



**NAVAL
POSTGRADUATE
SCHOOL**

MONTEREY, CALIFORNIA

THESIS

**DEVELOPING AN AFTER ACTION REVIEW SYSTEM
FOR A 3D INTERACTIVE TRAINING SIMULATION
USING XML**

by

Dimitrios E. Filiagos

March 2004

Thesis Advisor:

Rudolph Darken

Second Reader:

Joseph Sullivan

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE March 2004	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE: Developing an After Action Review System for a 3D Interactive Training Simulation Using XML			5. FUNDING NUMBERS
6. AUTHOR(S) Dimitrios E. Filiagos			8. PERFORMING ORGANIZATION REPORT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			12b. DISTRIBUTION CODE
13. ABSTRACT (maximum 200 words) An important capability that many modern 3D interactive training simulations lack is an After Action Review System (AARS) that helps both the trainer and trainee to conduct an After Action Review (AAR). Although AAR is not a new idea in the 3D simulation field, it is not widely used in training simulations. In real life training, AAR has been proven as one of the most important phases of the training procedure, sometimes taking the form of debriefing, or in other cases, by conducting a deeper analysis and discussion of the facts. In order to conduct an AAR, a well-designed system (AARS) must exist to keep track of the conditions and the actions during an exercise, so they can be available for review later. This thesis translates the idea of AAR for real training situations to the 3D interactive simulation domain and also develops an After Action Review System (AARS) using XML technology for capture, analysis, and interactive playback of an entire simulation training session. Users can change the point of view to any desired position and direction, something that is impossible in video streaming playbacks.			
14. SUBJECT TERMS After Action Review, After Action Discussion, Simulation Capture, Simulation Playback, Platform, Platform Path, HLA, Federation, DOM, SAX, XSLT			15. NUMBER OF PAGES 73
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release: distribution is unlimited

**DEVELOPING AN AFTER ACTION REVIEW SYSTEM FOR A 3D
INTERACTIVE TRAINING SIMULATION USING XML**

Dimitrios E. Filiagos
Lieutenant, Hellenic Navy
B.S., Hellenic Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

and

**MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS AND
SIMULATION (MOVES)**

from the

**NAVAL POSTGRADUATE SCHOOL
March 2004**

Author: Dimitrios E. Filiagos

Approved by: Rudolph P. Darken
Thesis Advisor

Joseph A. Sullivan
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

Rudolph P. Darken
Chair, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

An important capability that many modern 3D interactive training simulations lack is an After Action Review System (AARS) that helps both the trainer and trainee to conduct an After Action Review (AAR). Although AAR is not a new idea in the 3D simulation field, it is not widely used in training simulations. In real life training, AAR has been proven as one of the most important phases of the training procedure, sometimes taking the form of debriefing, or in other cases, by conducting a deeper analysis and discussion of the facts. In order to conduct an AAR, a well-designed system (AARS) must exist to keep track of the conditions and the actions during an exercise, so they can be available for review later. This thesis translates the idea of AAR for real training situations to the 3D interactive simulation domain and also develops an After Action Review System (AARS) using XML technology for capture, analysis, and interactive playback of an entire simulation training session. Users can change the point of view to any desired position and direction, something that is impossible in video streaming playbacks.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION.....	1
A.	PROBLEM STATEMENT	1
B.	MOTIVATION	1
C.	RESEARCH QUESTIONS	2
II.	BACKGROUND	5
A.	INTRODUCTION.....	5
1.	After Action Review.....	5
2.	After Action Review Methods	5
3.	After Action Review System (AARS)	6
4.	After Action Review Systems for Training Simulations	7
B.	THE VECQB SIMULATION	7
1.	Introduction - Characteristics	7
2.	Detailed Description.....	8
3.	Extended Features.....	8
III.	AAR SYSTEM DESIGN	11
A.	INTRODUCTION.....	11
1.	Purpose.....	11
2.	Overview.....	12
a.	Frequency.....	12
b.	Networking	12
B.	GENERAL DESIGN	13
1.	Data Collection and Storage Phase	14
2.	Process and Analysis Phase.....	15
3.	Presentation Phase.....	15
C.	AARS IN VECQB	16
1.	Overview.....	16
2.	Capture Phase	17
3.	Playback Phase.....	19
a.	Summary Report	21
b.	Interactive Playback.....	22
c.	Other Features	24
4.	The Role of XML.....	26
a.	Validation	26
b.	Transformations.....	27
c.	Error Handling	27
5.	File Formats	27
a.	XML Format	27
b.	Binary and ASCII Text Format.....	28
IV.	AARS IMPLEMENTATION	31
A.	INTRODUCTION.....	31
1.	Overview of VECQB Source Code	31
2.	Design.....	33

B.	MODULE CAPTURE	33
1.	Class Hierarchy	33
	<i>a. Notification.....</i>	<i>33</i>
	<i>b. Encapsulation.....</i>	<i>35</i>
	<i>c. Storing.....</i>	<i>37</i>
2.	File Conversion.....	38
3.	Capturing	39
C.	MODULE PLAYBACK.....	39
1.	Playback.....	40
	<i>a. Read Periodic Function.....</i>	<i>40</i>
	<i>b. Tick Periodic Function.....</i>	<i>41</i>
2.	Managing the Platforms	43
D.	XML LIBRARY.....	44
E.	GRAPHICAL USER INTERFACE.....	45
1.	Main Menu.....	46
	<i>a. Simulation Menu Item</i>	<i>46</i>
	<i>b. Edit Menu Item</i>	<i>47</i>
	<i>c. View Menu Item</i>	<i>48</i>
	<i>d. Actions Menu Item.....</i>	<i>51</i>
V.	CONCLUSIONS-FUTURE WORK	53
A.	SUMMARY.....	53
B.	FUTURE WORK.....	53
	APPENDIX GLOSSARY.....	55
	BIBLIOGRAPHY.....	57
	INITIAL DISTRIBUTION LIST.....	59

LIST OF FIGURES

Figure 1.	Screenshot from the VECQB Simulation.	2
Figure 2.	Equipment Used for the VECQB Simulation in Naval Postgraduate School's MOVES Lab.	9
Figure 3.	Phases of the AARS Architecture. 1st phase occurs at runtime (during an exercise), while the other two, after the end of it.	14
Figure 4.	How the Three Major Phases Fit into the VECQB AARS.	17
Figure 5.	Data Collection and Storage Phase. Combination of captured data from all cells creates the entire picture.	18
Figure 6.	Mechanism that Generates the Summary Report. An XSL Transformation is applied to the two input XML files.	22
Figure 7.	Mechanism that Generates the Platform's Waypoint File. The appropriate XSLT document is being applied to the two input XML files.	24
Figure 8.	Screenshot from the VECQB Simulation Demonstrating a Platform's Path. ...	25
Figure 9.	View of an XML Captured File. The document root element is Simulation. Then the Start element follows and a sequence of Events.	28
Figure 10.	A Representation of Binary and ASCII Files Line by Line. The binary file is a continuous binary stream. The ASCII has the same structure as the binary except the spaces between the data values.	29
Figure 11.	Event Class Hierarchy. Event is the base pure abstract class and the others are deriving from it.	35
Figure 12.	Stream Class Hierarchy.	37
Figure 13.	m_eventList Contains Pointers to the Events. m_eventPtr points to the event that is going to trigger next	41
Figure 14.	Box Diagram of the Event Latching Cycle.....	42
Figure 15.	Screenshot from the VECQB GUI.....	46
Figure 16.	Screenshot from the VECQB GUI. Simulation menu item.	47
Figure 17.	Screenshot from VECQB GUI. Edit menu	48
Figure 18.	Screenshot from VECQB GUI. Control Panel dialog.....	49
Figure 19.	Screenshot from VECQB GUI. Control Panel	51
Figure 20.	Screenshot from VECQB GUI. Actions menu item.....	52

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wife Marielena and my daughter Parita, for their patience during my studies at the Naval Postgraduate School.

I would also like to thank my advisors, Rudolph Darken and Joseph Sullivan, for their supervision and support. They were always available when I needed them.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

As computing power is increasing day by day, still following Moore's law, training simulations are becoming more sophisticated by incorporating various features. Some of the features are photorealistic rendering, agent technology for the computer driven entities, more accurate collision detection, and response, etc. Networking is another important feature that allows users from different global sites to become involved with the same simulation training session. The number of users that a networked simulation can support [Singhall and Zyda] is from just a couple to hundreds of thousands. It becomes immediately apparent that it is infeasible for someone to maintain a comprehensive view of what is happening in such a simulation, since the number and the pace of events are impossible to follow. On the other hand, the course of events in a simulation is valuable for an After Action Review (AAR) and analysis. After Action Review is a procedure that is widely acceptable in real training situations and perfectly fits the needs and the purpose of a networked virtual reality training simulation.

B. MOTIVATION

In order to introduce the idea and actually allow an After Action Review in a training simulation, it is important to develop a system that will track all the events of a session so that they can be reviewed later. Such a system must also provide mechanisms for the processing, analysis, and presentation of the data, and tools also to help the reviewer distinguish the events of interest from those that are insignificant. The challenge is to do that efficiently with the least possible impact on the simulation's frame rate.

The motivation of this thesis is to bring AAR one step closer to virtual reality and training simulations by designing an After Action Review System (AARS), discuss the most difficult parts of its design and then implement and test it, on one of DoD's training simulations, VECQB. VECQB (Virtual Environment Closed Quarters Battle) is currently under development by Lockheed Martin's laboratories. The design and implementation would also incorporate some advanced features such as the interactive playback of the entire simulation session as well as the drawing of the path followed by each entity.



Figure 1. Screenshot from the VECQB Simulation.

C. RESEARCH QUESTIONS

This research attempts to answer several key questions.

Initially, is it possible to develop a technique for capture, playback, and processing 3D data and events from a fraction of a simulation session in real time? The answer to this question is not an easy one because there are some obvious hardware limitations, mostly related with capturing. Capturing involves logging each entity's position, orientation, state, firings, detonations, entity "Creation", or "Destroy" events and so on via the network. In a simulation with a large number of entities, the network traffic can consume the bandwidth causing not only noticeable latencies to the simulation but also packet losses and inconsistencies to the producing logs. It seems that for a reasonable number of entities, capturing is possible without any problems and as the number is increasing, there are always countermeasures that will be discussed later such as Area-of-Interest Filtering or Group-per-Entity Allocation [Singhall and Zyda].

Second, is XML technology a good choice for that type of application? Why XML? In the past, After Action Review Systems made use of relational databases to log the events of a simulation. Then, to review the captured data, the users had to query the database. The relational database approach may seem the only reasonable solution to the problem, and indeed has some advantages, especially for a large number of entities simulations but it still requires extra overhead to develop, build (populate), use, and maintain. On the other hand, XML provides a very handy alternative with significantly lesser overhead and cost. XML today has somewhat revolutionized the build of lightweight databases, ensuring interoperability and extensibility, and providing powerful capabilities such as document validation and transformation allowing for data manipulation.

What can be done with the collected data from a simulation session? What conclusions can be drawn after the analysis of the data? The collected data are subject to further analysis in order to ascertain WHO did WHAT, WHEN and WHY, as well as comprehend and evaluate the actions taken by the participants in a training simulation, seek the presence of any tactic and/or strategic moves and patterns, and compare what was planned to what actually happened. It is also possible to detect problems related with virtual reality, cyber-sickness, and human factors. For example, it can be inferred that a participant did actually make a wrong decision because other issues such as a poor hardware configuration, motion sickness, limited field of view, poor image fidelity and so forth, influenced his actions.

Lastly, why and how can this technique help the trainer and the trainee? They both now have a common useful tool to discern what happened during the simulation and to find reasonable answers to their questions. The trainer will have a tool to evaluate the trainees' performance and the trainee, in turn, will be able to see and understand his own actions from a different point of view and maybe do some productive self-criticism.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. INTRODUCTION

1. After Action Review

After Action Review (AAR) is an important phase of the training procedure in every field. The simplest version is just a discussion (debriefing) that takes place after the end of a training session or an exercise. This tactic has been proven quite valuable for the trainees, especially novices, in order to understand the basic principles of training. A more advanced version of AAR uses existing technology such as slides, photos, video, charts, maps, sound, and computers to enhance the content of the AAR.

Virtual Reality is gaining day by day on traditional training methods. The use of AAR as a metaphor in Virtual Reality training simulations is not a risky attempt since it is a well-known procedure that will be used the same way as in the real world. The only thing that changes is the field of training, for example, a classroom or a battlefield in Virtual Reality is substituted for a virtual classroom or battlefield respectively.

2. After Action Review Methods

Perhaps the most important and productive method to conduct AAR today is After Action Discussion. It involves a discussion after a training session between the participants (trainees) and the instructor or any other experienced person, in the context of the training. The instructor ensures confidentiality and that everyone can talk and be heard. The goal is not to blame the participants who acted incorrectly but rather to spot any wrong decisions or actions, and analyze why they were wrong, what the consequences were, and how they could have been avoided. The discussion will have the desired results if it succeeds in providing answers to questions such as:

- What was planned?
- What really happened?
- Why did it happen?
- What can be done to make it better next time?

The discussion may examine a variety of subjects depending on the events and the interests of the participants. These may include:

- Methods used

- Lessons learned
- Communication among the participants
- Stress impact
- Fatigue impact
- Attitude of participants
- Adaptation
- Human factors
- Human performance
- Safety and organizational issues
- Roles and responsibilities

Duration and timing of the discussion [Allen and Smith, 1994] are very important. Only the most important facts should be discussed. If analysis degrades to trivia and unnecessary details, the participants will become bored. Timing is also important because people tend to forget the reasons why they followed a specific course of action because this is something that is recalled in the context of an emotion. Emotions, like fear, confusion, uncertainty, frustration and so on, influence the decision-making procedure.

The above principles and guidelines about AAR can be applied to the full extent possible in the Virtual Reality domain, and of course, in training simulations.

3. After Action Review System (AARS)

In many cases, AAR employs various technological innovations to improve and revolutionize the manner of conducting AAR. In other words, there are systems that are designed specifically to support AAR. Such systems are known as After Action Review Systems. An AARS may include cameras, closed circuit TVs for monitoring the training space, monitors, GPS (Global Position System) devices to log the location of each participant, microphones, wireless communication systems (e.g. walkie-talkie), computers for various purposes, etc.

AARS ensures that a training scenario is well covered and allows the users, during the AAR session, to go back and see the sequence of events that took place during the exercise. AARS can bring to light hidden aspects of training and help its users to identify the factors that dramatically changed the flow of events. Many sophisticated

AARS exist that utilize high-end technology, but their cost is high, not only to build them, but also to use and maintain them.

4. After Action Review Systems for Training Simulations

On the contrary, in the Virtual Reality domain, an AARS can be as powerful and flexible as it gets. Any aspect of a training simulation can be controlled and monitored. Cost is not a problem since the only thing needed is just some coding and, maybe, some extra computer equipment, in case the AARS code must be executed in a machine other than the one mentioned for the training simulation. The role of an AARS can be extended to allow for data processing and presentation. That allows the user to take a first glance of what happened. The system can also provide utility tools to conduct a better search of the sequence of events, a comparison between the pre-scheduled scenario to what really happened, quick access to a participant's movements and actions during an exercise, playback of the entire exercise, summary statistics for each participant or group of participants in the exercise and so on. The tools and features of such a system can be tailored exactly to fit the specific needs of the simulation for which they are designed.

The main focus of this research was to design and develop such an AAR System in the context of the VECQB simulation. The system utilizes some of the most common principles for that area, introduces some others, and demonstrates the power and advantages that an AARS can have in the virtual world, versus the real world.

B. THE VECQB SIMULATION

1. Introduction - Characteristics

A Virtual Environment Closed Quarters Battle (VECQB) is part of DoD's Virtual Technologies and Environments (VIRTE) project. VECQB is currently under development by Lockheed Martin Inc. It is a "first person shooter" type of application focusing mostly on closed quarters battles and tactics. A typical exercise scenario involves the clearing a building of enemy forces. As a "first person shooter" application, it only includes one local entity (platform in VECQB terminology).

The simulation keeps the user alert and focused all the time on any suspicious movement or sudden attack. The user must also cooperate with other remote participants to achieve his goal. It appears to be a great tool for training, especially in cases where there is a lack of space, such as on battleships or aircraft carriers. The simulation does not

need any special equipment, except of course the advanced configuration, and can be launched directly from a laptop. This makes the simulation portable to almost any place. The user can choose among several setup configurations starting from the simplest one that takes input from a keyboard and a mouse, to one that uses a game pad, and eventually, to the most advanced that includes replicas of M-4 and M-16 weapons, Head Mounted Displays (HMD) and inertial trackers for head and weapon tracking.

2. Detailed Description

VECQB is written in C++ and consists of a number of singleton classes (modules) each one responsible for an aspect of the simulation. Hence, there is a class responsible for the input phase capable of polling, initializing communication, and receiving any input event from the available input devices. Other classes exist to interface the HLA infrastructure and handle the networking part, compute physics, environmental settings, and so forth. The code currently has 30 such major classes-modules, which constitute the backbone of the system. The architecture of assigning different parts of the simulation to different handler-classes enhances the modularity of the code making it easy to maintain, modify, and extend.

VECQB is a fully networked application using DoD's standard networking infrastructure HLA (High Level Architecture). The rendering is done using NetImmerse[®], which is a product of Numerical Design Ltd. Physical-based events are handled by an Open Dynamics Engine (ODE). Extra weight has been given to sound due to the nature of the simulation. Thus, 3D spatial sound is used that requires extra processing power provided by a second computer or special equipment (Motu 800).

3. Extended Features

At the Naval Postgraduate School, two important features were added to the simulation by the author of this thesis. The first is the advanced interface configuration that includes HMD, a weapon replica (M-4 or M-16) and two inertial trackers, mounted on the HMD and the weapon respectively. The setup allows for independent 3-degrees of freedom motion of the user's head and weapon. The weapon's yaw rotation becomes the user's virtual body (avatar) heading. Head rotation does not change the body heading at all. That interface is very close to human nature, since someone can walk towards one direction while looking at another. No methodic testing was done in respect to human

factors for that feature, but the reader can review Steve Mathew's and Ken Miller's Master's thesis that did almost the same thing using exactly the same equipment in the Naval Postgraduate School's labs.

The second extended feature is the AARS, which is the main topic of this thesis.



Figure 2. Equipment Used for the VECQB Simulation in Naval Postgraduate School's MOVES Lab.

THIS PAGE INTENTIONALLY LEFT BLANK

III. AAR SYSTEM DESIGN

A. INTRODUCTION

1. Purpose

The purpose of the system is to provide the following features to the user:

- Information about each platform's activity in the simulation (summary report):
 - Platform's identification
 - Time of creation
 - Number of shots fired (or any other type of primary ammunition)
 - Numbers of hits
 - Number of kills (either mobility or operational)
 - Number of grenades thrown (or any other type of secondary ammunition)
- Events triggered by the system
 - Platform creation
 - Platform destroy
- Environmental settings
 - Time of day (for daylight settings)
 - Day of year (for sun and moon position and phase)
- Interactive playback of the exercise. Playback is designed to provide the following features:
 - Play back the simulation
 - Stop the playback at user request
 - Pause the playback because of a user request allowing viewpoint adjustments
 - Slow speed motion
 - Fast speed motion; this directly depends on the hardware the simulation playback is "hosted"
 - Draw the path followed by each platform (user is allowed to set a different color for each platform)
 - Set the viewpoint on any platform in the simulation (user is allowed to adjust the offset)

2. Overview

The design of an AARS is a very crucial. Modularity, efficiency, and extensibility are some of the ingredients. The problem is to find a way to monitor any activity, and any event that happens in the simulation and keep it in a log for further processing and analysis. The problem that arises next is what events to monitor and how often. Eventually, there are events in the simulation that are triggered by the users while some others are spawned by the system. The latter, no matter how many times they are launched, will work in the same manner each time, given that everything else remains the same, causing the same results. The dropping of a box from the same position many times will have the same effect each time. The box will follow the same trajectory, bounce the same way and land on the same spot, each time, and this is because the physics part of the simulation will always solve the problem the same way. It is obvious that the AARS must log only the initial conditions of those kinds of events and the rest can be reproduced by the system.

a. Frequency

Frequency is another keyword that must be given a great deal of thought and consideration. The sampling theory fits perfectly into the problem definition because, monitoring and logging a simulation is actually the same as sampling or taking snapshots of the simulation. The more samples (logs) taken, the more accurately the simulation can be analyzed later. On the other hand, as the number of samples increases, the danger of overwhelming the system with unnecessary details increases as well. It is not only that the logs may be larger in size and require more space, but also, the logging and analysis procedure will become computationally expensive causing some of the well-known consequences to the simulation such as latency, the slowing down of the frame rate and so on. Hence, there are some trade-offs between sampling frequency and efficiency or, between efficiency and accuracy.

b. Networking

A common characteristic of most simulations today is that they are networked allowing more than one remote user to engage in the same simulation session. This makes things easy as to where the AARS code will be executed. In a networked

training simulation, a separate machine “running” the AARS code can be employed to log the events that are communicated over the network without interfering with the machine that “hosts” the simulation. In cases where a simulation does not support networking and remote entities, the AARS must, if not run on the same machine with the simulation, at least interfere with it, e.g., form a cluster, or use some kind of connection and protocol to monitor and log the desired events. There are also many networked training simulations that use one or more servers (Centralized Repositories) to maintain the “world state”. In those situations, sampling frequency becomes even more important since, polling with high frequency for updates can bog down the server(s), decrease their response dramatically, and jeopardize the simulation’s state consistency. On the other hand, low update frequency will lead to inconsistencies and inaccuracies

B. GENERAL DESIGN

This research is focused on developing an AARS for a networked training simulation that uses HLA protocol and does not use any central server to maintain the “world state”. From the early stages of the design, three [Vasend, 1995] major phases prevailed (Figure 2):

- Data Collection and Storage
- Data Process and Analysis
- Presentation

The Data Collection and Storage phase is the most difficult to design and implement but is also the most important. It is the only phase that occurs during the simulation session. The other two phases can be executed conveniently after the simulation’s end.

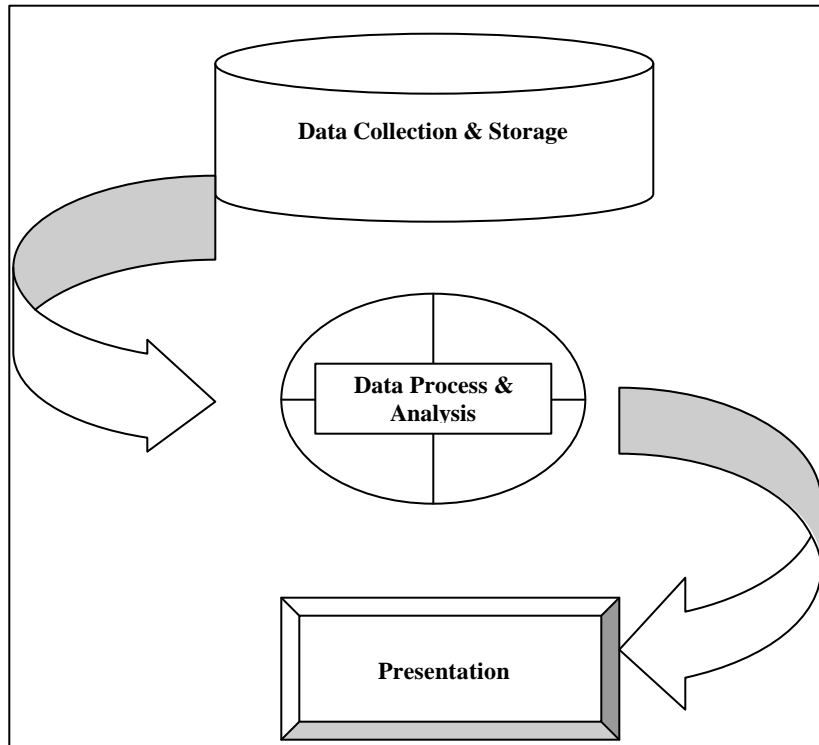


Figure 3. Phases of the AARS Architecture. 1st phase occurs at runtime (during an exercise), while the other two, after the end of it.

1. Data Collection and Storage Phase

The purpose of this phase is to generate the appropriate logs about the events of interest occurring in the simulation. The design here is critical and depends directly on the nature of the simulation. A different approach must be followed if the simulation is networked as opposed to one with no networking capabilities. In networked simulations, the way of maintaining the “world state” also influences the Data Collection and Storage phase’s design. The goal is to generate the logs with minimal impact to the simulation. A strategic point must be determined for this phase to be set up, for example, if the simulation accommodates a large number of entities, a good practice is to execute this phase on a dedicated machine. Another important issue is the format in which the collected data will be saved. Some systems write down all the incoming events to a file without any further processing whereas others build an entire database “on the fly”! Computation power and the expected volume of incoming events are very important at this point. If a high volume of incoming events is expected, it may be better to save the entire packets the time they come in, from the network, instead of unwrapping them,

extract only the desired information, and then save the information in the format of choice. Saving entire packets will require less computational power but more I/O operations, and of course will include excess and unnecessary information. Although this is not an elegant solution, it is considered viable in simulations where there is much heavy network traffic and the collection cannot occur in a dedicated machine. In cases where the collected events cannot be processed by the system during data collection, their process will be performed by the next phase or any other utility function after the Data Collection and Storage phase ends. Lastly, it appears that packet loss over the network may jeopardize the data collection procedure. This is truly an issue and one way to minimize it is to “run” any simulation instances and the data collection procedure on the same dedicated network. In cases where the simulation involves platforms that reside on remote networks, there is not much to be done. In any case, packet loss and latencies do not only affect the data collection procedure, but any instance of simulation as well, causing inconsistencies in the “world state”.

2. Process and Analysis Phase

The captured file, depending on the number of platforms, the pace of the events they generate and the capture duration may end up being too large. This phase will use this file to do some analysis and reveal the insights of the simulation to the end user. Process and Analysis is the second phase of the AARS and can take place after the end of the simulation session. Its purpose is to help the user process and analyze the events that occurred in the simulation. It may also provide mechanisms to process the collected data, and reformat it in the appropriate form, for further analysis. An example would be a mechanism to un-wrap the collected events and build some type of database. The user then will not have to go through the logs, but instead would be able to conduct a convenient database query. This phase should also provide handy tools for the analysis of the data, such as summary statistics for each platform, summary reports, scores, probabilities, performance measurements, etc.

3. Presentation Phase

The results from the process and analysis of the data conducted during the previous phase are presented in the Presentation phase. This phase is very important from

a human interaction point of view since it is the one with which the user actually interacts. Common ways of interaction/presentation are forms, spreadsheets, charts, and diagrams, etc.

C. AARS IN VECQB

1. Overview

VECQB was divided into two separate modes of operation:

- Normal mode
- AAR mode

The first is nothing more than just the normal simulation session where the user can engage in a scenario. The second is for After Action Review and the user cannot join in any scenario. The separation in modes of operation was necessary for safety reasons having to do with networking configurations and the local avatar behavior. The AAR mode uses a different networking configuration. It joins on a federation different from the one that Normal mode uses, so as not to interfere with other simulation instances that may run in Normal mode. The local avatar, which in a simulation represents the local user, cannot change anything in the environment during AAR to preserve the consistency of the playback.

In VECQB, the Data Collection and Storage phase can take place only during the Normal mode of operation and is called the “Capture” phase. The other two phases of the Process and Analysis phase and the Presentation phase, are combined into one phase, which takes place during the AAR mode of operation and is called the Playback phase, since playback is the major feature (Figure 4).

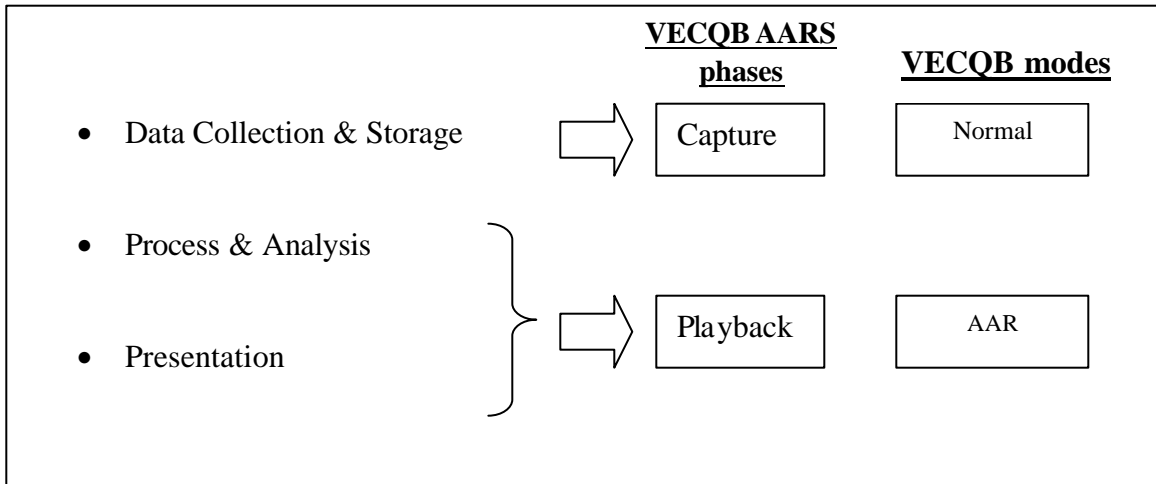


Figure 4. How the Three Major Phases Fit into the VECQB AARS.

2. Capture Phase

In the context of the VECQB simulation, this phase “runs” on the same machine as the simulation. Despite the fact that it may take place on a different machine, for experimental reasons, only one machine is used to accommodate both. The code of the Capture phase was incorporated in the simulation and the user with a click of a button can activate it.

In VECQB context, there is no need for polling a server to obtain updates or track any incoming packet. The latter actually happens but the simulation has an embedded mechanism that hides the details. This mechanism is called an Agile FOM Interface (AFI) developed at Lockheed Martin’s Inc. labs. AFI is an interface to the HLA protocol and handles the networking part of the simulation. The developer can write code that registers the events of interest to the AFI, and once an event occurs, AFI returns a notification, for example, when a platform update packet arrives, AFI notifies all registered routines about that event. In this manner, collecting data in VECQB works in an *event driven* fashion as opposed to *time driven* that requires more bandwidth, storage space and processing. For instance, in a time driven fashion, the procedure should log (sampling the simulation’s “world state”) every platform’s position and orientation many times a second even if the platform is not moving, whereas in an event driven fashion, only changes from the current state are logged. An event driven fashion filters out any unnecessary data and makes the process of data collection and storage more efficient.

VECQB virtual space (terrain) is divided into grids. Every instance of the simulation receives updates about platforms or other events occurring in the same grid with the local platform. This feature works in favor of the capturing procedure since it is acting as a filter trimming out any unnecessary data. There is no need to capture a fire event that took place a hundred miles away (in virtual space)! On the other hand, it is possible to obtain the whole picture of the simulation by simultaneously capturing every cell of the grid, and at the end, merging the data (Figure 5). This consists of a neat and clean way to capture a session in its entirety (all cells), without creating any “bottlenecks”, or stretching points in the simulation, since no instance of the simulation will have to anticipate the enormous volume of events, of the entire simulation session. It is important, though, to be careful of the merging procedure, and specifically, in cases where a platform changes grid.

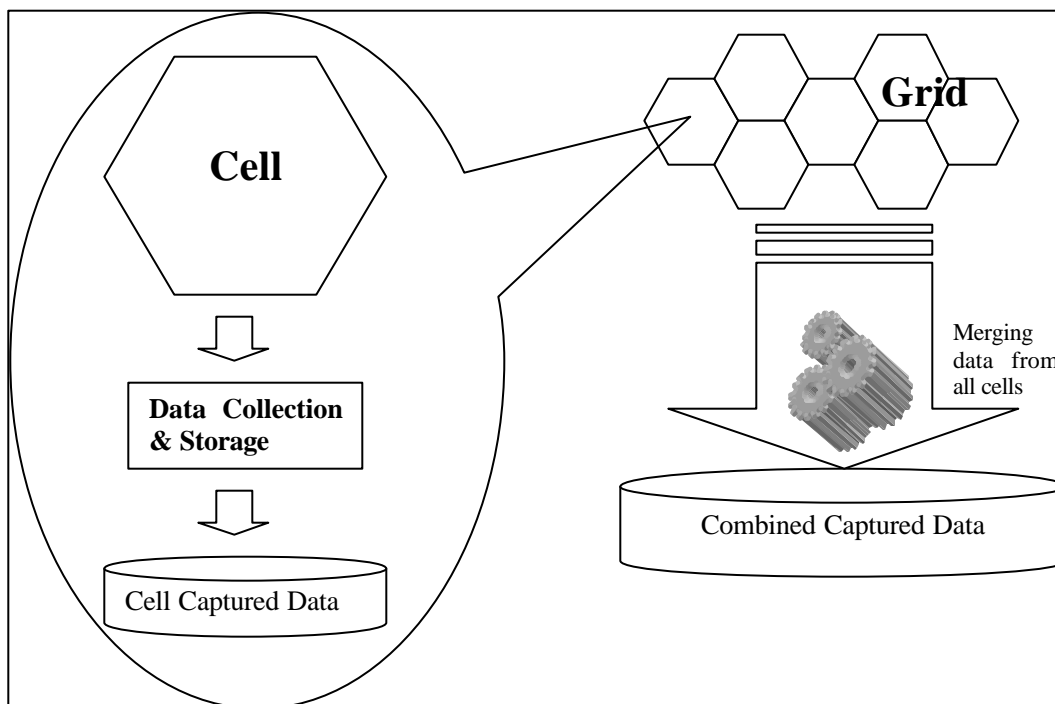


Figure 5. Data Collection and Storage Phase. Combination of captured data from all cells creates the entire picture

After specifying what data to collect, the next step is to ascertain what to do with it and how to store it, in terms of format. AARS exist that build a relational database after capturing the desired events. A database has certain advantages, allows for data

manipulation, and of course, in some cases, is the only choice. On the other hand, it takes some effort to design and it certainly takes much more time to build it in runtime.

XML technology is another choice that has some of the advantages of a relational database without very much overhead. XML appeared to be a viable and lightweight solution. Hence, in VECQB, the XML method is adopted and any captured event is written down to an XML file. The only overhead in that case is the XML element tags. As soon as an event arrives in the data collection procedure, it is wrapped into XML tags and forwarded into a stream. The stream keeps the data in a buffer, and from there, a thread that activates once every one or two seconds, flushes out the data in the hard drive. This turned out to be a quite robust design since it allows for infinite capturing without requiring too much of the system's memory or conducting too many write operations on the hard drive. Tags can be quite short, and especially those that wrap frequent events, such as a common platform update event, to decrease data volume.

In addition to XML format, the AARS is also able to store the data in a binary file. The binary file (**.virte*) is even more compact than the XML, since there are no tags. The only overhead, in this case, is just a short integer for every event record that is inserted right before the event's data, as a flag to specify the type of the event. This is necessary for recovering (reading) back the events from the file. Since not all of them are of the same type, they all are of different lengths and the use of a flag is the only way to read them back [Deitel & Deitel]. Binary files, of course, are not readable from the user, and they are not flexible in terms of integrity, transformation, and data manipulation. Two utility programs were incorporated in the AARS to interchangeably convert between XML and binary format. Thus, at runtime, it is possible to save the data in a binary file with just one short integer per event overhead, and then after the end of the Data Collection and Storage phase, to convert it into XML format. A successful validation of the produced XML file over the simulation's XML schema signifies the fact that everything worked fine during the Capturing phase, no errors are present and the file is ready to be forwarded to the next phase.

3. Playback Phase

The Playback phase is the second major phase of the VECQB AAR System. It combines both the Process and Analysis phase and the Presentation phase of a typical

AARS design. It only takes place in AAR mode, which means that the user cannot engage in any type of simulation session but can only review what happened in a previously captured one.

The AARS mentioned for VECQB was designed to provide a number of features-functions to the user, and the user decides whether to use each one of them, or not. A very simple graphical user interface (GUI) helps the user to interact with the simulation, choosing among various operations like:

- Summary report for each platform
- Show or hide a platform's path
- Specify the color of a platform's path
- Use the interactive playback's functionalities (interactive playback will be discussed separately in a following section)

These features are not the only features that can be added. They are only just the ones considered more important and “must” have in VECQB. Any operation in the Playback phase occurs in the imported file. The imported file contains all the events captured during the Capture phase. The file can be in either XML or binary format. The format determines what temporary files will be generated. The Playback phase needs two kinds of files:

- An XML file. Needed in any case for validation and data processing (applying XSLT documents, etc.)
- A file that contains the event's “raw” data and is going to be used from the interactive playback routines to read back the events

The second file can be either a binary file (*.virte), or an ASCII file (*.sdf, stands for simulation data file) which has exactly the same format as the binary, an integer as a flag to specify the type of the event and then the event data, but is just mere text. File types will be discussed in more detail in a later section) Although only one file type can be used for this purpose, for exploratory reasons regarding the file size and efficiency in reading back the events, both binary and text were used in the simulation. No formal research has yet been conducted, but the first impression was that the binary file is preferable since the data is more condensed and leads to the smallest possible file size.

Hence, if a binary file is imported, its XML version will be generated and validated. If no errors occur during validation, the application will use the binary to

stream-in the events and the XML for further data analysis. On the other hand, in case an XML file is imported, after validation succeeds, a text file will be generated (*.sdf) for the interactive playback's routines.

a. Summary Report

A summary report is a feature incorporated in the Playback phase. Its purpose is to inform the user about a platform's activity in the simulation. Currently, the report contains the following information:

- Platform's code name
- Platform's unique identity number
- Number of shots (primary ammunition)
- Number of shots (secondary ammunition e.g. grenades)
- Number of hits
- Number of mobility kills
- Number of operational kills

The mechanism that supports the summary report is very simple and involves two XML documents applied to an XSL Transformation producing a text file (*.stat) which contains the report (Figure 6). The application, in its turn, simply presents the contents of that file to the user. The two input XML documents are:

- The XML document that contains all the captured events
- A two-line XML document that is generated "on the fly" (as the user selects to view a platform's summary report) containing the id number of the platform.

The contents of the resulting document can be simply altered to meet any future needs by only changing the XSLT document, without having to change the C++ code at all. It is even possible to change the report to be an HTML, or any other XML supported document.

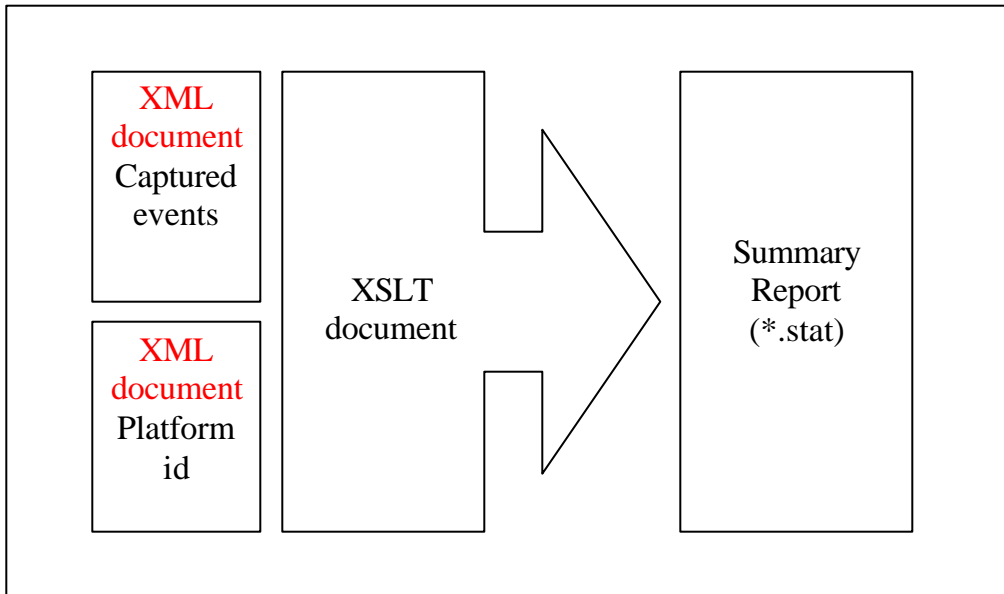


Figure 6. Mechanism that Generates the Summary Report. An XSL Transformation is applied to the two input XML files.

b. Interactive Playback

The ability to playback an entire simulation session was one of the main reasons for developing an AARS for VECQB. There are many ways to create a playback of a simulation. A very popular way, widely used in the gaming industry, is to generate a video file by capturing the frame buffer many times per second. Some video games even use MPEG2 format to decrease video size. This method works rather reasonably for video games but is considered obsolete for an AARS. An AARS needs an interactive playback that allows the user to change the point of view properly to ensure a better view, and not just a passive video stream. During the AAR session, all the participants should be able to see the playback of the events without anything obscuring their view.

In a video game, the main hero, the center of the world, is the player and the playback video is captured from such viewing angles that always keep the player in focus. On the other hand, a networking training simulation is completely different. There are many platforms and all of them require the same attention. In addition, many of them are deployed at different locations. Some may be near a building while others are near a beach or inside a tank, for example. The AARS should provide such a playback so that the actions of any platform can be observed. The idea of capturing a video file will not work well here, since the viewpoint cannot be in more than one location at a time, to

cover every platform. On the other hand, capturing a separate video file for each platform does not scale well either, since it is an extremely “expensive” operation. Capturing just a single video frame requires the entire scene be redrawn in the frame buffer in addition to the normal simulation rendering procedure. For instance, consider a “racing car” type of video game that provides a playback showing the car from above, say 30 feet height, the player though, sees the world behind the steering wheel. In order to produce the video playback, the scene showing the car from above must be rendered, in addition to the one that the player sees. When this procedure is repeated for every platform in the simulation, it becomes apparent that it is extremely inefficient, if not infeasible.

Another idea was adopted in VECQB. The simulation starts up in AAR mode, which is almost the same configuration as Normal mode except for the networking part, and the captured events are being launched one by one at the proper time. This approach is based on the already existing simulation infrastructure. It stimulates the system by firing up the captured events as if they were happening now. The system “knows” how to handle them and takes care of everything else, like physics, collision detection, collision response, animations, etc. Launching an entire sequence of captured events can regenerate the entire captured scenario.

All the remote and local captured platforms now reside on the same machine. The local avatar platform used in Normal mode to represent the “first person shooter” has now been given a different role, the one of “first person observer”. The user can “drive” the local avatar, which now represents the viewpoint, but cannot fire any weapon or change the flow of events in any way, to wherever he wants. The capability of wandering in the reproduced scenario and the ability to pause the playback and reposition the viewpoint to a new location, allowing for a better view, are what make the playback **interactive**. This method allows the user to see a scenario from many different angles, eliminating any ambiguities. A participant can see his actions from a different perspective and understand their impact. It is even possible to see how his actions look from the eyes of the enemy!

c. Other Features

During the interactive playback, the user, except for wandering around in the re-produced scenario, can also set the point of view at any platform in the simulation, and it is even possible to adjust the offset. Hence, the user’s viewpoint is not limited in the Playback phase.

Sometimes, while analyzing and discussing a platform’s role, except for the summary report, it is very helpful to see the entire path that the platform follows. Thus, the capability of drawing a platform’s path was added. The path is depicted in the simulation as a continuous line that connects all the platform’s waypoints. An XSLT document is being applied to the imported XML document to generate a text file (*.path) that contains all the waypoints.

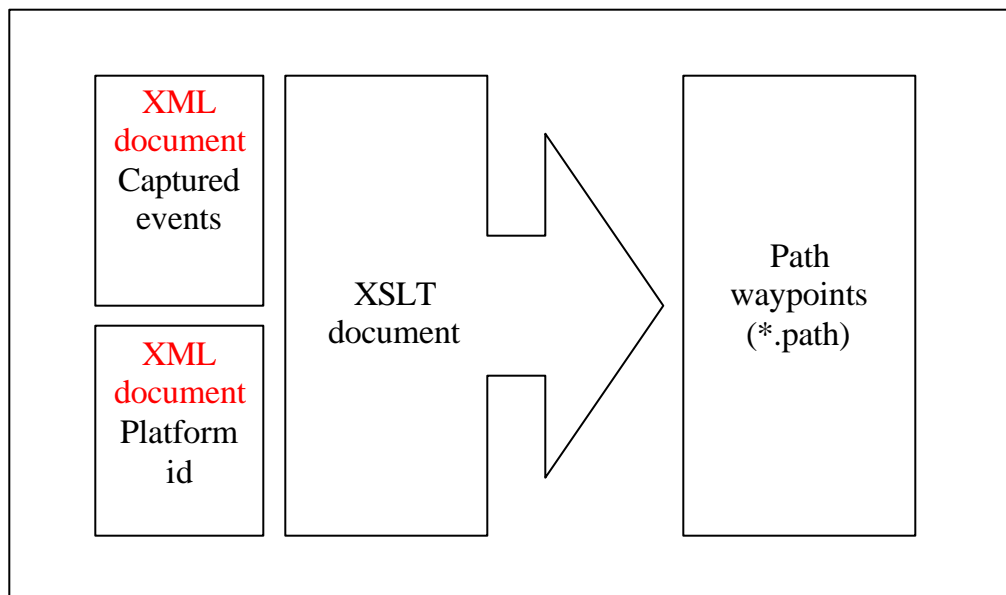


Figure 7. Mechanism that Generates the Platform’s Waypoint File. The appropriate XSLT document is being applied to the two input XML files.

The application then uses the waypoint file to draw the connecting line. Since the waypoints are derived directly from the position field of the platform’s Update events, some filtering occurs to trim out any repetitions or points that are too close to each other. The filtering defines a threshold value as the minimum distance between two successive waypoints. If A, B and C are three successive waypoints, the distance between

A and B is tested over the threshold value. In case the distance is greater, the line between those two points is drawn, otherwise B is discarded, and the distance between A and C is tested. The threshold value is chosen in such a way as to ensure that the rendering part is not overwhelmed with tens of thousands of vertices for drawing just a line, and on the other hand, to guarantee an accurate path. The path color can be changed allowing the user to specify a different color for each platform, and of course, the path can be easily removed from the screen.



Figure 8. Screenshot from the VECQB Simulation Demonstrating a Platform's Path.

The entire playback can be viewed from users at remote sites. A remote user has only to start the simulation in Playback mode and join in the same federation. The firing of events in the machine that hosts the playback will be visible from any other machine within the federation. The remote users, of course, cannot control any aspect of the playback as they do not have ownership over the platforms, but they can still wander

around in the re-produced scenario. Hence, it is possible to conduct AAR over distances using the interactive playback, in addition to other communication setups, a teleconference, for example.

4. The Role of XML

XML provides complete control over the captured data. Every event encapsulated in XML tags can be considered a record in a relational database. Applying an XSLT transformation to the XML file substitutes a typical database query. For instance, it is possible to extract a new one from the captured file that includes events of a particular platform or generate files containing information about different groups of platforms, such as dismounted infantry, tanks, ships, etc. In a VECQB simulation, the Xerces Parser for C++ [7] and Xalan for C++ [8] were used. They are both very powerful and cross-platform tools that are freely distributed on-line by Apache. They give the application capabilities for managing XML files, such as validation, transformation, read from and write to an XML file.

Neither Document Object Model (DOM) nor Simple API for XML (SAX) was used to read from or write to an XML file because they are considered unnecessary overhead that consumes the system's resources, especially in the Capture phase. The overhead in the Playback phase is also important as the size of the imported file increases. C++ standard methods for reading from and writing to a file were used.

a. Validation

An XML schema was written to validate the captured data ensuring that no invalid data is going in and out the simulation. The XML format is just plain text and thus readable from anywhere. This is important since it gives the user access to the real data. The XML Schema is also useful for authoring an XML file, because it can be used as a template by many products on the market and prompts the user for the right input. Thus, if the user wants to author a simulation scenario, or just setup the initial position, orientation, type and state of every platform in a large-scale simulation exercise, the XML schema simplifies the procedure. A more sophisticated graphical environment for authoring an exercise scenario is seriously considered as the next step of the VECQB AARS.

b. Transformations

XSL Transformation is a powerful feature of XML. It is a way to transform an XML file into an HTML, text or another XML file, and of course to various other types with the use of special plug-ins programs. VECQB's AARS uses XSL Transformations in a manner to control, convert, reshape, and, generally, manipulate the data. Thus, many XSLT documents have been written for that purpose. Actually, in any case a subset of information is needed from the captured file, an XSLT document is applied. If the captured file is considered a database and the captured events as the records in this database, then the XSLT documents are the queries to the database.

c. Error Handling

Error handling in an XML document is an important issue, and in the VECQB, is becoming even more important because just a simple error can cause the simulation to crash. Thus, given that the XML schema is correct, all errors must be caught during the XML validation procedure, and furthermore, if an error occurs, the system must produce a helpful message. For that reason, the error handling mechanisms of both products (Xerces and Xalan) were extended in the VECQB to provide the user with even more sophisticated error messages.

5. File Formats

The captured events are saved in XML or binary format depending on user settings. The default is XML.

a. XML Format

The structure of an XML captured file includes a root element called "Simulation". The first child element of "Simulation" is always the "Start" element, which contains information about the date and time, in simulation context, of the captured session. The "Start" element will be used in the Playback phase to set the scenario's date and time. The system will then appropriately configure the environment, for example, global illumination, moon, and sun position. After the "Start" element, any type of event can follow, usually some "PassiveBody" events to setup the objects, such as tables, chairs, doors, and boxes, on the terrain, then some platform "Creation" events and subsequently, the sequence of events is almost random.

The events are wrapped inside XML elements. The “Event” element is directly a child of root and wraps any other event. “Event” has an attribute called “time”, which in the Playback phase specifies the time of event occurrence in terms of passed ticks from capturing starting time. Children of the “Event” element can be any type of event, such as the platform “Creation”, “Update”, “Destroy”, “Fire”, “Detonation”, etc. The common attribute among these elements is “id”, which represents the id of the platform that is related to the event, for example, the platform that causes the event.

```

<?xml version="1.0" encoding="UTF-8"?>
<!--C:\Thesis\VEQCB_IFE2\Capture\Capture.xml-->
<Simulation xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\Thesis\VEQCB_IFE2\XML\SimML.xsd">
  <Start year="2003" month="6" day="20" hour="9" minute="45" second="34"/>
  <Event time="0">
    <RBody id="192138464" modelGuise="OfficeTable">
      <Pos Px="26698.1" Py="130641" Pz="10.9554"/>
      <LinVel Vx="6.44401e-007" Vy="-1.96502e-007" Vz="2.43342e-005"/>
      <AngVel Vx="3.53784e-007" Vy="1.16018e-006" Vz="1.98898e-013"/>
      <Rot Rw="0.965926" Rx="-1.57335e-008" Ry="2.96504e-009"
Rz="0.258819"/>
    </RBody>
  </Event>
  <Event time="0">
    .
    .
    .
  <Event time="19.6824">
    <Update id="2" cell="-842150451">
      <Pos Px="26710.3" Py="130640" Pz="10.4363"/>
      <Vel Vx="0" Vy="0" Vz="0"/>
      <Rot Rw="0.820409" Rx="0.0071596" Ry="0.00498924" Rz="0.57171"/>
    </Update>
  </Event>
</Simulation>

```

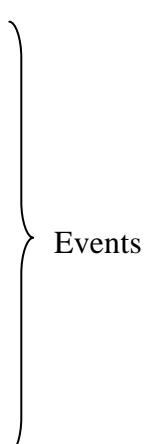


Figure 9. View of an XML Captured File. The document root element is Simulation. Then the Start element follows and a sequence of Events.

b. Binary and ASCII Text Format

The binary format contains “raw” event data (Figure 10). It is the most compact captured file since it does not contain significantly excess information such as the XML file. The file is generated using the standard C++ input/output streams. The structure starts with the date and time of the captured scenario (“Start” element in XML format), and continues with the event data. Since the events are written sequentially, and there are many event types of different structure and length, a flag was inserted before each event’s data to specify the type. During the read procedure, the flag informs the

application what type of event to read next from the stream. Furthermore, how many bytes to read and what object to generate using those bytes is also determined.

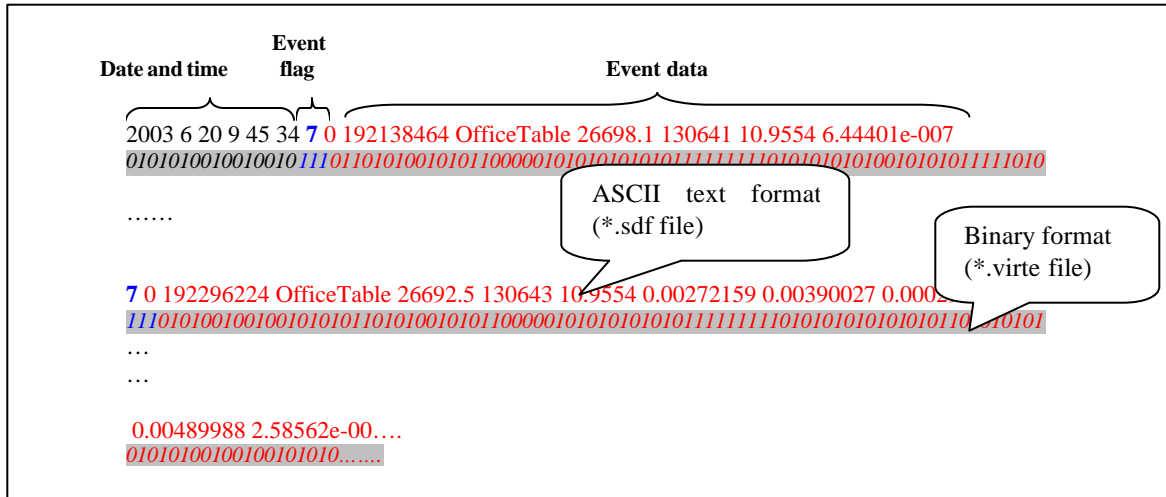


Figure 10. A Representation of Binary and ASCII Files Line by Line. The binary file is a continuous binary stream. The ASCII has the same structure as the binary except for the spaces between the data values.

The ASCII text file has the same structure as the binary file except that the information is in ASCII format and there are spaces between the numeric values. It is generated during the AAR mode in case an XML file is imported by applying an XSLT document to the imported file. What generally happens next is that the XSLT strips-off the XML tags. The ASCII file is used from the playback as a stream to read back the events. It is possible to use a binary file (*.virte) instead and not have to use ASCII at all. This is exactly what happens when a binary file is imported since binary is used to read back the events, but the ASCII format is used during development for experimental purposes to see how the simulation behaves and how the frame rate is impacted using different file formats. Conducting formal research on this topic should be considered future work.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. AARS IMPLEMENTATION

A. INTRODUCTION

1. Overview of VECQB Source Code

The VECQB source code currently consists of 22 modules, including the modules of the AARS, and a number of supporting libraries. The modules are classes following the singleton pattern. They constitute the application's backbone, and each one handles a specific aspect. Every module is instantiated when the simulation starts. The initial values for every module are read from a text file, and of course, each one has default values.

The modules are:

- **ModuleAFI:** Primary RTI (Run-Time Infrastructure of HLA) interface. This class provides all necessary interfaces for setting and retrieving RTI data.
- **ModuleAFIWorldState:** Notifies registrants with user input information.
- **ModuleVisual:** 3D visual windows
- **ModuleWorld:** Primary world state interface
- **ModuleVisDB:** Visual database
- **ModuleCollision:** Collision detection. Notifies registrants with collision detection information
- **ModulePhysics:** Physics detection. Notifies registrants with physics detection information
- **ModuleAvatar:** Avatar simulation
- **ModuleInput:** User input data
- **ModulePlatforms:** Platforms scene graph. This class provides the interface to the platform update support
- **ModuleVisEffects:** Visual effects
- **ModuleTexture:** Texture manipulation and screen text
- **ModuleModelVis:** Visual model scene graph. This class provides the interface to the platform update support
- **ModuleModelSim:** Physical simulation. This class provides the interface to the physical simulation support
- **ModuleMaterial:** Material detection
- **ModuleAudioDB:** Audio database

- ModulePlatformAudio: Create and track audio entity for platforms
- ModuleListener: Initialize spatial sound and track avatar
- ModuleEnvironment: User environment data
- ModuleOcean: Dynamic ocean visualization
- ModuleCapture: Event data capturing
- ModulePlayback: AAR and playback

The simulation is based on two schedulers, the Real-Time, and the Simulation-Time schedulers. The schedulers are software components that control the time of execution of any registrant function. Registering functions to schedulers is a very robust way to control the flow of execution in such a complex application. Through the schedulers, it is possible to set-up a:

- Periodic function. The function will be executed periodically, after a period's number of seconds have passed
- Phased function. The function will be executed in a specific order in relation to other phased functions
- Delayed function. The function will be executed after the specified amount of time has passed

It is also possible to set the execution priority of each scheduled function, and of course, remove any scheduled function from being executed.

Each scheduler is based on a separate timer. The timer of the Real-Time scheduler runs in real time, and the one for the Simulation-Time runs in simulation time respectively. A function that must be executed regardless from the simulation rate, e.g. visual rendering, is registered in the Real-Time scheduler. On the other hand, a function that must be executed with respect to the simulation time, e.g. detonation of a grenade, should be registered in the Simulation-Time scheduler.

The simulation, except for the working directory that contains the executable and a number of “.dll” (dynamic linked library) files, uses four additional subdirectories for various purposes:

- data. This directory includes files with geometric data for the models, textures, audio, material, network configuration (HLA federations), etc.
- XML. Includes the appropriate XSLT documents and the schema used by the AAR implementation.

- Capture. It is the default directory for storing a capture file.
- Workspace. The intermediate files produced from the simulation after an XSLT transformation are saved in this directory. Although these files are considered “garbage”, the user can review them and use them as input to other applications, for example, a spreadsheet.

2. Design

The implementation of an AARS design follows the same programming practices and guidelines as the rest of the VECQB source code. Extra care was given to fit the AARS’s functionality without making any changes to the rest of the code. Eventually, some minor changes in just a few isolated cases are required that were considered as absolutely necessary, such as for instance, a function that queries the simulation wall time from *ModuleEnvironment* every time a new capture session begins.

Thus, one module was inserted for each phase of the design. “ModuleCapture” and “ModulePlayback” were added in the VECQB source code to implement the Capture and Playback phase respectively. In addition to those modules, a library was developed, called “libXML”, to support the application in handling XML documents. Code was also inserted in the graphical user interface of the application to allow user interaction with the AARS such as importing a captured file, or set the output format, the name and location of a capture file, etc.

B. MODULE CAPTURE

1. Class Hierarchy

ModuleCapture implements the Capture phase in VECQB. It is a class that follows the singleton pattern, and it is derived from the *BModule* class as any other module. *ModuleCapture* provides classes to obtain notification, encapsulate, and store the occurring events.

a. Notification

ModuleCapture receives notification about any event in the simulation from the following classes:

- PlatformCaptureCB derived from AFIPatform::Callback
- FireCaptureCB derived from AFIFire::Callback
- DetonationCaptureCB derived from AFIDetonation::Callback

AFIPlatform::Callback, AFIFire::Callback, and AFIDetonation::Callback are abstract classes of *ModuleAFI*, which is an interface to the network. Any derived class from those three classes will be able to receive notifications about any platform, fire, and detonation event occurring in the simulation. In order to receive the events, the derived classes must be registered with the *ModuleAFI*, because only registrants can receive notification, and of course, any abstract methods must be implemented in the derived classes. The methods that are implemented for each derived class are:

- PlatformCaptureCB
 - *PlatformCreateEvent* (*AFIPlatform**) notification about any platform creation event
 - *PlatformUpdateEvent* (*AFIPlatform**) notification about any platform update event such as position, orientation, velocity, articulation, etc.
 - *PlatformDestroyEvent*(*AFIPlatform**) notification about any platform destroy event
 - *PlatformAppearanceChanged* (*AFIPlatform *platform, unsigned int old appearance, unsigned int new appearance*) notification about any platform appearance change. As appearance change is considered, any change in the platform animation, e.g. from walking to running
- FireCaptureCB
 - *FireEvent* (*AFIFire**) notification about any fire event
- DetonationCaptureCB
 - *DetonationEvent* (*AFIDetonation**) notification about any detonation event

The *AFIPlatform**, *AFIFire**, and *AFIDetonation** pointers passed into the methods point to objects that encapsulate all the information related to those events (fires, detonations, platform updates, etc.). Hence, that information is available to the methods for any further use. In the case of capturing, this information will be encapsulated inside other objects before it is stored in the captured file.

In *ModuleCapture*, when a capture session starts, instances of those three classes are created and registered with the *ModuleAFI* to receive any events. The platforms that already exist in the simulation are captured directly from the *ModuleAFI* that keeps track of them. Similarly, position and orientation of any rigid body is captured

by querying the *ModuleModelSim*, which maintains a list of all rigid bodies in the simulation. Thus, by the time a capture starts, the current state of existing platforms and rigid bodies is saved, and mechanisms that will notify the AARS about any future event are set up. The next step is how to handle those notifications, in other words, what to do when an event notification occurs.

b. Encapsulation

A hierarchy of classes is used to encapsulate only the necessary information of an incoming event and some extra information that is needed to reproduce that event later, e.g. the time of occurrence. The foundation of the hierarchy is the class *Event* (Figure 11). *Event* is a pure abstract class that specifies a simple interface for any derived class. The interface includes the following functions:

- *read (std::ifstream &, FORMAT)* reads the values of *this* event from the specified input stream
- *write (std::ofstream &, FORMAT)* write *this* event to the specified output stream
- *GetTime()* returns *this* events time of occurrence (simulation time)
- *Launch ()* triggers *this* event by sending an interaction to the system. The system responds to the event as if it is happening now

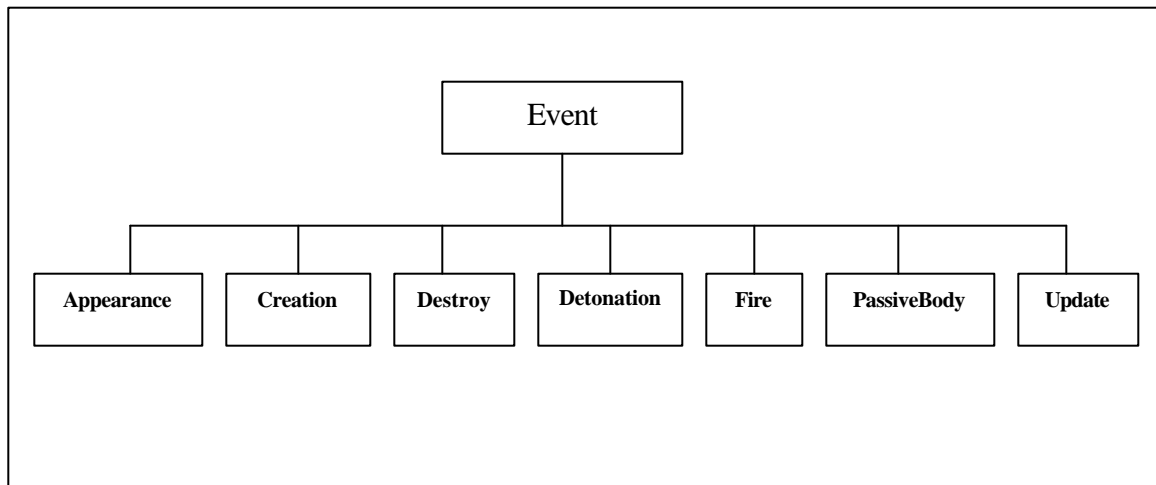


Figure 11. Event Class Hierarchy. Event is the base pure abstract class and the others are deriving from it.

The parameter *FORMAT* in *read* and *write* functions specifies the format type of the stream. The input or output stream can be either in binary or ASCII format. Hence, the function becomes aware of the stream type that it must handle and uses the appropriate methods each time.

The derived classes are mapped directly to the type of the events to be captured. Thus, the following classes were defined:

- *Appearance*: maps to the appearance changed event that changes the current animation of a platform
- *Creation*: encapsulates a platform creation event
- *Destroy*: encapsulates a platform destroy event
- *Detonation*: encapsulates a detonation event
- *Fire*: encapsulates a fire event
- *PassiveBody*: encapsulates information about a passive rigid body in the simulation. This type of event is actually captured right at the start of a capture session and triggered first on a playback to set the passive bodies appropriately. As passive bodies are considered, the bodies that are not driven by the user or the computer, e.g. an office table, a chair, a door etc., are just sitting there and passively respond to any interaction.
- *Update*: encapsulates a platform update event

Each of the derived classes has a different set of data members depending on the event they are meant to encapsulate. Hence, all of them are of different lengths, and all must implement the functions of the *Event* class, since the latter is a pure abstract class.

In the capturing phase, each time a notification about an event arrives, objects of the *Event* hierarchy classes are generated “on the fly” and then are saved to the specified format. The object generation occurs inside the functions of the registered class with the *ModuleAFI* classes for receiving notification as follows:

- *PlatformCreateEvent (AFIPlatform*)* generates a Creation object
- *PlatformUpdateEvent (AFIPlatform*)* generates an Update object
- *PlatformDestroyEvent(AFIPlatform*)* generates a Destroy object
- *PlatformAppearanceChanged (AFIPlatform*,...)* generates an Appearance object
- *FireEvent (AFIFire*)* generates a Fire object

- *DetonationEvent* (*AFIDetonation**) generates a Detonation object

c. Storing

In the beginning of a capturing session, an output file stream is created to save the events. Actually, the format type determines the type of the stream that is created. The default is XML. There are two output file streams in the simulation, and both are derived from the *FileOutputStream* class:

- XMLFileOutput
- BinaryFileOutput

FileOutputStream wraps the *std::ofstream* class and provides an interface for the two deriving classes. The *XMLFileOutput* stream opens a stream in text mode and writes some XML initial headers, XML Schema field and some comments in the capture file. Then, it is ready to receive any other information from the application. The *BinaryFileOutput* initiates a *std::ofstream* in binary mode. Both streams implement the pure virtual functions *Start* and *Stop* of *FileOutputStream* for writing the start and stop time of the capture (in simulation wall-time).

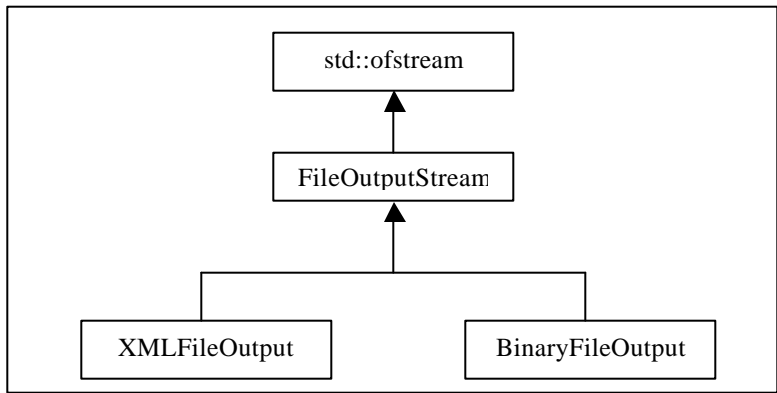


Figure 12. Stream Class Hierarchy.

Additionally, a periodical function is scheduled in the simulation scheduler that flushes the stream out to the hard drive. The stream, before being flushed out, keeps the data in the system’s memory. In order to achieve the least possible impact to the simulation, the period must be adjusted to such a value that not very many physical writes are executed, while not very much of the system’s memory is used to keep the stream’s data.

2. File Conversion

ModuleCapture, except for capturing the events in a simulation, also provides utility functions to convert a captured file from one type to another. File conversion is important since the user can convert a binary file and be able to read its contents, or even better, validate it over the simulation's XML Schema document. If a converted to XML file does not pass validation, this does not mean that something is wrong in the file necessarily, but if it does, then it is a good indication that everything is right and no file corruption or error fields exist. The XML version of a file also gives the user the ability to apply the XSLT documents and derive any kind of information. On the other hand, the binary file is more condensed without any XML tags and headers. A typical binary file containing approximately five minutes of simulation capturing that contains a couple of platforms is four times smaller than its XML equivalent. As the number of platforms, the events they generate and the duration of a capture, are increasing, this difference in size between the two versions deviates even more.

There are two functions to convert between file types. These are:

- *ConvertToXML (const char * binaryFile)*
- *ConvertToBinary (const char * textFile)*

ConvertToXML opens a standard binary input stream (*std::ifstream*) and reads the events off of the stream. The events then are saved in a separate file, using an *XMLFileOutput* stream, in XML format. The reading is done as follows:

- At first, the start time is read from the binary stream and is written in XML format by calling the *Start* function of the *XMLFileOutput* stream.
- Then to read each event, the event type flag must read first
- According to the flag, the appropriate Event object is created as follows *Event* event = new Update ();*
- Using polymorphism, the read function of the event is called passing the binary input stream and the type, which in this case is BIN (for binary). The read function will then read the values from the stream and assign them to the events attributes e.g. *event->read (binaryStream, BIN);*
- The next step is to save the event in XML format which is done by calling the write function, e.g. *event->write(xmlStream, TEXT);*
- The event is no longer needed after the write function so it is deleted to de-allocate resources, and the next event is then read until the end of the file.

The *ConvertToBinary* function works exactly the same way. In order to convert an XML capture file, the application at first applies an XSLT document to strip off the XML tags and then passes the produced text stream to the *ConvertToBinary* function. The *ConvertToBinary* follows exactly the same steps as the *ConvertToXML* but, now it reads the events from a text stream, e.g. *event->read(textStream,TEXT)*; the TEXT flag is used instead of BIN and saves them in a *BinaryFileOutput* stream by making the following call: *event->write(binStream,BIN)*;

3. Capturing

The sequence of actions that *ModuleCapture* does during a capture is the following (in time order)

- Opens an output stream for saving the capture. The stream of the type, and location (path) in the hard drive are determined by the user (default values are XML and “<working directory>\Capture” respectively)
- Queries *ModuleEnvironment* about the simulation wall time and saves it in the output
- All passive rigid bodies are logged into the output (as *PassiveBody* events). *ModuleModelSim* has a list that keeps track of them
- All platforms currently in the simulation are logged into the output (as platform *Creation* events). *ModuleAFI* maintains a list that keeps track of them
- The appropriate callback objects are instantiated and registered for further event notification
- A periodic function call is scheduled to flush the output stream to the hard drive

When the capture stops, the callback objects are unregistered, the periodic function call removed from the scheduler and the output stream is closed.

C. MODULE PLAYBACK

ModulePlayback implements the Playback phase of the AARS design. It is derived from the *BModule* class and adopts the singleton pattern as well. Its purpose is to provide the appropriate functions to conduct process and analysis of a simulation capture file, and playback the whole sequence of events. This module is initiated in the AAR mode of the simulation. The user must import a capture file (*.xml, or *.virte), otherwise, the simulation is going to start in Normal mode. *ModulePlayback* internally uses two kinds of files to operate, a file that contains the sequence of the events that will be used to

stream in the events and the equivalent XML document for any further use such as a summary report, platform path, etc. The application, depending on the type of the imported file, generates the other one.

1. Playback

In order to read the captured events, *ModulePlayback* uses the *Event* class hierarchy defined in *ModuleCapture* (*Appearance*, *Creation*, *Destroy*, etc.) to read and trigger the events. In fact, all these classes have been declared friend classes in the *ModulePlayback* class definition.

Once *ModulePlayback* is initiated, or in case a new capture file is imported, the function *Start* is executed to do some extra initializations:

- Call the *Reset* function to clear the simulation level. The *Reset* function destroys all platforms and rigid bodies, removes any previous scheduled callbacks, and resets the camera position back to the local avatar
- The next step is to generate the second needed file, according to the one that is imported
- The simulation's wall time is adjusted (by calling the *SetSimTime(tm & time)* function of *ModuleEnvironment*) to be the same as the scenario's wall time that is read directly from the imported file. Thus, the simulation is configured to have the same environmental settings (e.g. global lighting, moon phase, sun position etc) as the captured session
- Two periodic function calls are scheduled with the simulation scheduler, one for reading the events (*Read*) from the imported file and one for triggering the events (*Tick*)

a. Read Periodic Function

The *Read* periodic function call reads the events off of the imported file the same way as the *ModuleCapture*'s functions *ConvertToXML* and *ConvertToBinary* do. This is a long switch statement that, according to the event flag, creates and reads the appropriate event. As the events are created, they are stored in a standard list (of type *std::list<Event*>*) that is a data member of the *ModulePlayback* class called *m_eventList*. The function is scheduled to be executed twice a second. Each time it reads a finite number of events, currently set to read the next 500 events, from the capture file and returns to the "sleeping" state. The scheduling allows for an immediate start of the playback. There is no need to wait until all the events are read in order to start the playback, which may take a while, especially if the capture file is big enough.

b. Tick Periodic Function

The *Tick* periodic function is scheduled to execute in every tick of the simulation timer. This function triggers the *m_eventList*'s events and it is not executed until the user presses the "Play" button to start the playback. The mechanism (Figure 13) used to trigger the events on the appropriate time involves a list iterator that starting from the head iterates through the nodes (*Event* objects) of *m_eventList* and a variable called *m_timeOffset* that keeps the simulation time when the playback starts playing. The list iterator is a data member of *ModulePlayback* and is declared as follows:

```
std::list<Event*>::iterator m_eventItr;
```

The *m_timeOffset* variable is used to calculate the offset playback time from the current simulation time. The simulation still keeps the wall and simulation timers in AAR mode. The *m_eventList* contains the events in the same order as they were read from the capture file, which in turn, preserves the same event order, as they occurred in the simulation session. If the event timeline is preserved, there is no room for errors in the playback, e.g. changing the order of events and, of course, there will not be a problem in rearranging them in the right order. The *m_eventItr* points to the first event of the *m_eventList* at the time the playback starts playing. At that time, when the user clicks-on the "Play" button, the *m_timeOffset* is assigned the value of the simulation timer, and any further time reference in the playback is calculated from that time:

$$\text{time} - m_timeOffset$$

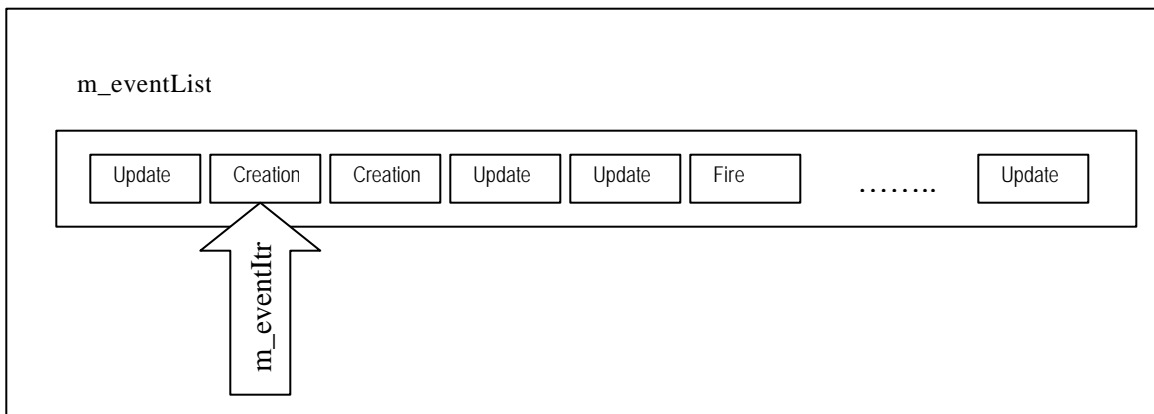


Figure 13. *m_eventList* Contains Pointers to the Events. *m_eventItr* points to the event that it is going to trigger next

The *Tick* function starts execution, and on every tick, just one comparison is conducted. The comparison compares the current playback time with the time of the event that is pointed by the *m_eventItr* as follows:

```
*m_eventItr -> GetTime() >= Current time - m_timeOffset;
```

If the playback time gets greater or equal than the time of an event, then the event must be triggered, for instance, if an event has a time field of 10.000 seconds (in simulation time) then during playback, this event must be triggered when the playback time becomes 10.000. In practice, since the time clock uses ticks, there is some tolerance on event execution, but in any case, the maximum error is not more than one tick or 1/30 of a second.

The triggering of an event occurs by calling the *Launch* function as follows:

```
*m_eventItr->Launch()
```

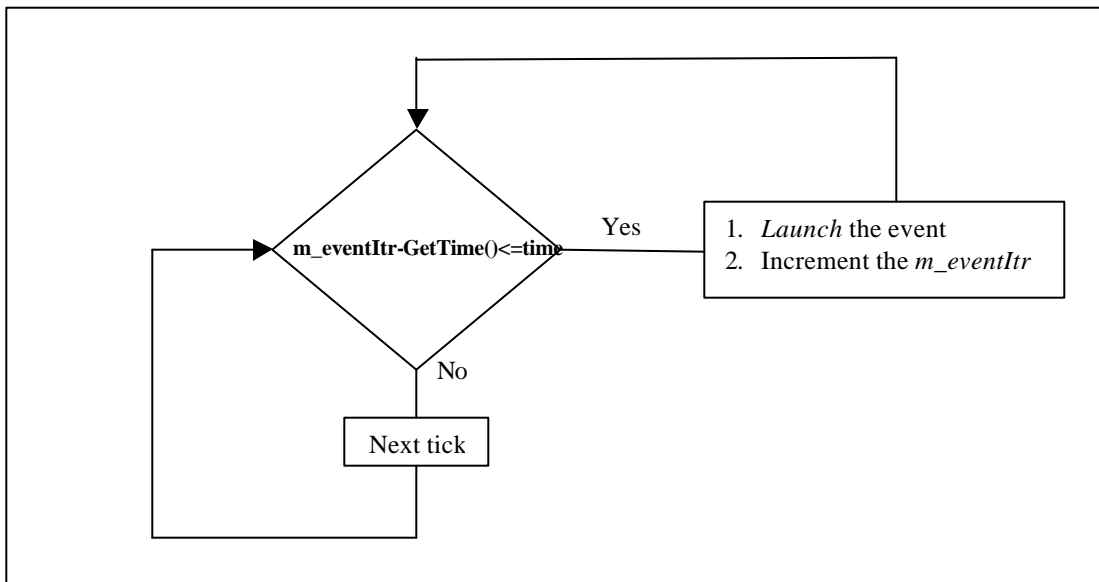


Figure 14. Box Diagram of the Event Launching Cycle

Depending on the type of event, the *m_eventItr* pointing to the appropriate *Launch* function will be executed (polymorphism). For instance, if the event is a *Creation*

event, then a new platform will be created. After launching an event, the *m_eventItr* goes to the next event in the list and checks again (during the same tick) if the next event must also be triggered (Figure 14).

2. Managing the Platforms

As the events are spawning at the proper time by the *Tick* function, there is a problem that has to do with platform management. In the Normal mode, the simulation has own-ship only over one platform that is controlled by the user. In AAR mode, the simulation has own-ship over any platform, and thus, should “drive” all of them. In addition to managing the platforms, *ModulePlayback* should manage the rigid body representation for each platform and also the passive rigid bodies. There is also another technical problem concerning the platform id number. The system assigns a unique number to a newly created platform, which is different on each run. Hence, there must be a mechanism that maps a platform’s id number as it appears in the capture file to its new id number that the simulation playback assigns to that platform. Therefore, when a platform with id, say 1234, causes an event, and in playback is assigned an id of 4321, the appropriate mapping occurs. Thus, the event is still caused by the same platform.

In order to overcome these problems, a standard C++ map container is used that has the platform id numbers as keys as they appear in the capture file and as values pointers to *PlatformNode* objects. The map container is declared in the *ModulePlayback*’s class definition as follows:

```
typedef std::map <int , PlatformNode*> PlatformMap;  
static PlatformMap    m_platMap;
```

PlatformNode is a C++ structure that has an *AFIPlatform** and a *ModuleModelSim::Body_Handle* as attributes, which is a handle to a rigid body.

The *Launch* function of a *Creation* event creates an *AFIPlatform* and a rigid body for that platform, and then encapsulates them in a *PlatformNode* object. A pointer to that *PlatformNode* object and the platform’s id number (the one in the *Creation* event) are inserted as a pair to the *m_platMap*. The *Launch* function of the other events

(Appearance, Destroy, Update etc) can query the *m_platMap* to gain access to the appropriate *PlatformNode* and from there do the appropriate changes to the platform and the rigid body.

PlatformNode is used also for some secondary operations such as inserting or removing the platform's name from the list box in the GUI, or keeping a pointer to the platform's path since it is a node in the scene-graph so that the path is managed via the *PlatformNode*. In other words, from the *PlatformNode*, it is easy to manage any aspect of a platform, the network representation (*AFIPlatform*), the physics representation (rigid body), its path in the simulation, update the GUI, and also provide functionality to safely destroy a platform.

D. XML LIBRARY

In order to be able to introduce XML capabilities into the simulation, a library was written (libXML) on top of Xerces C++ Parser 2.4 [7] and Xalan-C++ 1.7 [8]. Both projects are under the open source public license of Apache.org. The libXML allows the application to validate an XML document over a schema and also allows for XSLT transformations. Additionally, a helpful error handler was written, in order to help the user locate any errors. The library includes the following two major functions:

- *bool Validate(const char * xmlfile, const char* schemaFile, char chkMsg[]);*
- *bool Transform(const char *xmlFile, const char*xsltFile ,const char*outFile ,char chkMsg[]);*

The first function validates an XML document over the given schema. If the schema location is declared in the XML document, then this location is used internally from the parser to locate the schema file. Otherwise, the passed schema document is used. The *Transform* function takes the XML document, the XSLT document, and a file as input to store the product of the transformation.

In both cases, a buffer is passed to store any message from the operation, either successfully if there is no error, or information about the last error that occurred in the validation or transformation procedure respectively. A class called *SimErrorHandler*, which is a wrapper of the Xerces API class *BaseHandle*, traps and stores (<working directory>\xmlError.log) a complete record of all warnings, errors and fatal errors

occurred in the procedure in a log file. The error logs come with the line and column of occurrence and a reasonable description. Both functions return true if the operation is completed with no errors or warnings. Otherwise, false is returned.

E. GRAPHICAL USER INTERFACE

The Graphical User Interface (GUI) of the simulation was altered from its original version to accommodate controls for the Capture and Playback phase. Currently, two windows constitute the GUI. One is the MANSim window that appears when launching the simulation. The other is available only in the AAR mode of operation and is a window called “Control Panel”.

The GUI in the main (MANSim) window includes a menu and two panels and a status bar at the bottom. The first panel is a list of all the modules in the simulation. It gives information about each one of them. The information includes the state (running, stopped, or paused) and the latency that each of the modules introduces to the system. Some modules give more information than just their state such as the number of the platforms in the simulation (*ModuleAFD*), or the frame rate (*ModuleVisual*). The second panel is a console used mostly for debugging purposes and provides more detailed information about the initialization phase of the modules and any other event that occurs in the simulation after that. The information may overwhelm the front end user but is valuable to the developer. The status bar at the bottom of the window gives the global status of the simulation, for example, Running, Capturing, or Playback.

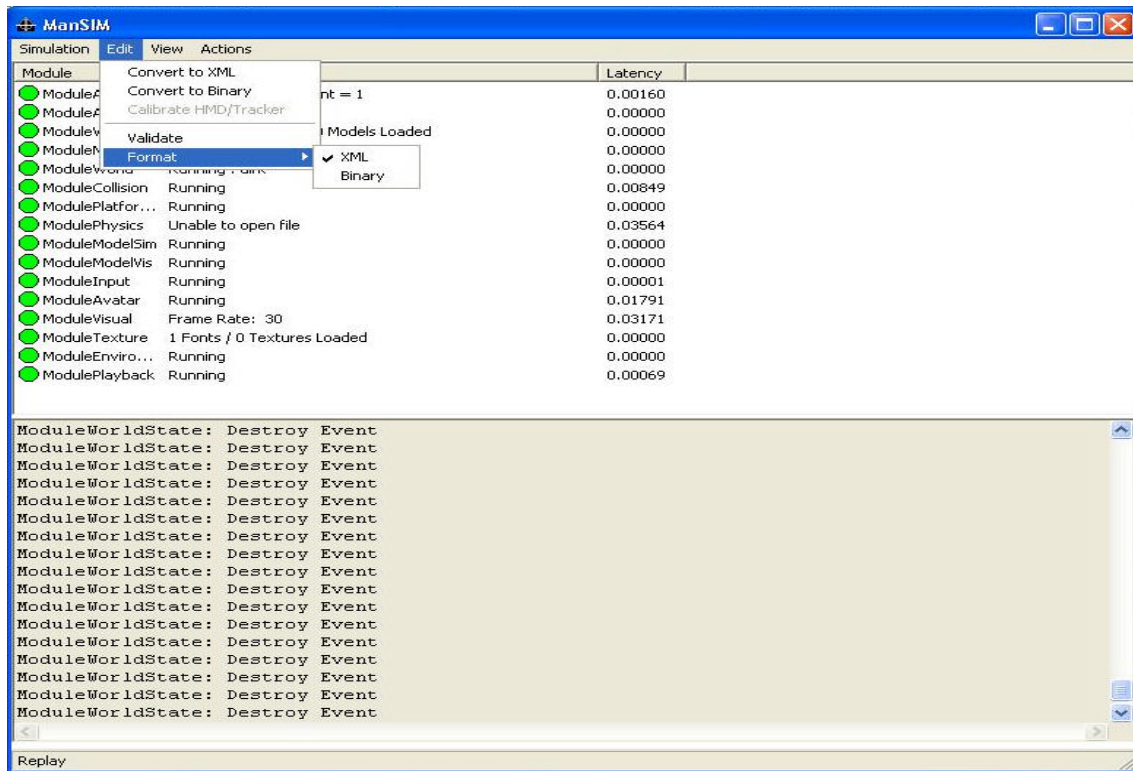


Figure 17. Screenshot from VECQB GUI. Edit Menu

c. View Menu Item

The View menu item currently contains only the “Control Panel” choice, which brings up a dialog window (Figure 18) that allows the user to control the flow of the playback and activate some extended features during the AAR mode of operation.

The Control Panel has three sections:

- Replay flow control. The user can control the flow of the playback
- Actions. Actions that the user can take in the simulation
- Console. Provides the user some feedback messages and the Summary Report

The Replay section has buttons such as Play, Rec, Stop, Pause that can start playing, recording (not yet implemented), stop, and pause the playback of the simulation respectively. There are also the (+) and (-) buttons and the Replay Speed slider that control the simulation speed. Clicking the (+) button makes the simulation play faster while pressing the (-) button slows down the playback, and the same applies to the slider. What actually happens when the playback’s speed changes is that the simulation timer

rate changes accordingly. Playing the simulation fast may result in rough movement of the simulation contents (platforms, object, etc.). This occurs because the time interval from frame to frame has been increased.

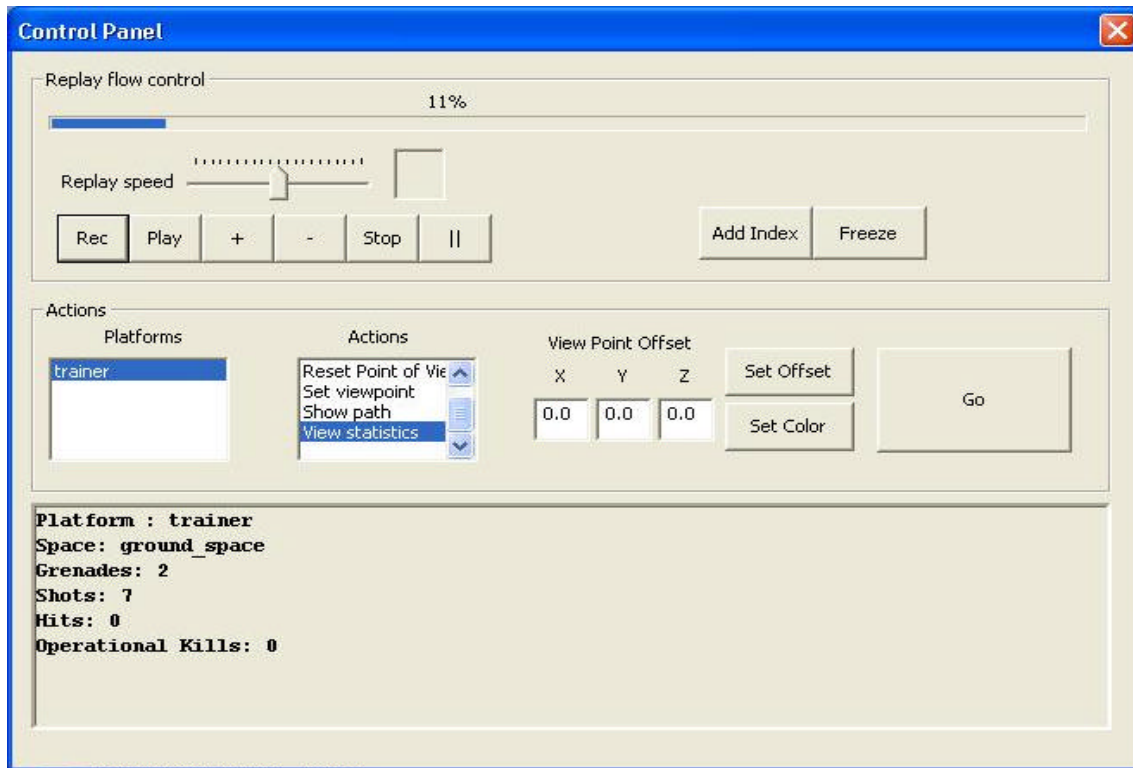


Figure 18. Screenshot from VECQB GUI. Control Panel Dialog

In the Replay section, there is also a progress bar that shows the percentage of the triggered events providing insight about the duration of the playback as well as the remaining time to the user. The progress bar is driven directly from the position of the `m_eventPtr` pointer, in the `m_eventList`. Two additional buttons are “Freeze” and “Add Index”. The first pauses the simulation timer and does the same as the Simulation→Pause selection from the main menu. The second is for future use and will insert an index point in the playback so that the user will be able to come back exactly to that point later. Both the “Freeze” and “Pause” buttons in the “Control Panel” dialog pause the simulation. However, the main difference is that the first pauses the

entire simulation so that nothing is able to move, whereas the latter pauses only the playback. Hence, the user can still move and wander around and maybe try to find a better view angle.

There are two list boxes in the Action section. One contains all the platforms currently in the simulation and the other the actions that the user can do to a selected platform. Thus, the user can select a platform from the platform list, an action from the action list and then press the “GO” button to execute the action over the selected platform. The actions are:

- *Hide Path.* Hides the selected platform’s path
- *Reset Point of View.* Resets the point of view back to the local avatar
- *Set Viewpoint.* Mounts the viewpoint to the selected platform. This feature allows the user to see the “world” from the selected platform’s eyes
- *Show Path.* Shows the selected platform’s path
- *View Statistics.* Show the Summary Report in the console (Figure 18)

The “Set Color” button opens up a color chooser dialog (Figure 19) for the user to choose a color. The platform paths drawn from this point forward will use that color. Hence, the user can assign a different path color to each platform. The “Set Offset” button changes the offset of the viewpoint from the platform on which this viewpoint is mounted. This does not work for the local avatar, but only works for the other platforms.

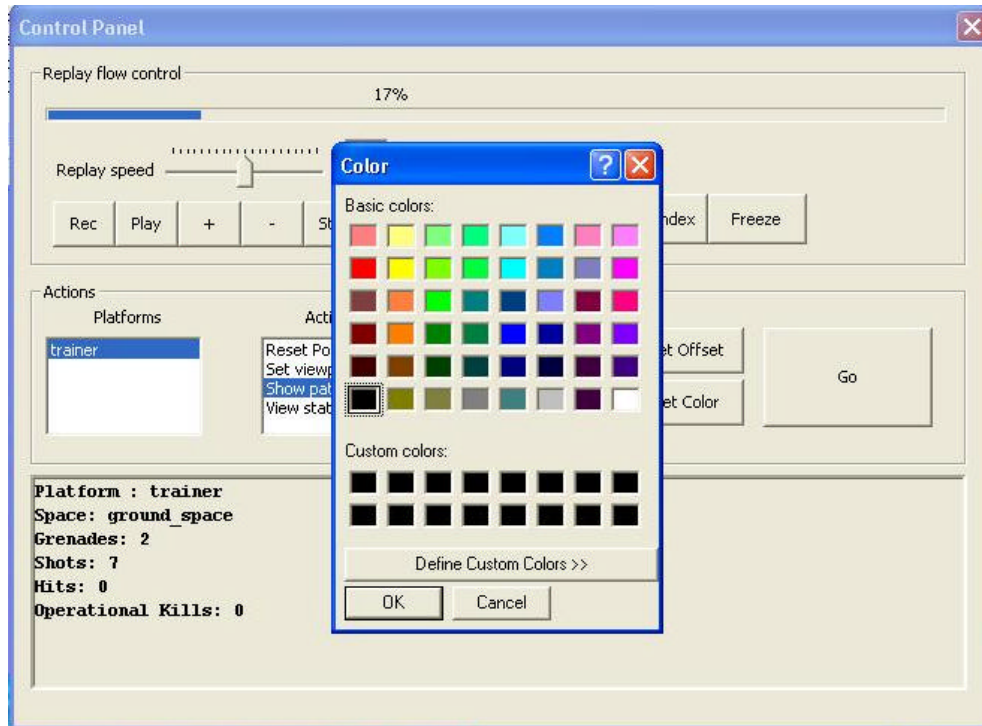


Figure 19. Screenshot from VECQB GUI. Control Panel

d. Actions Menu Item

The Actions menu item is used in both the Normal and AAR mode of operation to execute some generic actions such as capture or playback a session. This menu provides the following choices (Figure 20):

- *Rec.* Starts capturing using the specified format (default is XML). Saves the data into the specified capture file. If none is specified, it saves the result in the “Capture.(xml, or virte) in the Capture directory.
- *Play.* In the AAR mode, it starts playing the imported capture file
- *Pause.* In Normal mode, it does nothing while in the AAR mode, it pauses the playback
- *Stop.* In Normal mode, it stops the capture while in the AAR mode, it stops the playback
- *Save Capture File.* Opens a save dialog to allow the user to specify a location and name for the capture file

V. CONCLUSIONS-FUTURE WORK

A. SUMMARY

This thesis designed and implemented an After Action Review System for a 3D networked training simulation. As simulations become more sophisticated and complex, applications become important to provide tools for After Action Review. AAR has been used very successfully in real type of situations, so why not in a virtual environment? AAR will help both the trainer and trainee reveal the hidden aspects of a played scenario. It is a valuable tool that provides feedback to the users about how well they are doing in the simulation, and also helps in spotting virtual environment weaknesses and human factor problems.

The implementation of the VECQB After Action Review System showed that the design is not only viable but also expandable in many different directions. The use of XML technology led to a lightweight solution without very much overhead and without sacrificing any of its flexibility.

B. FUTURE WORK

Future work will focus on incorporating more features in the AARS implementation and conduct scientific research in order to determine how and to what degree AAR improves training in a 3D training simulation. More specifically, future work will include the following:

- Scenario Editor. The scenario editor will help the user to setup the initial platform position, orientation, type, and status for a simulation scenario or even write an entire simulation scenario for demonstrating purposes. This is something that can be done even in the current implementation since the user can author a new capture scenario based on the XML schema document that will be used to initialize and “drive” around the platforms in the simulation. What actually is needed is a graphical representation of the simulation’s level where the user will just go and set the platforms to the desired positions without struggling with numbers such as a platform’s terrain coordinates, or orientation angles measured in rads.

- Enhance the current Summary Report to a Statistic Report. Hence, the user will be able to see statistical data instead of only a summary for each platform. It will be also useful to extend the Statistic Report to include more than one platform.
- Activate the Add Index button in the Control Panel. The functionality of this button will include the ability to mark a time point in the simulation during a playback and be able to come back directly to that point later. This will make the playback even more flexible, since the user will be able to go back and forth with ease to exactly the points of interest.
- Allow capturing during a playback or implement another way to extract a new capture file from the one that is currently used for playback. This feature will allow the user to trim out any unnecessary data from a capture file and keep only what is needed.

APPENDIX GLOSSARY

After Action Review	The procedure that is conducted after an exercise or an activity that helps the participants to discover what really happened and why
After Action Discussion	The most popular method to conduct After Action Review. Involves a discussion guided by someone who has the appropriate experience or authority e.g. an instructor or a team leader. Discussion's main purpose is to clarify every hidden aspect of an act, give answers to what was really happened and why
Simulation Capture	The procedure of logging events and/or other simulation specific characteristics of a simulation for further use and analysis
Simulation Playback	The procedure of replaying or reproducing a simulation using a capture file. The idea is similar to replaying a video file
Platform	An entity in the simulation, it may be human or computer driven. Its actions may affect other platforms and/or the environment
Platform Path	The path followed by a platform in the simulation. It is consisted by the set of a platform's waypoints. In the simulation is represented by a colored line that connects the waypoints
HLA	High Level Architecture currently DoD's standard protocol for networked virtual environment simulations
Federation	In HLA the set of simulation instances along with the RTI (Run Time Infrastructure) constitutes a federation. Simulations in the same federation can interchange information through the HLA's infrastructure without

interfering with other simulations that are joined in different federations

XSLT Extensible Stylesheet Language Transformation (XSLT) is an XML language for transforming XML documents into other XML documents. More can be found at <http://www.w3.org/TR/xslt>

DOM The Document Object Model (DOM) is an API for HTML and XML documents. It defines the logical structure of the documents, and the way they can be accessed. It is implemented in many programming languages like Java, C++, Perl etc. More information can be found at http://www.xml.org/xml/resources_focus_dom.shtml

SAX Simple API for XML (SAX) is an XML API that allows for event-driven XML parsing. Unlike the DOM specification, SAX doesn't require the entire XML file to be loaded into memory. More information can be found at http://www.xml.org/xml/resources_focus_sax.shtml

BIBLIOGRAPHY

- [1] Sandeep Singhal and Michael Zyda (2000). *Networked Virtual Environments Design and Implementation*. Addison Wesley © 1999 ISBN 0-201-32557-8
- [2] H. M. Deitel and P. J. Deitel, Deitel and Associates Inc.(2001). *C++ How to Program 3rd Edition*. Prentice Hall ©2001 ISBN 0-13-089571-7
- [3] Frank M. Carrano Janet J. Prichard (2002). *Data Abstraction and Problem Solving with C++ Walls and Mirrors 3rd Edition*. Pearson Education Inc. © 2002 ISBN 0-201-74119-9
- [4] Gerald Vasend (1995) *After Action Review System Development Trends*. Proceedings of the 27th conference on Winter simulation Arlington, Virginia, United States
- [5] Major Gary W. Allen, Roger D. Smith (1994) *After Action Review in Military Training Simulations*. Proceedings of the 26th conference on Winter simulation Orlando, Florida, United States
- [6] Documentation of VECQB simulation's source code.
- [7] The Apache Software Foundation. *Xerces C++ Parser* subset of "The Apache XML Project". Distributed under GNU Library General Public License at <http://xml.apache.org>
- [8] The Apache Software Foundation. *Xalan C++* subset of "The Apache XML Project". Distributed under GNU Library General Public License at <http://xml.apache.org>
- [9] David Hunter, Kurt Cagle, Chris Dix, Roger Kovack, Jonathan Pinnock, Jeff Rafter (2001). *Beginning XML 2nd Edition*. Wrox Press Ltd © 2001 ISBN 1-861005-59-8

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California
3. Professor Rudolph Darken
Naval Postgraduate School
Monterey, California
4. CDR Joseph Sullivan
Naval Postgraduate School
Monterey, California
5. Dylan Schmorrow
Office of Naval Research
6. Denise Nicholson "Nicholson, Denise NAVAIR"
NAVAIR Orlando
7. Mike Bailey
TECOM, Quantico
8. Joseph Cohn
Naval Research Laboratory
9. Robert Armstrong
TECOM, Quantico
10. DI.K.A.T.S.A.
Inter-University Center for the Recognition of Foreign Academic Titles
Athens, Greece