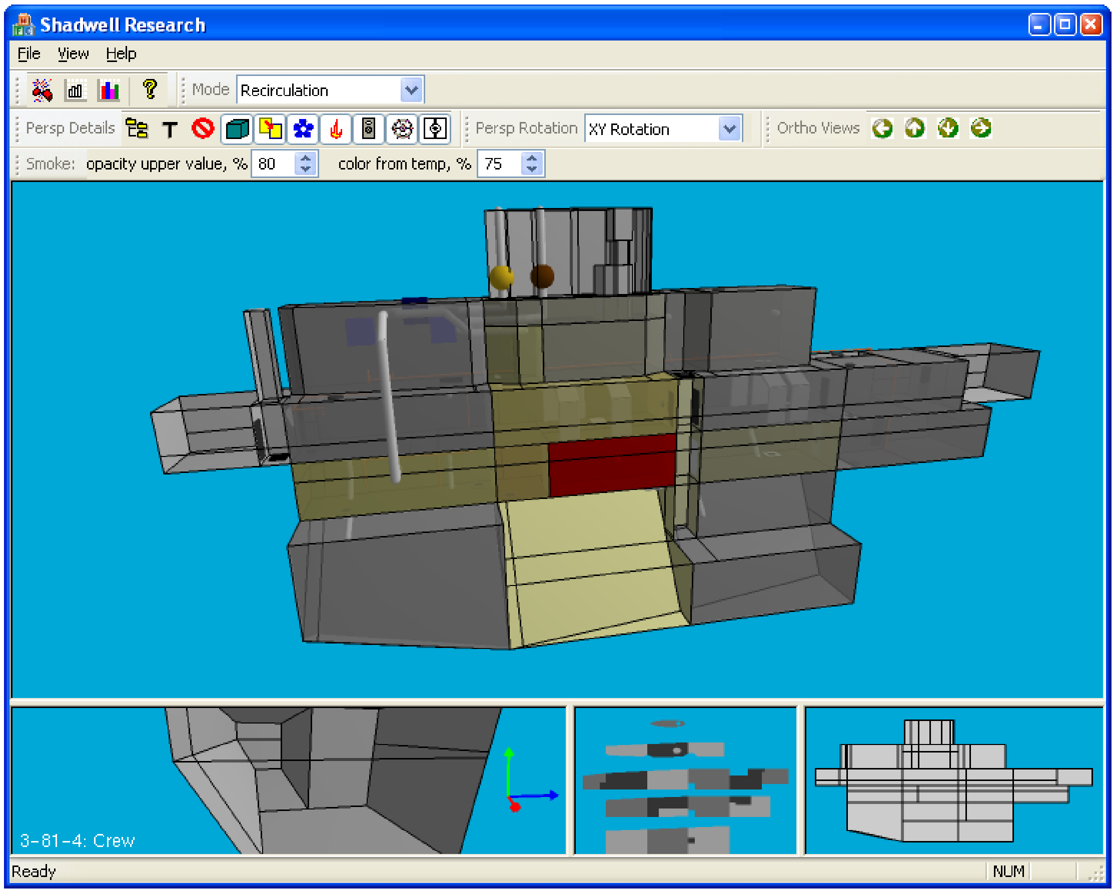
Persp Rotation XY Rotation

Smoke: opacity upper value, % 80 color from temp, % 75

Figure 42: Fire simulation: visualization of smoke.

In the simulation window (Figure 20), the user controls running, replaying and comparing of simulations. In any case, the simulation results appear in the main application window. Scalar values (e.g. temperature) are represented as color maps that affect the color of walls

(Figure 41), whereas smoke is represented as a transparent object inside of each compartment (Figure 42).



Mode Recirculation

Persp Rotation XY Rotation

Smoke: opacity upper value, % 80 color from temp, % 75

3-81-4: Crew

Ready NUM

Figure 43: Fire simulation: visualization of temperature and smoke.

It is possible to view parameters, temperature and smoke at once. In this case, the color of smoke of a compartment is affected by the temperature in that compartment (Figure 43). Analyzing this view does require some eye training to be able to adequately estimate a scalar value of compartment temperature and amount of smoke. The user must remember that parameters have two critical levels that have the greatest impact in the color and level of transparency. The *Legend* window offers assistance in identifying values of parameters as well as values of critical levels (Figure 32).

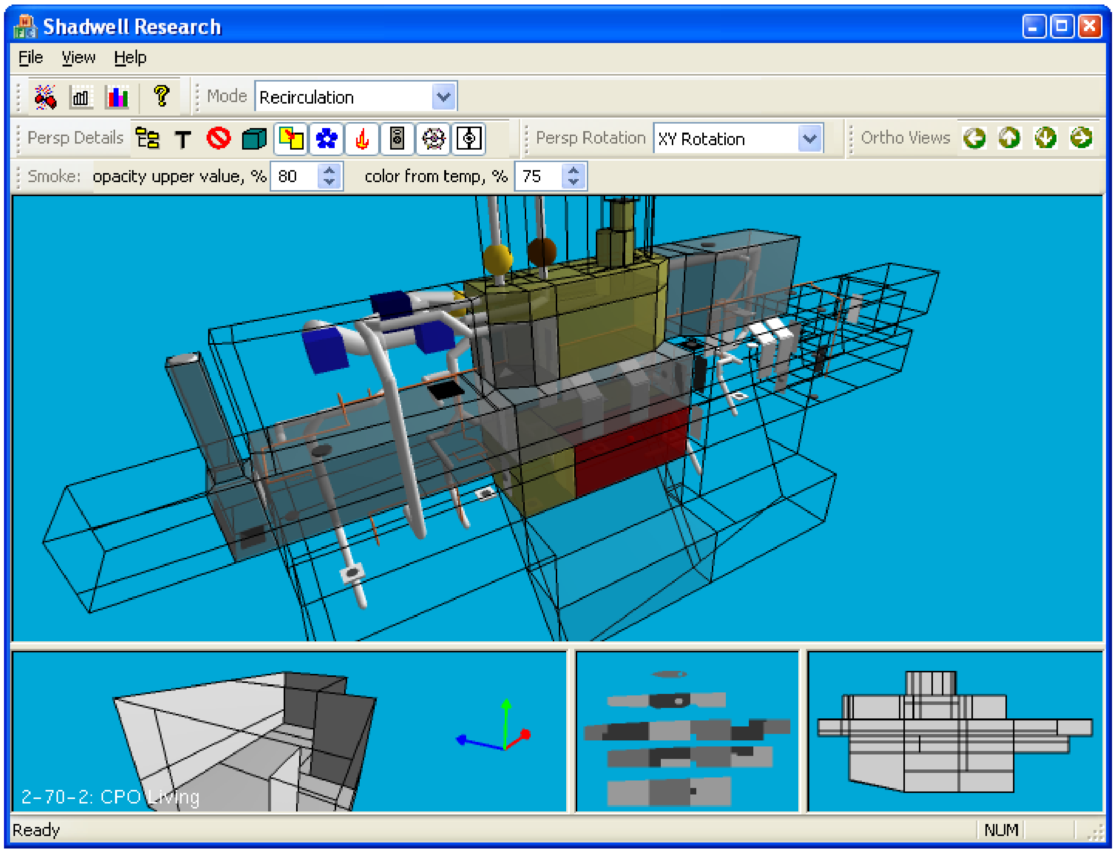
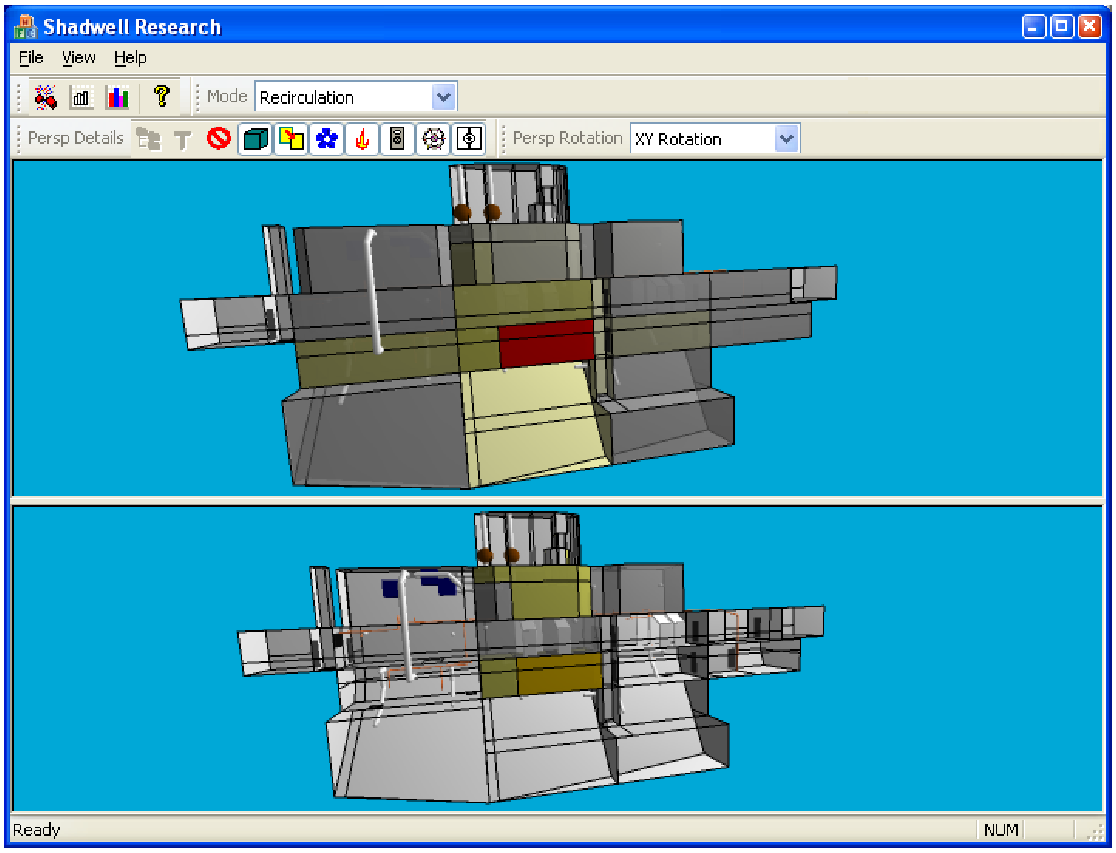


Figure 44: Fire simulation: smoke.

Switching off compartment walls produces a very good perception of smoke, as a resulting color is a mixture of the smoke color and a clean background, unaffected by the color of the walls. The effect is even stronger due to increased number of visual clues revealed by the absence of the walls (Figure 44).

Finally, the user is able to compare two simulations for better analysis and optimization. This simulation mode s represented by two similar perspective views separated horizontally (Figure 45). It provides the same interaction and functionality as described before.



Mode: Recirculation
Persp Rotation: XY Rotation
Ready

Figure 45: Fire simulations: comparison of two simulations.

This work is a small part of complex onboard ship fire suppression system. Currently, it provides functionality for:

- Import geometry and duct information from a third party CAD system.
- Visualization of the ship's geometry and systems.
- Simulation a fire with a visual feedback to a process.
- Comparison of previously run simulations.

Visualization of fire parameters includes two techniques – color maps and variable transparency objects. Color maps helped to represent scalar values like temperature, oxygen and etc., whereas transparency produced very convincing results for the smoke. Design of color maps required developing map function that realizes transition between a value and a color.

Visualization of smoke was and is a challenge. Currently, an object with transparent walls helps to produce effect of smoke. This effect is very convincing, but it is expected to fail on the later developmental stages, when a walking inside of the smoked volume is considered. Currently, such a mode is not required, but it may be extremely useful for training purposes and understanding processes, happening during a fire.

Another challenge was processing the visualization input data and producing an input file for the Network simulator. Currently, the input data comes from a ship model designer in a text format, which due to its nature is very unsteady, thus, unreliable. Hence, creating reading routings required higher concerns about formats of the files. With a properly organized database, this issue will be a straightforward task. Moreover, using a database will be also very beneficial from other aspects of the system development beginning from a simplicity and higher speed of data access and ending with a flexible data structure management and deeper results analysis.

The project at its current state was demonstrated at the Workshop on Fire Suppression Technologies held in February, 2003.

5.2 Hypothesis Validation

The goal stated the hypothesis is to prove that simulations in near to or faster than real time are possible. The visualization algorithms demonstrated a good performance reserve by running much faster than real time. This fact can be clearly observed by recording replay speed of simulations. Step-by-step replays on average run 40 times faster than real time. Hence, it is practically possible to achieve running simulations with such speed. Visualization produces a relatively low load on the main CPU during simulation runs – around 15-20% – whereas the Network model consumes the rest available, which is usually close to maximum 80%. Thus, even though the Network model is a current bottleneck, the presented software is able to process, store and finally render data in a much higher speed than the Network model can offer. Consequently, the stated hypothesis of developing a visualization system that may run near to or faster than real-time can be claim proven.

CHAPTER VI

FUTURE WORK

The project is still on its early development stages so there is a wide range of activities in the nearest future.

This paragraph presents brief descriptions of thing under consideration. Design a database structure has a high priority due to performance penalties causes by using text files as a data source. A proper and sophisticated error handling may help to turn this prototype into a robust and reliable product. The interaction with the user while a simulation is running does not exist due to inability the Network simulator to handle the user's requests. An example of such an iteration process is changing object states through the GUI.

The next direction of work is improving or changing currently used visualization techniques like:

- Custom color maps for visualizing different simulation species. Currently, two critical levels seen represented with yellow and red.

- Simplification of graphical duct representations, namely, removing spheres as section joiners.
- Plane representation of 3-D networks, e.g. ducts, which is simple, thus, helpful and important in critical situations.

A wall is built with some material. Different walls may use different materials, but a few are used during ship construction. They play an overwhelming role in fire rise and distribution. Providing the user with the ability to analyze simulation scenarios with different wall material settings may reveal valuable information for ship builders. It requires creating a mechanism for editing materials parameters, wall structures, and the ability to change wall settings interactively.

Smoke representation raises two issues: correct transparent color and using volumetric smoke.

The implemented approach of smoke representation uses the transparent polygons to obscure objects and produce a smoke effect. It requires sorting. Recently, Cass Everitt represented a method for rendering transparent objects order-independently [31]. He has described how hardware of a new generation can help to avoid a great deal of headache by using depth peeling mechanism. Depth peeling is a fragment-level depth sorting technique described by Mammen using Virtual Pixel Maps [9] and by Diefenbach using a dual depth buffer [1].

The peeling of a layer requires a single order-independent pass over the scene.

In addition to programming efforts, volumetric smoke requires considering its applicable use cases. Being very resource demanding due to its complexity, it must be used only in rare cases like gaming environment (walking through a smoked volume) or animations.

Representing an opening as an independent object is arguable. Indeed, each opening connects two or more compartments and it belongs to a wall. Thus, it may be thought as a property of a wall. Consequently, it may be preferable to aggregate an opening class to compartment. Aggregation would allow retrieval of additional information like adjacent compartments. Furthermore, it would assist in drawing only the compartment related objects, as well as in a depth resolution problem.

REFERENCES

[1] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Publishing Company, 2001.

[2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley Publishing Company, 1998.

[3] P. Diefenbach, *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering*, doctoral dissertation, Dept. Computer Science, University of Pennsylvania, Philadelphia, PA 1996.

[4] J. Floyd, T. Haupt, H. Habbash, B. Hodge, O. Norton, F. Williams, and P. Tatem, *Requirements and Development Plan fro a Shipboard Network Fire Model*, tech. report 6180/0469A:FW, Hughes Associates Inc., Baltimore, MD, 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Publishing Company, 1995.

[6] W. Jones, G. Forney, R. Peacock, and P. Reneke, “A technical reference for CFAST: an engineering tool fro estimating fire growth and smoke transport”, *NIST TN 1431*, Building and Fire Research Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2000.

[7] N. M. Josuttis, *The C++ Standard Library: a Tutorial and Reference*, Addison-Wesley Publishing Company, 1999.

[8] J. Klote and J. Milke, *Design of Smoke Management Systems*, American Society of Heating, Refrigeration and Air-Conditioning Engineers, Atlanta, GA, 1992.

[9] A. Mammen, “Transparency and antialiasing algorithms Implemented with the virtual pixel maps technique”, *IEEE*

Computer Graphics and Applications, vol. 9, no. 4, July 1989, pp. 43-55.

[10] E.J. Marey, *La Methode Graphique*, Paris, 1885.

[11] K. McGrattan, H. Baum, R. Rehm, A. Hamins, G. Forney, J. Floyd, and S. Hostikka, "Fire Dynamics Simulator (version 2) – Technical reference guide", *NISTIR 6783,* Building and Fire Research Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2000.

[12] T. Möller and E. Haines, *Real-Time Rendering*, A K Peters, Ltd., Natick, MA, 1999.

[13] G. Murch, ed., "Color graphics – Blessing or Ballyhoo (Excerpt)", in R. M. Baecker, W. Buxton, and J. Grudin ed., *Readings in Human-Computer Interaction: Toward the Year 2000,* San Francisco, CA, Morgan Kaufmann, 1985.

[14] J. Prosise, *Programming Windows with MFC*, 2nd ed., Microsoft Press, Redmond, WA, 1999.

[15] J. Richter, *Programming Applications for Microsoft Windows*, 4th ed., Microsoft Press, Redmond, WA, 1999.

[16] S. R. Schach, *Classical and Object-Oriented Software Engineering with UML and Java*, 4th ed., WCB/McGraw-Hill Company, 1999.

[17] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit. An Object-Oriented Approach to 3D Graphics*, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[18] B. Stroustrup, *The C++ Programming Language*, 3rd ed., Addison-Wesley Publishing Company, 2000.

[19] E. R. Tufte, *The Visual Display of Quantitative Information*, 2nd ed., Graphics Press, 2001.

[20] R Vieira, *Professional SQL Server 2000 Programming*, Wrox Press Inc., Chicago, IL, 2000.

[21] M. Woo, J. Neider, T. Davis, and D. Shreiner, ed., *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Addison-Wesley Publishing Company, 2000.

[22] http://java.sun.com/products/java-media/3D

[23] www.ati.com

[24] www.cd-adapco.com/products/physical.htm

[25] www.cs.berkeley.edu/~bukowski/wkfire

[26] www.cti-simulation.com/ctisimulation/nuclear.htm

[27] www.fire.nist.gov/fds

[28] www.linas.org/gle

[29] www.microsoft.com/windows/directx/default.asp

[30] www.nvidia.com

[31] www.nvidia.com/view.asp?IO=Interactive_Order_Transparency

[32] www.opengl.org

[33] www.opengroup.org/motif

[34] www.vib-mrfc.de