5-7-2011

Animation and Visualization of 3D Underwater Sensor Networks

Matthew T. Tran
University of Connecticut - Storrs, mtranim8rr@gmail.com

Animation and Visualization of 3D Underwater Sensor Networks

Matthew T. Tran

B.S., University of Connecticut, 2009

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2011

APPROVAL PAGE

Master of Science Thesis

Animation and Visualization of 3D Underwater Sensor Networks

Presented by

Matthew T. Tran

Major Advisor _____
Jun-Hong Cui

Associate Advisor _____
Swapna Gokhale

Associate Advisor _____
Zhijie (Jerry) Shi

University of Connecticut

2011

ACKNOWLEDGEMENTS

There are many people whom I must thank for supporting me throughout my entire graduate career. Without them, I would not have been able to advance this far.

First and foremost, I am deeply grateful to my major advisor, Dr. Jun-Hong Cui. It was her who took me into the graduate program and accepted me into the UWSN Lab to continue my project after I had worked with her on my Honors Thesis. Throughout my two years in the program, she has provided me with much guidance and was always looking out for me. She also helped me find a graduate assistantship at UITS, which is a wonderful place to work.

I also need to thank my associate advisors Dr. Zhijie Jerry Shi and Dr. Swapna Gokhale for being on my thesis committee.

I must also thank my colleagues in the Underwater Sensor Network Lab, James, Son, Yibo, Haining, Jun, Lina, and Michael, as well as former colleages Robert and Zheng, for supporting me all this time, helping with my work, and overall making the lab an enjoyable place to be.

I need to thank my co-workers at UITS, Yi, Haleh, Andrew, Jim, and everyone else, for working with me and making my job at UITS productive, educational, and enjoyable.

I definitely need to thank all my friends for making my entire college experience one of the most enjoyable times of my life. All my friends from the UConn Capoeira Club and the UConn Taiko Club have been incredibly supportive and I will cherish them for the rest of my life. Their friendship has allowed me to stay positive and continue looking forward.

Last but definitely not least, I need to thank parents for supporting me throughout my life and for doing all they can to help me. I will definitely make them proud.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Simulation and visualization are critical for the development of new systems and protocols in the area of computer networking. As real-world field testing is expensive and time-consuming, simulations are often preferred as they can be performed repeatedly and inexpensively while still reflecting the outcome of field tests to an extent. Visualizations of the simulation and field test results often follow to provide researchers with a vivid animation of the events, allowing for a much more intuitive understanding of the system than tediously reading through trace files. While there are currently a multitude of simulators and animators for land-based networks, few such tools exist for the emerging field of underwater networks, which differ significantly from land-based networks in that they are essentially 3D networks where the depths of the nodes are considered. An underwater simulator called *Aqua-Sim* had already been in development by the University of Connecticut's Underwater Sensor Network (UWSN) Lab. However, an accompanying visualization tool was not yet available.

Thus, this thesis work developed *Aqua-3D* as a brand new animator specializing in the visualization of underwater networks. Written in a Linux environment with C++ and OpenGL, *Aqua-3D* is able to read trace files outputted by *Aqua-Sim* and animate the simulation in full 3D graphics. Network events are represented by animations that are intuitive and easy to recognize by the user. The tool also provides users with full control over the playback and speed of the animation, full control over the viewing camera, and selectable windows that display additional useful information about the simulation. The accuracy of *Aqua-3D*'s visualization has been verified by the UWSN Lab members through the use of test scenarios generated by *Aqua-Sim* and the use of trace files from real-world field tests. It can be concluded that the current version of *Aqua-3D* is a robust tool, with the ability to correctly visualize trace files of underwater network simulations as

well as provide a number of additional useful features. While further improvements and modifications are necessary and new features may be required before it can become a truly powerful and respected research tool, the *Aqua-3D* software can serve as the necessary milestone.

Chapter 1

Overview

1.1 Motivation

In the research area of computer networks, simulations, field tests, and visualizations are important techniques for the development of protocols and systems. Simulations allow for inexpensive and repeatable experiments, which facilitates debugging. However, they cannot reproduce realistic conditions with high fidelity. Field testing on the other hand does allow for practical evaluation of research under real-world conditions, but is also expensive, time consuming, and subject to unpredictable problems.

To analyze the events of a simulation or field test, packet traces are typically generated. Unfortunately, these often contain large amounts of static detail that is tedious and difficult to comprehend. Additionally, manual analysis can be more prone to human error. To provide assistance in analyzing and understanding the outcome of simulations and field tests, visualization tools are used to provide researchers with vivid images and animations of the events. This allows for a quicker and more intuitive identification of patterns, which facilitates the understanding of how the network and protocols function. [12].

Traditional terrestrial networks, such as LANs and WiFi networks, are essentially two-dimensional. Underwater communication networks on the other hand, which are the focus of the University of Connecticut's Underwater Sensor Network (UWSN) lab, are actually three dimensional networks. This is because the depth at which a network sensor node is located is important, as the network itself may have different qualities depending on the depth.

While many simulation and visualization tools exist for 2D terrestrial networks, few such tools exist for 3D underwater networks. The UWSN Lab has been able to extend the popular terrestrial network simulator *NS-2* [10][1][7] so that it can simulate 3D underwater networks. However, additional work has to been done to efficiently visualize these 3D networks. The popular visualization tool *NAM* [12][3][7] currently only supports 2D networks, with no extensions implemented that allow it to render 3D networks. Because visual information is lost when projecting a 3D space onto a 2D plane, *NAM* is therefore unable to accurately visualize 3D underwater networks. There are a few other 3D network animators currently in existence on the web, but they have been found to be inadequate for the needs of underwater networks. Therefore, it was necessary to develop a brand new animator that can fit the requirements of underwater networks and the UWSN lab's research efforts.

1.2 Contribution of This Thesis

Over the course of the past two and a half years, we have been developing *Aqua-3D* for the purpose of visualizing the simulations produced by *Aqua-Sim*, which is an underwater sensor network simulator developed by the University of Connecticut's UWSN Lab [18][5]. *Aqua-3D* is inspired by *NAM* but does not use any of its source code; instead it was rebuilt completely from scratch to satisfy the new requirements for visualizing 3D networks.

Aqua-3D is written in C++ using an object-oriented paradigm, where the GUI components, major internal modules, and all network objects and events, are represented by classes. The GUI is built using the wxWidgets library, which is powerful and features many built-in classes to make designing and implementing a GUI that uses the machine's native API easy. Mesa-3D, which is the open-source version of OpenGL, is used to render the 3D graphics of the animation.

While *Aqua-3D* is still relatively incomplete compared to *NAM*, it is capable of reading a trace file produced by *Aqua-Sim* and animating the simulation events in full 3D graphics. It also has a fully controllable camera such that researchers are able to view the simulation from any angle. A number of other features, such as windows that display additional information, jumping to annotated events, and a variety of preferences for customizing the look and feel of the program, have been implemented as well. As for the appearance, intuitiveness, and accuracy of *Aqua-3D*'s animations, they have been verified by the researchers currently in the UWSN Lab.

However, since *Aqua-3D* hasn't been released to the general public yet, it still has to pass the test of wide-spread public acceptance in the research community. Additional features, such as time-plots and protocol-specific graphs, can be implemented in *Aqua-3D* as well. Overall, *Aqua-3D* is still a relatively young application that requires further development. However, it may also serve as a starting point for the development of future 3D visualization tools for underwater networks.

1.3 Roadmap

This thesis will primarily cover the design, implementation, and evaluation of *Aqua-3D*.

Chapter 2 will begin with a brief introduction to underwater networking, their applications, and the challenges the aquatic environment presents. It will also discuss how simulations, field

tests, and visualizations play critical roles in the research and development of new systems and protocols for underwater networks.

Chapter 3 will discuss the design of *Aqua-3D* in detail. It will start with the requirements and objectives *Aqua-3D* needs to fulfill as a research tool and then move on to describe the design of its graphical user interface, internal components, and animations. The chapter will conclude with a discussion on several difficulties encountered when designing *Aqua-3D*.

Chapter 4 will describe the implementation of *Aqua-3D*'s classes and modules in code. It will begin with a brief discussion of the development environment, followed by detailed descriptions of the class relationships, the general control flow of the program, and the control flow of the animation process. The chapter will conclude with an evaluation of *Aqua-3D*'s accuracy in visualizing a simulation trace file, its memory usage, and the results of attempts to install it on several machines and virtual machines.

Chapter 5 will discuss *Aqua-3D*'s ability to visualize trace files from a field test. First, an overview of the test bed will be provided, including their locations, node layout, and topology. Then the results of *Aqua-3D*'s visualization of the field test trace files will be presented and described. Finally, the issues that were encountered and possible corrections for them will be discussed.

Chapter 6 will present the conclusions and discuss possible future enhancements and additions to *Aqua-3D*.

Lastly, Chapter 7 will contain an appendix with important links, the requirements to install and run *Aqua-3D*, the installation instructions, key excerpts from *Aqua-3D*'s user manual, and other important information.

Chapter 2

Background

Covering approximately 71% of the Earth's surface, the oceans remain largely unexplored even to this day. They are vast but hostile environments, with unpredictable conditions, high pressures, and increasing darkness the deeper one dives. Human exploration is often too dangerous and expensive and thus, the need for technology that can perform unmanned expeditions becomes necessary [11]. With the possibilities of controlling and communicating remotely with underwater sensors or robots, the growing field of underwater sensor networks (UWSNs) aims to advance the technology forwards such that exploration of the unknown depths can be safe, easy, and cost effective.

This chapter will first present a general overview of underwater sensor networks, discussing their useful applications and describing the challenges to research and development because of the aquatic environment. The following section with briefly discuss the importance of simulations and field tests to the research and development of new technologies. The final section will describe how visualizations are critical to the evaluation and understanding of simulation and field test results.

2.1 Underwater Sensor Networks

2.1.1 Applications

Advancements in the field of underwater sensor networking can contribute to a multitude of applications. Many current underwater research methods require the use of large and tethered equipment, which is expensive to deploy, maintain, and recover in volatile conditions. Being able to use untethered, wireless devices that can monitor and detect events in their local environments would vastly reduce cost while also increasing reusability [11]. For example, seismic monitoring of undersea oil fields can be done more cheaply and frequently with an array of wireless sensor nodes [13]. Current methods require towing arrays of hydrophones along the surface which is very expensive in terms of time and money and therefore, can only be done once every two to three years. Sensor nodes distributed in the area of interest will be able to provide longer periods of study at a lower cost, which will be more useful for judging the performance of the reservoir. An example of a military application for UWSNs is submarine detection, where numerous sensor nodes are distributed in a critical area as seen in Figure 1. As submarines with modern stealth technology are visible only within very short ranges, a mesh of sensors can increase the probability that an enemy submarine will travel close enough to a node to be detected. Furthermore, detection will be reinforced with multiple observers. Once detection has occurred, the alert can then be forwarded along the multi-hop network of nodes to a surface transmitter to inform the base on shore [11]. Other applications include the monitoring of underwater structures or equipment, detecting oil leaks, studying phytoplankton concentrations [13], pollution monitoring, disaster prevention, and assisted navigation [9]. Further uses include space and time sampling which is needed by many aquatic systems studies and ship accident and wreckage investigation [11].

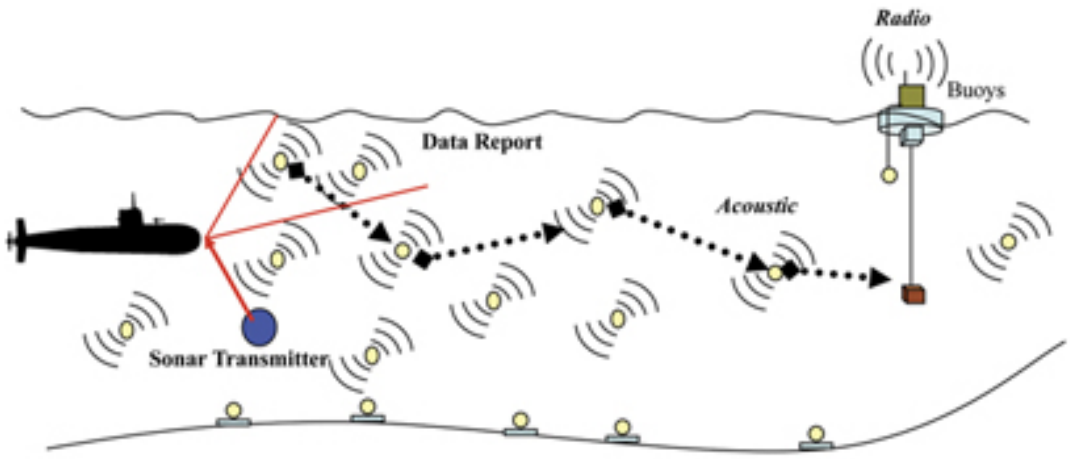


Figure 1: Submarine Detection with Sensor Mesh

2.1.2 Challenges

There are tremendous research and development challenges due to the nature of the underwater environment itself. Radio signals have a very limited range underwater, and thus are unsuitable for underwater communication [13]. The alternative that is currently being utilized is acoustic signals which unfortunately has its own complications. Whereas radio signals travel at the speed of light (3×10^8m/s), acoustic signals underwater travel at 1.5×10^3m/s, which is five orders of magnitude slower. This results in a much longer propagation delay for acoustic signals. In addition, the propagation speed may vary significantly depending on the conditions of the water, with increasing temperature, pressure, density, and salinity resulting in increasing speed [16]. Such variations can cause the paths of acoustic waves to curve and create dead zones where signals cannot reach [13]. The current available bandwidth of acoustic signals is also limited and varies based on its range and frequency, with rates between several tens of kbps over long ranges (several kilometers) to hundreds of kbps over short ranges (several meters) [11]. Signal attenuation is much greater underwater as well and is caused by a variety of factors. There is absorptive loss due to the water, multipath reflections from the ocean surface and floor, obstacles, and temperature

variations, and scattering from reflections caused by rough surfaces. These losses can also vary with distance and frequency, with reflections, scattering, and temperature variations being worse at longer distances while only spreading and absorption are significant at short distances [13]. Noise from underwater machinery or weather such as rain and wind can negatively effect acoustic signals as well [9]. Another issue with the underwater environment is that ocean currents along with other disturbances may cause the majority of the sensor nodes to have low to medium mobility, with a speed of around three to five knots, which presents a critical issue for protocols as they cannot assume that all nodes will be stationary (as is the case with typical terrestrial networks) [15]. It is also possible that autonomous underwater vehicles are used instead of stationary sensors, which are sparsely deployed and move on their own [17]. These issues of mobility are further complicated because global positioning systems are currently unusable underwater [15]. Lastly, underwater nodes may be lost due to a greater number of factors, such as being caught by fishing ships, damaged by aquatic life, or failure due to poor waterproofing [13].

Due to the plethora of issues caused the underwater environment as described above, it is difficult, if not impossible, to adapt existing terrestrial protocols to underwater networks. Therefore, new technologies must be developed that consider those different conditions.

2.2 Simulations and Field Tests

Simulations and field tests are both critical steps in evaluating the correctness and functionality of newly created systems and protocols.

Simulators such as *NS-2* [1] which provide testing environments are able to sufficiently replicate real-world events for development, testing, and debugging. They have the benefits of being inexpensive in terms of time, money, and man-power, easy to schedule so that they can be done overnight or whenever the simulator is available, and with repeatable network occurrences and

variables such as interference. Simulators may also provide additional features that are useful for evaluation. *NS-2* allows for adjustable abstraction levels, where less important details are hidden to improve performance, and emulation where the simulator interacts with a live network and subjects traffic passing through it to various dynamic situations. However, simulators overall are unable to reproduce real-world conditions with high fidelity and thus cannot be used for research alone.

Real-world conditions are unpredictable and there can be many causes of failure, such as those that were unforseen by researchers or those that simulators were unable to replicate. Modems may malfunction, batteries may fail, there could be poor weather, etc. It is for that reason that field tests are vital to the practical evaluation of new systems and protocols. Unfortunately, field tests are expensive both in terms of money and time. The physical equipment needs to be bought or built, ships need to be scheduled, and crews need to be hired. Time-wise, field tests may last for several days to weeks in total, with each day involving many hours of testing. In addition, in the event that something goes wrong, such as a malfunction or poor conditions, time and money may be lost with few meaningful gains. In spite of all that, it is absolutely necessary to test whether or not new research will function correctly under such events, and failures can be used as important learning experiences as well.

2.3 Visualizations

When simulations and field tests are completed, packet traces are typically generated that contain large amounts of details on the events that occurred. Unfortunately, manually reading and analyzing these files is tedious, difficult to comprehend, and prone to human error. With visualization tools however, the process of trace file analysis is greatly improved. The visualization tool will be able to read the trace files, interpret all the data, and present vivid images and animations of

the events that occurred. This greatly improves the manner in which researchers identify patterns and observe behaviors by making the process much faster and more intuitive [12]. Visualization overall drastically facilitates how researchers are able to understand and analyze the results of simulations and field tests.

However, for the field of underwater networking, visualization tools designed for 2D terrestrial networks such as *NAM* [3] are unusable. That is because underwater networks are essentially 3D networks, where the depths of the nodes are important. For instance, network conditions, such as the propagation speed of signals, may change with depth as the pressures increase and temperatures decrease the deeper one goes. The physics of underwater networks are also drastically different than terrestrial networks, for instance due to the aquatic conditions and slower propagation speed, and so the visualization tools must be designed with those physics in mind. In addition, 3D networks cannot be accurately represented by 2D visualizers because visual information is lost when a 3D space is projected onto a 2D plane. Due to those reasons, and because there exist few other, if any, 3D underwater network visualization tools, that *Aqua-3D* was developed.

Chapter 3

Aqua-3D Overview and Design

Although specializing in the visualization of 3D underwater sensor networks, the design of *Aqua-3D* was heavily inspired by *NAM*, the widely used 2D network animator. Its appearance and functionality was originally planned to be very similar so that researchers currently using *NAM* would not have to learn a completely new program. In addition, it was originally intended to *Aqua-3D* to be an extension of *NAM*, reusing most of its source code and only implementing the functionality for 3D visualization. However, due to the inability to reverse-engineer *NAM*'s code and the lack of developer documentation, this could not be accomplished. Instead, the primary functionality of *Aqua-3D* was completely re-written, using a rudimentary understanding of *NAM*'s functionality. Similarly, while the appearance *Aqua-3D*'s GUI was initially similar to *NAM*'s, further additions and modifications to its features eventually lead to a change in design. In terms of overall functionality however, *Aqua-3D* is sufficient to the needs of its intended users in that it reads trace files produced by *Aqua-Sim* and visualizes the simulation such that researchers can observe the events and freely manipulate the camera for the best possible viewpoint.

This chapter will first go over the requirements of *Aqua-3D*'s target users, which are underwater network researchers as well as general objectives it needs to satisfy as a well-designed

application. The next section will discuss the various aspects of *Aqua-3D*, starting with the design and functionality of its GUI. An explanation on the feature set of the program will follow, along with a description on the basic elements that make up the 3D visualized environment. Afterwards, the format of the trace file *Aqua-3D* uses as input, the key internal modules that provide the core functionality of the program, and how the network events are represented and animated will be discussed. The chapter will conclude with a discussion on the conceptual challenges that were encountered when designing *Aqua-3D*.

3.1 Design Requirements and Objectives

Researchers require that *Aqua-3D* be able to present an accurate and attractive visualization of their simulations for the purposes of facilitating research and development as well as presentation of their work to others. Thus, *Aqua-3D* must be able to correctly interpret the simulation trace files and render the animations in such a way that is immediately intuitive and recognizable to the viewer as well as being correct to the contents of the file. It is also desirable for the program to be able to handle many different kinds of networks and events so that it may be used for different areas of network research.

In addition to the requirements of the researchers, *Aqua-3D* must satisfy a number of other objectives. First, the graphical user interface must be intuitive so that users do not have a difficult time learning or remembering how to use the program. It should follow accepted standards, be clear and consistent with word usage, provide good visual feedback, and logically arrange fields and controls [14]. At the same time, the program must be able to provide users with many helpful options for playing, viewing, and manipulating the environment, such as a freely controllable camera and adjustable animation speed.

Lastly, the program must be well written so that it is efficient in its computations and memory usage so that the animations run smoothly without overworking the machine. In addition, the code and documentation must be easy to read and understand as well so that future developers can work with the program with little difficulty. As with any application, development and refinement of *Aqua-3D* will continue on for a long time, so its code must be written with future additions and modifications in mind.

To satisfy these requirements and objectives, *Aqua-3D* was designed as such.

3.2 Graphical User Interface

The graphical user interface of *Aqua-3D* is divided into several primary areas as seen in Figure 2. They include the animation viewing window (Environment Panel) in the center, the playback controls (Playback Panel) at the top, the camera controls (Control Panel) to the left of the Environment Panel, and the information viewing windows (Information Panel) to the right of the Environment Panel.

As *Aqua-3D* is an visualization tool, the most important component of its GUI is the window in which the animation is shown: the Environment Panel. It is the focal point of the program, being the biggest section of the GUI and also centered in the layout. In additional to displaying the animation, users are able to click and drag anywhere within the window. Doing so with the left mouse button rotates the camera so that users can view the animation from different angles. Clicking and dragging with the right mouse button pans the camera so that different parts of the environment can be centered in the window. Scrolling the mouse wheel (while the mouse cursor is positioned over the window) zooms the environment in and out. With these controls, users are able to focus the camera on key areas of the animation when studying the functionality of their systems and protocols.

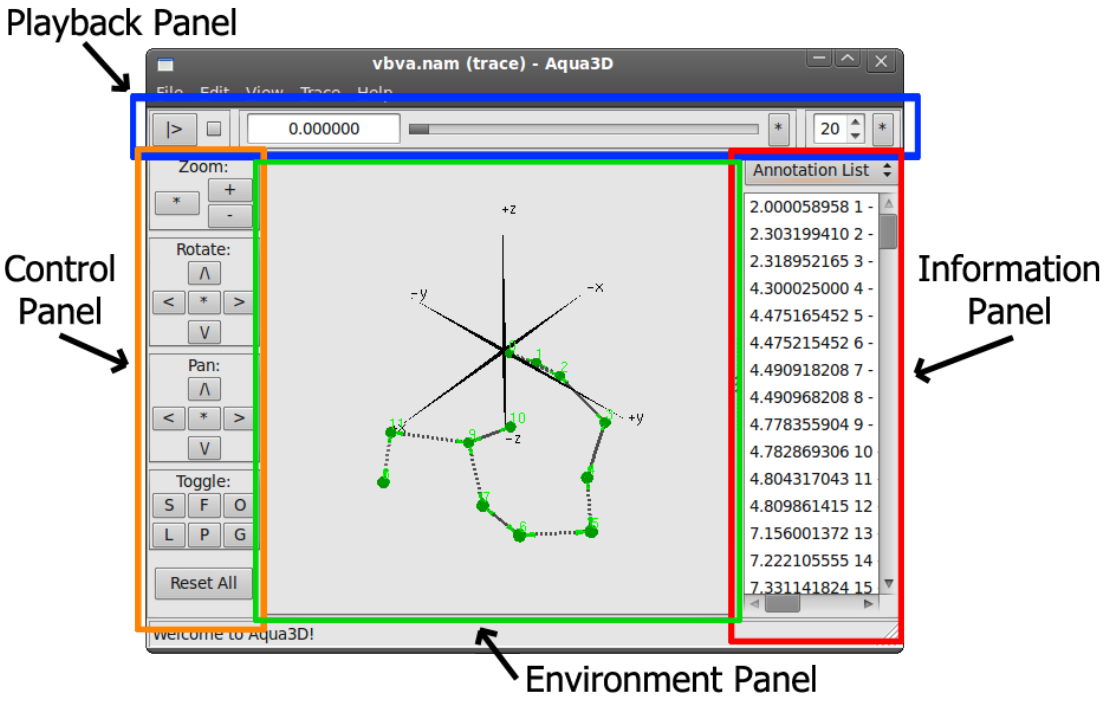


Figure 2: Graphical User Interface

The Playback Panel holds the controls for playing and pausing the animation. Originally, there were five buttons that handled playing the animation: fast rewind, play backwards, pause, play forwards, and fast forward, because *NAM* had those buttons as well. However, the ability to play backwards was deemed unnecessary and thus the fast rewind and play backwards buttons were removed. The fast forward button was also eventually removed because the functionality to control the animation's speed was implemented as well, and so having a fast forward button was redundant. Lastly, the stop and play buttons were combined into a single toggle button that switches between the two functionalities. When the animation is paused, the button shows a "play" icon and begins playing the animation when pressed. When the animation is playing, it shows a "pause" icon and stops the animation when pressed. Combining the two allows users to quickly switch between playing and pausing as needed without having to move the mouse cursor to another button. Added later to *Aqua-3D* was a checkbox that indicates whether or not the animation will

automatically be replayed when completed. Checking the box will make it so that the animation is reset and played again from the beginning when it is completed, whereas unchecking it will tell the program to stop playing the animation once the end is reached. In addition to the player controls, the Playback Panel includes a window that displays the current time in the animation as well as a scroll bar showing the progress of the animation. Users are able to jump to specific times in the animation either by entering a time directly into the display window or by clicking and dragging the scroll bar to a specific point. Lastly, a control to adjust the animation speed in included so that users can watch the animation in super slow motion for a more careful look, or to view it closer to real time.

While users are able to manipulate the viewing camera with the mouse, the Control Panel contains additional buttons for camera manipulation as well as buttons that control appearance of the environment. The buttons that control the camera zoom, rotation, and pan move the camera in exactly the same way as using the mouse, but each button press moves the camera a specific (user customizable) amount. Having buttons for camera manipulation in this manner is useful so that users can more precisely adjust the camera small amounts than clicking-and-dragging. Each group also has a button that resets their respective camera movement, which is useful for restoring independent aspects of the camera. There are also buttons for toggling the visibility of certain environment objects so that users can choose to see only what they need. Users are able to toggle the surface and floor polygons that indicate the orientation as well as the size of the simulated environment, the grid lines that allow users to see the positions of the network elements more accurately, the marker that shows the position of the environment's origin (0, 0, 0), the labels for the nodes and axes, and the transmission path lines between communicating nodes. Lastly, there is a button that resets all the camera angles and toggle settings to their default values.

In addition to viewing the simulation, *Aqua-3D* displays various sets of information to the user in the Information Window. The different information windows are selectable via a drop-down menu at the top of the panel. The first window displays a list of annotated events, which are events that have been designated in the trace file to mark important times in the simulation. Users can double-click on an event line in this window, which will pause the animation if it is playing and jump to that point in time in the animation. The window normally only displays annotated events that have already been encountered, because getting all the annotated events in the trace tile requires that the program read the entire trace file at the start. For small files, this is not a big issue, but as the size of the file increases, so does the loading time. There is an option however to load a trace file with all annotated events loaded as well, but this will require the overhead that the entire file be read at the start. The second selectable window simply displays a list of events that have already been encountered or events that are currently loaded and waiting to be played. As with the annotation events, *Aqua-3D* cannot show all the events in the trace file because that will require reading the entire file at the start. The third window presents a list of the nodes in the simulation. A user can double-click on an individual node to view additional details, such as its color, size, position, and velocity. Also, clicking on a node will draw a black box around it in the Environment Panel so that the user can see where that node is in the environment. The last window simply shows the dimensions of the simulated environment as well as the ending time of the animation.

3.3 Additional Features

Aqua-3D includes some additional features that make the program more helpful, customizable, and easier to use.

There are numerous preference options to adjust the functionality and appearance of *Aqua-3D*. For the camera control buttons, users can adjust the granularity of movement so that each button press moves the camera more or less, as well as inverting the direction of movement so that it can either appear as if the camera is moving around a stationary object or that the environment is moving relative to a stationary camera. There's is also a switch to change between a 3D perspective camera angle or a 2D orthographic angle for times when a 2D top-down view of the network is necessary. In addition, there are options to change the location of the origin marker so that the network objects can be positioned correctly within the bounds of the environment. Furthermore, users are able to show or hide specific animations so that users can view only relevant events. Lastly, there are options to change the colors of the background, the surface polygon, and the floor polygon for added customization as well as possibly being useful for users who are color blind.

Aqua-3D has the ability to save camera angles to an external file for later reloading. This can be handy for a user who has found a "perfect" camera angle to view his or her network simulation and would like to easily go back to that angle between program instances. Instead of having to manually readjust the camera and change the toggle settings back to those values, the user can simply save those settings to an external file and then load that file again at a later time.

As with *NAM*, *Aqua-3D* has the functionality to jump to interesting events that are annotated in the file [12]. When annotated events are encountered during the reading and parsing of the file, they are added to a special page in the Information Window that will display them all. The page will allow users to double-click on an event, which will stop the animation and quickly update the visualization to the selected event. The user can then resume the animation from that point.

As packet transmission send events occur during the course of the animation, *Aqua-3D* will draw lines between senders and receivers to indicate the transmission paths of the nodes. For unicast transmissions, only the path between the sender and the intended receiver is drawn For

broadcasts, lines are drawn from the sender to all nodes that are within the transmission signal's range. An arrow head can be drawn as well to indicate the direction of transmission and also be optionally colored with the color of the sender node. With this feature, the topology of the network can be seen as the animation progresses.

When a node attempts to receive more than one transmission signal at the same time, a packet collision will occur at that node. *Aqua-3D* has the ability to detect such packet collisions and draw an appropriate graphic over the node to indicate the event. When a new transmission signal event is created, it first checks to see which other nodes are within its transmission range and saves them to a list. That list is given to each of the signal spheres that are created as part of the event animation, who will individually check whether or not it has reached any of the nodes. When the sphere does reach the node, if the sphere is the first (starting) sphere of the event, it will notify the node so that the node will know that it is now receiving a transmission. Once the last (ending) sphere of that event reaches the node, it will notify the node so the node will know it is done receiving that transmission. If at any point, the node receives more than one transmission, meaning it encounters the starting sphere of another signal before the end sphere of the previously received signal is encountered, the node will know a collision has occurred and will draw a red explosion-like graphic over itself to inform the user. As more starting spheres are encountered, the node will increment a counter to keep track of how many transmissions it is currently receiving, and will decrement that counter once the corresponding ending sphere is encountered. Only once all the corresponding end spheres are received will the node stop drawing the collision indicator. For the time being, this only works when the nodes are stationary, as the event only checks for node within its transmission range once when it is initially created. If the nodes were moving, nodes will enter and leave range, which will cause the collision detection to work incorrectly.

Finally, for loading trace files, users have the additional option of loading all the file's annotated events and/or drawing all of the transmission lines between the nodes immediately from the start. Choosing to do this has the additional overhead that the entire trace file must be read through in order to get the necessary information. For loading all the annotated events, the Parser simply extracts only those event lines and passes them to the Event Manager so that they will be added to the special page in the Information window. For loading all the transmission path lines, the Parser only extracts transmission events and passes them to the Event Manager so that the transmissions between the nodes can be determined. For either option, all other event lines are ignored. The benefit of loading each of the annotated events is that they are all available for selection immediately so users can easily select later events to jump to. The benefit of loading all the transmission paths is that the layout of the network can be seen immediately.

3.4 Environment Appearance

While the most important objects and animations in *Aqua-3D* are for the nodes and network events, there are several base objects that serve necessary purposes as well. They can be seen in Figure 3 as follows.

The surface and floor polygons are simply flat rectangles that are positioned at the top and bottom of the visualized environment respectively. The surface polygon is colored blue by default and represents the ocean surface. The floor polygon is colored brown by default and represents the ocean floor. Together, they allow users to easily discern the orientation of the environment as well as its size.

The origin marker is a black star object with points radiating along both directions of the x-, y-, and z-axes. The object itself is positioned at the (0, 0, 0) point in the OpenGL space so that users can easily recognize the coordinates of the network objects. The preference options allow users to

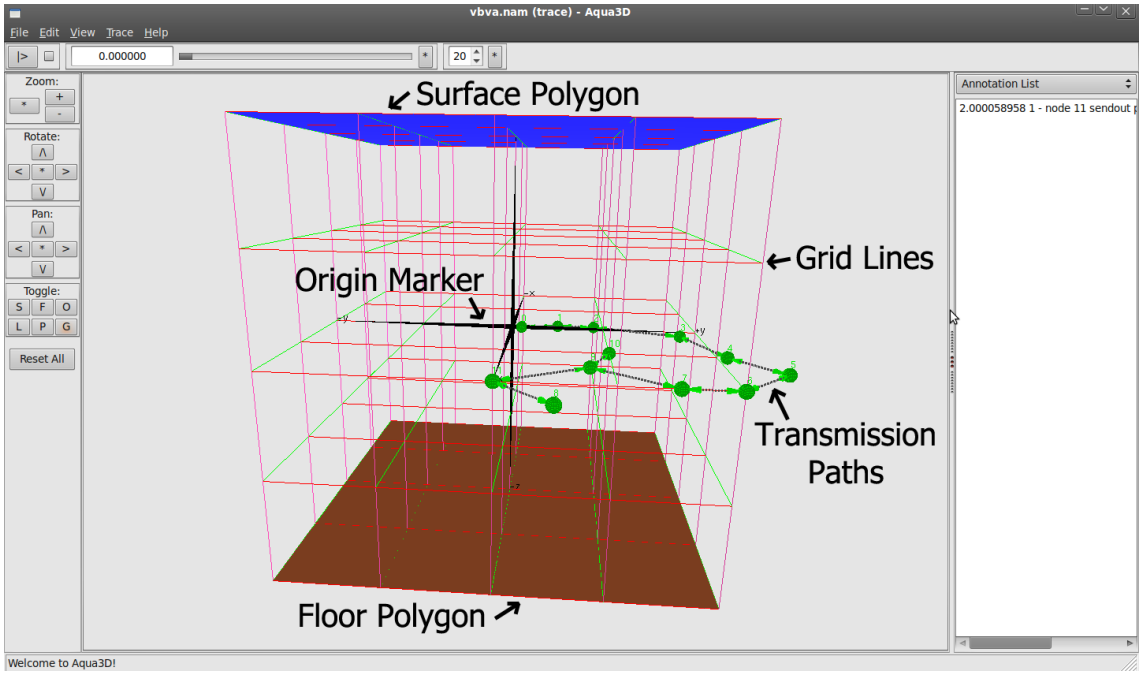


Figure 3: Base Environment Elements

place the origin marker in three preset positions: at the center of the environment, at the top center of the environment with the surface polygon, or at the lower corner of the environment with the floor polygon. The three positions are set to accommodate for different possible values of object coordinates. The center position is best when the coordinate values of the objects are both positive and negative for all three axes. The surface position is best when the x- and y- coordinates of the objects are both positive and negative while the z- coordinate (depth) is either strictly positive or strictly negative. The downward direction of the z-axis is also changeable (strictly positive or strictly negative) for either case. Lastly, the corner position is best when the coordinate values of the objects are strictly positive.

The grid lines are a series of lines that run along the x-, y-, and z- planes to divide the space into smaller subsections. They provide a visual scale to the environment and allow users to see the relative positions or sizes of network objects. There is a preference option to adjust the spacing between the lines to adjust the granularity of the scale. Also, the Control Panel contains a button

that will cycle through different visibility settings for the grid lines. It will cycle from showing no lines, to showing all lines, to showing lines only along the x-axis, then only along the y-axis, and finally only along the z-axis before returning to showing none.

Lastly, the transmission path lines are drawn to connect all sender and receiver pairs that have been determined by signal transmission events as described in Section 3.3. Also, by default, arrow heads are drawn at the end of the paths, pointing to the destination node so that users can see the direction of transmission. The arrow heads are also colored with the color of the sending node to indicate the source of the path. The preference options allow users to adjust how the transmission paths are drawn. Broadcast and unicast paths can be turned on or off so that only one or both of them are shown. The arrow heads themselves can be turned on or off and whether to color them as described above can be set. Finally, the option to draw only the paths that originate from the selected node (selectable in the Information Window) can be set so that users can see all the nodes the selected node has communicated with.

3.5 Trace File Format

The trace files that *Aqua-3D* takes as input to generate the visualization are generated by *Aqua-Sim*. The extension used for the file is .nam because *Aqua-3D* uses the same file format as *NAM*, only with some additional information. The top of the file lists the layout information for the network including the 3D positions, shapes, and colors of the nodes and the dimensions of the environment. The remainder and vast majority of the file contains a time-indexed list network events such as packet transmissions and packet drops [12]. An example of a trace file can be seen in Figure 4.

In general, a line in a trace file has the form <***event type flag***> **-t** <***time***> **-s** <***node***> <***other flags and attributes***>, where <***event type flag***> specifies the network event the line represents,

```
n -t * -s 0  -x 0 -y 10 -Z 0 -z 10  -v circle -c green
n -t * -s 1  -x -10 -y 50 -Z 0 -z 10  -v circle -c green
n -t * -s 2  -x -10 -y 90 -Z 0 -z 10  -v circle -c green
W -t * -x 400 -y 400
+ -t 2.000058958 -s 0 -d -1 -p vectorbasedforward -e 320 -c 2 -a 0 -i 0 -k MAC
- -t 2.000058958 -s 0 -d -1 -p vectorbasedforward -e 320 -c 2 -a 0 -i 0 -k MAC
h -t 2.000058958 -s 0 -d -1 -p vectorbasedforward -e 320 -c 2 -a 0 -i 0 -k MAC -R 100.000000
v -t 2.000058958 -e sim_annotation 2.000058958 1 node 11 sendout packet (vectorbasedforward) 0
r -t 2.303199410 -s 1 -d -1 -p vectorbasedforward -e 320 -c 2 -a 0 -i 0 -k MAC -R 100.000000
v -t 2.303199410 -e sim_annotation 2.303199410 2 node 8 receive packet(vectorbasedforward) 0
r -t 2.318952165 -s 2 -d -1 -p vectorbasedforward -e 320 -c 2 -a 0 -i 0 -k MAC -R 100.000000
v -t 2.318952165 -e sim_annotation 2.318952165 3 node 9 receive packet(vectorbasedforward) 0
...
```

Figure 4: .nam Trace File

<***time***> is the time in the simulation that the event takes place, and thus when the event's animation should begin, and <***node***> is the source node where the event occurs, and thus determines the position of the event in the environment. If <***time***> is a '*' that means the line is an *initialization* line which is used to set up the environment when the trace file is initially loaded. The two *initialization* lines currently implemented in *Aqua-3D* are the lines that specify the dimensions of the environment (**W**) and that specify the details of the nodes (**n**). If the <***time***> is a number, then it is a normal event line. Occasionally, there are events whose <***time***> values are not a '*' but appear above the initialization lines and have a time value of 0. These have been called "pre-initialization" lines and typically are node events (**n**) that specify the velocity of nodes that move during the simulation.

3.6 Animations

The Event and Node classes are vital to the visualization of the trace file.

Each individual node in the simulation is represented by a Node object which stores its name, position, size, shape, colors, and velocity, among various other attributes. The Node class is also responsible for the OpenGL rendering of itself as well as the graphic indicating a collision at the

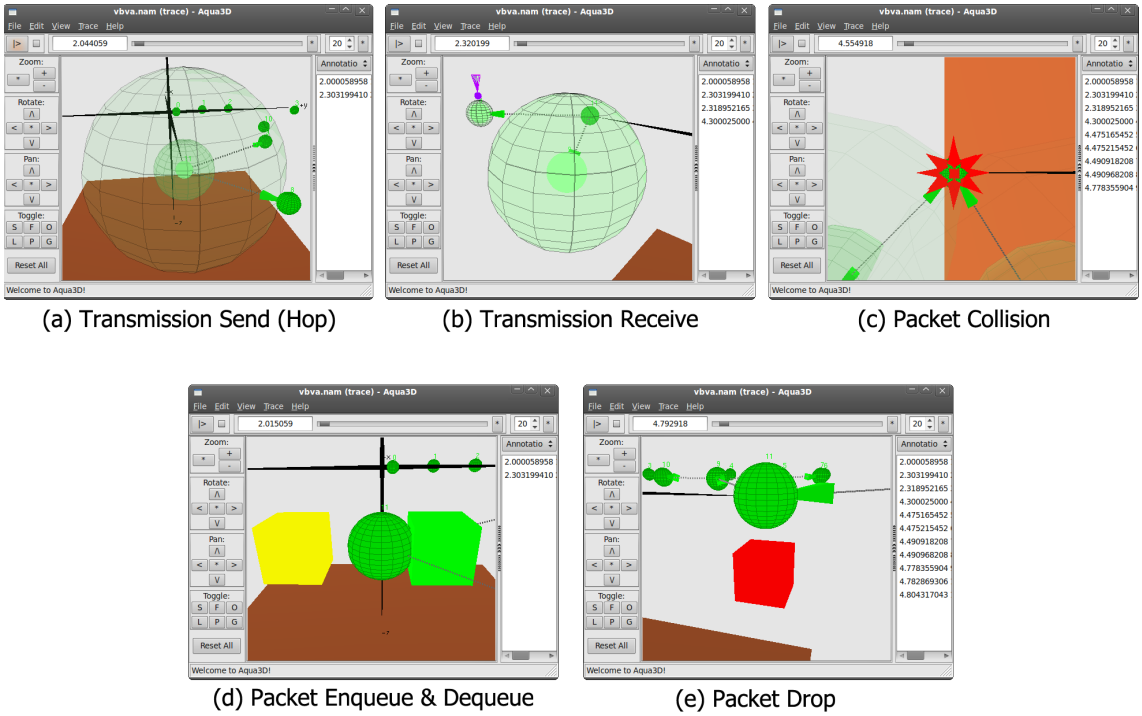


Figure 5: Currently Implemented Animations

node and the wire frame cube surrounding it when the node is selected from the node window in the Information Panel.

Each individual event animation is represented by a subclass of the Event class. The Event class specifies common event attributes such as its type, starting time, and source node. The class also declares two important functions that all Events must implement: the update function and the animate function. The update function modifies the values that are used by the Event to determine how its animation is drawn, while the animate function handles that actual OpenGL rendering. Each event subclass is updated and animated differently.

It is imperative that the animations for the network events are intuitive so that what they are representing is as obvious and recognizable as possible. Currently, there are five animated events implemented as shown in Figure 5: packet enqueue, packet dequeue, packet drop, packet transmission, and packet receiving.

The first and most important event is packet transmission send, and it was also the most difficult event to represent. While it is easy to visualize in 2D as circles expanding outwards from the transmission source node, in 3D it is more complicated. To achieve a similar looking animation, a series of semi-transparent spheres are drawn, expanding outwards from the transmission source node as seen in image (a) of Figure 5. The semi-transparency allows users to still see the signals while also able to see all objects inside of them. Wire frame outlines around the spheres are also drawn to better illustrate the 3D shape of the spheres. The spheres move at speed representative of the speed of an underwater signal, which is 1,500 meters per second. Since one pixel represents one meter in *Aqua-3D*, the spheres grows at a speed of 1.5 pixels per millisecond in animation time. The animation also takes into account the effective range of the signal in that its strength diminishes the farther it travels. As the spheres expand, they fade until becoming completely transparent once they have reached their maximum range. In addition, since the animations happen much slower than real time, to show a continuous transmission, multiple spheres are drawn during the course of the event. A formula is used to control the spacing between them so that not too many spheres are drawn at the same time, while also ensuring that at least two are always on screen to indicate that they all belong to one continuous signal. To indicate when a single transmission event begins and ends, the series of signals are book-ended with spheres that have a darker outline.

The second important event is the transmission receive event. Initially, the event was interpreted as the node beginning to receive a transmission and was animated with the receiving node simply changing its color during the course of the receive event. Once a certain amount of time had passed, the receiving node would return to its normal color. However, it was later realized that a receive event meant that the node had finished receiving the entire transmission. So, its animation was changed so that while the node still changes color, a small, semi-transparent sphere appears

around the source node and shrinks in towards that node to represent the receipt of the transmission signal. For receiving broadcast signals, that is the entirety of the animation. However, if the receiving node is the transmission's intended destination, a graphic similar to an exclamation-mark is drawn above the node for a short amount of time before disappearing. The semi-transparent sphere as well as the exclamation-mark icon can be seen in image (b) of Figure 5.

The next three events, packet enqueue, packet dequeue, and packet drop are currently only represented using a very simple animation. For packet enqueue, a green cube moves horizontally towards the event's source node to represent a packet being added to the node's queue. Once the cube is completely inside the node, it disappears. For a packet dequeue, the animation is the opposite, where a yellow cube moves horizontally out and away from the node for a set distance until it disappears, to represent a packet being removed from the node's queue. Packet enqueue and dequeue can be seen in image (d) of Figure 5. A packet drop, as seen in image (e) of Figure 5, is similar in that it shows a red cube leaving and moving vertically downwards from the node to represent a packet being dropped. These animations are currently only placeholders and future work will involve redesigning them so that they are much more representative of their respective events.

The last animation, the animation for packet collisions, is not actually specified within the trace file. As discussed in Section 3.3, packet collisions are actually detected by *Aqua-3D* using the signal spheres from transmission send events. The animation itself can be seen in image (c) of Figure 5 and involves a red explosion-like star object to represent the collision. It is drawn the moment the node detects the collision and is removed once the node is no longer receiving any transmissions.

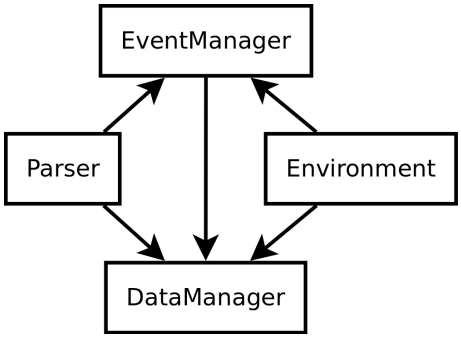


Figure 6: Key Internal Modules

3.7 Key Internal Modules

Aqua-3D has four key internal modules that are vital to its correct functionality: the Data Manager, the Parser, the Event Manager, and the Environment. Their relationships can be seen in Figure 6.

The Data Manager maintains many of the shared variables that are used by the other modules of the program. It stores numerous values that are used to properly render the scene, including the environment dimensions extracted from the trace file by the Parser, all the camera zoom, rotation, and pan values that are set by the Control Panel, top menu bar, or mouse click-and-drag, and the current visibility settings of the environment elements such as the surface and floor polygons, the grid, and the origin marker set by the Control Panel or the top menu bar. The Data Manager also stores a multitude of other values set by the preference changing window. These include color values for the background, surface, and floor polygons, toggle flags that control the visibility of event animations and transmission paths, spacing between the grid lines, and values determining the offsets and positions of the environment and the origin marker. Lastly, the Data Manager contains the current animation time and animation ending time values set by the Playback Panel, flags indicating whether or not the animation is paused or playing set by the Playback Panel, and the names and paths of the trace file and preference file.

The Parser is responsible for reading the trace file and extracting the event lines. When a trace file is loaded into *Aqua-3D*, the Parser first reads the initialization lines from the file and stores them into a list. The list is then passed to the Event Manager, which will decide how to interpret the lines. One of the lines will determine the dimensions of the environment in the Data Manager while the rest will create Node objects that represent the network nodes. When the Event Manager finishes processing the initialization lines, the Parser finishes setting up the environment by finding the ending time of the simulation, loading the first event lines of the file, and establishing starting camera angles and positions for the Nodes and environment elements. For the event lines, the Parser maintains a buffer to store a maximum of 500 lines for quick access when it is time to animate them. When the time comes to create more event animations, the Parser removes all the events currently in its buffer that have the same starting time and passes them to the Event Manager, which will create the appropriate Event objects and prepare them for animation. The Parser will then read and extract more lines from the file and load them into its buffer so that the buffer remains full.

The Event Manager is responsible for creating and updating the Event objects during the course of the animation. It receives the event lines passed from the Parser and extracts the relevant information from them to create the appropriate Event objects. When a new Event object is created, it is saved in a buffer that contains Events that are not yet being animated. All the Events in this "waiting" buffer should have the same starting time. Once the time comes for those Events to be animated, they are moved to another buffer which contains all the Events that are currently animating. Events in that buffer are processed by two functions. The first is responsible for updating the Event object attributes that determine how the animation is progressing, such as the positions of individual elements of the animation. Once an event's animation is complete, it is removed from the buffer. The second function is responsible for telling the Event objects to draw

themselves if their animation visibility is turned on. This is separate from the update function because there are many instances where the events need to be drawn but not updated, such as when the animation is paused but the camera is being moved. If the two functions were not separated, then moving the camera would also cause the animations to be updated, which is undesirable. In addition to handing the creation and updating of Event objects, the Event Manager also stores all of the Node objects created during the Parser's initialization process of loading the trace file and is responsible to telling them to draw themselves as well. Lastly, the Event Manager communicates with the Information Panel by updating its various display windows whenever new events are created.

The Environment is responsible for everything OpenGL related except for drawing the Nodes and Event animations. When the Environment is first created, it initializes all the necessary OpenGL settings required for an attractive presentation of the visualization. For example, it sets the lighting of the scene, the shading of the polygons, and perspective viewing angle of the camera for the 3D effect. It is also responsible for rendering the base environment elements described in Section 3.4: the floor and surface polygons, the origin marker, the grid lines, and transmission path lines. Whenever the scene is redrawn, the Environment first establishes the camera angle using the current rotation, pan, and zoom values obtained from the Data Manager. It then draws all the base elements and notifies the Event Manager, which will tell all the Nodes and Events to draw themselves. In addition, the Environment is the module that receives the click-and-drag and scroll wheel movements from the mouse for manipulating the camera. The Environment passes these values to the Data Manager and then redraws itself using the new camera values.

3.8 Lessons Learned

There were three major hurdles during the designing process of *Aqua-3D* that had to be overcome in order for the program to work as intended.

The first difficulty was how to represent a transmission signal in 3D. The decision to use semi-transparent spheres as explained in Section 3.6 came about only after several other attempts and a long period of conceptualizing. The important considerations were how to show the signal while also making sure the signal didn't block the visibility of the other graphics. Also, without adequate shading, the sphere would still look two-dimensional. In addition, the transmission of the signal would not be instantaneous and would actually occur over a relatively longer period of time depending on the size of the packet being transmitted. Lastly, a transmission signal's propagation speed in water as well as its effective range had to be taken into account as well. The rate at which the signal graphic would grow was simple enough to calculate: if the propagation speed of a sound wave in water is 1,500 meters per second, then if one pixel in the OpenGL space is a meter, then the signal representation would grow at a rate of 1.5 pixels (meters) per millisecond in animation time. The earliest representation of a signal was simply three circles that lied along the x- y- and z- axis that radiated out from the source node as seen in image (a) of Figure 7. It was not adequate at all however because it had far too much open space and was difficult to determine its orientation and position relative to other objects in the visualization. Afterwards, a black wire-frame sphere was attempted as seen in image (b) of Figure 7, but it also had too much open space and was difficult to see its position and orientation. Eventually, the semi-transparent sphere was decided on, which a thicker wire-frame sphere over it to better illustrate that it's 3D. For the signal's range, the sphere simply disappeared in the early iterations of the animation. However, for a more accurate and visually pleasing appearance, it was modified so that the sphere would fade gradually over time

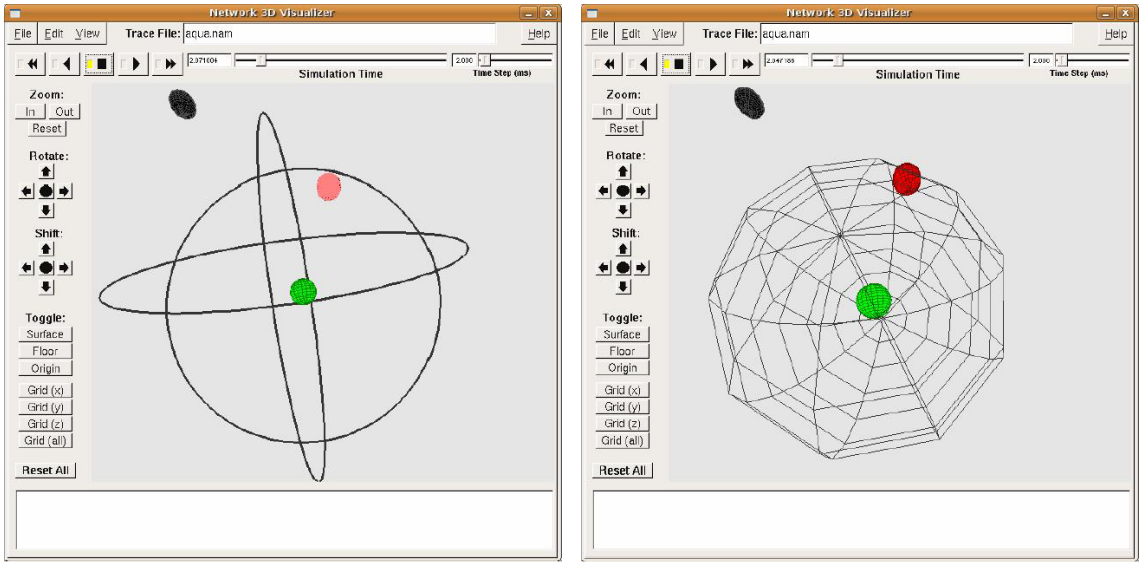


Figure 7: Early Attempts at 3D Transmission Signals

until it reaches its maximum range, where it would be completely gone. In order to visualize the constant transmission of a signal, it was originally planned to somehow draw a "thick" sphere radiating from the source node. However, illustrating a "thickness" was impossible, since there was no way to show that thickness from outside the sphere; the sphere would look the same no matter how "thick" it was because the outer-most sphere graphic would encapsulate the rest of the sphere graphics. Instead, numerous spheres are created and radiate outwards from the node, with spheres constantly being drawn for the duration of the transmission. Drawing too many spheres at a time would over work the system while drawing too few would risk having a sphere disappear before the next one appeared, which may lead the user to think the second sphere was a separate transmission. Eventually, a formula was developed that would space the creation of the spheres apart enough so that the minimal amount would be drawn while still enforcing that at least two spheres be on screen at the same time to show that it is still the same transmission. In addition, differentiated spheres with darker wire-frame outlines would "bookend" the transmission to make it more explicit where one signal began and ended.

The second major hurdle was figuring out how to quickly and correctly update events and animations when the user jumps to different point in time. When the user selects a specific time, the program must update and show the animation as it should be appear at that point, as if it had been reached normally by playing it from the beginning. All events and node positions (if the nodes are moving) must be in their correct positions, sizes, etc. Simply loading event lines from the selected point in the trace file would not work because there may be events that are supposed to be animated at that time which would not be loaded and thus not visible. So, all the events starting from the beginning of the file needed to be loaded and have objects created and updated again so that all positions and animations would be at their correct places. Initially, there was an additional value passed to each of the Events' update functions that specified the "interval" of updating, meaning instead of updating an animation's position at one pixel per frame, it would update at 100 pixels per frame, essentially speeding up the animation by 100. However, it never worked out completely correct because the timings changed and led to strange and undesirable animation artifacts, such as collision graphics that were not deleted properly. Eventually, the same process that is used to quickly go through the file to load all annotated events and transmission path lines was adapted for quickly reloading all the trace file events up to the selected point. That is, *Aqua-3D* will read the trace file again from the beginning and quickly update all the events without animating them until the selected time is reached. There is a moment of unresponsiveness while the program quickly reloads and updates all the events, but the end result is that the animation up to the selected point is guaranteed to be correct since it is essentially playing and updating the events as normal, only without drawing any of the animations and doing it at a much faster speed. Unfortunately, since this currently requires reading from the very beginning of the trace file, this method is impractical for large trace files as it would take far too long. Thus, a method of

determining a good starting point in the middle of the file, but before the selected time, is required so that *Aqua-3D* can start reading somewhere within the file instead of from the very beginning.

The final major problem was how to handle extremely large .nam trace files. While all of the files used to test the program were relatively small, it is possible that a single file can contain thousands to millions of lines of events. Obviously, it would be impractical to load the entire file into the program because it would be incredibly time-consuming as well as consume a lot of system memory. So, the method that was implemented was for the Parser to load a batch of lines at a time and store them in a buffer. Then, those lines will get passed to the Event Manager for it to create the appropriate Event subclass objects while the Parser refills the buffer with more lines from the trace file. While this may not be any more efficient than simply reading a line at a time and passing it to the Event Manager immediately without storing it in a buffer, the method currently used allows for the possibility that the loading of the event lines into the Parser's buffer can be done in parallel while the animation is playing if multi-threading is implemented.

Chapter 4

Aqua-3D Implementation and Evaluation

This chapter will go into detail with *Aqua-3D*'s implementation in code. First, the development environment will be quickly described. Then, the classes and their relationships will be discussed, followed by a walkthrough of the program's general control flow and animation process. The chapter will conclude with an evaluation on how well *Aqua-3D* fulfills the requirements established in Section 3.1 as well as discussion on the program's performance, memory usage, and installation on other machines.

4.1 Development Environment

Aqua-3D was coded primarily on two machines during the course of its development. It started on a virtual machine emulating Ubuntu 9.04 that was run on an HP laptop before being transferred to a Dell Latitude E6500 laptop running Ubuntu 10.04 with 3.4 gigabytes of RAM and a 3.06 GHz Intel Core 2 Duo CPU. The purpose for using Linux as the primary operating system is because researchers in the underwater sensor network field typically use Linux and because *NS-2*, *Aqua-Sim*, and *NAM* are all Linux-based applications as well. For the language, C++ was chosen for its strength, portability, and familiarity. Mesa3D, a completely open-source implementation of

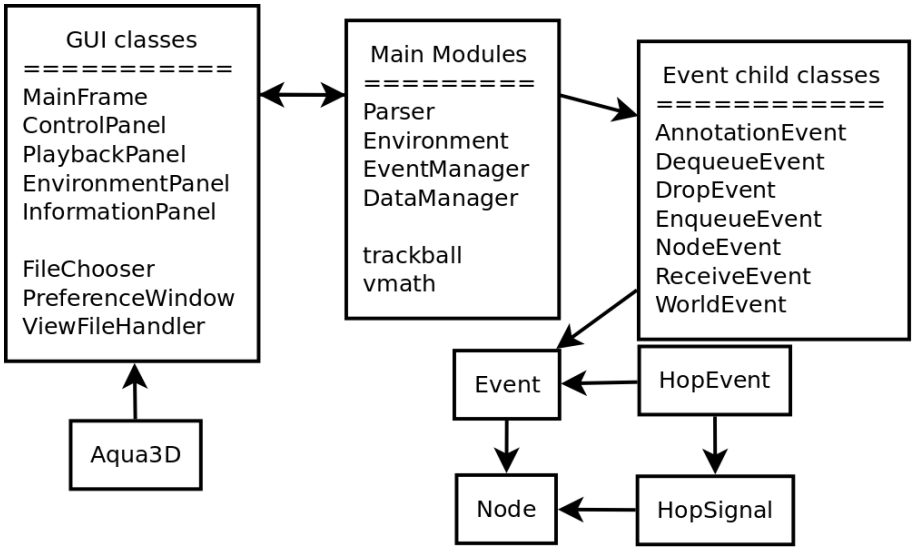


Figure 8: Class Relationships

OpenGL, is used to render the 3D graphics because it is powerful, platform-independent, widely used, and easily compatible with C++. Freeglut is also used because it is an open-source utility toolkit that provides additional features to Mesa3D/OpenGL. For instance, it provides management for windows containing OpenGL contexts and reading of keyboard and mouse functions. The GUI is implemented with a library called wxWidgets because it is compatible with C++, easy to use, cross-platform, and provides many built-in classes for common program features, such as buttons, scroll bars, and file selector dialogs.

4.2 Classes

Aqua-3D is implemented using the object-oriented programming paradigm with classes representing the various modules of the program. The classes can be grouped into three main sets as seen in Figure 8.

The Aqua3D class is the main entry point of the program and is responsible for reading and interpreting the command line arguments, setting up log file output, and initializing the creation of the GUI.

4.2.1 GUI Classes

The GUI classes are responsible for the appearance and functionality of the interface. The wxWidgets library is used in all of the GUI classes because it provides many built-in classes and functions that make it incredibly easy to implement a powerful graphical user interface while also retaining the machine's native interface API. The four classes that compose the main GUI as described in Section 3.2 are the PlaybackPanel, ControlPanel, EnvironmentPanel, and InformationPanel classes. The other three classes, the FileChooser, PreferenceWindow, and ViewFileHandler are smaller components that are created only when needed.

The MainFrame class represents the application window frame itself and is thus the parent component of all other GUI components of the program. In addition, it creates the top menu bar along with the bottom status bar and performs the necessary actions whenever the menu options are selected. In the "File" menu, there are the options for opening a trace file, opening a trace file with special options, closing the currently loaded trace file, and quitting the program. When a user selects "Open Trace File", the typical file selection dialog window opens, allowing the user to select a trace file to load. For the "Open With Options" selection, a FileChooser object will be created, which will allow the user to load a trace file with all annotation events loaded and/or all transmission paths loaded and drawn. Close trace file is close the currently loaded file and disable most of the GUI except for the "File" menu and the "Help" menu. In the "Edit" menu, the only option currently implemented is "Preferences", which will create a PreferenceWindow object and allow users to customize a variety of settings in the program. In the "View" menu, users are able to save the current view settings and camera angles to an external file, load a view settings file, angle the camera to specific viewpoints, toggle various environment graphics, or reset all camera angles and toggles to the default. When the user selects "Save" or "Load" view file, a ViewFileHandler

object is created to handle the selection appropriately. For the camera angles, users can select one of six preset angles: front, back, top, bottom, left, or right, and the camera will be positioned appropriately to show that side of the environment. The toggle and "Reset All" options in the main menu function exactly the same as the buttons in the ControlPanel, as explained in Section 3.2. The "Trace" menu currently has only one option, which is to open a simple dialog window that shows the name and path of the currently loaded trace file as well as the file's size. Lastly, the "Help" menu currently only has "About", which opens the typical window presenting the name and version of the program, the license of the program, and the credits for the developers. The "About" window itself is generated using a built-in wxWidgets class.

The ControlPanel contains the buttons for manipulating the zoom, rotation, and pan values of the camera. Whenever a button is pressed, the ControlPanel first calculates the new camera value, checks the new value to make sure it is within bounds if necessary. If the new value is valid, it is passed to the DataManager and then Environment is redrawn using that value. The controls also include buttons that reset the zoom, rotation, and pan values to their defaults. At the bottom of the ControlPanel, there are buttons for toggling various environment graphics described in Section 3.4.

The PlaybackPanel is responsible for playing and pausing the animation as well as informing the other modules when a new animation time or speed is selected so that the Events are updated accordingly. The PlaybackPanel contains a timer that is able to repeatedly call a function called Notify() every X-milliseconds when started. The code within Notify() checks the current animation time and will tell the Parser and EventManager to get more lines from the trace file and update the Event animations if necessary. Section 4.4 will discuss this process in more detail. The interface itself contains the button that toggles the animation between paused and playing, the window displaying the current animation time, the slider showing the progression of time, and a control to

adjust the animation's speed. In addition, there is a button to reset the animation back to the beginning and a button to reset the animation speed. When the user clicks into the time display window, the animation will pause, allowing the user to directly change the animation time. When the user clicks out of the window, the program will check to see if the entered time is actually a number and within bounds. If so, the animation will be jumped to the selected time. Similarly, when the user clicks on the progression slider and drags it to a new point, the animation is paused and the animation is jumped to that time. When the user clicks on the animation speed control, either into the window showing the current value or the adjust buttons, the animation will pause as well. The user can then either directly enter a new speed value or use the buttons to increment or decrement the value. The values in the control range from 1 to 58, with the lower values slowing down the animation and the higher ones speeding it up. When the selected values are less than or equal to 36, they can be seen as the desired frame rate and used to calculate the time interval (in milliseconds) with the formula $interval = 1000/value$. So for example, if the value selected were one, then the calculated interval would be 1000, meaning Notify() would be called every 1000 milliseconds (every second, so one frame per second). When the control values are higher than 36, then if the previous formula were used, many of calculated intervals would result in the same animation speed. So, the interval is directly calculated with the formula $interval = 26 - (value - 37)$. So, if the selected value is 58, the calculated value is five, meaning Notify() will be called every five milliseconds (200 frames per second). The PlaybackPanel is also responsible for creating the Annotation List window that contains the double-clickable list of annotated events. It will pass the page to the Parser, which will then pass it to the InformationPanel for it to display. This is done so that the PlaybackPanel can handle the actions when a user double-clicks on an annotated event, which is to pause the animation, and jump the animation to the selected time.

The EnvironmentPanel is responsible for creating, containing, and destroying the OpenGL Environment object. When no trace file is currently loaded, the panel simply displays a "Welcome!" text. Once a file is loaded and the environment values are initialized by the Parser, the Parser will tell the EnvironmentPanel to create and display a new Environment object. When the trace file is closed, the Environment object is destroyed. The EnvironmentPanel also creates the Node Details window and node-deselect button that will display the list of nodes in the InformationPanel. This is done so that the EnvironmentPanel can also handle the actions of the window and the button without needing the InformationPanel to directly communicate with it. The EnvironmentPanel will create the window and the button and pass them to the EventManager, which will then pass them to the InformationPanel. This way, the EnvironmentPanel can contain the functions that handle when the user clicks and selects a node in the window or when the user clicks the button to deselect the node.

The InformationPanel contains four selectable windows of information for the user to view. The windows are selected using a drop-down menu at the top of the panel. The class contains several functions that allow other modules to write values to the windows so that they can be continuously updated when necessary. Two of the windows are created within the InformationWindow class itself: the *Event Details* window, which contains a list of past events and immediately upcoming events, and the *Environment Details* window, which displays the dimensions of the environment and the stop time of the animation. The other two windows, the *Annotation List* and the *Node Details* page, are created in other modules because those modules need to handle the controls of the window. The Annotation List page is created by the PlaybackPanel so that when a user double-clicks on an annotated event, the PlaybackPanel can receive that action and be told to pause so that the animation can be set to the selected time. The Node Details page is created by the EnvironmentPanel so that when a user clicks on a node in the list, the EnvironmentPanel

will receive that action and tell the selected node to draw the “is selected” indicator around it. The button to deselect the selected node is also created by the EnvironmentPanel so that it can handle the button press accordingly. The button is passed to the InformationPanel in the same way as the Node Details page.

The PreferenceWindow displays tabbed pages of settings that the user can adjust to customize the look and feel of *Aqua-3D*. A PreferenceWindow object is created whenever a trace file is loaded so that the preference settings can be initialized. In the PreferenceWindow’s constructor, if the object is created when a new trace file is loaded, then the preference settings are loaded from an external preferences file. If the file doesn’t exist, the default values for the settings will be used instead. If the object is created because the user selected the Preferences menu option, then the current preference values are loaded from the DataManager. The controls of the PreferenceWindow will be set to the values that are loaded. After the values have been read either from the external file or the DataManager, they are validated to ensure that they are actually numbers and within their proper range. When the user selects Apply or Save, the PreferenceWindow will write all the new values to the DataManager as well as to the external preferences file.

The ViewFileHandler does not actually have a visible graphical interface. Instead, it only presents file selection dialogs (built-in by wxWidgets) that allow users to load an existing view settings file or save a new view file. When the menu option to load an existing file is selected, the MainFrame will show a file selection dialog for the user to select the file. The ViewFileHandler will open the file and extract the values. After extraction, the values are checked to make sure they are valid numbers and within their correct range. If any values fail validation, the setting’s default value is used instead. Once the values have been validated, they are passed to the DataManager to be set. Immediately, the Environment scene will be redrawn to show the loaded camera values. When the user selects the menu option to save the current view settings, the MainFrame will first

open a file selection window for the user to specify the filename to save. Then, the current camera and toggle values are retrieved from the DataManager and written to the new file.

The FileChooser is a simple dialog window that includes three elements: a button that opens a file selection dialog that allows users to pick a trace file to load, a check box for loading all annotation events immediately, and a check box for loading all transmission path lines immediately. The class itself only contains a constructor that builds the window and the controls, and accessor functions to get the selected trace file path and options. When the FileChooser is created by the MainFrame, the window is shown and the user picks the file and selects any options. When the window is closed with the "Load" button, the MainFrame uses the accessor functions to get the selection information and uses them to load the trace file accordingly. The FileChooser object is then destroyed.

4.2.2 Main Modules

The four key internal modules, the DataManager, Parser, EventManager, and Environment are the core components of *Aqua-3D* as described in Section 3.7.

The DataManager is responsible for storing all of the shared variables that are used by multiple components of the program. The class provides get methods for all of them as well as set methods for the variables that need them. For camera movement, while zooming and panning is simply an increment or decrement of their respective values, rotation is handled using quaternions so that mouse-controlled camera rotation would work. When a rotation is done, either with the mouse or with the ControlPanel buttons, a new rotation quaternion is created and passed to the DataManager. It will add the new quaternion to the current rotation quaternion using functions from the trackball class, which will result in a new quaternion representing the new rotated camera angle. This new quaternion is then set as the current quaternion. The DataManager makes sure

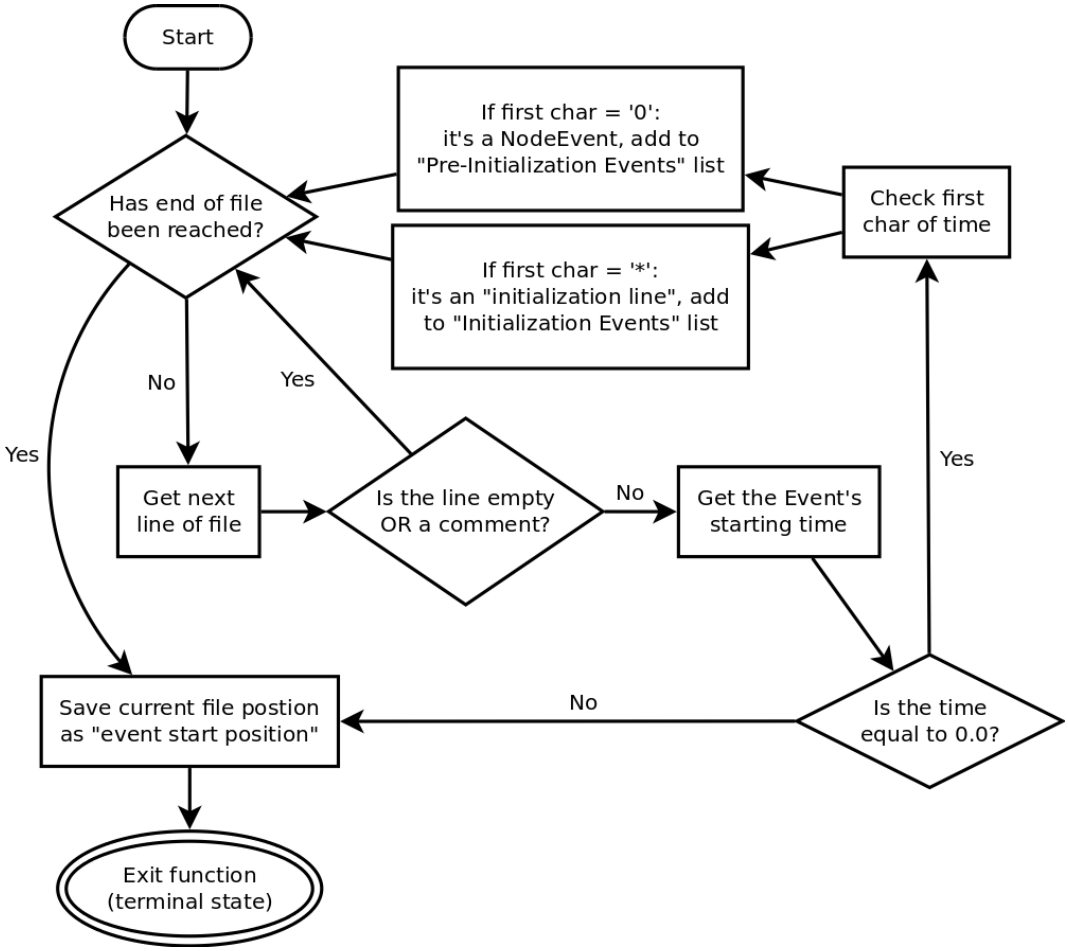


Figure 9: Parser::cacheInitializationLines()

the camera movement remains within bounds and looks appropriate whenever preference settings change, since the movement amounts vary depending on the dimensions of the environment and the current viewing mode (if it changes between orthographic and perspective). The function will calculate new values by which the camera zoom, rotation, and pan will change when their respective buttons are pressed, set the default and minimum and maximum bounds for the zoom, and reset the zoom if needed.

The Parser is the module that is responsible for reading the event lines from the trace file. When it is constructed, it is given pointers to the EventManager, DataManager, EnvironmentPanel, and InformationPanel so that it will be able to communicate with them. There are four main

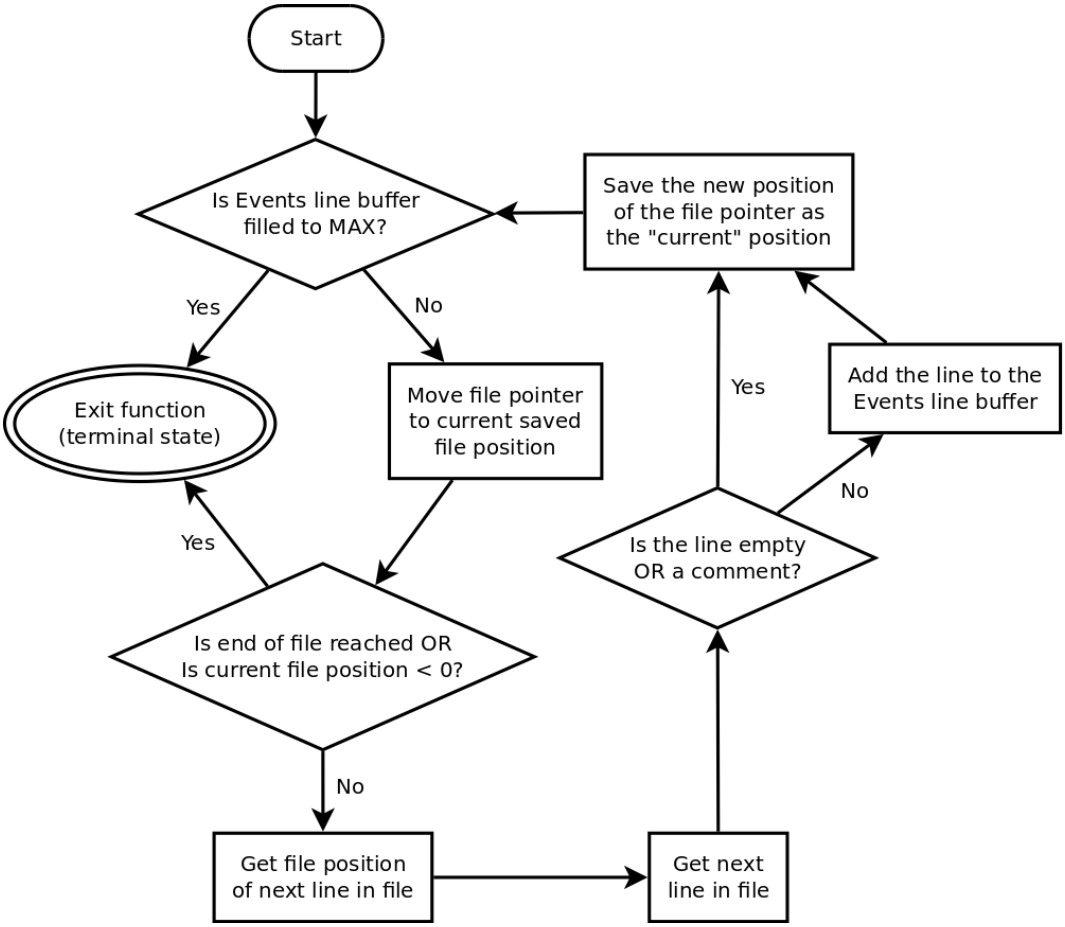


Figure 10: Parser::bufferEvents()

functions the Parser uses to read and extract information from the trace file. The first is called cacheInitializationLines(), which is used when the trace file is first loaded. A flow-chart of the function can be seen in Figure 9. While the end of the file has not been reached, the function will continue to get lines from the file. Each time, it will check if the line is empty or commented out, and if so, it will skip it and get the next time. If not, the function will look at the event's starting time. If the time is a '*', it will be added to the list of initialization events, which will be used to set up the environment. If the time is a 0, then add it to another list containing "pre-initialization" events. These events typically assign motion to specific nodes and appear at the very top of the file. For these to be reused whenever the animation is reset to the beginning, they need to be saved in a

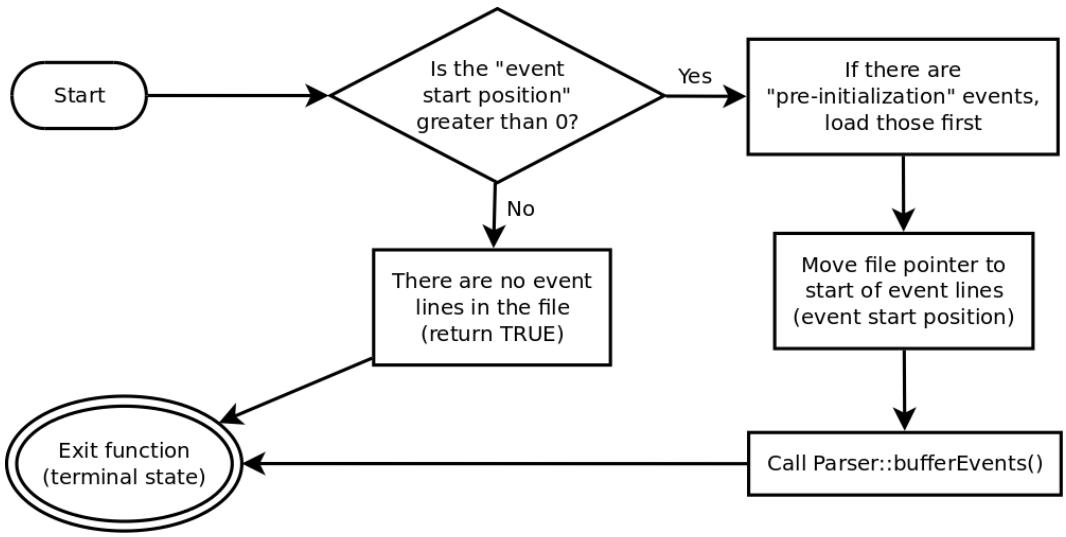


Figure 11: Parser::cacheFirstEventLines()

separate list. Once a line is reached that has a time that is greater than 0, that means the remaining lines in the file represent network events. The current file pointer is then saved as the "event start position" to mark the start of the network events so that it can be returned to easily. After this function completes, the lines in the list of initialization events will be passed to the EventManager so that the node objects can be created and so the dimensions of the environment can be extracted and saved. The second function is called bufferEvents() and is responsible for extracting lines from the trace file and storing them into the Parser's buffer. While the buffer has not been filled to capacity, it will move the file pointer to the currently saved file position and check if the end of file has been reached or if the current position is less than 0 (no event lines). If either are true, then the function exits. Otherwise, the next file position and next line are extracted. If the line is neither empty nor commented out, then the line is added to the buffer and the file position is saved. The Parser saves the file positions so that it can know where its current position in the trace file is. The third function is called cacheFirstEventLines() and is used to load the first network events in the trace file, after environment initialization is complete. A flow-chart of the function can be seen in Figure 11. If the "event start position" is not greater than 0, then there are no lines for

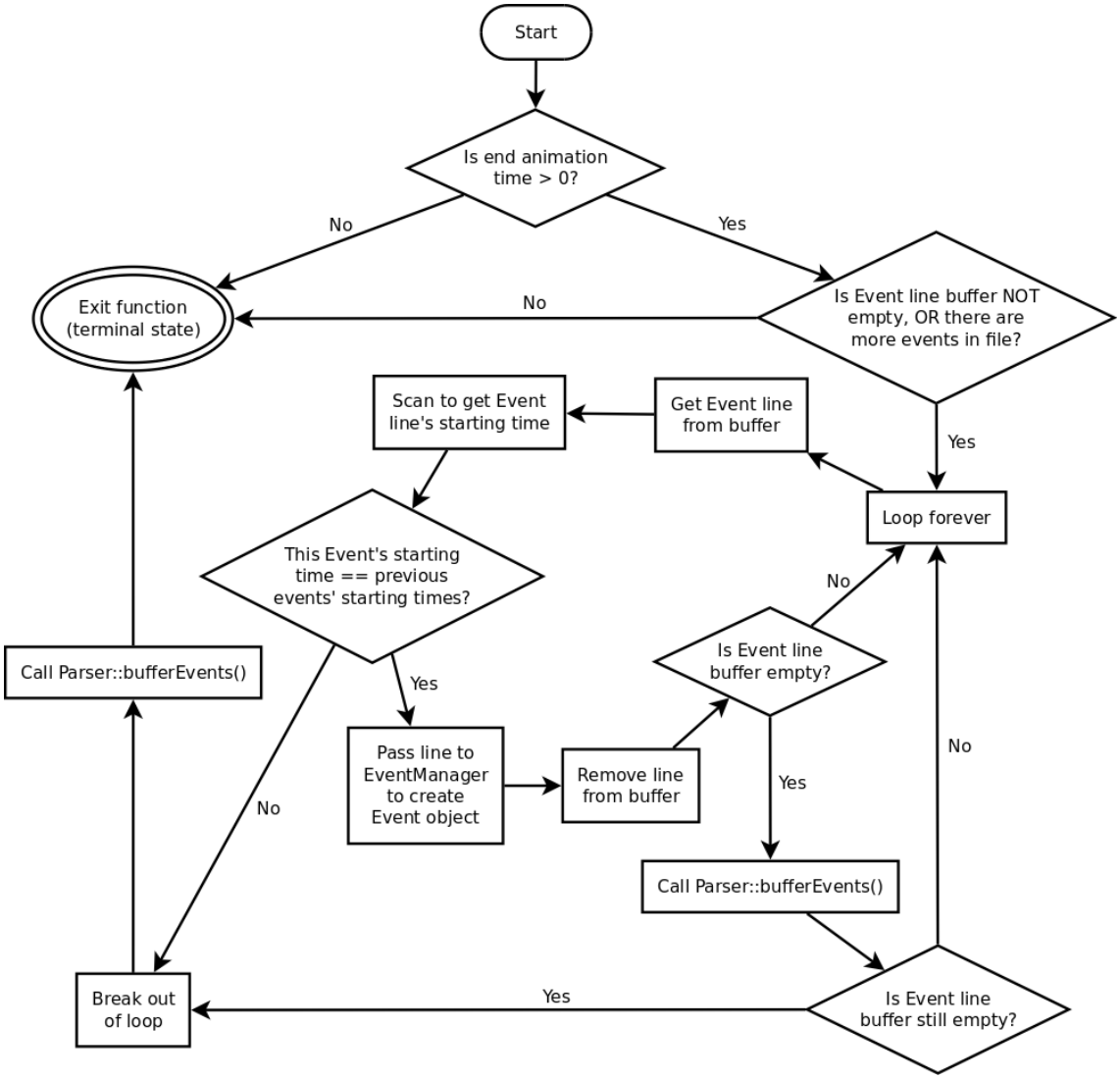


Figure 12: Parser::queueNextEvents()

network events in the file and the function immediately exits. Otherwise, it then checks to see if there are any “pre-initialization” events, and if so it will pass those to the EventManager first. Then, the file pointer will be moved to the “event start position” and bufferEvents() is called. The fourth function is queueNextEvents() is responsible for passing event lines stored in the Parser’s buffer to the EventManager. It first checks to make sure there are actually events in the trace file by checking if the saved end time is greater than 0 (ending time is 0 when there are no network event lines). If so, the function then checks if the Parser’s buffer isn’t empty or if there are more

event lines in the file. If the buffer is empty and there are no more events in the file, then the function exits. Otherwise, it will begin extracting lines from the buffer and passing them to the EventManager. It will only do so for all events that have the same starting time however, meaning the events towards the front of the buffer (since they are added in chronological order). Once an event is encountered that starts at a different time, the loop is exited and bufferEvents() is called. If the buffer empties before an event with a different time is seen, then bufferEvents() is called to add more lines to the buffer and loop continues, in case there are other events with the same starting time. If the buffer is still empty, then there are no more event lines in the trace file and the loop exits.

The EventManager is the module that extracts the important values from the event lines, creates Event objects for the animations, tells the Events to update or animate, and then deletes them once they are done. To extract the event information from the lines, the Parser will pass the line to a function in the EventManager, which will check the first character of the line to determine the type of event the line represents. Then, the appropriate function is called that will go through the line character by character to look for flags and values. The proper Event subclass object will be created accordingly using the extracted values and information about the event will be passed to the InformationPanel for it to display. The two functions that the EventManager uses to update and animate all the Events in its buffers are called updateEvents() and animateEvents(). The updateEvents() function first checks the "waiting" buffer to see if it is time to update the stored Events. If so, then the Events are removed from the "waiting" buffer and placed into the "animating" buffer. Then, updateEvents() will go through the "animating" buffer and check whether or not any of the Events animations have completed. If an Event has finished, it is deleted from the buffer. If not, the Event's update function is called to advance the Event's animation. The animateEvents() function is similar, in that it first goes through the "animating" buffer and checks each Event to

see if it has finished. If it has not, then if the animation is set to be visible (as selectable in the preferences), the Event will be told to render itself using its animate function. The EventManager is also responsible for storing all the Node objects that are created and tells them whenever they need to update their positions or render themselves.

The Environment is the module that is responsible for setting up the OpenGL rendering settings as well as drawing the environment graphics, which include the surface and floor polygons, the grid lines, the origin marker, transmission path lines, and the labels for the axes. The first time it is drawn, the Environment will establish all the OpenGL settings, such as depth testing which does not draw objects that are behind other objects, blending for transparencies, lighting, shadows, and material (texture) properties for how color will be shown. All subsequent redraws will not require those settings to be reestablished. The function also sets the camera's viewing mode and angle to either orthographic for a flat two-dimensional look, or to perspective for a three-dimensional look, depending on the user's preference setting. The class then renders all the base environment objects before telling the EventManager to draw the Nodes and Events. The Environment is also the object that receives the mouse-actions from the user and determines the appropriate change in rotation, pan, and zoom for the camera. When the user scrolls the mouse wheel, the new zoom value is calculated based on the direction the wheel is scrolled. If the new value is within bounds, it is passed to the DataManager and then the Environment is redrawn to reflect the change. When the right mouse button is clicked, the position of the mouse cursor when the button is clicked is obtained. Then, if the mouse is dragged, the amount by which the mouse is being dragged will be calculated and passed DataManager to update the camera pan values if they're within bounds. If the left-mouse button is clicked-and-dragged, the trackball class is used

to calculate the new rotation values. The new rotation quaternion is then passed to the DataManager, which will add the new quaternion to the current one to show the changed rotation. The Environment is then redrawn to reflect the change.

The Node class represents the network nodes in the environment and is responsible for drawing itself, updating its position if it is moving, and checking for signal collisions. The class contains numerous get and set methods for retrieving and setting its attributes such as its color, speed, and position. During the process of the animation, HopSignals during a HopEvent may pass through the Node. If so, the HopSignal will notify the Node if it is the starting or ending signal of the Event. The Node keeps track of the number of such signals that have passed through it to know how many signals it is currently receiving. A counter is incremented whenever a start signal is received and decremented whenever an ending signal is received. As long as it is receiving at most one at a time (counter is less than or equal to one), there are no collisions. However, if the Node receives more than one signal (two or more starting signals are received before an ending signal closes out the first one), then the Node detects a collision and will start drawing the collision graphic. The graphic used to represent the signal collision is a large red explosion-like polygon as described in Section3.6. Also, when the Node is selected in the InformationPanel's Node Details window, the EnvironmentPanel detects that selection and tells the Node that it is selected, which will prompt the Node to draw a black wireframe cube around itself as well.

4.2.3 Events Classes

The Event class is the abstract parent class for all of the classes representing specific network events. It defines the attributes which all events share, such as their event type, position, source node, and starting time. It also implements the get functions for all of them, a function to set the position values, and a function to check if the event is complete. The two most important

functions that the Event class declares are the virtual methods animateEvent() and updateEvent(). Since they're virtual, the Event class does not actually implement them; each event subclass is responsible for implementing their own version of those two methods. The event subclasses will store their own specific attributes and implement additional functions if necessary.

The DequeueEvent, EnqueueEvent, and DropEvent classes are implemented in practically the same way, with the only difference being how the event's animation is updated. All three event animations are currently very similar; they all involve a colored cube moving in some direction relative to the event's source node. They all have the same private attributes that include the current position of the moving cube, the maximum position possible for the moving cube, and numerous other attributes that are extracted from the trace file line but are not actually needed for the animation. The three classes' animateEvent() functions are almost completely the same, where they all draw a cube at some position relative to the source node. The only differences are the color of the cube that is being drawn and the axis and direction the cube is drawn. For a DequeueEvent, the cube is yellow, moving along the y-axis, and moving away from the node. For an EnqueueEvent, the cube is green, moving along the y-axis as well, but moving towards the node. For a DropEvent, the cube is red, moving along the z-axis, and moving away from the node. The updateEvent() functions for the three classes are implemented slightly differently as well in regards to how they calculate the cube's current position. All three classes have the same maximum cube position, but for DequeueEvents and DropEvents, the cube's current position is initialized to zero while the cube for EnqueueEvents are initialized to the maximum. Then in the updateEvent() function for both DequeueEvent and DropEvent, the current packet position is incremented each frame until it has reached the maximum packet position, meaning it has moved far enough away from the source node. Once this happens, the animation is complete and the Event object will be deleted by the EventManager. The updateEvent() function for EnqueueEvent

decrements the cube's current position each frame until it reaches zero, meaning it has completely entered the source node. At this point, the animation is complete and the Event object will be deleted by the EventManager.

The ReceiveEvent class has a more complicated animation than that of DequeueEvent, EnqueueEvent, and DropEvent and involves at least two separate animations. The first is simply drawing a polygon the same size and shape as its source node on top of the source node, but with a brighter color to indicate that the node is in a "receiving" state. The second is drawing a small semi-transparent sphere that shrinks in towards the source node until it fully disappears, to indicate the node receiving a signal. A third optional animation is drawn when the event's node is the recipient of the signal. When this is the case, an exclamation-mark-like polygon with the same brighter color as the "receiving state" indicator and with a purple wireframe outline is drawn above the node to indicate that it's the signal's destination. The class has a number of attributes that are extracted from the trace file, but few are actually used for the animation. The important attributes are its source node's shape, size, and color and the initial size of the sphere. The class's updateEvent() function will decrement the size of the sphere while its size is greater than zero, and once the sphere's size is zero, the function will decrement another timer until that timer reaches zero. At that point, the event will be complete. The purpose of the second timer is to extend the animation slightly so that the changed color indicating the "receiving state" and the "signal destination" polygon can be seen after the sphere disappears. The class's animateEvent() function draws the three animations as described earlier. It first draws the "receiving state" indicator, then the exclamation-mark-like polygon, and lastly the shrinking semi-transparent sphere around the node. The sphere has to be drawn last so that the other two animations are visible through the sphere.

The HopEvent class currently has the most complex animation. As explained in Section 3.6, the event's animation consists of a series of semi-transparent spheres that radiate out from the source node and gradually fade the farther they get until they disappear completely. Exactly the same as the ReceiveEvent, a polygon the same size and shape as the HopEvent's source node is drawn on top of the node with a brighter color to indicate its "sending" state. The key attributes of the class include the size of the packet, which controls how long the event lasts; the transmission speed of the signal, which also controls how long the event lasts; the range of the signal, which control how large the spheres get; and a value controlling the amount of space between each of the spheres, calculated by range*0.3 (0.3 chosen through trial and error to look the best) so that there are at least two spheres on screen at any time. The HopEvent also creates a list of nodes that are within its signal range so that it can detect signal collisions by calculating the distance from itself to the other nodes. If the distance to a node is less than or equal to its signal range, it saves a pointer to that node in a list. To update the event's animation, the class's updateEvent() function first increments a counter that is keeping track of the number of bytes that have been sent so far using the bit rate value. Once the counter equals the size of the event's packet, the event has finished transmitting the signal and will set its source node's state to UP. Otherwise, the source node's state will be SENDING. If the event has not completed transmitting, the function will create a new signal sphere if one of three criteria are met: if the spacing counter that keeps track of the space between the spheres is zero, which means the sphere is the event's STARTING sphere; if the sending counter is greater than or equal to the event's packet size, which means it is the END sphere of the event; or if the spacing counter is equal to the spacing value, which means it is one of the spheres in the MIDDLE. The signal spheres are actually represented by an additional class, called HopSignal, and will be explained in the next paragraph. When any of the three criteria are met, a new HopSignal is created and it's type (START, END, or MIDDLE) is set

accordingly, depending on the criteria that lead to its creation. If the signal is a MIDDLE type, the spacing counter is reset to zero. The HopSignal is then added to the front of the HopEvent object's list of signals. Newer (smaller) HopSignals are positioned in front of older (larger) ones so that when drawn from first to last (newest to oldest), the newer ones can be seen through the transparency of the older ones. Regardless if a new HopSignal was created or not, the spacing counter is incremented (and this way, the counter is only zero exactly once for the START signal). After this, all the HopSignals currently in the event's list are told to update themselves and check for collisions. If any of the HopSignals indicate that they are done (fully transparent), they are deleted from the list. Lastly, if the signals have actually been drawn, and there are no more signals in the list (all signals have finished animating), and if the event is done transmitting (all bytes have been transmitted), then the HopEvent itself is done and will be deleted by the EventManager. The HopEvent class's animateEvent() function draws the "sending state" indicator if the event is not done transmitting (not all bytes sent) and then tells each HopSignal in its list to draw itself.

The HopSignal class represents the semi-transparent spheres that are drawn during HopEvents. When constructed by its HopEvent object in the updateEvent() function, it is given a position (the same as the event's at that time), a color (the same as the node's "sending state" color), the signal's range, and a list of nodes that are within the signal's range. The constructor also initializes the sphere's starting transparency value (alpha) and the speed at which the sphere will grow, which represents the acoustic signal's propagation speed. The constructor then calculates the rate at which the sphere will fade, which is obtained with the formula $fadeRate = alpha/(range/speed)$. With this, the sphere will completely disappear once its size has reached its range. Similar to the other Event classes, the HopSignal class includes an updateSignal() function and a drawSignal() function. The updateSignal() function simply increments the sphere's size with $newSize = currentSize + speed$, where "speed" represents the rate of growth of the

sphere, and decrements the sphere's alpha value with $newAlpha = currentAlpha - fadeRate$, where complete transparency is reached once the alpha value equals zero. The drawSignal() function draws the semi-transparent sphere in two steps. First, it draws a semi-transparent solid sphere with the current alpha value and color. It then draws a wireframe sphere whose color depends on the type of signal it is: if it's a START signal, it is black and less transparent (darker); if it's an END signal, it is also black but more transparent (lighter); else, it's a MIDDLE signal and its color is lighter than the solid part's color. Every time a HopSignal object is updated by the HopEvent's updateEvent() function, the HopSignal will check for collisions by going through the list of nodes within range that was given to it and seeing if it has passed through it. If so, it will tell that Node, which will then determine if a collision has actually occurred. The HopSignal's animation is considered complete once its alpha value is zero or less, meaning it is fully transparent and has essentially grown past its effective range. So, once the HopEvent's updateEvent() function detects this, the HopSignal is removed from the HopEvent's list and deleted.

The creation of HopEvents also leads to the creation of transmission path lines, which are lines (optionally with arrow heads) that connect the source node of the transmission to its destination node(s). The lines themselves are represented by a struct and are stored in a list in the EventManager object and then retrieved by the Environment object, which will render the lines appropriately. The line structs actually contain pointers to the line's source and destination nodes, whose positions are used by the Environment to draw the lines. Because of this, the drawing of transmission lines will work even when the nodes are moving, since the drawing function will always use the most up-to-date positions of the nodes. Drawing the line itself is very simple, as it is just drawing an OpenGL line with the appropriate starting and ending coordinates. Drawing the arrow head is more complicated as it is actually a cone object that is positioned and rotated so that it is located at the correct end of the line and pointing at the destination node.

The AnnotationEvent and WorldEvent classes are not handled in the same manner as other event classes because they don't represent events that need to be animated. The function used by the EventManager to create the event parses the line from the trace file to extract the relevant information and then actually passes that information to the InformationPanel, which will add the double-clickable line to its Annotation Events window. An AnnotationEvent object is still created and added to the EventManager's buffer anyway. Similarly, the function for creating a WorldEvent will instantiate a WorldEvent object that will not actually do anything. Instead, if the WorldEvent line from the trace file is encountered during the initialization phase of loading a new trace file, the environment dimensions are extracted from the line and passed to the DataManager while those dimension values are also passed to the InformationPanel for it to add to its Environment Details window. Otherwise, if the WorldEvent line is located among the other event lines (it is typically the very last event in the file), the function will create the WorldEvent object and add it to the EventManager's buffer. When it comes time to animate the AnnotationEvent objects and WorldEvent objects, since there will be nothing for those events to animate, the objects will be deleted from the buffer immediately. The reason for still including objects for them and adding them to the buffer is for completion's sake so that ALL events are handled the same way, and also for the possibility that it may be necessary for those objects to do something later.

The NodeEvent class is handled differently as well because it also doesn't represent an event that is animated. Instead, it modifies the animation of the Nodes and tells them to start moving. NodeEvent lines that are encountered during the environment initialization phase of loading a new trace file are responsible for creating new Node objects for the environment and not for animating any other events in particular. So, the relevant node information is extracted from the line to create the Node object and then the line is discarded. When a NodeEvent line is encountered during the animation of file, the function to create the event object in the EventManager will

extract all the information from the line, create the NodeEvent, and then add it to its buffer. The NodeEvent object will store numerous other attributes about the node when it is created, such as its color, label, shape, etc., but since those attributes are only for creating the Node object during initialization and not for animating, they are not used for anything. The only important attributes from the line are the node's velocity values (one along each axis) and stop time. The velocity values will specify how many pixels the node will move along each axis during every frame of animation, while the stop time will be the time in the animation when the node stops moving and its velocities will be reset to zero. When it comes time for the NodeEvent to be animated, the NodeEvent's updateEvent() function will pass these velocity and stop time values to its source node, after which the Node will start moving as instructed. The NodeEvent is then considered complete by the EventManager and deleted.

4.3 General Control Flow

The general control flow of *Aqua-3D* can be seen in Figure 13.

When the user first starts *Aqua-3D* either through the command line or by opening the executable, the base GUI is created and displayed. While no trace files are loaded, the GUI is displayed with all of the main controls disabled. The only options available are the ones in the "File" menu and the "Help" menu. Once a trace file is loaded, either as a command line argument or using the file selection dialogs, a single instance of each of the Parser, EventManager, DataManager, and Environment objects are created to handle the animation process. The entire GUI is then enabled to provide the full functionality of the program to the user. While the trace file remains loaded, the animation can be played or paused, the camera can be manipulated, the information windows can be viewed, the preference settings can be changed, etc. When the user decides to close the trace file, the Parser, EventManager, DataManager, and Environment objects

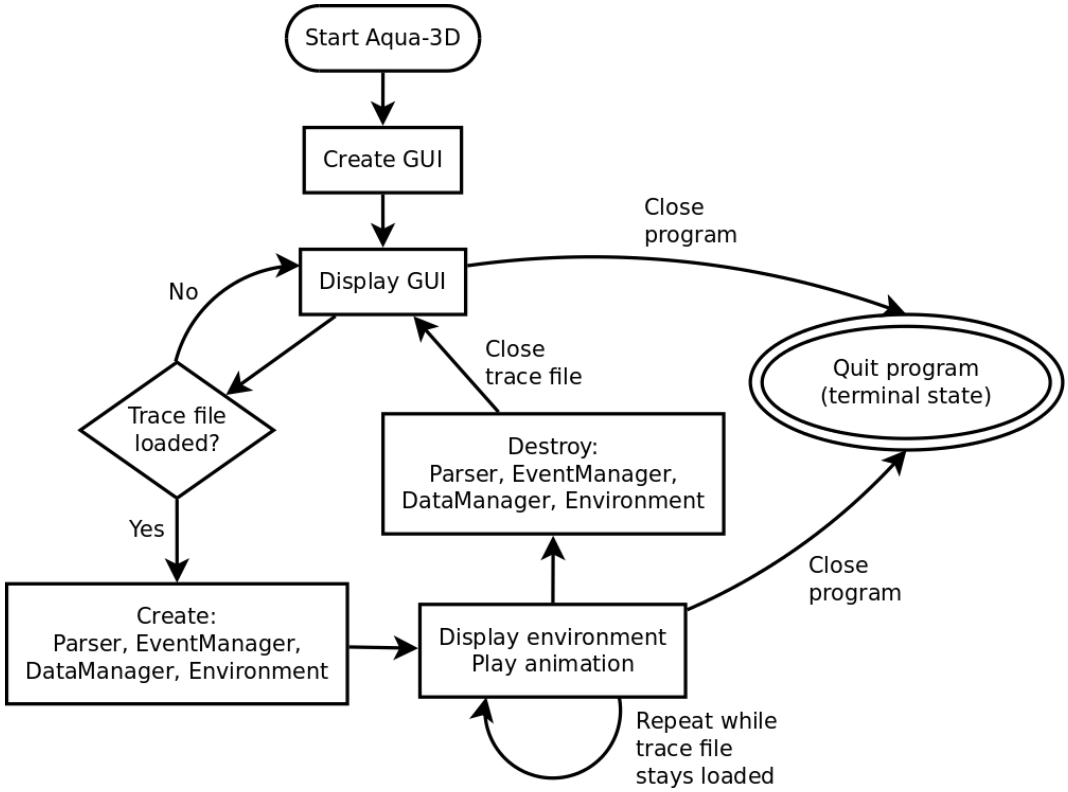


Figure 13: General Control Flow

are destroyed, all the GUI components are reset to their default values, the entire GUI except for the "File" and "Help" menus is disabled again, and the base GUI is displayed. When the user decides to quit the program, the entire program is destroyed.

4.4 Animation Process

Aqua-3D takes as input a trace file from *Aqua-Sim* that contains a time-indexed list of events. When a file is loaded, it first begins by initializing the topology layout before moving on to animate the events [12]. The parsing and animation process is shown in Figure 14.

When the user pressed the "Play" button on the PlaybackPanel, a timer is started that will call a function called Notify() every X-milliseconds until it is stopped. The Notify function() works as follows. If the animator is performing a "quick update", meaning the user selected a new time in

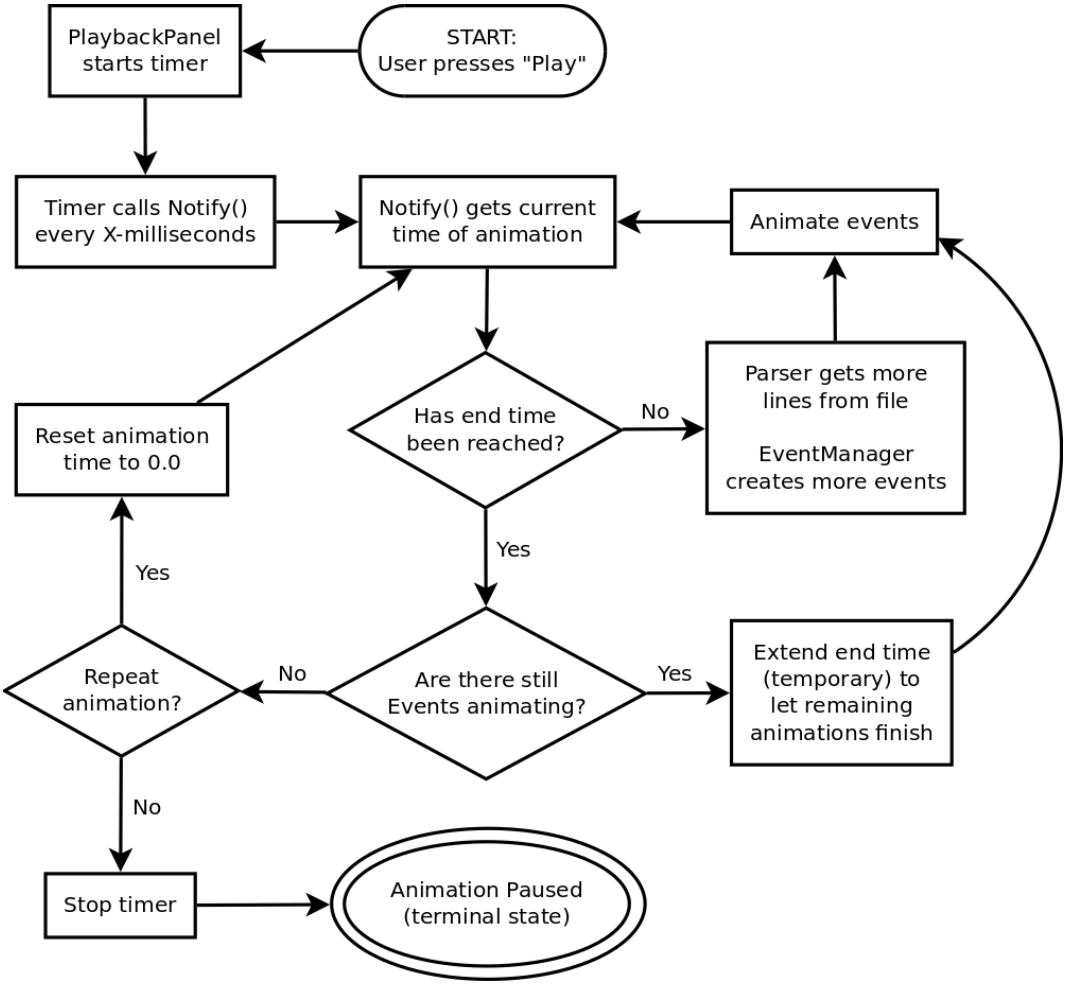


Figure 14: Parsing and Animation Process

the animation and the program has to quickly update all events leading up to that time, the function will simply tell Parser (which will tell the EventManager) to simply start at the beginning of the file, go through the events and update them without animating them. Once the selected time has been reached, the timer will be stopped. If the program is animating normally, Notify() will first check to see if the ending time of the animation has been reached. If not, it will tell the Parser to extract more lines from the trace file and tell the EventManager to continue creating, updating, and animating the Events. Once the ending time is reached, Notify() will check if all Events have finished updating and animating. If there are still Events that have not yet finished animating, the end time is temporarily extended by some set amount (currently 50 milliseconds) to allow those

animations to finish. Once all the animations are complete, Notify() will check if the user selected the option to loop the animation. If so, the time is reset to 0 and the animation will replay. If not, the timer is stopped and Notify() will no longer be called. Also, at any given time, the user is able to press the "Pause" button, which will immediately stop the timer and pause the animation.

4.5 Evaluation

To summarize Section 3.1, *Aqua-3D* was designed to satisfy the following objectives:

- Correctly interpret the trace files
- Present accurate, attractive, and recognizable visualizations
- Handle many different kinds of networks and events for different areas of network research
- Provide an intuitive graphical user interface
- Follow accepted GUI standards such as those explained in [14]
- Provide users with many options for playing, viewing, and manipulating the environment
- Achieve efficient computations and intelligent memory usage
- Code and documentation that is easy to read and understand
- Code is extensible to facilitate future additions and modifications

This section will present a qualitative analysis on how well *Aqua-3D* has achieved those objectives. However, as *Aqua-3D* has not yet been released to the public for extensive review and critiques, all of the following evaluations are solely from researchers at UConn's UWSN Lab.

To test *Aqua-3D*'s correctness in visualizing a simulation, an example scenario was produced by a researcher where he knew exactly how the protocol should have behaved and how the resulting

network topology should have appeared. After running the simulation and generating the trace file, he observed the visualization by *Aqua-3D* to see if it conformed to his expectations. He commented that the events and resulting network topology did indeed match his mental image of the network. Thus, it can be safe to assume that *Aqua-3D* is accurate in its animations to an acceptable degree. Extensive evaluation with many different kinds of trace files has not yet been done, but for the time being it can be said that *Aqua-3D* will correctly visualize a trace file, assuming the trace file contains all the necessary information.

The only animations that have had extensive thought put into them are HopEvent, ReceiveEvent, and signal collisions. HopEvent received the most consideration and feedback from the researchers at the lab and was constantly modified until it's current form. Having the semi-transparent spheres radiate from the node at regular intervals to indicate signal transmission was said to be easy to recognize, although using darkened spheres to indicate the beginning and end of a transmission is not entirely intuitive and may not be understood unless it was explained to the user. Perhaps that could be additional work for the future. The animation for ReceiveEvent was said to be intuitive, as it shows a small semi-transparent sphere shrinking inwards towards the node, to indicate receiving a signal. Likewise, showing a red explosion-like graphic at nodes where a signal collision occurred was said to be easy to notice and easy to understand that something undesirable happened at the node.

For now, *Aqua-3D* was written to handle only the minimum requirements of the UWSN Lab, which was to visualize underwater networks and animate HopEvents, ReceiveEvents, EnqueueEvents, DequeueEvents, DropEvents, and collisions. Additional underwater network events have not yet been considered but should not be difficult to implement. Terrestrial networks however, both wired and wireless, will require significant additional work, as the behavior and dynamics of terrestrial networks are far different than underwater acoustic networks.

Researchers at the lab have not made any criticisms of the layout of *Aqua-3D*'s GUI so it may be safe to say that there is nothing horribly wrong with it. Control groups are separated and arranged properly and are clearly labeled. In addition, almost every control will show a descriptive tool tip if the mouse cursor is hovered over it for a couple seconds, so the functionality of a control will never be completely unknown to the user. However, in the future it would be beneficial to give the buttons proper icons rather than using the letters and symbols they currently have.

Aqua-3D provides many useful features, the first of which are numberous preference options that researchers can use to customize the appearance and functionality of the program. Attributes such as the color of the environment elements (floor polygon, surface polygon, etc), the visibility of specific network events, the appearance of the grid lines, or the amount and direction each control button moves the camera are several examples, and more options can always be added if user demand is high. In addition, *Aqua-3D* allows users to save and load camera settings so that good camera angles can be easily reselected for later use. *Aqua-3D* can also dynamically visualize important elements of a network, such as the topology by drawing lines between transmission senders and receivers and packet collisions at nodes.

Lastly, evaluations regarding the cleanliness and documentation of the code as well as its extensibility are subjective to other developers, not users. Since there have been no other developers for *Aqua-3D* at the time of this writing, such evaluations cannot yet be meaningfully performed.

4.5.1 Software Performance

Aqua-3D is able to run smoothly on the development machine with very little lag in the camera controls or in the updating of the time displayed, even at full speed animation. On other machines however, some performance issues were observed depending on the hardware specifications of the

Test #	# of Allocations	Total Bytes	Bytes per Allocation
1	37,882	4,261,348	112.49
2	38,502	4,377,247	113.69
3	40,967	4,659,270	113.73
4	42,805	4,820,565	112.62

Table 1: Memory Usage

machine. Weaker machines had problems updating the time displays correctly at max speed and would lag somewhat when trying to manipulate the camera during an animation.

To perform a rudimentary evaluation on the memory usage of *Aqua-3D*, two profiling tools were used. The first was MemProf, which is able to generate a profile on the amount of memory allocated in a program as well as detect memory leaks [6]. *Aqua-3D* was run through the profiler several times and in several different ways in order to get a variety of performance differences and results. Table 1 shows the number of allocations made, the total number of bytes allocated, and the average number of bytes per allocation after an animation in *Aqua-3D*. All four tests were performed using the same trace file and running the animation at maximum speed. The following numbered list provides more detailed descriptions of the test scenarios.

1. No initial loading options (so annotation events and transmission paths were NOT loaded), animation was run from start to finish.

2. Both initial loading options (so annotation events and transmission paths WERE loaded), animation was run from start to finish.

3. Both initial loading options, time was skipped to near the very end, animation was run until the end, camera was constantly rotated with mouse.

4. No initial loading options, time was skipped to near the very end, ran animation for a short time at max speed, then closed and reloaded the same trace file again, this time with both

initial loading options, ran animation at normal speed, then skipped to near the very end, ran animation until the end, camera was constantly rotated with mouse.

So, the four profiling tests from MemProf indicates that the average memory usage during animation is around 4.5 MB of memory, with an average of around 40,000 allocations and thus an average of 113.13 bytes per allocation. Considering the typical size of RAM currently in most machines, 4.5 MB of memory used is not terrible. However, the profiling indicates that there are still many memory leaks that need to be carefully investigated and rectified.

The second analysis tool used was called Valgrind which can detect memory management and threading bugs as well as generate a detailed profile of the program [8]. It was used in conjunction with another tool called Alleyoop that provided a GUI front-end for Valgrind [4]. *Aqua-3D* was profiled by Valgrind only once, since it was only able to detect memory leaks. The same trace file from the MemProf test was used and the animation speed was set to maximum as well. The animation was run normally at first and then the time was skipped to near the very end. The animation was then allowed to complete. After *Aqua-3D* was exited, Valgrind produced the following summary:

- definitely lost: 396,304 bytes in 2,515 blocks
- indirectly lost: 100,182 bytes in 1,431 blocks
- possibly lost: 1,725,951 bytes in 15,022 blocks
- still reachable: 556,724 bytes in 7,141 blocks

The documentation on the Valgrind website provided the following explanations for the “lost” descriptions:

- Definitely lost: no pointer to the memory block can be found, possibly due to losing the pointer at some earlier point in the program, so the programmer could not possibly have freed it at program exit
- Indirectly lost: the memory block is lost because all the blocks that point to it are themselves lost (for example, losing the pointer to the root of a tree will cause all its children to be lost as well)
- Possibly lost: a chain of one or more pointers to the memory block has been found, but at least one of the pointers is an interior-pointer, which could just be a random value in memory that happens to point into a block
- Still reachable: pointers to the block of memory is still found, so the programmer could, at least in principle, have freed it before program exit

So, while Valgrind did not provide empirical results on the memory usage of *Aqua-3D*, it did provide a very helpful summary of memory leaks in the program. Individual results in the Alleyoop GUI can be expanded for a detailed stack trace, which will provide additional help in locating the source of the leak.

To summarize, while *Aqua-3D* is able to perform acceptably well, it is not without issues. There are still a plethora of memory leaks which will need to be rectified. It is worth noting that these memory values are only from the laptop that *Aqua-3D* is being developed on. Empirical performance tests have not yet been run on other machines, so these results cannot be taken as completely representative of the program overall. So, while *Aqua-3D* runs well enough with little speed/performance issues, even at max speed, on development laptop, it is currently unknown how it will fare on other machines.

4.5.2 Testing

Once the development and implementation of the core *Aqua-3D* functionality was completed, it needed to be tested on other machines to make sure it could be installed and work on different systems. To facilitate the process, virtual machines were used to run multiple operating systems on the same computer so that different physical machines were not necessary. VMWare Player was the virtual machine program of choice as it was free and could run all the important Linux operating systems. The following Linux OS's were used:

- Ubuntu 8.04, 9.04, 9.10, 10.04, 10.10
- Debian 5.0.6
- Fedora 14

Of all the Linux distributions that were used to test *Aqua-3D*, Ubuntu 9.04, 9.10, and 10.04 on virtual machines were unable to successfully run the program without problems. All three encountered two significant issues that unfortunately remain unresolved.

The first is what has been described as the **X Window System error**. The affected distributions of Ubuntu may require the installation of *nvidia-current* or *nvidia-glx-<version_number>* drivers to function correctly, otherwise a segmentation fault will occur when attempting to load a trace file. After the necessary driver is installed, starting *Aqua-3D* with the command line *./aqua3d* or by double-clicking on the executable will open the program as normal and loading a trace file via the "File -> Open" window will work. However, attempting to start *Aqua-3D* with the command line *./aqua3d -t <trace_file_name>*, which will try to load a trace file immediately upon startup, will give the following error and terminate the program:

```
The program 'aqua3d' received an X Window System error.
```

```
This probably reflects a bug in the program.
The error was 'BadMatch (invalid parameter attributes)'.
    (Details: serial 3534 error_code 8 request_code 66
        minor_code 0)
    (Note to programmers: normally, X errors are reported
        asynchronously; that is, you will receive the
        error a while after causing it. To debug your
        program, run it with the --sync command line option
        to change this behavior. You can then get a meaningful
        backtrace from your debugger if you break on the
        gdk_x_error() function.)
```

The only work-around for this issue is to simply start *Aqua-3D* in any way except with the command line option to load a trace file immediately.

The second problem, described as the **extension "GLX" missing on display**, is much more detrimental and will completely prevent *Aqua-3D* from functioning in any manner. The issue occurs when the X Server is restarted, either by restarting the (virtual) machine or by using *Ctrl+Alt+Backspace*. Once that happens, any attempt to run *Aqua-3D* will result in the following error and terminate the program:

```
Xlib:  extension ``GLX" missing on display ``:0.0".
freeglut (? ??): OpenGL GLX extension not supported by
    display ':0.0'
```

There are only two work-arounds but neither is very elegant. The first is to simply run *Aqua-3D* on a virtual machine with Fedora 14, Debian 5, or Ubuntu 10.10 installed. Those operating systems

have been able to successfully run *Aqua-3D*. The other option is to remove and reinstall the nvidia driver, restart the X Server, and then reinstall the nvidia driver. Running *Aqua-3D* again (without trying to load the trace file immediately as explained in **X Window System error**) should work. Unfortunately, every time the X Server is restarted from this point, *Aqua-3D* will fail to start again and the uninstalling/reinstalling process will need to be repeated.

Strangely, actual machines running Ubuntu 9.04, 9.10, or 10.04 (including the development laptop, which is running 10.04) were able to successfully run *Aqua-3D*. Therefore, it may be safe to hypothesize that the issue might have to do with virtual machines themselves and how they handle display and graphics drivers.

Chapter 5

Field Test Visualization

Whereas simulations provide inexpensive and repeatable environments for testing and debugging, larger-scale field tests are necessary to evaluate the performance of new research under real-world conditions. Field test results are often recorded in trace files, and similar to simulation trace files, they often contain large amounts of detailed data that are tedious and difficult to understand. Thus, visualization again plays a vital role. It is able to clearly illustrate the underwater events that researchers were unable to observe and present them in a way that makes understanding the system quicker and more intuitive.

This chapter will discuss *Aqua-3D*'s ability to visualize trace files obtained from field tests conducted by researchers at UConn's UWSN Lab. The first section will present an overview of the field test beds, including the motivation behind conducting the experiments and a description of each of the test scenarios. The next section will present *Aqua-3D*'s visualizations of the field test trace files. The last section will evaluate *Aqua-3D*'s correctness in visualizing the field test files.

5.1 Test Bed Overview

It is extremely important to note that *Aqua-3D* was first and foremost designed to visualize trace files from *Aqua-Sim*. Attempting to utilize *Aqua-3D* for field test visualization came about only after its primary functionality was complete. There are several key differences between how events are controlled in simulations versus field tests. In simulations, nodes are positioned using Cartesian coordinates, network events are centrally controlled, and everything is perfectly time synchronized. On the other hand, the events occurring during a field test are individually controlled by the separate nodes, real time is used (Unix time stamps), and the timings themselves may be out of sync. Because of this, additional work was required to adapt the existing *Aqua-3D* software to work with field test trace files.

5.1.1 Trace Files

During the course of the experiments, each of the nodes maintained log files that recorded every action and event. Also, for any packet enqueue, packet dequeue, packet drop, signal transmission, and signal receive events that occurred, an additional line was written using the *NAM* trace file format as described in Section 3.5. These lines were then extracted and combined into separate .nam trace files.

Additional preprocessing was required to make the files readable by *Aqua-3D* however. First, separate trace files were produced for each node, and the nodes themselves each maintained different trace files for different protocols being tested. So, it was necessary to combine all the separate files from the nodes for a specific protocol test into one singular trace file containing all the events of that test for all the nodes. Next, the time stamps recorded by the nodes were Unix time stamps in the format of *seconds.microseconds*, which is the number of seconds elapsed since January 1, 1970. As the starting time values of the events were in real-time, it was necessary to offset all the

time values so that the first event in the trace file started at time 0. To do this, the event lines of the merged file were first sorted in chronological order with respect to time, with the earliest event at the top. Then, the time value of the earliest event (the lowest numerical value) was subtracted from the time values of every event in the file. This maintained the timing between all events while making the first event begin at time 0. Lastly, the individual trace files did not have information on the topology of the network, meaning the positions of the nodes and the dimensions of the environment were unknown. GPS coordinates of the nodes were available, however they needed to be converted into Cartesian coordinates (using the formulas found in [2]) so that they could be positioned correctly in OpenGL. Also, as the resulting calculated positions tended to have large coordinate values, they were all scaled to be positioned closer to the origin (0,0,0) while still maintaining their relative distances from each other. Afterwards each of the world width, height, and depth dimensions were established simply by being set slightly larger than the largest coordinates in each of the three corresponding axes. Once all the position and dimension calculations were complete, that information was added as initialization events to the top of the trace file.

5.1.2 Test Locations

The first field test was conducted from July 23, 2010 to July 25, 2010 in the Atlantic Ocean approximately 120 kilometers off the coast of Atlantic City, New Jersey. The layout of the test bed can be seen in image (a) of Figure 15. On the ocean surface were six buoys (nodes), each approximately 2 to 2.5 kilometers apart from each other (in the circle). The network had a fully connected topology, meaning every node was able to communicate with every other node. Each node had an underwater acoustic modem suspended beneath it in the water and had a radio frequency antenna on top. The buoys were monitored and controlled by the researchers on a boat via radio signals. The subject of the experiment was a dynamic routing protocol.

(a) Atlantic Ocean (b) Chesapeake Bay

Figure 15: Field Test Beds

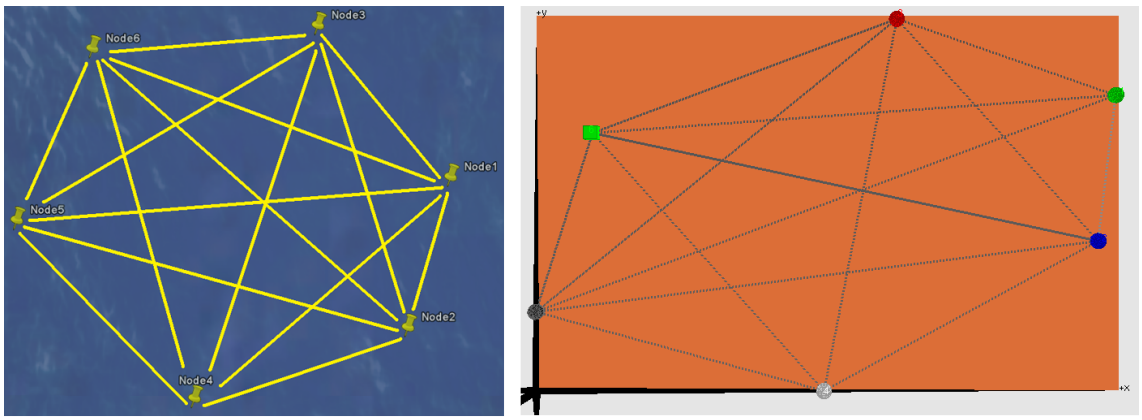
The second field test was from March 7, 2011 to March 11, 2011 in Chesapeake Bay off the coast of Maryland, near Chesapeake Beach. The positions of the buoys can be seen in image (b) of Figure 15, with each of the nodes being approximately 1 to 2.5 kilometers from shore. The nodes in this experiment also had an acoustic modem suspended underneath it in the water and a radio frequency antenna on top. In addition, the CBD (Chesapeake Bay Detachment of the Naval Research Laboratory) station on shore had an RF antenna that allowed the researchers to monitor and control the nodes. Numerous protocols were being tested during this experiment, including ALOHA, Broadcast MAC, and Slotted FAMA. It is also very important to note that while all the nodes were within range of each other (permitting a fully-connected topology), the actual topologies during the experiments were controlled using configuration files. The files allowed or restricted communication between nodes, creating cluster, lattice, or string topologies and establishing the possible routes packets were allowed to traverse. In terms of the conditions, one of the UConn UWSN Lab researchers who participated in the experiment recounted that Monday through Wednesday had good weather and good acoustic communications between the nodes.

However, Thursday was struck with heavy rain and windy conditions, which made communications much more difficult. On Friday, the weather improved but there were issues with batteries running out of energy or having too low voltage. For instance, the low voltage in one modem was triggering the modem's reboot threshold whenever it tried to receive a signal so it was constantly and unintentionally restarted whenever a signal was sent to it. This first hand account illustrates the unpredictability of the real-world environment and thus the need for running field tests to make sure systems and protocols can function in those conditions.

5.2 Visualization

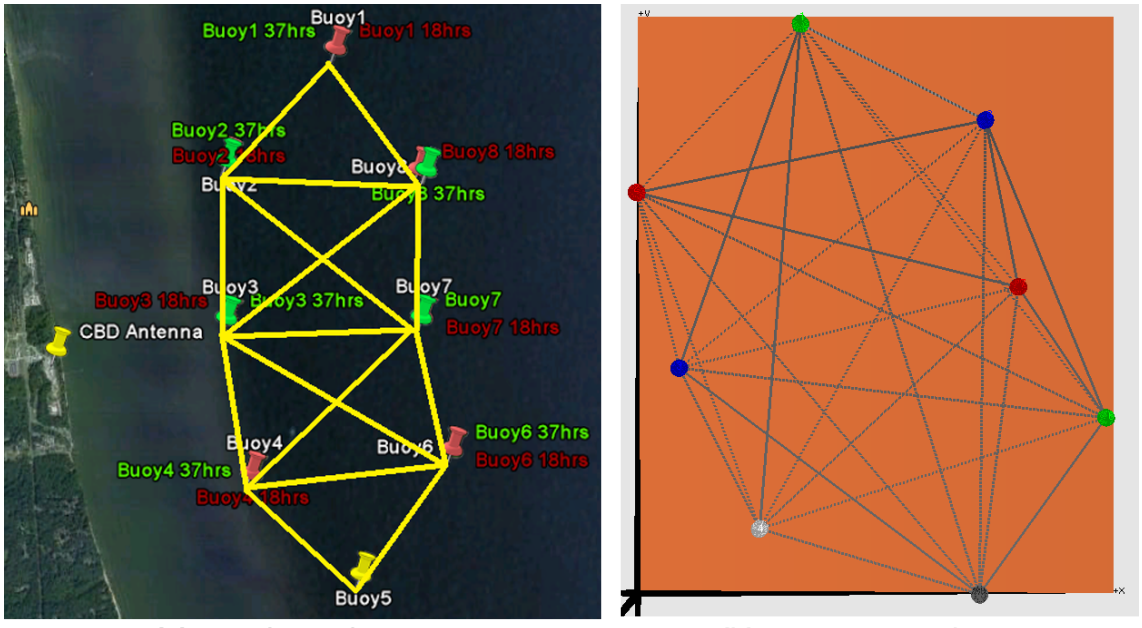
Both field tests recorded multiple trace files on multiple days and for multiple protocols. For the Atlantic Ocean test, the trace file from the first day (July 23) was selected to be visualized because it contained the greatest number of events. The fully-connected topology of the network can be seen in image (a) of Figure 16. For the Chesapeake Bay test, two trace files were selected, each with different topologies as established by the configuration files. The first one was testing the ALOHA protocol and with a lattice topology, as seen in image (a) of Figure 17. The second one was testing the Slotted FAMA protocol and with a string topology, as seen in image (a) of Figure 18.

After the selected trace files from the field test were processed, they were loaded by *Aqua-3D* and the animations were run. The visualization for the Atlantic Ocean test can be seen in image (b) of Figure 16. The visualizations for the lattice topology and string topologies of the Chesapeake Bay tests can be seen in image (b) of Figure 17 and in image (b) of Figure 18 respectively.



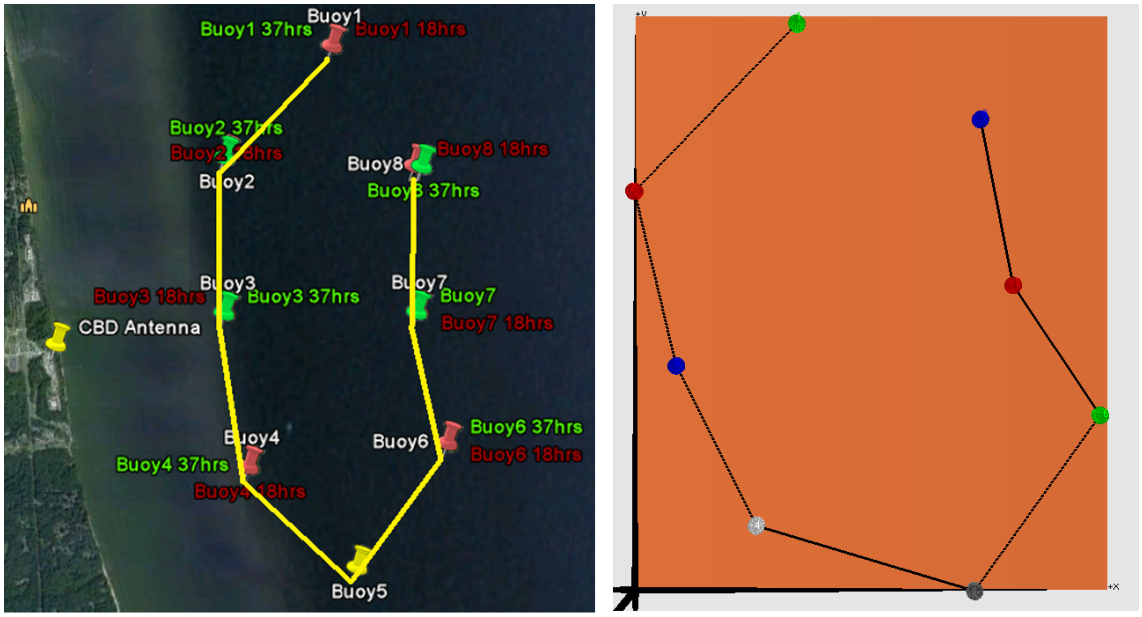
(a) Google Earth image (b) Aqua-3D visualization

Figure 16: Atlantic Ocean Test - *Aqua-3D* Visualization vs. Actual Topology



(a) Google Earth image (b) Aqua-3D visualization

Figure 17: CB Test (ALOHA, Lattice) - *Aqua-3D* Visualization vs. Actual Topology



(a) Google Earth image (b) Aqua-3D visualization

Figure 18: CB Test (Slotted FAMA, String) - *Aqua-3D* Visualization vs. Actual Topology

5.3 Evaluation

5.3.1 Atlantic Ocean Test

As seen in Figure 16 for the Atlantic Ocean test, the resulting *Aqua-3D* topology in image (b) is fairly accurate to the actual topology in image (a). The transmission path lines were able to faithfully replicate the layout of the network. It is important to note that nodes 1 and 4 were down during the actual experiment, explaining the missing path line between them. Other nodes were able to transmit signals to nodes 1 and 4, but since 1 and 4 did not generate any transmissions, no transmissions occurred between them. However, the node positioning is not completely perfect because, as can be seen visually or calculated using the coordinates, the distances between some nodes are larger or smaller than their actual distances in the field test.

5.3.2 Chesapeake Bay Test 1

For the first Chesapeake Bay test with the lattice topology, a comparison of the images in Figure 17 shows that *Aqua-3D* was unable to accurately visualize the field test. The positions of the nodes were fairly accurate, although with similar discrepancies in distances as the Atlantic Ocean test. However, the visualization as seen in image (b) shows a fully-connected topology whereas the lattice topology in image (a) does not have connections between all pairs of nodes. This was primarily because *Aqua-3D* did not know about the configuration file that established the topology. How *Aqua-3D* currently determines transmission paths is by drawing a line from the sender to the receiver during HopEvents, which is when a transmission is sent. For unicasts, only one line is drawn from the source to the intended destination. For broadcasts, lines are drawn from the sender to all other nodes that are within transmission range, regardless of whether or not those nodes actually received the packet. So in this trace file, since all the HopEvents were broadcasts and all the nodes were within transmission range of each other, lines were drawn between every pair of nodes, even if communication between them was not allowed by the configuration file. The reason *Aqua-3D* interprets the HopEvent lines this way is because there is currently no information in ReceiveEvent lines that indicate who the original sender of a transmission was; the line only specifies the node at which the event occurs. If receive events did include that information, then drawing the transmission lines based on ReceiveEvents would result in completely accurate topology visualizations. That is because the existence of a ReceiveEvent indicates that communication between the receiver and original sender was allowed by the configuration file. However, this will require modifications to how *Aqua-Sim* outputs receive event lines as well so that *Aqua-3D* will work for both the simulator and field tests.

5.3.3 Chesapeake Bay Test 2

For the second Chesapeake Bay test with the string topology as seen in Figure 18, a comparison of the images shows that *Aqua-3D* was able to correctly visualize the topology of the field test. As described in the previous paragraph, how *Aqua-3D* currently determines transmission paths for HopEvent unicasts is by drawing the line only between the transmission's source and its intended destination. Since all the HopEvents in this trace file were unicasts, there was no need for *Aqua-3D* to know about the configuration file and thus it was able to correctly visualize the topology using only its current functionality.

5.3.4 Issues Encountered

When visualizing the field tests in *Aqua-3D*, the primary requirements were:

- The positions of the nodes in the visualization must match their corresponding real-world positions.
- The visualized network topology (drawn by the transmission path lines) must match the intended actual topology.
- The event animations must be correctly timed so that they start and end when they are supposed to.

Unfortunately, several issues were encountered that hindered the accuracy of the visualizations.

First, the conversion from the GPS coordinates of the nodes to cartesian coordinates was not precise. Shorter distances between nodes seemed to be better preserved, such as between nodes 3 and 7 in the Chesapeake Bay test. However, the greater the distances between the nodes, the larger the errors. For instance, nodes 1 and 5 in the Chesapeake Bay test were approximately 3.3

kilometers apart according to Google Earth, while calculating their distance using their Cartesian coordinates resulted in a distance of approximately 2.1 kilometers. This will cause problems with determining the topology of the network using the transmission paths because *Aqua-3D* currently draws lines to other nodes that are within transmission range of the HopEvent's source node. If the distances were incorrect, then paths may be drawn where they are not supposed to be or may not be drawn where they should. This is not entirely the fault of *Aqua-3D* however, as it was originally intended to visualize simulation trace files where node positions would already be in Cartesian coordinates.

Secondly, for the Chesapeake Bay test, all the nodes used a configuration file to determine the topology of the network. This means that while all the nodes were in transmission range of each other (would be a fully-connected topology), the configuration file allowed or restricted communication between certain nodes. Unfortunately, *Aqua-3D* was not aware of this file and thus will draw transmission paths between all nodes within range if the transmission is a broadcast. This explains why the resulting visualization of the lattice network in image (b) of Figure 17 does not match the intended layout in image (a) of the figure. All of the HopEvents in this trace file were broadcasts, and because all the nodes were in range of each other, lines were drawn between all of them, incorrectly resulting in a fully connected network. The string network in image (a) of Figure 18 was rendered correctly however, as seen in image (b) of the same figure, because all of the HopEvents in that trace file were unicasts and thus lines were drawn only between the transmission's sender and its intended recipient. If *Aqua-3D* instead determined transmission paths based on ReceiveEvents as discussed in Section 5.3.2, then the visualized topologies would be completely accurate. This cannot be seen as a complete fault of the current *Aqua-3D* however, as it was never designed to accommodate a topology configuration file. Similarly, the way it

determines transmission path lines was based on the current format of the .nam trace file, and so modifying that will also require modifications to *Aqua-Sim*.

Lastly, field tests may take several hours to complete, with individual events possibly lasting several minutes and with even more time between events. Because events were time stamped by the nodes using real time, the resulting animations from the trace files would also require hours to complete with many long periods where nothing would occur. So, in an attempt to speed up the animation so that the time required to view the animation would be drastically reduced, the time stamps were scaled down by 1000 so that a animation lasting 4000 seconds would only last four seconds. Unfortunately, this was detrimental to the timing of events, with animations beginning or ending before they were supposed. The actual time stamps of the events may also be slightly incorrect in relation to other events due to mistimed startups or restarts. This will cause such events to be animated out of order even though they occurred correctly during the actual field test. However, it still can be assumed that the events themselves were animated correctly, as *Aqua-3D* has successfully animated shorter simulations. So, matters regarding how *Aqua-3D* handles visualizations of very long trace files will need to be modified.

To summarize, while the attempts at visualizing the field tests were not completely successful, they did bring to light a number of improvements and modifications that can be made to *Aqua-3D*. As field tests will make use of GPS coordinates for the positions of their nodes, a much more accurate conversion method to transform GPS coordinates into cartesian coordinates is needed. Also, while *Aqua-3D* can correctly determine topologies using HopEvents for simulation trace files to an extent, more accuracy can be achieved by using ReceiveEvents instead. Lastly, a much better way of adjusting the animation speed and jumping forward in time is needed, especially for long trace files, as the current methods can take too much time and be too slow.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

With design inspirations from the 2D animator *NAM*, *Aqua-3D* has been written from the ground up to visualize underwater simulations in full 3D. It is able to accurately visualize trace files generated by *Aqua-Sim* and present a freely controllable camera that allows users to view the animations from any angle. Additional functionalities, such as windows that display other important information on the simulation, the ability to jump to annotated events, the ability to save and load camera angles, and a wide range of options for customizing the look and feel of the program, make *Aqua-3D* a feature-rich tool that is capable of satisfying the needs of underwater network researchers. As the field of underwater networking continues to develop and mature, *Aqua-3D* will also mature and grow as a respectable research tool.

6.2 Future Work

Although it accomplishes its primary objective, which is to accurately visualize trace files from *Aqua-Sim*, there are many ways in which *Aqua-3D* can be enhanced.

Many improvements to the current code and functionality can be done, the first of which is of course locating and eliminating the memory leaks described in Section 4.5.1. There may be additional optimizations that can be made to the code as well to improve its performance overall, such as multi-threading. It is possible to improve all the animations as well, in terms of their design and accuracy. While the animations for HopEvents and ReceiveEvents are completed, further refinements to their appearance and animations may be done to make them even more attractive and intuitive. Additionally, the animations for EnqueueEvents, DequeueEvents, and DropEvents are still very basic and need to be fully realized. As mentioned in Section 3.8, the method for updating the animations to a selected time is currently impractical for large trace files because it needs to start reading the file from its beginning. Changing the method so that it intelligently selects a point in the middle of the file will improve the speed of the animation updating greatly while also making it usable for large trace files. The difficulty will lie in calculating which part of the file to begin reading from so that the animations at the selected time are perfectly correct, as if the user had played the animation to that point from the beginning.

Along with improvements to the current functionality, there are many additional features that can be implemented to increase *Aqua-3D*'s feature set. First, *NAM* allows users to click and interact with visualized elements in its animation window. For instance, a user is able to click on a link or a packet to open a new window that displays graphs or time plots related to the clicked-on object. An additional feature of *NAM* is the ability to run and synchronize multiple *NAM* instances visualizing different trace files, which allows for easy side-by-side comparisons. Such features may be valuable to researchers using *Aqua-3D* as well. Lastly, as discussed in Section 5.3, the current method in which field test trace files are processed to be readable by *Aqua-3D* is not yet completely correct. While not entirely a feature of *Aqua-3D* itself, such functionality could be integrated into *Aqua-3D* or at least used as a starting point for other researchers to implement

their own processing applications to make field test trace files readable by *Aqua-3D*. Once that is able to function properly, it may even be possible to provide an interface in *Aqua-3D* that can accept data streamed to it from nodes during a field test in order to visualize the on-going test in real time!

Overall, with the functionality currently implemented along with the wide range of possible enhancements and additions, *Aqua-3D* can serve as a milestone for the future development of 3D visualization tools for underwater networks.

Chapter 7

Appendix

7.1 Relevant Links

- The *Aqua-3D* package can be downloaded at

  http://ubinet.engr.uconn.edu/mediawiki/images/a/a8/Aqua3d-1.0.0-src.tgz

- The Wiki for the site is located at

  http://ubinet.engr.uconn.edu/mediawiki/index.php/Aqua-3D

- The online user's manual is located at

  http://ubinet.engr.uconn.edu/mediawiki/index.php/Aqua-3D_manual

- Doxygen documentation for the source code can be found at

  http://ubinet.engr.uconn.edu/ matthewtran/aqua3d/documentation/html/index.html

7.2 System Requirements

- **Linux** operating system
  - Has been installed and tested successfully on the following:

    * Ubuntu 8.04, 10.04 LTS, 10.10
    * Debian 5.0.6
    * Fedora 14
  - Installing onto an OS run by a virtual machine (like VMWare) is not recommended because the animator's performance may be severely affected

- **g++** compiler
- **NVIDIA binary Xorg driver**
- **freeglut-2.6.0** for the 3D rendering
- **GTK+ 2.0** for the GUI
- **wxGTK-2.8.11** for the GUI

7.3 Installation Instructions

1. Install **freeglut-2.6.0** if not already installed
2. Install **nvidia-glx** if necessary
3. Install **GTK+ 2.0** if not already installed
4. Install **wxGTK-2.8.11** with the following steps (**if not installing from package manager**):
   (a) Extract the package with *tar -xzf* and *cd* into the folder
   (b) Use *./configure –with-opengl –enable-shared –enable-unicode* to configure the installation
   (c) *make*

(d) *sudo make install*

5. Extract the Aqua-Sim package with *tar -xzf* and *cd* into the folder

6. Enter *./install* and the install script will build the entire package (including Aqua3D)

7. Try running the program by navigating into the Aqua3D directory and entering *./aqua3d*

   - If it outputs an error such as "*error while loading shared libraries... cannot open shared object file: No such file or directory*":
     - Enter */sbin/ldconfig -v* to set up the links for the shared binaries and rebuild the cache
   - If it still doesn't work you will also need to add the following line into your *.bashrc* file (located in your *home* directory):
     - **export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH**

8. Aqua3D is now ready for use

Bibliography

[1] Ns-2 official website. http://www.isi.edu/nsnam/ns/.

[2] Spherical coordinates and the gps. http://www.math.montana.edu/frankw/ccp/cases/Global-Positioning/spherical-coordinates/learn.htm, 1996.

[3] Nam official website. http://www.isi.edu/nsnam/nam/, July 2002.

[4] Alleyoop official website. http://alleyoop.sourceforge.net/, November 2009.

[5] Aqua-sim official wiki. http://ubinet.engr.uconn.edu/mediawiki/index.php/Aqua-Sim, November 2010.

[6] Memprof official website. http://projects.gnome.org/memprof/, July 2010.

[7] Ns-2 and nam official wiki. http://nsnam.isi.edu/nsnam/index.php/Main_Page, October 2010.

[8] Valgrind official website. http://valgrind.org/, February 2011.

[9] Ian F. Akyildiz, Dario Pompili, and Tommaso Melodia. Underwater Acoustic Sensor Networks: Research Challenges. *7*, 3(3):257–281, 2005.

[10] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[11] Jun-Hong Cui, Jiejun Kong, Mario Gerla, and Shengli Zhou. Challenges: Building Scalable Mobile Underwater Wireless Sensor Networks for Aquatic Applications. *IEEE Network, Special Issue on Wireless Sensor Networking*, 20(3):12–18, June 2006.

[12] Deborah Estrin, Mark Handley, John Heidemann, Steven McCanne, Ya Xu, and Haobo Yu. Network visualization with the VINT network animator nam. Technical Report 99-703b, University of Southern California, March 1999.

[13] John Heidemann, Wei Ye, Jack Wills, Affan Syed, and Yuan Li. Research challenges and applications for underwater sensor networking. In *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, volume 1, pages 228–235, April 2006.

[14] James Hobart. Principals of good gui design. http://www.classicsys.com/css06/cfm/article.cfm?articleid=20, October 1995.

[15] Jiejun Kong, Jun-Hong Cui, Dapeng Wu, and Mario Gerla. Building Underwater Ad-hoc Networks and Sensor Networks for Large Scale Real-time Aquatic Application. In *Proceedings of IEEE Military Communications Conference*, MILCOM'05, pages 1535–1541, October 2005.

[16] Lanbo Liu, Shengli Zhou, and Jun-Hong Cui. Prospects and Problems of Wireless Communication for Underwater Sensor Networks. *Wireless Communications & Mobile Computing*, 8(8):977–994, May 2008.

[17] Jim Partan, Jim Kurose, and Brian Neil Levine. A Survey of Practical Issues in Underwater Networks. In *Proceedings of the 1st ACM international workshop on Underwater networks*, WUWNet'06, pages 11–24, 2006.

[18] Peng Xie, Zhong Zhou, Zheng Peng, Hai Yan, Tiansi Hu, Jun-Hong Cui, Zhijie Shi, Yunsi Fei, and Shengli Zhou. Aqua-Sim: An NS-2 Based Simulator for Underwater Sensor Networks. In *IEEE/MTS Oceans 2009*, pages 1–7, 2009.