NAVAL POSTGRADUATE SCHOOL
Monterey, California

19980811 125

THESIS

REAL-TIME 3D SONAR MODELING AND VISUALIZATION

by

Timothy M. Holliday

June 1998

Thesis Advisor:	Don Brutzman
Thesis Co-advisor:	Kevin B. Smith

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 1998	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Real-Time 3D Sonar Modeling And Visualization			**5. FUNDING NUMBERS**
6. AUTHOR(S) Holliday, Timothy M.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT

Virtual world simulations are realistic when each individual component is simulated in a manner that reflects reality. For an underwater virtual world that simulates acoustic detection, a physically based sonar propagation model is required if ranges in excess of tens of meters are expected.

This thesis creates an application programming interface (API) for real-time 3D computation and visualization of acoustic energy propagation. The API provides features for generating complex physically based sonar information at interaction rates, and then visualizing that acoustic information. The simulation is programmed in Java and runs either as a stand-alone program or as a script in a web browser. This program generates Virtual Reality Modeling Language (VRML 97) compliant code that can be viewed from any VRML-capable browser. This approach allows the characteristics of the energy propagation to be calculated with high precision and observed in 3D.

As sonar system information bandwidth becomes larger, more intuitive ways of presenting information to a user will be required. Higher information density in a more intuitive format can free the user from integrating the data himself and allow quicker reaction times. This thesis and the API provide the foundation for fundamental advances in sonar modeling and visualization.

14. SUBJECT TERMS sonar, ray tracing, modeling, visualization, VRML			**15. NUMBER OF PAGES** 250
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

REAL-TIME 3D SONAR MODELING AND VISUALIZATION

Timothy M. Holliday
Lieutenant, United States Navy
B.S., University of New Mexico, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
June 1998

Author: [illegible]
Timothy M. Holliday

Approved By: [illegible]
Don Brutzman, Thesis Advisor

[illegible]
Kevin B. Smith, Thesis Co-advisor

[illegible]
William Maier, Chair
Department of Physics

ABSTRACT

Virtual world simulations are realistic when each individual component is simulated in a manner that reflects reality. For an underwater virtual world that simulates acoustic detection, a physically based sonar propagation model is required if ranges in excess of tens of meters are expected.

This thesis creates an application programming interface (API) for real-time 3D computation and visualization of acoustic energy propagation. The API provides features for generating complex physically based sonar information at interaction rates, and then visualizing that acoustic information. The simulation is programmed in Java and runs either as a stand-alone program or as a script in a web browser. This program generates Virtual Reality Modeling Language (VRML 97) compliant code that can be viewed from any VRML-capable browser. This approach allows the characteristics of the energy propagation to be calculated with high precision and observed in 3D.

As sonar system information bandwidth becomes larger, more intuitive ways of presenting information to a user will be required. Higher information density in a more intuitive format can free the user from integrating the data himself and allow quicker reaction times. This thesis and the API provide the foundation for fundamental advances in sonar modeling and visualization.

TABLE OF CONTENTS

ACKNOWLEDGMENT

I would like to first thank my wife. Without her constant support and loving care of our family, none of this would be possible. Thanks also go to my thesis advisors Dr. Brutzman and Dr. Smith. They spent much time with me on this thesis, even though it was not funded.

I. INTRODUCTION

A. OVERVIEW

This thesis provides an interactive three-dimensional (3D) sonar propagation and rendering application programming interface (API) for use in virtual worlds as well as stand-alone simulation and visualization programs. The interactive program allows a simulation to send virtual sonar energy pulses into a virtual world for detection of target objects. This allows the simulation to have real-time and realistic sonar information from the virtual world. The stand-alone programs allow the compilation of data on how the sonar pulse energy interacts with the ocean environment. These programs allow the user to view the data as 3D virtual worlds. These worlds allow the researcher the ability to immerse himself in the data and develop a deeper understanding of the data set. The accuracy of this physically based sonar model is very comparable to other widely accepted sonar propagation models. Its accuracy is achieved by using the recursive ray acoustics (RRA) algorithm [Ziomek, 1996].

The *Phoenix* autonomous underwater vehicle (AUV) is a research robot at the Naval Postgraduate School (NPS). It is designed to navigate autonomously and perform a variety of tactical functions in the ocean. In order to perform complex testing without getting the robot "wet," a virtual world for the robot to interact with was created. This world responds to the robot's control signals, allowing it to navigate in the virtual world, and it provides external sensor stimulation information to the robot for feedback and information collection. In this type of world two models are highly important: the hydrodynamics and the sonar. The hydrodynamics model is a full-featured six degree-of-freedom model which accounts for cross body-flow. The current sonar model is a straight-line geometric model. In essence, the model depicts acoustic energy as traveling in a conical beam through the ocean. This sonar model is satisfactory for distances on the order of a few hundred meters, but for ranges on the order of kilometers will not provide realistic results.

B. MOTIVATION

Real time simulation of sonar information and visualization of that information are important for several reasons. The use of a sonar simulator for training of crews and AUV's is enhanced by physically based, realistic sonar information. In this instance not only must the data be realistic but it also must be real time. Tactical planning and visualization is another field that benefits from realistic 3D sonar information. If an area of ocean can be accurately simulated then more detailed plans can be made for the searching and controlling of that area. The ability to generate an API that can be used in these capacities is the motivation behind this thesis.

C. PURPOSE

The purpose of this thesis is to provide an application programming interface (API) for use in investigating the propagation of acoustic energy in shallow water regions of the ocean. This API is to provide a convenient and intuitive interface to the user to allow him to easily integrate physics based ocean acoustics into a simulation. The API will provide methods for the user to specify a virtual world of arbitrary complexity, propagate acoustic energy in the world, collect information on any objects encountered and visualize the energy as it propagates in the world. The user is able to vary all aspects of the virtual world and of the acoustic energy pulse.

D. ORGANIZATION

This thesis provides all the information necessary to understand the motivation for accurate sonar simulations, the construction of the sonar model, the implementation of the model in Java, 3D visualization and the accuracy of the model compared to other generally accepted models.

Chapter II, Related Work, provides information about the motivation for this thesis. It describes previous work in the fields of tactical, virtual world and sonar modeling and simulation. Chapter III, Problem Statement, relays in detail the problem to be solved, possible solutions and the chosen solution. Chapter IV, Recursive Ray Acoustics (RRA) Derivation, describes the development of the ray acoustic approximation to the linear acoustic wave equation. It develops the equations that govern the propagation of acoustic energy in the sonar model. Chapter V, Modeling of Recursive Ray Acoustics (RRA), describes each sub-model in the RRA, recursive ray acoustics, model and the methods and data required in each of these sub-models. Chapter VI, Sonar Visualization, provides the same information on visualization that Chapter V provided on RRA. Chapter VII, Model Implementation and Integration, discusses the method and vehicle of implementation of the RRA and visualization models. Chapter VIII, Simulation Results, compares time, position and energy transport results from the model to other generally accepted models. Chapter IX, Conclusions and Recommendations, summarizes the thesis and provides information about improvements and extensions that can be made to the simulation.

II. RELATED WORK

A. INTRODUCTION

Many past works play an important role in the development of this thesis. They are summarized in this chapter in order to provide the motivation and background needed by the reader. The Phoenix AUV virtual world is the original project motivating this thesis. The linearly geometric sonar model of the virtual world needs to be made more robust so that real-world results can be obtained. In the Manta UUV project, tactical visualization simulations show that multiple mine clearing tactics can be evaluated both analytically and visually to find an optimal tactic. This project also lacked a realistic sonar model and again provided motivation to pursue a real-time sonar simulation. The Manta UUV concept vehicle shows that implementation of an autonomous unmanned vehicle is seriously being considered from a military stand point and the requirements placed on this type of vehicle helped to refine the features of the shallow water sound propagation model. An essential overview of acoustic modeling [Etter, 1996] presents the advantages and disadvantages of all types of sonar models. The model evaluation criteria provided in Etter [1996] show that the choice of RRA is equal to or better than all other choices, based on the criteria of environment complexity, range of propagation and calculation speed. Seminal work by Stewart demonstrates the importance of 3D visualization and the derivation of target solutions from 3D data via the stochastic back-projection algorithm [Stewart, 1988].

B. AUV VIRTUAL WORLD

1. What It Is

The NPS AUV virtual world, [Brutzman, 1994] was created to provide a realistic "out of the water" simulation environment for the *Phoenix* AUV. The *Phoenix* AUV was created at the Naval

Postgraduate School in Monterey, California and a description of the vehicle is given in the dissertation. Figure 2.1 is a drawing of the *Phoenix* AUV.

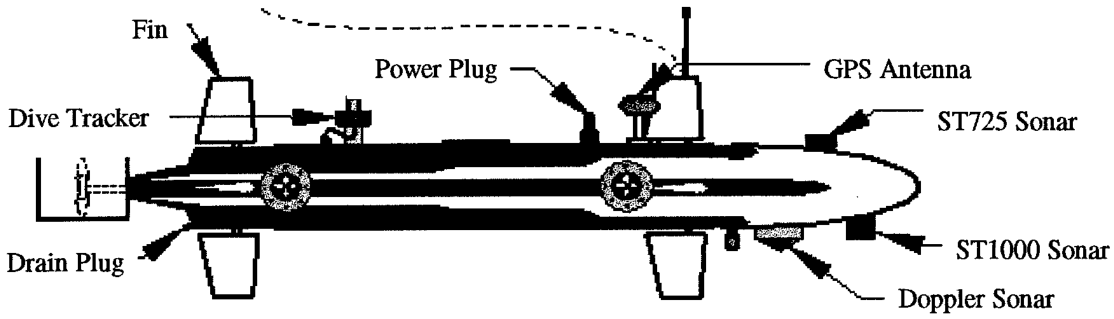


Figure 2. 1 External drawing of the *Phoenix* AUV

The basic idea behind virtual simulation is that every time a new change is made to the hardware or software of the AUV, testing of the change can be made without the time, expense and inherent danger of putting the vehicle in the water. The simulation models all aspects of the effects of the vehicles control surfaces on its position, orientation, speed and rate of change of orientation. In addition it models several aspects of the ocean environments effects on the AUV. For example, the effects of cross vehicle fluid flow are simulated. By attaching the computer in the Phoenix AUV to the computers simulating the ocean environment, simulated operation of the AUV in water is achieved. Previous papers [Brutzman, February 1997] and [Brutzman, 1994], have been devoted to validating the virtual environment against actual operation of the AUV in the water. More information on the Phoenix AUV is found in [Brutzman, 1998].

2. What Are Its Acoustic Limitations

The original modeling of sonar detection in the virtual world simulation was rather simplistic. Sonar geometry of the test tank was trigonometrically derived for any posture inside the tank. This approach is insufficient for most applications, but for the initial abilities of the Phoenix AUV the sonar was acceptable. The more general model derived in [Davis, 1996] can handle arbitrarily large and cluttered environments. In essence the sonar propagation was just a sonar beam comprised of several straight-line ray segments. These ray segments were "shot" into the 3D graphics scene database which comprised the virtual world and any object that they encountered was detected. Figure 2.2 shows the AUV in a virtual test tank, with sonar rays coming from its virtual sonar and lines of fluid flow coming from its cross-body thrusters and propellers. This is an accurate solution within a few hundred meters.

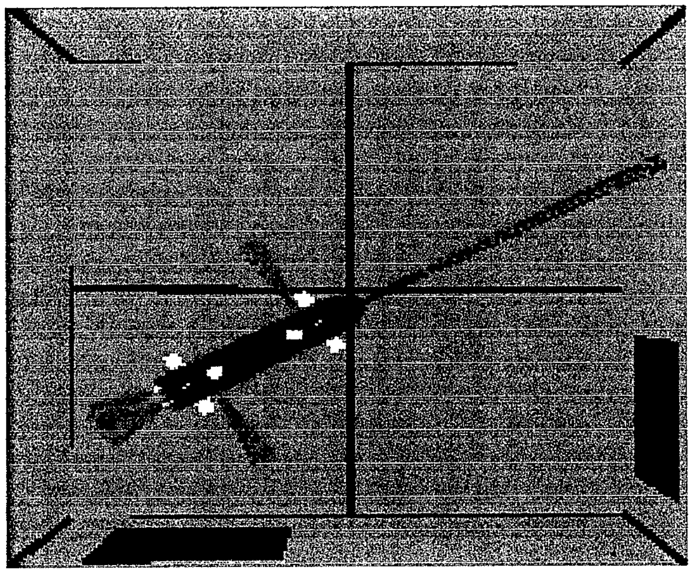


Figure 2. 2 Virtual Phoenix AUV in a virtual test tank. The red lines are sonar rays and the green lines are water flow from the propellers and cross-body thrusters. [Brutzman, 1994]

For several reasons this scheme will not work for long-range propagation. As is described in later chapters, sound speed in the water varies tremendously and because of this variability the rays undergoing long-range sound propagation do not travel in straight lines. This effect typically is noticeable (i.e. > 1 meter in variation) at ranges on the order of hundreds of meters. Also when distances from the source of the sound get large enough, the area front of the beam tube gets large. If this area starts to get larger than the object the sonar beam is trying to detect, missed detections occur in the numerical modeling. The reason for missing returns in a ray-trace model is that the rays may surround but not intersect the target even though the beam front might intersect. This poses a serious problem if simulated detections are required at ranges on the order of a kilometer or so, depending on the size of the target and the beam width of the sonar beam. Improved sonar modeling to cope with such scenarios is the next logical step for the *Phoenix* AUV sonar model, if *Phoenix* is to realize its role as an autonomous mine-hunting platform.

C. AUV TACTICAL MINEFIELD SEARCH

1. What It Is

The AUV tactical minefield search program is another simulation developed by students working with Dr. Brutzman. The program is detailed in [Brutzman, February 1997], where the efficiency of nine different minefield search maneuvers is presented. In the case of clearing a mine field with a presupposed mine distribution, having a good idea of what search tactics produce the best results is essential. If an exhaustive search of the area is desired then all tactic types will produce essentially the same results. However, if there is a time or spatial constraint placed on the situation, not all tactics will produce the same results. The tactical minefield search simulation shows and analytically compares search effectiveness to determine optimal tactics. Though simple in form, it goes a long way toward showing that computer simulations can give meaningful structure to the amorphous question of which tactic is best for a variety of situations. Figure 2.3 shows a tactical search in progress.

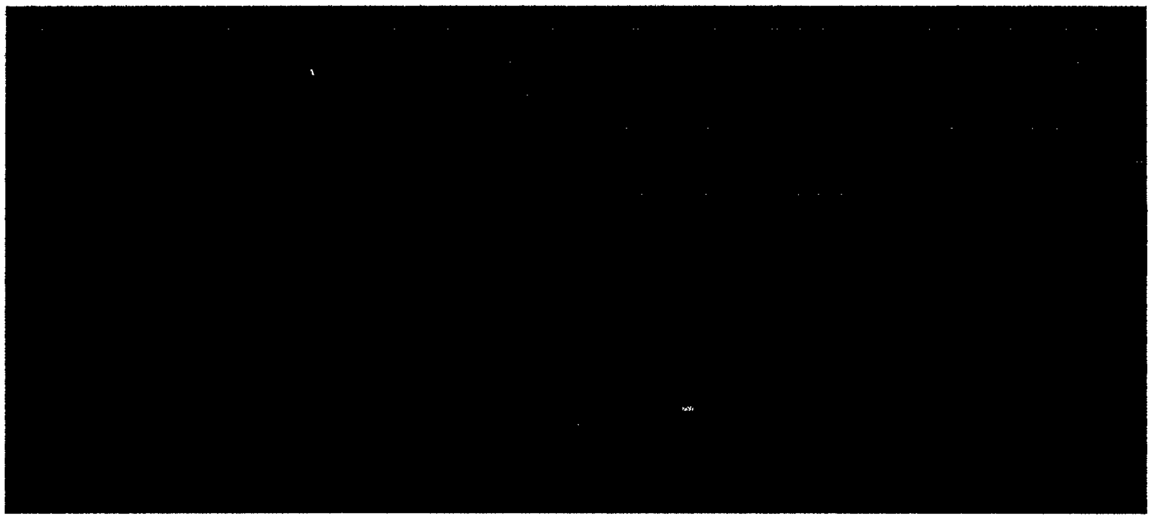


Figure 2. 3 Simulation - tactics evaluation for mine hunting. This 3D rendering allows the tactician to see the simulated search in progress. The search vessel is in the foreground with an oval sonar beam front approaching mine targets

2. What Are Its limitations

Again the limitations of this simulation apply mostly to the area of sonar detection. In this simulation the area to be searched is split into a two-dimensional grid of smaller areas. The areas are chosen so that an AUV traversing the area has a high probability of detecting a mine in the area. No account for sound speed profile (SSP), search depth or bottom composition was made. To get a truly accurate indication of efficiency of detection, these are parameters for which account must be made. For example, if there existed a strong surface layer and the AUV is in that surface layer, almost all of the energy projected by the AUV stays in the layer. Any mines that are below this layer may remain undetected, giving the particular search tactic a lower efficiency than indicated.

D. MANTA CONCEPT

1. What It Is

Manta is a concept unmanned underwater vehicle (UUV) that is being examined by engineers at the Naval Undersea Warfare Center (NUWC) Newport, Rhode Island. The concept of operations has four Mantas attach to a submarine hull for transit and then detach to augment the submarine capabilities when in the area of operations. Figure 2.4 shows a VRML rendering of the Manta UUV. The role of this vehicle is still in development but may include mine detection, surveillance and prosecution of hostile targets. In any of these situations sonar is important for target acquisition, target tracking and navigation. While there exist many sonar models that are used for sonar performance evaluation and off-line interaction between vehicle and environment, there are few (if any) models that provide accurate real-time active acoustic simulation of the ocean environment. Such a capability is very important as money to develop and test new concepts remains scarce. Research time and resources can be utilized more efficiently if the initial mechanical, acoustical and tactical testing of the autonomous vehicle are performed with high fidelity in the laboratory environment.

Figure 2. 4 Manta Concept Vehicle, a) is starboard beam and b) is above the forward port quarter

2. Why Is It Important

Since the break up of the Soviet Union, the U.S. Navy's strategic role has shifted from deep water operations to a shallow water littoral environment. As several conflicts in recent years have shown, minefields can have a devastating effect on the projection of naval power ashore as well as a devastating effect on the ships that encounter them. While the traditional means of mine sweeping work to a limited degree, in most cases a covert mapping and sweeping ability might enable rapid neutralization of a minefield. The Manta vehicle is the first look at an unmanned underwater vehicle (UUV) capable of operating in the littoral environment. While submarines can and have worked in the littoral environment, the potential cost both in loss of human life and monetary loss has made this option very undesirable. Although a fully functioning Manta is not cheap, it is less expensive than a submarine because there are no operators on board. Developing such a capability is invaluable and inevitable. More information on Manta minefield searching is found in [Brutzman, April 1998].

E. SONAR MODELING AND VISUALIZATION

Sonar modeling attempts to recreate the highly variable ocean acoustic environment in an abstract form. Many different sonar models exist primarily because the real world is complex and no one model simplification is adequate to describe the real world in a reduced form. Sonar visualization applies scientific visualization techniques to the results of the sonar simulation, with the goal of enhancing the understanding of the interested researcher. This section identifies related work in sonar modeling and visualization.

1. Sonar Modeling

Deciding on the proper sonar model is very important and highly dependent on how the model is to be used. [Etter, 1996] states that no one sonar model is adequate to describe all situations and that a model (or combination of models) must be chosen to fit the particular set of environmental and operational conditions. Etter thus identifies a model hierarchy that can be used to systematically define the model to be used in a specific situation. Figure 2.5 shows Etter's modeling hierarchy.

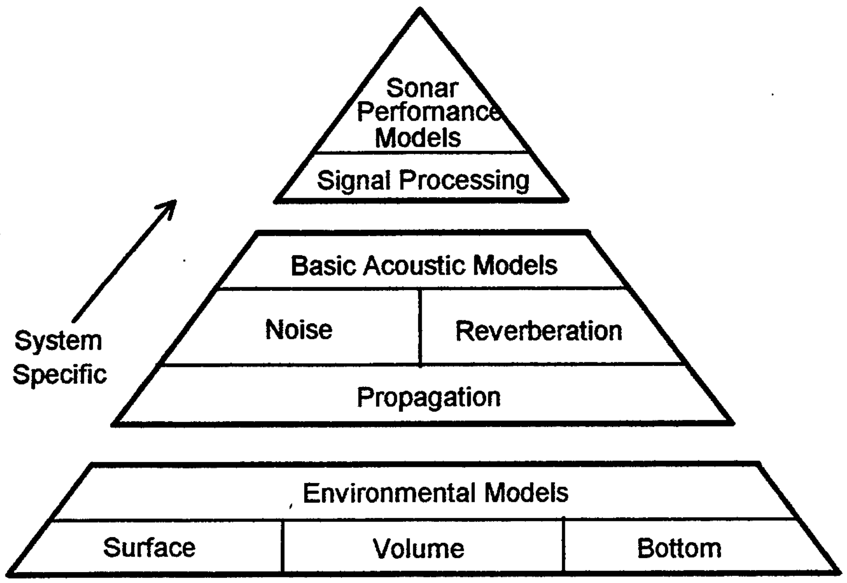


Figure 2. 5 Sonar Modeling Hierarchy [Etter, 1996]

The figure shows how the model proceeds from general to specific as higher and higher layers of abstraction are applied. The lowest level of the model deals with the specific geographic and physical constraints in the problem to be studied. The next layer identifies the particular propagation routine used to transport acoustic energy through the environmental model defined on the lower level. On top of this layer are built models of noise and reverberation that directly integrate into the specific acoustic propagation model. The last layer of the overall sonar solution is the performance model, which receives accurate simulated sonar information from the basic acoustic model and performs signal processing on the data. Etter then examines many example models for each layer in the hierarchy. The number of permutations of this hierarchy is staggeringly large, which accounts for the relatively few attempts made to form a cohesive model. Some comprehensive model operating systems exist that address these issues of complexity and interoperability by switching among models, but they lack scalability and generality. Few (if any) models are computationally tractable in real time. The overall message in [Etter, 1996] is that the ocean is a complex acoustic environment and that models must be carefully considered to ensure realistic results. The implication for real-time 3D sonar modeling and visualization is that model selection is challenging and critically important.

2. Sonar Visualization

[Stewart, 1988] presents a largely statistical approach to modeling, analyzing and visualizing sonar return data. His stochastic back-projection technique adaptively forms a visual model of the world from inherently noisy sonar information. This sonar data is characterized as high bandwidth, high noise and highly redundant. It is redundant in the fact that many propagation paths are available for acoustic energy in the ocean environment. The particular advantage of this model is that immediate results are always available and that the accuracy of the visualization increases as time progresses.

Extensive research in the area of sonar visualization shows that a lack of other resources are available. As evidence of this statement, a five year literature search of the Acoustical Society of America Journal (JASA), IEEE Transactions on Visualization and Computer Graphics, and most published books on scientific visualization revealed only a handful of diagrams rendered in 3D. It is proposed that real-time 3D sonar visualization is an important new field of study lacking computational foundation.

F. SUMMARY

This chapter reviewed the work related to the design of a shallow water sonar propagation and visualization model. Several current virtual world simulations and tactical simulations were presented, as well as two systematic approaches to sonar and visualization modeling. For those interested in tactical search evaluation, Appendix D and Appendix E contain an annotated slide set and HTML pages on tactical visualization using the Manta UUV. Specific faults of various models were addressed and a strategy for addressing these faults was presented. In particular [Etter, 1996] describes a systematic technique for the development of an accurate and realistic sonar model. Finally, real-time 3D sonar visualization is proposed as an important new field of study.

III. PROBLEM STATEMENT

A. INTRODUCTION

Chapter II, Related Work, showed that there are several programs and simulations in existence that have limited or no realistic sonar modeling. When a virtual world is created to mimic the real world, an account of all of the important physical parameters must be made. Performing an underwater acoustic simulation with an unrealistic acoustic model ensures unrealistic results.

B. PROPOSED FRAMEWORK

A sonar model that has its origins in the physics of underwater acoustics is necessary for any high-resolution virtual world simulation of the real world. All of the parameters vital to underwater sound propagation must be included: bottom type, water column sound speed profile (SSP) and surface interactions. Due to the complexity of acoustic energy propagation in real-world underwater environments, the sonar model solution must possess robust visualization capabilities as well as the ability to deliver the vast amount of information in the simulation to an outside client. The sonar model needs to be realistic in energy transfer through the water and must also provide real-time response for user visualization and robot interaction.

C. FACTORS AFFECTING THE SOLUTION

The problem at hand is to generate a real-time, three-dimensional (3D) acoustic energy propagation simulation. This is in response to the need for autonomous underwater vehicles that can perform tactical operations in the underwater environment. This environment is highly complicated, and therefore acoustic simulation in other than a physically rigorous manner is of little use. Additionally, due to the complexity of the real-world environment, it is desirable to build realistic physics-based virtual worlds. Why is a virtual world desirable? The development of a virtual world provides many features.

First, visualization of the acoustic energy propagation in a spatially oriented manner becomes possible. Second, since the scene can be animated, visualization of the time-dependent nature of the acoustic wave front can be seen as it develops in both time and space. Lastly, when the sonar model is allowed to run in an interactive mode as a server to some client program, then more complex real-world simulations can be created. In this last form, the ability to enhance the *Phoenix* AUV virtual world and the AUV search tactics evaluation simulation programs results in data that is more consistent with the real world and its processes. However, physics-based real-time solutions to any wave-energy propagation model requires sufficient computing power. Surprisingly, adequate computational power can be provided by currently available personal computers to give the amount of resolution required for sufficiently accurate results. To increase the computational power further, either throughput in an individual processor needs to be increased or the number of processors working on the job needs to be increased. Both of these options are viable and need to be explored. Increasing throughput is a job being handled quite well by microprocessor manufacturers, so no further discussion is warranted here. Increasing the number of processors falls into one of two categories: building a massively parallel computer or interconnecting many individual computers via a high speed local-area network (LAN). For most researchers , the second option may already exist or can be created relatively easily. With the real-time 3D sonar visualization problem well defined, how is the solution to be obtained?

D. POSSIBLE SOLUTIONS

This section addresses many possible ways of solving the problem. Several different sonar propagation models exist, such as ray tracing, normal mode, parabolic equation and others. In addition many different ways of simulation exist; direct analog simulation and digital computer simulation are but two of the many ways to simulate a physical process. Once the specifics of the implementation are worked out, what to produce as the results of the simulation must be decided. In other words, what useful function does the simulation provide that was not available before implementation. These aspects and others are explored in this section.

1. Available Sonar Models

Deciding on the proper sonar model is very important and very dependent on how the model is to be used. [Etter, 1996] Chapter 4 is devoted to explaining the advantages and disadvantages of the common sonar-propagation models. The chapter presents ray models, normal mode models, multipath expansion models and parabolic equation models. A short description of each of the model types is given in the following paragraphs.

Ray theory models start with the Helmholtz equation and make the assumption that a propagating wave can be thought of as a surface of constant phase emerging from a source. This surface, called the eikonal, can be represented by rays that travel perpendicular to the surface of the wave front. The direction of travel of these rays depends completely on the initial direction of travel and the sound speed profile (SSP) in the water column. Ray theory requires that the frequency of the acoustic signal be large compared to the rate of change of sound speed in the water column and that the wavelength be small compared to the characteristic length of structures on the ocean surface and bottom. [Etter, 1996] develops Equation (3.1),

$$f > 10c / H, \tag{3.1}$$

where f is the minimum accurate frequency, c is the speed of sound in the ocean and H is the depth of the ocean. This gives a good rule of thumb for the low limit of the accuracy of the ray-tracing algorithm. Another important characteristic of a sonar modeling theory is that of range dependence. Ray theory models are generally range-dependent models. This means that the acoustic structure of the ocean can vary with distance from the source in a 2D or 3D manner.

Normal mode methods are derived from an integral representation of the linear acoustic wave equation. To arrive at practical solutions, cylindrical symmetry is assumed which limits range dependence to two dimensions. This equation is then solved using separation of variables. This technique develops a solution that depicts acoustic waves as traveling waves in the horizontal direction and standing waves in the vertical. Many times in order to develop a closed form solution to the normal mode problem, no dependence is placed on range, this is termed range independence. For many shallow water regions, especially near shelf break zones, range independence is an invalid assumption.

Multipath expansion methods expand the acoustic field integral representation into an infinite orthogonal set of integrals. This sounds like a bad idea; to go from one integral to an infinite set of integrals. However, each integral is simpler than the original and with the proper approximations only a certain number are used in the final solution. By integrating only a limited number of integrals, an angle-limited source model is produced. This solution technique has many similarities to ray theory, but is better suited to deep water. One advantage over ray theory is that caustic and shadow zones are properly computed, instead of incorrectly producing infinite pressure fields as ray theory can.

Parabolic equation methods find their roots in the 1940's when it was first applied to long-range radio propagation. The first application to acoustic waves is in [Hardin and Tappert, 1973]. The basic theory is to make an approximation that there is little back scattering of acoustic energy in the water column. This approximation transforms the elliptic wave equation into the parabolic wave equation. From the form of the parabolic wave equation, many other assumptions about symmetry and sound speed profile can be made to simplify the mathematics. Several different numerical integration techniques are then used to evaluate the parabolic equation form of the wave equation. Two of the most popular are the split-step Fourier algorithm and the finite-element method. The split-step Fourier method has the distinct advantage of speed of computation, while the finite-element technique is typically more accurate at higher propagation angles.

[Etter, 1996] provides a table that summarizes the applicability and practicability of each of the models in relation to water depth, source frequency and environmental dependence on range. A summary of the table from the book is presented in Figure 3.1. As was described earlier, the focus of this model and simulation is for the shallow water environment and so the deep-water portion of the figure was left off. Therefore, the two independent variables in this table are frequency and range dependence. A range-dependent environment has fundamental characteristics that change as the distance from the source varies, whereas a range-independent environment does not. Given that the goal is to develop a shallow-water sonar model that is robust in a range-dependent environment and which works well with a frequency in the 1000 Hz range and higher, then Figure 3.1 indicates that a ray-theory model is best suited. If however the frequencies of interest are below 500 Hz and range dependence is required, the parabolic equation model is best.

Model Type	Applications			
	Shallow Water			
	Low Freq.		High Freq.	
	RI	RD	RI	RD
Ray Theory	○	○	⦶	●
Normal Mode	●	⦶	●	⦶
Multipath	○	○	⦶	○
Fast Field	●	○	●	○
Parabolic Eq.	⦶	●	○	○

Low Freq. - < 500 Hz
High Freq - > 500 Hz

RI - Range Independent
RD- Range Dependent

● Applicable and Practical
⦶ Limitations in Accuracy or Speed
○ Neither Applicable or Practical

Figure 3. 1 Applicability and practicability of model types, adapted from [Etter, 1996]

Of particular note is a paper published in the Journal of the Acoustical Society of America, [Smith et al., 1991], which states that due to sensitive dependence on initial conditions, ray-theory solutions tend to exhibit chaotic behavior. The paper goes further to show that the chaotic behavior begins to become apparent somewhere around 100 km from the source in deep ocean environments with no boundary interactions. Chaotic behavior for most shallow-water environments will occur at shorter ranges due to extensive ocean bottom interaction. For this reason, care must be taken in evaluating the results of complex tactical shallow-water environments.

2. Type Of Simulation

Choosing the type of simulation is important to any modeling problem. To simulate a shallow water environment one might use a physical simulation or an abstract simulation. An example of a physical simulation of a shallow-water environment is to construct a tank of water with a scaled version of the environment to be studied. An example of an abstract simulation is a computer simulation of a sonar computational model, where the actual simulation is occurring as a bunch of electrons flowing through computer chips. Each method of simulation has its own advantages and disadvantages. The physical simulation has the advantage of being closely linked to reality, providing that proper scaling and terminating techniques are used. However physical simulations require massive amounts of material and time reconfiguring for a new environment. Abstract simulations are just the opposite in that they are further removed from reality even when the initial and boundary conditions are specified properly, but they are easily reconfigured for different environments by changing the initial and boundary conditions.

3. Networking Considerations

The choice here is to opt for or against network interaction. Networked simulations have a distinct advantage over single-computer simulations. Networked simulations can be made more widely available and can as a result have more computers working on a given problem. In large-scale simulations this increased computing power and ease of access allow the formation of multiple participant simulations. The advantage here is that the simulation is closer to reality and any simulation that is closer to the real world progression of a system is a better simulation. However, networked models must also pay performance penalties associated with bandwidth capacity, message delivery latency and synchronization overhead costs. The decision to network or not is primarily a function of whether or not computing power needs to be increased or if interaction with distant parties is desired.

4. Visualization

The options for visualization range far and wide. Output options include text-based tables, two-dimensional graphs and three-dimensional virtual worlds. The first two options are modest and well within the capabilities of most computer software that is available. The third option requires specialized rendering software. This software comes in many forms, such as stand-alone programs that read in textual tables and generate information, interactive programs that are directly called from the simulation, or a combination of the two as can be found in common virtual reality (VR) browsers. Each form has its distinct advantages and disadvantages. One recent advantage, that stands out, is provided by VR browsers: widespread availability of a common standard for 3D graphics representation, using the Virtual Reality Modeling Language (VRML 97). This compatibility supports the needs of large-scale simulations presented in the previous section.

E. SUMMARY

Implementation considerations for a shallow water sonar propagation and visualization simulation are many. Proper choices for propagation model, simulation type, networking and visualization must be made. The remainder of this thesis discusses and develops these important choices in detail.

IV. RECURSIVE RAY ACOUSTICS (RRA) DERIVATION

A. INTRODUCTION

This chapter is included as a benefit to the reader interested in the mathematical derivation of the Recursive Ray Acoustics (RRA) equations. The nomenclature used is similar to [Ziomek, 1996] and each equation in the stated derivation comes from Chapter 5 of that text book. The flow of the mathematical development presented here is straight to the point. Some of the more difficult (and long) tangential derivations are left for the reader to explore in Ziomek's text. Although this derivation is streamlined, it is a complete and rigorous treatment of the transformation of the linear acoustic wave equation into a linear ray acoustic equation.

B. THE DIFFERENTIAL EQUATION

[Ziomek, 1996] Chapter 5 is devoted to solving the linear acoustic wave equation using ray theory simplifications. This section derives the pertinent equations of ray acoustics. The first assumption is that the acoustic source is a time-harmonic oscillator. When this is the case, the starting point of the derivation is the Helmholtz equation:

$$\nabla^2 \varphi(\mathbf{r}) + k_0^2 n^2(\mathbf{r}) \varphi(\mathbf{r}) = 0 . \tag{4.1}$$

The acoustic pressure is then assumed to have the simple geometrical form which locally appears as a plane wave front of amplitude a(r), as in:

$$\varphi(\mathbf{r}) = a(\mathbf{r}) e^{-jk_0 W(\mathbf{r})} . \tag{4.2}$$

The function W(r) defines a surface of constant phase and typically is called the eikonal, which means *image* in Greek. Upon substituting Equation (4.2) into Equation (4.1) and simplifying, the following equation is obtained:

$$\left[\nabla^2 + k_0^2\left[n^2(\mathbf{r}) - \left|\nabla W(\mathbf{r})\right|^2\right]\right] a(\mathbf{r}) - jk_0\left[a(\mathbf{r})\nabla^2 W(\mathbf{r}) + 2\nabla a(\mathbf{r}) \bullet \nabla W(\mathbf{r})\right] = 0 . \tag{4.3}$$

Now since the left hand side of the equation is identically zero, both the real and imaginary parts of Equation (4.3) must be zero. This provides two equations:

$$\nabla^2 a(\mathbf{r}) + k_0^2\left[n^2(\mathbf{r}) - |\nabla W(\mathbf{r})|^2\right]a(\mathbf{r}) = 0 \tag{4.4}$$

and

$$a(\mathbf{r})\nabla^2 W(\mathbf{r}) + 2\nabla a(\mathbf{r}) \bullet \nabla W(\mathbf{r}) = 0\,. \tag{4.5}$$

Equation (4.4) is the equation that tells how the wave front of the acoustic signal travels through the water and Equation (4.5) describes the flow of energy of the acoustic signal. Equation (4.4) is now the focus of the following development.

Rearranging this equation yields an alternate form:

$$1 + \frac{1}{k_0^2 n^2(\mathbf{r})}\left[\frac{\nabla^2 a(\mathbf{r})}{a(\mathbf{r})} - k_0^2|\nabla W(\mathbf{r})|^2\right] = 0\,. \tag{4.6}$$

From this form if it is assumed that the amplitude function of Equation (4.2) varies much more slowly than the phase function, then

$$\left|\frac{\nabla^2 a(\mathbf{r})}{a(\mathbf{r})}\right| << k_0^2|\nabla W(\mathbf{r})|^2 \tag{4.7}$$

is true and Equation (4.6) reduces to

$$|\nabla W(\mathbf{r})|^2 = n^2(\mathbf{r})\ , \tag{4.8}$$

the eikonal equation. As a direct result of the assumption in Equation (4.7) Ziomek shows in his book that

$$\left|\frac{\nabla^2 a(\mathbf{r})}{a(\mathbf{r})}\right| << k^2(\mathbf{r}) = \left[\frac{2\pi f}{c(\mathbf{r})}\right]^2 \tag{4.9}$$

and also indicates that the approximation amounts to a high-frequency approximation.

From vector mathematics it is known that the gradient of the eikonal is

$$\nabla W(\mathbf{r}) = \frac{\partial}{\partial x}W(\mathbf{r})\hat{x} + \frac{\partial}{\partial y}W(\mathbf{r})\hat{y} + \frac{\partial}{\partial z}W(\mathbf{r})\hat{z} . \qquad (4.10)$$

Since it is a vector it can be expressed as a magnitude and a direction,

$$\nabla W(\mathbf{r}) = |\nabla W(\mathbf{r})|\hat{s}(\mathbf{r}) , \qquad (4.11)$$

where $\hat{s}(r)$ defines the local direction of the ray path. This can be rewritten, using Equation (4.8), as

$$\nabla W(\mathbf{r}) = n(\mathbf{r})\hat{s}(\mathbf{r}) , \qquad (4.12)$$

where n(r) is the index of refraction. To further clarify Equation (4.12) a ray is defined to be a path perpendicular to the phase fronts at a given position, as in Figure 4.1.

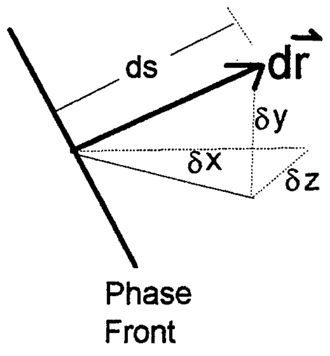


Figure 4. 1 Ray Definition

From this graphic it is easy to see that the following equation is true:

$$\hat{s}(\mathbf{r}) = \frac{d\mathbf{r}}{ds} = \frac{\partial x}{\partial s}\hat{x} + \frac{\partial y}{\partial s}\hat{y} + \frac{\partial z}{\partial s}\hat{z} , \qquad (4.13)$$

since |dr| = ds. Equation 4.13 is the unit vector in the direction of Equation (4.10). Computing the directional derivative of *W*(r) and using the chain rule yields the change in phase along the ray path,

$$\frac{\partial}{\partial s}W(\mathbf{r}) = \frac{\partial}{\partial x}W(\mathbf{r})\frac{dx}{\partial s} + \frac{\partial}{\partial y}W(\mathbf{r})\frac{dy}{ds} + \frac{\partial}{\partial z}W(\mathbf{r})\frac{dz}{ds} . \qquad (4.14)$$

From inspection of Equations (4.10) and'(4.13), it can be seen that

$$\frac{\partial}{\partial s} W(\mathbf{r}) = \nabla W(\mathbf{r}) \bullet \hat{s}(\mathbf{r}) \tag{4.15}$$

is equivalent to (4.14). Substitution of (4.12) into this result produces

$$\frac{\partial}{\partial s} W(\mathbf{r}) = n(\mathbf{r}) . \tag{4.16}$$

This equation can be integrated quite easily to yield the solution of the eikonal equation. In essence the equation says that a small change in the wave front can be generated from the product of the index of refraction and a small change in the path length of a ray traveling normal to the front.

C. THE DIFFERENCE EQUATION

To develop this equation into a form better suited for calculation on a digital computer and to solve for a more meaningful physical quantity, the gradient operator is applied to Equation (4.16) to yield,

$$\nabla \frac{\partial}{\partial s} W(\mathbf{r}) = \nabla n(\mathbf{r}) . \tag{4.17}$$

Since the two operators preceding W(r) commute, Equation (4.17) can be rewritten as,

$$\frac{\partial}{\partial s} \nabla W(\mathbf{r}) = \nabla n(\mathbf{r}) . \tag{4.18}$$

Equation (4.12) is then substituted into Equation (4.18) yielding,

$$\frac{\partial}{\partial s}\left(n(\mathbf{r}) \hat{s}(\mathbf{r}) \right) = \nabla n(\mathbf{r}) , \tag{4.19}$$

which can be rewritten as a difference equation,

$$\Delta\left(n(\mathbf{r}) \hat{s}(\mathbf{r}) \right) = \nabla n(\mathbf{r}) \Delta s , \tag{4.20}$$

instead of a differential equation. If the path length step Δs is kept small, then the path can be considered a straight line, and Equation (4.20) can be rewritten again as,

$$n(\mathbf{r}_f) \hat{s}(\mathbf{r}_f) - n(\mathbf{r}_o) \hat{s}(\mathbf{r}_o) = \nabla n(\mathbf{r})(\mathbf{r}_f - \mathbf{r}_o) . \tag{4.21}$$

Substitution of,

$$n(\mathbf{r}) = \frac{c_0}{c(\mathbf{r})}, \tag{4.22}$$

and solving for the normal vector to the wave front at the final position yields,

$$\hat{s}(\mathbf{r}_f) = \hat{s}(\mathbf{r})\frac{c(\mathbf{r}_f)}{c(\mathbf{r}_o)} - (\mathbf{r}_f - \mathbf{r}_o)c(\mathbf{r}_f)\frac{1}{c^2(\mathbf{r}^*)}\nabla c(\mathbf{r}^*), \tag{4.23}$$

where c(r*) is the average of the sound speed at the final and initial points.

This is the final form of the solution and is referred to as the recursive solution, since to get the final solution the previous step must be solved recursively all the way back to the starting position. The recursive technique is relatively simple in that, if one starts with a normal to the wave front and takes small-path-length steps, calculating a new normal at each step, the solution can be achieved. It is this difference equation that is implemented in the RRA code of this thesis.

D. SOLUTION TO THE TRANSPORT EQUATION

Equation (4.5) is now addressed to discover how the acoustic energy is transferred (i.e. transported) with the ray. The first step is to multiply the equation by a(**r**) which yields,

$$a^2(\mathbf{r})\nabla^2 W(\mathbf{r}) + 2a(\mathbf{r})\nabla a(\mathbf{r}) \bullet \nabla W(\mathbf{r}) = \nabla \bullet \left[a^2(\mathbf{r})\nabla W(\mathbf{r})\right] = 0. \tag{4.24}$$

Calculating the time averaged intensity vector of,

$$\varphi(t,\mathbf{r}) = \varphi(\mathbf{r})e^{j2\pi ft}, \tag{4.25}$$

where φ(**r**) is Equation (4.2), the following average intensity is arrived at;

$$\mathbf{I}_{avg}(\mathbf{r}) = \frac{1}{2}k_0^2\rho_0(\mathbf{r})c_0 a^2(\mathbf{r})\nabla W(\mathbf{r}). \tag{4.26}$$

[Ziomek, 1996], Chapter 5, page 336 shows the development of the time-averaged intensity vector in detail. When Equation (4.26) is substituted into Equation (4.24), the result is,

$$\nabla \bullet \left[\mathbf{I}_{avg}(\mathbf{r}) \big/ \rho_0(\mathbf{r}) \right] = 0. \tag{4.27}$$

Now suppose that instead of a considering single ray proceeding along a path, a bundle of rays such as Figure 4.2 exists. When Equation (4.27) is integrated over the volume defined in the figure and the divergence theorem is applied,

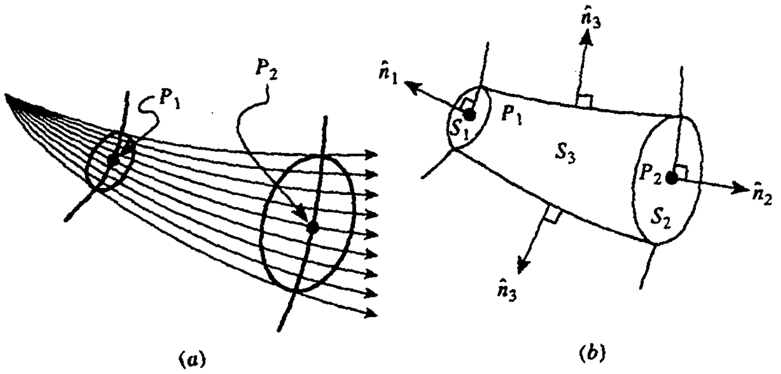


Figure 4. 2. (a)Rays forming the bundle, (b) Surfaces enclosing the bundle [Ziomek, 1996]

$$\int_V \nabla \bullet \left[\mathbf{I}_{avg}(\mathbf{r}) \big/ \rho_0(\mathbf{r}) \right] dV = \oint_S \left[\mathbf{I}_{avg}(\mathbf{r}) \big/ \rho_0(\mathbf{r}) \right] \bullet dS = 0, \tag{4.28}$$

is the result. The integral over the surface S_3, in Figure 4.2, is assumed to be zero since the rays themselves form the surface and by definition cannot leave the surface. Therefore, it is easily seen that the integral over the three surfaces result in showing that all of the energy stays in the bundle and as such the energy in the bundle crossing any plane is a constant. In equation form this concept is written as,

$$I_{avg}(\mathbf{r}_1) S_1 \big/ \rho_0(\mathbf{r}_1) = I_{avg}(\mathbf{r}_2) S_2 \big/ \rho_0(\mathbf{r}_2), \tag{4.29}$$

where the S's are the surface areas perpendicular to the ray bundle and the ρ's are the density at the two different positions. This equation allows the calculation of the intensity of the bundle at any position along its path, at any time. It is important to note that no summations of incremental intensity variations along the path are necessary, only the knowledge of the values of three of the four quantities allow calculation of the other at any two positions in the water column.

E. SUMMARY

As can be seen from the preceding derivation, the Recursive Ray Acoustic (RRA) approximation is straight forward. The particularly fascinating outcome of this entire chapter is that when any acoustic wave meets the assumptions made in the mathematical development, the wave can be completely described by just two simple Equations, (4.23) and (4.29). These correspond directly to Equation (3) in [Ziomek, 1993] and Equation (5.2-170) in [Ziomek,1996]. Thus the RRA formulations appear to be both very general and computationally efficient. These attributes make the RRA model an excellent (and perhaps optimal) choice as a computational engine for real-time 3D sonar beam generation

V. MODELING OF RECURSIVE RAY ACOUSTICS (RRA)

A. INTRODUCTION

This chapter extends the recursive ray acoustics (RRA) algorithm into a complete model suitable for real-time simulation and 3D sonar visualization. The full set of equations that define the RRA model are stated and elaborated upon. Each sub-model of the RRA (sonar server) model is explained from a modeling perspective. The composition of the state vectors and operations for each sub-model are defined and the mathematical developments for curvature determination, reflection detection and target detection are given. This chapter lays out the formal structure of the RRA model, which is further described in Chapter VII, Model Implementation and Integration.

B. MODEL ENGINEERING

At the outset, the difference between the term 'model' and the term 'simulation' needs to be defined. A model is the abstract representation of the dynamic system in question. Simulation simply refers to a the performance of a model over time. Implementation of the model can predict the dynamic characteristics of the system, from which we gain insight about the original model. It is quite clear from the definitions that the formulation of a model must precede the development of a simulation. Therefore, before one can jump into the mechanics of programming the entire model into a simulation, model engineering must be carefully considered before a single line of code is written. Fishwick states,

> How do we engineer models? While there are many modeling techniques for simulation, we are often in a quandary as to which model technique to use and under what conditions we should use it. First start with a concept model of whatever dynamic system is being investigated. Break the system into a hierarchy of abstractions and then choose models to represent those abstraction levels. The final solution to a model will be a multi-model of the system since no one model type will be sufficient to describe a system except in only the most elementary circumstances. [Fishwick, 1995] (page 5)

Figure 5.1 shows the top-level architecture of the shallow-water sonar model. The sonar server model is considered in this chapter, the sonar visualization model is considered in Chapter VI and the virtual world display model is considered in Chapter VII.

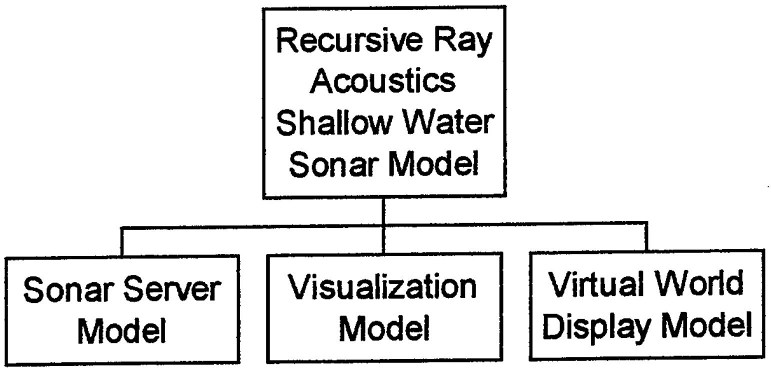

Recursive Ray Acoustics Shallow Water Sonar Model
Sonar Server Model
Visualization Model
Virtual World Display Model

Figure 5. 1 Shallow-water RRA sonar model architecture

The sonar server model is further decomposed into the hierarchy of models depicted in Figure 5.2. As an overview, the base models in this hierarchy are the target model, the ray model, the ocean bottom model, the ocean surface model and the sound speed profile model. From the ray model the beam model is derived and from the beam model the lobe model is derived. Each sub-model and its implementation in the simulation is discussed in detail in the following sections. Figure 5.3 shows the relationship among rays, beams and lobes.

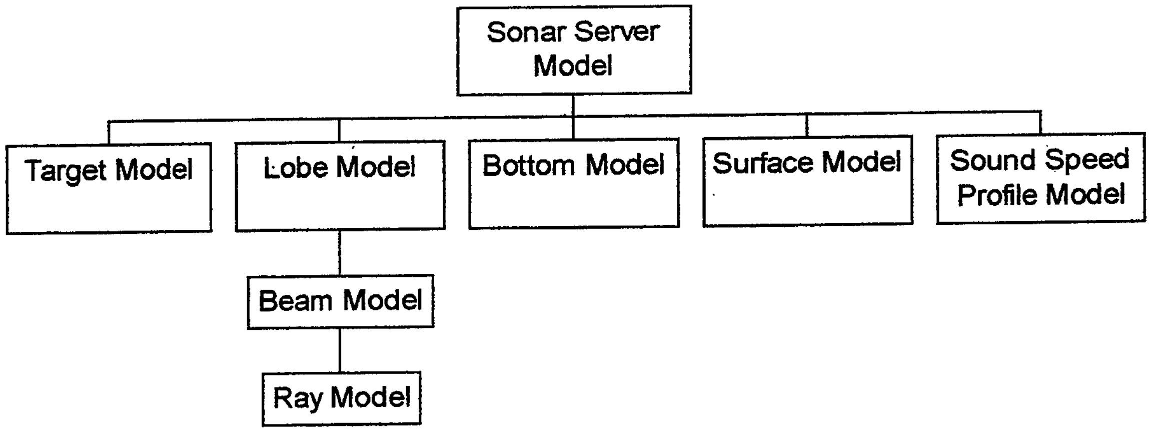


Figure 5. 2 Sonar Server Class Hierarchy

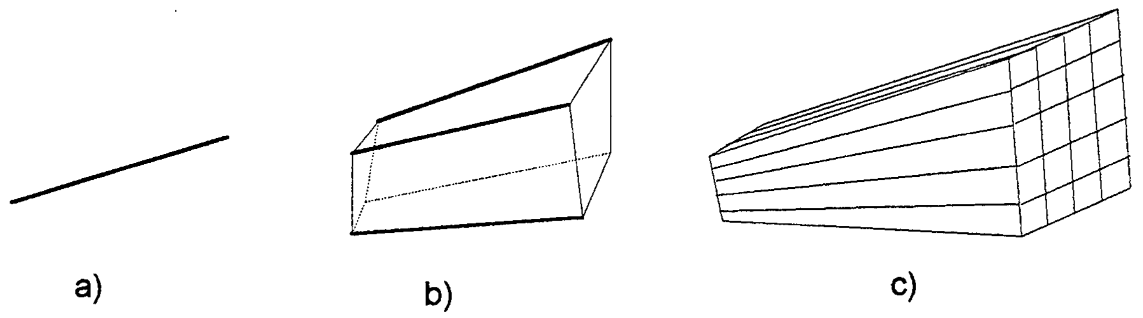


Figure 5. 3 a) A single ray, b) Four rays forming a beam and c) Twenty beams forming a lobe

C. THE COMPLETE RRA EQUATIONS

Before further elaboration on sub-model design and implementation, it is necessary to revisit the ray theory solution to the wave equation. Equation (4.23) is the central equation in the algorithm intended for eventual implementation. [Ziomek,1996] shows the final form of the equations in a finite difference format vice the differential format. The equations are restated with minor variations from the originals for ease of understanding.

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \Delta\mathbf{r}_{i-1}, \tag{5.1}$$

$$\Delta\mathbf{r}_{i-1} = \Delta r_{i-1}\hat{s}_{i-1}, \tag{5.2}$$

$$\hat{s}_i = \hat{s}_{i-1}\frac{c(\mathbf{r}_i)}{c(\mathbf{r}_{i-1})} - \Delta r_i c(\mathbf{r}_i)\frac{1}{c^2(\mathbf{r}^*_{i-1})}\nabla c(\mathbf{r}^*_{i-1}), \tag{5.3}$$

$$\mathbf{r}^*_{i-1} = \frac{\mathbf{r}_{i-1} + \mathbf{r}_i}{2}, \tag{5.4}$$

where the unit normal vector S is the same as Equation (4.13) and i = 1,2,3,... Since most real-world rays are represented as an elevation and an azimuth angle, a coordinate system conversion must be made. These conversions are summarized in the following equations,

$$\frac{\partial x}{\partial s} = \sin(\beta)\cos(\phi), \tag{5.5}$$

$$\frac{\partial y}{\partial s} = \cos(\beta) , \quad (5.6)$$

and

$$\frac{\partial z}{\partial s} = \sin(\beta)\sin(\phi) , \quad (5.7)$$

where β is the elevation angle and ϕ is the azimuth angle relative to the initial ray position, as shown in Figure 5.4.

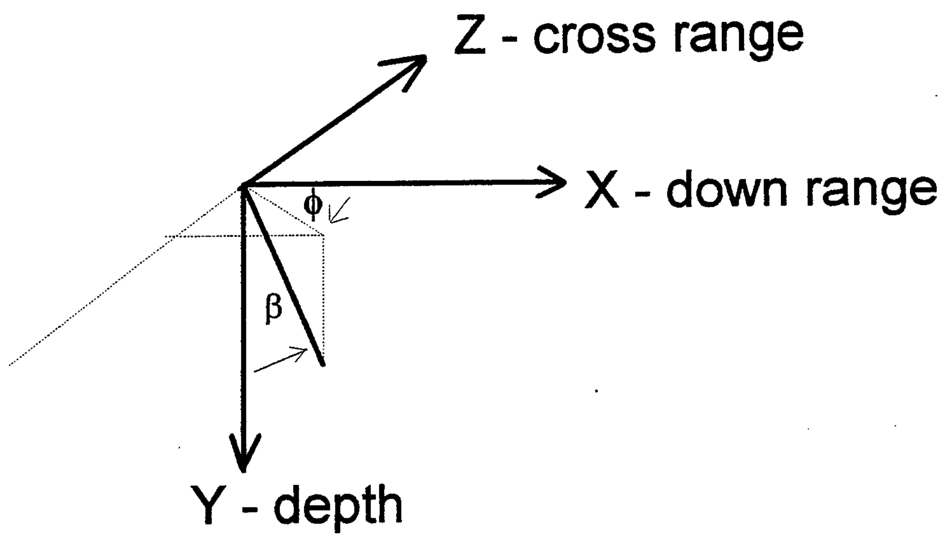


Figure 5. 4 Ziomek coordinate system definition of β and ϕ. β is measured from the y-axis and ϕ is in the x-z plane

Equations (5.5)-(5.6) assume the model uses the coordinate system of Figure 5.4, referred to as the Ziomek coordinate system to distinguish it from systems used later. Figure 5.5 shows the Ziomek coordinate system and the traditional naval coordinate system. Using the preceding equations and conventions specified it is now possible to unambiguously specify each sub-model of the RRA model.

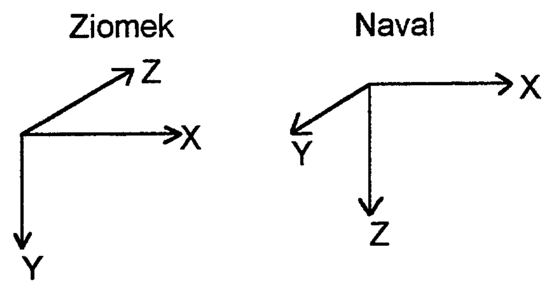


Figure 5. 5 Ziomek and Naval coordinate systems

D. RAY MODEL

With the construction of any model, one must first work out what data is important and limit the model to produce just this set of information. While the ability to calculate every foreseeable parameter hypothetically might be desirable, the reality of having to implement a practical model to such a level of detail proves difficult. In general, with an excessive level of detail, the time taken to run the simulation and collect the data increases dramatically. Therefore, what parameters must be modeled in the case of a sound wave traversing an ocean environment? To adequately describe the state of the model, and how it transitions from one state to the next, the designer must clearly state what the desired output of the model is. As seen in the previous chapter, a wave front traveling through the ocean can be thought of as a collection of sound rays traveling through the water. It is just these rays that are considered in this model.

From the ray model the information that is available at each time step includes position, duration, direction of propagation, phase, time, total distance traveled, and total attenuation of the ray. In an ideal case, all information at every step of the calculation might be recorded, but for a single ray this can add up to megabytes of data. Obviously, this is not practical due to the excessive disk access and computational overhead involved in keeping that amount of data. Instead the model only keeps the amount of data necessary to visualize accurately what is developing in the model. Since a ray traces out a curved line in the ocean with occasional abrupt changes in direction at reflections, the model needs to retain enough points to faithfully represent the ray curvature, especially for neighboring points in the vicinity of an abrupt change in direction of the ray. Therefore, in addition to the required output information, local curvature and reflection information must be kept. Thus the state vector of the ray model must include the position of the leading edge of the ray segment, position of the trailing edge of the ray segment, direction that the ray segment is traveling, simulation time, phase of the ray at the wave front, total distance traveled, and total attenuation. With the state vector now defined, the operations on this vector must be defined to enable the state vector to change in time. The desired operations needed are to initialize, propagate, reflect, calculate attenuation, calculate total curvature, save the current state and print any required state of the ray. Some of these operations are self-explanatory while others (propagation, reflection, attenuation calculation and curvature calculations) require more discussion. The reflection operation is discussed as part of the surface and bottom models later in this chapter. State vector parameters and operations for the ray model are enumerated in Figure 5.6

Ray Model	
State Vector	Operations
Position	Set Any Parameter
Trailing Edge Position	Get Any Parameter
Propagation Direction	Propagate
Simulation Time	Reflect
Phase	Calculate Attenuation
Attenuation	Calculate Curvature
Curvature	Save State
	Print State

Figure 5. 6 Ray model state vector and required operations summary

The propagation of a ray first requires implementation of Equations (5.1)-(5.4). Equations (5.5)-(5.7) are used to initialize the unit normal vector to the wave front, and find no further use during the propagation of the ray. Starting with a given state vector, propagation involves applying Equations (5.1) - (5.4) (in reverse order) for every given change in path length. In order to keep the simulation time between rays the same, the incremental path length, Δr, in any given time step, Δt, is calculated as follows,

$$\Delta r = \Delta t \cdot c\left(\mathbf{r}^*\right). \tag{5.8}$$

This substitution allows for stepping in regular discrete time intervals to calculate changes in the state vector.

Attenuation of sound in sea water has been extensively studied and is due to many physical phenomena occurring simultaneously in various parts of the ocean. Relaxation processes and scattering of energy from resonant micro-bubbles [Clay and Medwin, 1977] are but a few of the loss mechanisms in the ocean volume. This model is concerned solely with the relaxation process, which occurs when a passing sound wave interacts with a media in a non-adiabatic fashion. In other words a net transfer of energy from the sound wave to molecules in the water has occurred, which results in an attenuation of energy in the wave. In Chapter 7 of [Kinsler et al., 1982] sound attenuation due to relaxation processes in sea water are shown empirically to follow Figure 5.7. Since the value of the attenuation changes with temperature and pressure, those effects must be taken into account during every time step of the propagation. Therefore a simple summation of the product of the change in path length and the local value of the attenuation coefficient must be made. The formula for calculating the attenuation coefficient is complex and requires excessive processing time so an approximation is made. At the frequency of most active sonar systems (less than 100KHz), the attenuation coefficient is fairly negligible for all temperature and pressure conditions, thus a representative value is chosen from the range of possible values. This value is the attenuation coefficient as read from Figure 5.7. A more detailed discussion of attenuation and the relaxation attenuation coefficient is found in [Kinsler et al., 1982].

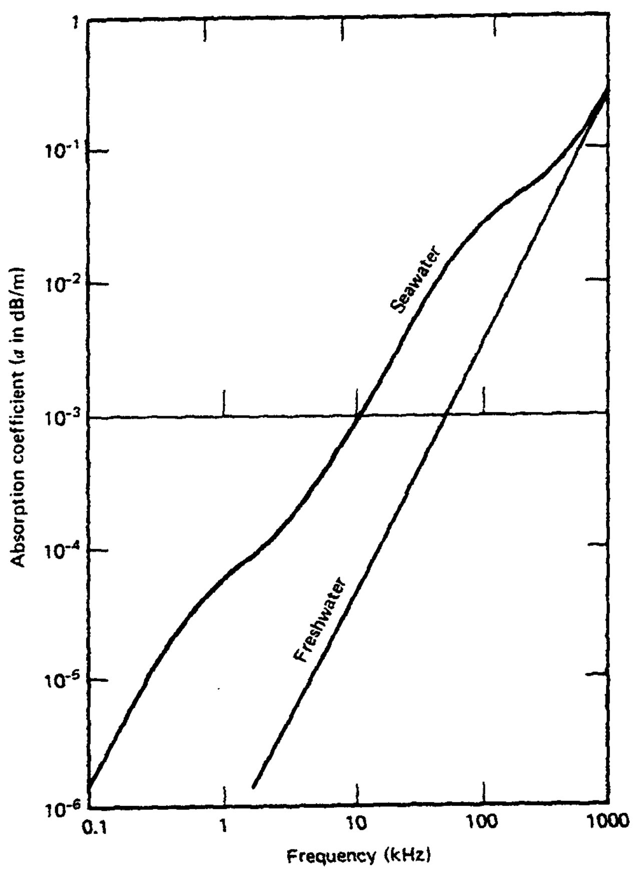


Figure 5. 7 Attenuation coefficient in sea water [Kinsler et al., 1982]

Curvature calculation for the ray is based on storage efficiency and the aesthetic appearance of visualizing the ray. As stated before, it is important to minimize the amount of information one carries along in a simulation. In the case of visualizing sound waves in the ocean, only the number of points necessary to make the ray appear smooth when seen from a distance must be saved for later use. The trick then is to calculate the cumulative radius of curvature of the ray and when it reaches a predefined small value, a detailed vector of state data is saved and the cumulative curvature is reset to zero. The development of this simple curvature relationship is presented in the rest of the section. In Figure 5.8, the left-hand graphic shows two ray segments laid head to tail, as they are when the ray propagates. The right-hand graphic shows the angular difference between the two ray segments.

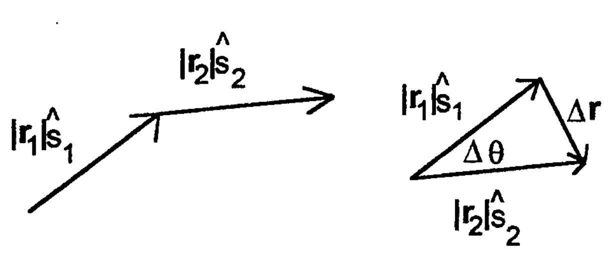


Figure 5. 8 Successive segments in the ray path

These vectors represent two successive steps in the tracing of a ray. If the vectors are rearranged as in the right-hand graphic in the figure, it is easily seen that

$$\Delta\theta = \frac{|\Delta\mathbf{r}|}{|\mathbf{r}_1|} = \frac{|\Delta\mathbf{r}|}{|\mathbf{r}_2|} = \frac{|\mathbf{r}_2 - \mathbf{r}_1|}{|\mathbf{r}|} = \frac{|\mathbf{r}|\left|\hat{\mathbf{s}}_2 - \hat{\mathbf{s}}_1\right|}{|\mathbf{r}|} = \left|\hat{\mathbf{s}}_2 - \hat{\mathbf{s}}_1\right|, \tag{5.9a}$$

where $|\mathbf{r}| = |\mathbf{r}1| = |\mathbf{r}2|$ when the constant time step is small and there are no abrupt changes in direction of propagation. In Equation (5.3), the values of $c(\mathbf{r}_1)$, $c(\mathbf{r}_2)$ and $c(\mathbf{r}^*)$ are assumed to be equal, provided that Δt is small, therefore they disappear from Equation (5.3). Then Equations (5.3) and (5.8) are substituted into Equation 5.9a, yielding

$$\Delta\theta = \left|\hat{\mathbf{s}}_1 + \Delta t\nabla c - \hat{\mathbf{s}}_1\right| = |\Delta t\nabla c| \ . \tag{5.9b}$$

It is also apparent that if the second normal does not change much from the first normal as the ray propagates, and the path length of each ray is small and nearly equal, then the distance from the mid-point of the first vector to the second is equal to the arc length of one of the rays,

$$\Delta s = \Delta t \cdot c(\mathbf{r}) . \tag{5.10}$$

Curvature then can be defined as the amount of angular change per amount of change in distance traveled. Thus Equations (5.9b) and (5.10) can be combined to arrive at

$$K \equiv [curvature] = \frac{\Delta\theta}{\Delta s} = \frac{|\Delta t\nabla c|}{\Delta t \cdot c} = \frac{|\nabla c|}{c}, \tag{5.11}$$

which shows that for small changes in ray position with small changes in direction, the increment in curvature can be defined completely in terms of the sound speed profile of the medium. Since it is already necessary to take small step sizes and there are no abrupt changes in sound speed in the water column, this approximation is satisfactory. The next task is to then find the limit to which the cumulative curvature is to be taken before the ray state vector is recorded. This tends to be a matter of preference as to how smooth one wants the curve and how much overhead is desired. Evaluating visual renderings through a series of empirical measurements, a value of .004 radians/meter gave good results of curvature versus overhead. Now that the state vector and the processes to change the state vector in time are known, the ray tracing model is ready for simulation. This simulation is described in Chapter VII.

E. BEAM MODEL

The beam model is similar to the ray model in that it is just one level of abstraction away from the specifics of how rays are calculated. In other words, the beam model composes multiple ray models. The state vector for a beam is that of a composition of four matched rays which form a quadrilateral ray bundle. The only fields added to the state vector that have not been previously discussed are transmission loss, detect time and echo level. The transmission loss part of the state vector is used to define the logarithmic change in the intensity of the energy contained within the ray bundle, while the detect time and echo level parameters are used to document the time and intensity of a target detection. These new fields and operations are listed in Figure 5.9.

The methods needed to change the beam state vector over time and to produce usable output are thus extensions of some of the methods developed for the individual rays. These methods are initialization, propagation, calculation of transmission loss and echo level, and production of usable output.

Beam Model	
State Vector	Operations
Ray1	Set Any Parameter
Ray2	Get Any Parameter
Ray3	Calculate a Beam
Ray4	Calculate TL
Transmission Loss (TL)	Calculate Echo Level
Detect Time	Save State
Echo Level	Print State

Figure 5. 9 Beam model state vector and required operations summary

Initialization and propagation are just extensions of the ray model initialization and propagation methods; these methods perform the initialization and propagation methods on each of the four rays in the beam state vector. The visualization methods are used to generate abstract visualizations of the data stored in the beam's state vector. The explanation of the visualization model is provided in Chapter VI. The only methods remaining are those of calculating the transmission loss of the bundle of rays and the echo level of the target. Since the echo level is determined from the transmission loss, the transmission loss is discussed first.

In order to calculate the transmission loss of the ray bundle, it is essential that the ray segments are synchronized in time. As shown in Equation (4.29), given knowledge of the initial intensity, initial area and final area, the value of the intensity of the acoustic signal in the ray bundle can be calculated. This allows direct calculation of the transmission loss which is,

$$TL = 10\log_{10}\left(\frac{I_{avg}(\mathbf{r}_2)}{I_{avg}(\mathbf{r}_1)}\right) = 10\log_{10}\left(\frac{S_1}{S_2}\right), \tag{5.12}$$

where the areas, S_1 and S_2, are the same as in Equation (4.29). The values of the initial intensity and the initial area can be known from the sound source, while the value of the final area must be calculated for each spatial position of interest.

The method of calculating an approximation to the area of the ray bundle is to divide the quadrilateral beam front into two triangles. The area of these two triangles are then added together as the total area. The question is, how is an area calculated when all that is known is the positions of the four corners of the area? The straightforward solution is to use vector algebra to find the area. Figure 5.10 a) shows a quadrilateral formed by four points in space. These points are not necessarily coplanar, so they are divided into two triangles that are guaranteed to be coplanar. Figure 5.10 b) and c) show the two triangles as being transformed into parallelograms. Vector V1 is crossed into vector V2, which gives the vector whose length is equal in magnitude to the area of a parallelogram defined by V1 and V2. This value is then halved to form the area of triangle T1. The area of triangle T2 is found in the same manner as that of T1 using vectors V3 and V4. It is noted that the area of a parallelogram being equal to the magnitude of the cross product of the vectors defined by two adjacent sides is a fundamental property of the parallelogram.

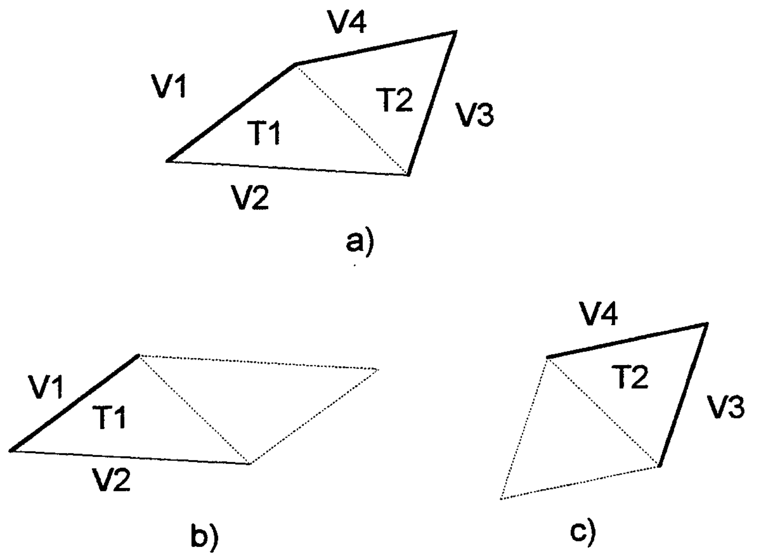


Figure 5. 10 Division of beam front into triangles for area calculation. a) beam front b) triangle T1 transformed into a parallelogram c) triangle T2 transformed into a parallelogram

The echo level, EL, is defined as

$$EL = SL - 2TL + TS, \tag{5.13}$$

where SL is the source level and TS is the target strength. For this model, the users calling program keeps track of the SL. Thus, the model calculates EL-SL. The target strength as found in Figure 15.51 from [Kinsler et al., 1982] is,

$$TS = 10\log\left(\frac{\sigma}{4\pi}\right). \tag{5.14}$$

Where the symbol 'σ' is defined to be the acoustic cross-section of the target.

Again it is shown that the beam model propagation of sound in the water column is defined by the RRA model plus the addition of three simple equations, Equation (5.12), (5.13) and (5.14). Here the power of ray tracing begins to show, through the simplicity and ease of calculation of pertinent parameters. Simple calculations provide both reduced simulation time and the ability to propagate more rays in real time.

F. LOBE MODEL

This model is the simplest by far. The lobe model merely composes several beams. A lobe is defined as a horizontal and vertical beam width and a horizontal and vertical count of the beams in each width. Thus the state vector of a lobe consists of a matrix of beams. This matrix of beams determines the accuracy of the simulation. Since ray tracing is a space-time accurate algorithm, the number of beams calculated has no bearing on position and time accuracy. When the number of beams is increased, the size of the area of the tube bundles is kept small and proper interference patterns can be maintained. Minimization of beam front area is also important for reflection purposes. As the beam gets larger the individual beams that comprise it get farther apart. The beams can, if large enough, strike a surface at significantly different times or can even strike completely different surfaces. The current reflection and detection models do not account for the size of the beam fronts and thus drastic errors can be introduced into the simulation if beam front sizes are not maintained small. The operations that can be performed on a lobe are initialization, resetting, calculating and printing. All of these operations are self evident in that they will rely upon the same type of operations in the beam sub-model to perform their operations. Each operation will call the sub-model operation for each beam in the lobe. The initialization and resetting operations also perform an energy distribution function. Each beam receives a fraction of the energy of the lobe in accordance to the position of the beam in the lobe. In this model all beams receive the exact same energy, but this can be changed to any energy distribution pattern desired. Figure 5.11 shows the fields and operations for the lobe model.

Lobe Model	
State Vector	Operations
m by n matrix of beams Beam widths	Set Any Parameter Get Any Parameter Calculate a Lobe Print State

Figure 5. 11 Lobe model state vector and required operations summary

G. SURFACE MODEL

The surface model defines the ocean surface with which the rays interact. The current model specifies the ocean surface as a smooth pressure-release surface. For most cases this is sufficiently accurate. However, if surface effects are being studied, the surface model can be easily changed and re-implemented, since it has been constructed as a separate entity from the beginning. The state vector of the surface model is that of a variable that indicates whether or not the ray in question has intersected the surface. There are two methods associated with this state vector and one utility method that outputs a representation of the surface itself. This final method will not be considered in this discussion since it is self explanatory. The first method is deciding when an intersection of the ray with the surface has occurred. The second method is the actual performance of the reflection. The reason for thinking of this as two methods instead of one is simply semantics. The first performs a query into the current state of the system and the second actually makes changes to the state. This separation was provided so that if the user model referencing the surface model needed to make this distinction, the option was available. The reflection query operation is simply performed using vector algebra. Figure 5.12 provides a graphical representation of how a surface interaction is determined to have occurred and how a reflection is made, with arrows indicating the reflected ray path.

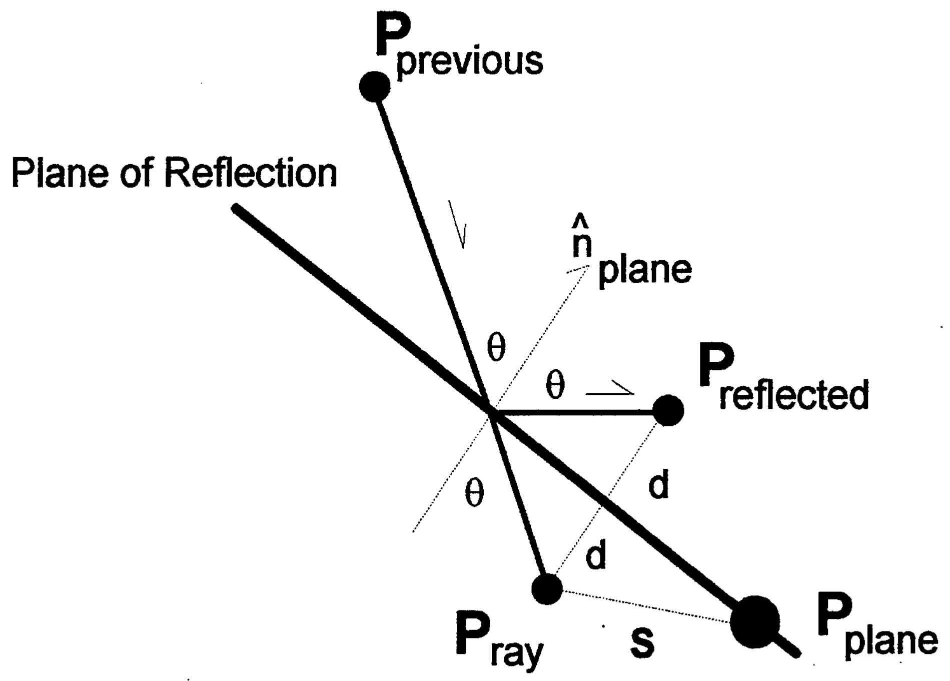


Figure 5. 12 Reflection of a ray from a plane

From the ray model it is known that the current state of a ray can be partially defined by a position and a direction normal that indicates what direction the point is moving. It is also known from vector algebra that a plane surface can be represented as a point and a normal to the surface. From this information a detection can be determined. First the vector from the defining point of the plane to the current ray position is constructed as,

$$\mathbf{S} = \mathbf{P}_{\mathbf{ray}} - \mathbf{P}_{\mathbf{plane}}. \tag{5.15}$$

Then the perpendicular distance of the ray from the plane is calculated using,

$$d = -\mathbf{S} \bullet \hat{\mathbf{n}}_{\mathbf{plane}} . \tag{5.16}$$

This distance then indicates whether or not the ray has penetrated the bottom. If the value of d is negative an interaction has not occurred and if d is positive the ray has penetrated into the bottom.

The reflection problem is founded in Snell's law, so it is there that the derivation must start. Snell's law is stated here as,

$$\frac{\sin(\theta_i)}{c_i} = \frac{\sin(\theta_r)}{c_r} . \tag{5.17}$$

Notice, however, Snell's law is stated in an inherently two-dimensional scalar form and that the ray model is stated in a three-dimensional vector form. The question is then applying Snell's law to ray model vector form. It is important to note that the law was derived for three-dimensional space and this two dimensional-simplification is valid for the reflection since it occurs in the plane formed by the incoming ray and the normal to the surface. Figure 5.12 shows a two-dimensional representation of the three-dimensional problem. Since Ci equals Cr, both the incident and reflection angles in Equation (5.17) are equal as well. From the figure it can be seen that the only correction to the ray needed is in the depicted plane in the direction of the normal to the reflecting surface. This reflection position can be found by adding twice the perpendicular distance of the ray position from the surface, to the ray position, which results in the reflected ray position. Mathematically this is shown by,

$$\mathbf{P}_{\mathbf{reflected}} = \mathbf{P}_{\mathbf{ray}} + 2d\,\hat{\mathbf{n}}_{plane} . \tag{5.18}$$

This forces the reflected normal to change in a like manner as shown,

$$\hat{\mathbf{n}}_{reflected} = \hat{\mathbf{n}}_{ray} - 2\left(\hat{\mathbf{n}}_{ray} \bullet \hat{\mathbf{n}}_{plane}\right)\hat{\mathbf{n}}_{plane} . \tag{5.19}$$

These few equations govern all reflections in the surface model and enable precise calculation of reflected rays at the ocean surface.

H. BOTTOM MODEL

The ocean bottom model is similar to the ocean surface model. The major difference is that the bottom is modeled as a rigid surface instead of a pressure-release surface. Thus the only difference is in the phase of the reflection from the bottom surface. Instead of a reflection phase change of π, the reflection phase change is zero. Also the bottom has two contour configurations. The "no slope" contour is much the same as the surface, but a depth needs to be specified, unlike the surface model. The "sloped" contour is a little more complicated; it has a flat deep bottom, a steeply sloping wall and a gently sloping shelf that tapers off to a depth of zero. All of the parameters of the "sloped" contour are changeable which makes for a highly versatile model. Figure 5.13 shows a three-dimensional representation of the surface and the "sloped" bottom models. The bottom model uses the exact same state vector and operations as the surface model. Therefore,. it also uses the exact same equations as the surface model and as such a discussion of the specifics of a bottom model is not warranted.

A final statement as to the generality of the bottom model needs to be made. Any surface can be broken down into a near plane surface by subdividing the surface into infinitely small pieces. The bottom model can be broken into a piece-wise planar (gridded) surface, with each plane specified by a point on the plane and a normal to the plane as well. With careful planning, almost any bottom interface can be modeled with ease and low computational overhead. Care must be taken however to ensure that the characteristic size of the bottom structure does not become so small that it approaches the scale of the sonar wavelength. This is because errors due to phase interactions become large, in this model, when bottom structure and wavelength are of the same order. An important area of future work is the addition of methods which can import standard bathymetric databases and compute corresponding normal vectors.

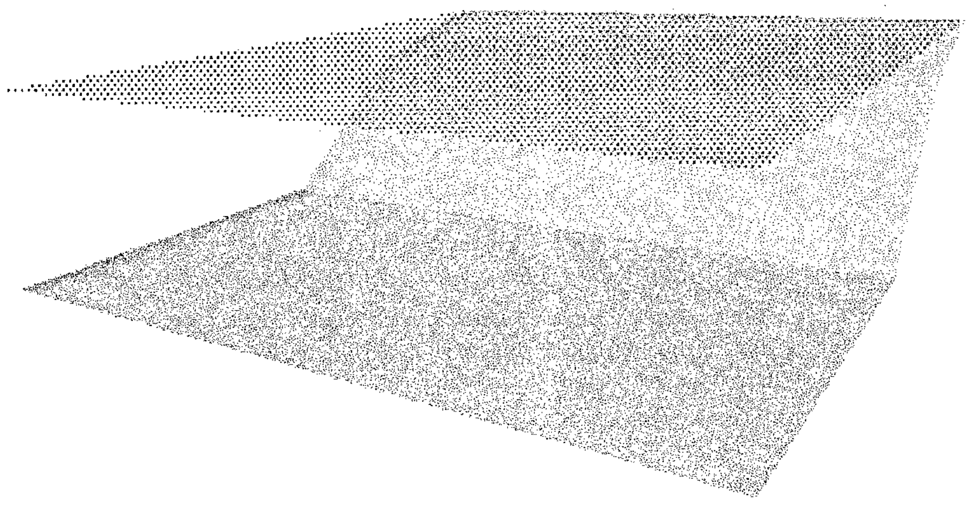


Figure 5. 13 Example sloped bottom contour with surface model

I. SOUND SPEED PROFILE (SSP) MODEL

This model is a simple one. It has no state vector and thus supplies information about the current time and position only. Its purpose is to return the sound speed and the rate of change of the sound speed with position. A detailed discussion of SSP is available in [Urick, 1975]. It has five basic profiles that come from [Ziomek, 1996] as the standard profiles. Again due to the highly modular breakdown of the RRA model, changes in the sound speed profile model are made easily. Chapter VIII Simulation Results, details in specific the SSP models used in the current simulation. An important area of future work is the development of an SSP database linked to the bottom model, import of real-time SSP data and SSP data visualization utility operations. It is important to note that the broad generality and accuracy of RRA is directly dependent on the quality and accuracy of local bottom, surface and SSP models.

J. TARGET MODEL

The target model is designed to keep track of all targets in the virtual world and to provide operations that determine if a target has been detected by a sonar pulse. A target is defined as any object that is not environmental, geological or biological. Therefore the target model gives us information about the man-made objects in the virtual world. It does so by keeping information about each target in the virtual world and comparing that to position information provided by the beam model. The state vector of the targets contains all the information about the targets.

In the current implementation the targets are randomly placed moored mines and submarines that have no movement. Thus in the current target model the target state vector does not change. This state vector is composed of the position and radial size of the mine. As for the submarine, it is considered to be composed of several mines and as such just generates a chain of mines the size of the submarine. This model is intentionally simplistic and much further work is warranted in target modeling. Additionally, information pertaining to the cumulative area of targets detected in the current time step in the beam is saved. The operations affecting the state vector are initializing, resetting, detecting a collision, getting the area of detection and outputting the target information. Again initializing, resetting, getting detection area and outputting are self explanatory.

The only operation left is computationally efficient collision detection. This model uses the same equation for detection of a possible interaction between the sonar beam and the targets as the surface model uses to detect interactions between the ray model and the surface. Equations (5.15) and (5.16) are used to indicate when the plane of the beam front passes through the target. Therefore the perpendicular distance from the plane of the target, d, is used to tell if the target has the possibility of being detected. When the sign of d changes from positive to negative this means that the wave front has encountered the target in the current time step. Since the plane passing approximating the wave front of the beam is infinite in extent further operations are necessary to check if the target (or any portion of it) is in the beam tube. This detection is performed by checking to see if the target lies within the X, Y and Z coordinates of each pair of points on the box that make up a diagonal in the box. Figure 5.14 visually depicts the bounding box concept. The presence or absence of a target is reported at the end of every check.

Determining the distance to every object is also useful in another way. The distance of the closest target during each check is recorded and this distance is divided by a nominal ocean sound speed of 1500 meters per second. This time is then divided by the time per propagation step. The result is the number of iterations that can occur without checking to see if a collision occurs. Essentially, this number of time steps that can be skipped in between checks for target interactions is yet another member of the state vector. This has the potential to be a dramatic speed enhancement over checking all targets every time step.

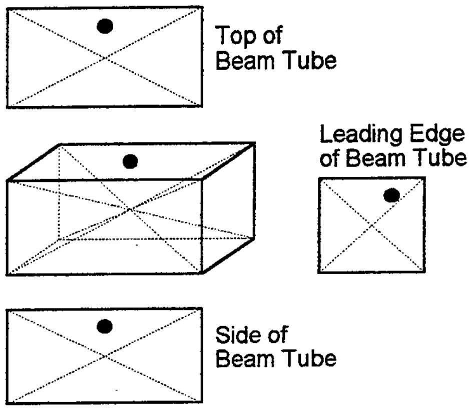


Figure 5. 14 Bounding box target detection. A target in the sonar ping volume is considered detected

K. PINGSERVER AND PINGER MODELS

The pingserver and pinger models are included as an example to show how the RRA class library can be used. The pinger model is a stand-alone sonar ping model that produces text files that are either raw data or three-dimensional renderings of the sonar lobe. The pingserver model is where the true power of the RRA model comes into play. The pingserver uses a client-server communications approach, where the client asks for a sonar beam to be projected into the virtual world and then the server calculates the necessary data and returns this data back to the client. A major benefit of this approach is that a proxy server can be placed in between the client and server to improve overall system scalability. This proxy server appears to the server as a client and to the client as the server (hence the name). Since the RRA algorithm is a space-time based algorithm (no transforms to other domains), the propagation calculations can easily be partitioned and shared among multiple pingserver implementations. Thus, the proxy server has the job of getting the request for the sonar ping and then distributing the calculation of the many beams required to the many available servers. With the advent of large-scale network clustering, the possibility of distributing real-time RRA calculations is highly attractive.

At this point, a step back from modeling to discuss computer implementation seems appropriate. Network clusters, [Hill, et al., January 1998], are collections of large numbers of moderately fast general purpose computers connected via a fast local-area network (LAN). The motivating idea is that when a highly parallel calculation needs to be done, the task can be split up among the many computers. Such clustering can provide gigaflop performance from machines that provide only megaflop performance individually. The most amazing aspect of this clustering is that since the individual computers are mass produced and general purpose, each one has a very low price. The cost for the creation of one of the clusters, Loki, was around $50,000 for a peak performance of 1.4 gigaflops. This cluster was initially made around 1996 and production of this same system as of 1998 is estimated to cost 50% less. It is seen that the RRA model lends itself to this type of massively parallel computing and it is the pingserver model that provides the example of how easy this parallel programming can be. Thus, the prospect of high-resolution sonar beam calculation in real time appears quite feasible. Distributing RRA sonar beam calculations over many computers is an important area for future work.

L. SUMMARY

This chapter has presented the basic model for the implementation of a RRA simulation. The use of the models presented has provided the ability to simulate a virtual sonar environment. The ray model provides the basic structures to propagate a sonar ray in real-time through a virtual world. The beam model combines four rays to form a beam tube and then adds operations for attenuation and collision detection. The lobe model combines a variable number of beams and propagates all of them as one unit. The ocean bottom and surface models provide reflection information to the ray model. The SSP model provides information on sound speed to the ray and target models. The target model maintains a target database and provides efficient operations to the beam model to determine if a detection has occurred. The pingserver and pinger models combine the various models into a cohesive simulation and provide example implementations.

The next step in the process is to convert these model ideas into a working simulation. Chapter VII addresses the topic of implementation and integration of the RRA model with the visualization model. Appendix A contains the simulation code for the RRA models.

VI. SONAR VISUALIZATION

A. INTRODUCTION

Simulations typically require some form of visualization. Whether the visualization is in the form of text tables or graphic plots, the simulation must produce data in a logical and orderly grouping of similar information. In the early days of scientific discovery, information was collected in tables and occasionally plotted on paper, usually by hand. With the coming of the digital computer, the mainstay of data visualization shifted from text tables to graphic plots, usually in two dimensions (2D). With the increase in computational power of digital computers and the widespread availability of three-dimensional (3D) rendering software, the time has come to shift once again to a higher form of visualization. That higher form of visualization is 3D rendering.

In the fleet today, sonar tactical planning is still performed from text tables and 2D plots. This approach was fine for twenty years ago when less data was processed and evaluated, but today the sheer volume of data and the finer resolution of information obtained demand a more robust visualization of the incoming data. Since we live in a 3D world, the next step ought to be 3D plots. [Karahalios, 1991] discusses the history of data visualization. In that thesis she quotes from [Defanti, 1987] stating that half of the human neo-cortex is devoted to visual information processing. Thus, since our brain is wired for visual data input and that typically is 3D in nature, the natural conclusion is again 3D representations. This chapter explores the possible 3D visualization model for presentation of the results of the RRA simulation.

B. VISUALIZATION CONSIDERATIONS

1. Tactical Visualization

In the U.S. Navy fleet today, visualization of four dimensional space-time is performed using two dimensional plots. These plots have various combinations of positional and time information as the axes of the plots. Time-range, time-bearing, and geographic plots are but a few of the examples in widespread use. In addition to these time-space plots there are also time-frequency and space-frequency plots, again in two dimensions. The usual practice for evaluating these plots in real time is to show several varieties of them at the same time on separate computer displays. There are typically several console operators and one supervisor interpreting such plots. The job of the supervisor is to integrate the plots mentally and develop a four-dimensional (space-time) mental model of the current sonar-based tactical picture in the real world.

In years past when shipping was not nearly as heavy and detection ranges were much shorter, the incoming volume of sonar information was not overwhelming. Today, with increased shipping and increased detection ranges, the volume of information processed by a ship's sonar system has dramatically increased. This has often led to an overload of information on the operators themselves. To complicate matters, the quieting of modern submarines has led to decreased detection ranges for the primary threat of interest. With the increase in noise and decrease in signal, these shortened detection ranges result in shortened reaction time. Though the human brain is wonderfully created, expecting an information-overloaded supervisor to make snap decisions while mentally integrating several plots into one multiple-dimension plot is extremely difficult.

While Gary Kasparov proved that a man using intuition could frequently outthink Deep Blue, IBM's chess-playing supercomputer, one must remember that a chess board is infinitely less complex than the real world. Deep Blue on the other hand proved that raw processing power and proper programming eventually can defeat one of the best chess-playing minds. This digression points to the fact that the human brain has limitations in its ability to process large volumes of information and infer likely outcomes. This fact leads one to believe that more innovative means of visualization are necessary to allow the operator and supervisor to step one more level of abstraction away from the raw data, so that timely and accurate decisions can be made.

In the scientific research community, sonar visualization has really not progressed much past the state of the art in the Naval Service. Typically, the extent of visualization is colorful 2D plots, which in essence are pseudo 3D plots. Nevertheless, when complex space-time processes are being investigated, ingenious use of 3D plots, color, intensity, and transparency can provide the researcher with pseudo four and five-dimensional plots. All of these properties can be displayed using existing rendering programs. As with most physics solutions, judicious simplification and abstraction are a must if a solution is to be obtained. Scientific visualization of sonar data can benefit from careful simplification and abstraction as well.

2. High-Dimensional Space

Acoustic data is typically a volumetric data set rather than a surface data set. In addition sonar has various parameters of interest that one can consider as additional dimensions. Sound intensity level, time of travel, and frequency are but a few examples of these extra dimensions. In the visualization discipline this is termed a "high-dimensional data space." For the dimensions just mentioned a six-dimensional plot needs to be generated to simultaneously render these parameters. How can such a plot be conceived? In the world of 2D plotting, special techniques of coloring and use of contours transform these plots into pseudo three-dimensional plots. Similar techniques can be applied in 3D renderings. By mapping the additional dimensions to other graphics quantities besides spatial, pseudo high-dimensional space plots can be generated.

3. Available Graphics Parameters

The question then is how to map these extra-dimensional parameters. In most 3D rendering languages, the objects that are created have rendering properties themselves that can be used as dimensional extensions, much the same as contour lines are used on geographic mappings. The properties that objects possess include color, intensity, transparency, smoothness, reflectivity and emissivity. The natural tendency at this point is to imagine that with all of these parameters a pseudo nine-dimensional space is possible. This is just not so. Many of the parameters rendered at the same time produce unpredictable results or do not elucidate the information intended. For example, no matter what color is given to an object, if the object is transparent, it cannot be seen. So indeed, when one decides to implement high dimensional space data rendering, careful attention must be given to ensuring that the parameters employed do not overlap in such a way as to result in incomprehensible data. These fundamental challenges form the basis of scientific visualization research.

4. Mapping Data to Graphics Parameters

The task of mapping, for the reasons stated in the last section, must be undertaken with great care. Suppose that time of travel of a sonar pulse was linked to the transparency parameter and that the intensity of the sonar pulse was mapped to the color. If care is not taken, long before the sonar pulse becomes insignificant, as far as intensity is concerned, the pulse may not be visible in the rendering. This premature invisibility produces a loss of information in the scene that cannot be recovered. Now suppose that the two rendering dimensions and parameters are swapped. Thus, when the sonar pulse intensity becomes negligible the time (mapped to color) will no longer be visible. This situation is of little concern since the information on intensity is of greater importance that the information on time. In other words, if there is no perceptible sonar pulse, the time that it arrived does not matter. This example indicates that a hierarchy of informational importance and a hierarchy of object parameters, likely exists, and is variably dependent on the goals of the user. The hierarchy of parameters must be chosen so that the parameters on the lower end of the hierarchy have little influence on the parameters that are on the higher end. Careful use of such a hierarchy may prevent the rendering engine from eliminating important data from the visualization.

5. No Single Right Answer

Numerous scientific visualization research projects are presently being conducted that are investigating high-dimensional data space and the techniques used to make sense of the vast quantities of data contained therein. A particularly interesting finding is that there is no single right answer as to how the data must be viewed. The type of data and how it is to be used directly influence the way in which the data is presented. The format for presenting information from which snap decisions are made will likely prove to be different from the format from which keen insights into physical processes are made. Indeed, it is often user interaction with the visualization data that provides insight rather than any single presentation method. Thus, the most important goal and contribution of this project is support for user exploration of high-dimensional sonar information using real-time 3D graphics and scientific visualization techniques.

6. Initial Visualization Recommendations

The initial recommendation for visualizing sonar data is that the three spatial components are mapped to the three spatial coordinates of the visualization system. The sound intensity is mapped to transparency in a dynamic visualization, while color is mapped to detectability. In a static beam, intensity, distance from source and detectability are mapped to color. All three can be mapped to color by providing a means of selecting the desired parameter. A full investigation of sonar parameters and the means of best visualizing them definitely needs to be continued as future work. These initial visualization recommendations are provided as a starting point for evaluation of the most important sonar parameters provided by the RRA model.

C. RRA SONAR VISUALIZATION MODEL

1. Interactive Server Model

Figure 6.1 depicts the standard visualization model. This hierarchy provides the general high-level concept of what modules comprise a visualization model. Interactive visualization also suggests the need for an interactive user interface model, which is depicted in Figure 6.2.

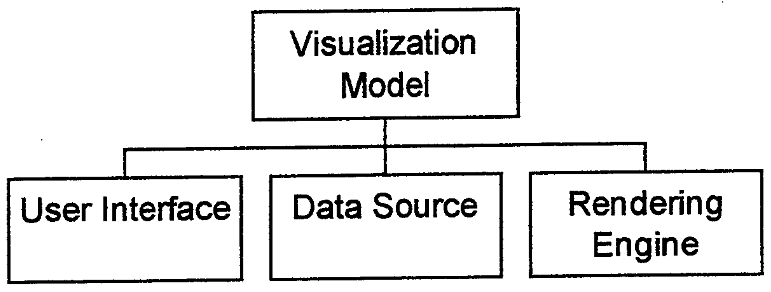


Figure 6.1 Visualization conceptual hierarchy

a. Data Flow

The data flow in the interactive model is quite simple. After all communications between the sections have been established, which happens transparently to the user, the simulation is waiting for user interaction. Figure 6.2 shows the hierarchy of the user interface model. The user enters the specifics of the characteristics of the sonar pulse desired and the desired parameters of the platform containing the virtual sonar system. When all initialization data is entered, the user can execute the ordered platform and sonar pulse commands by pressing the appropriate input buttons. This returns control from the control panel to the battle scene manager which relays the sonar ping requests to the sonar server and executes the platform commands. The battle scene manager then waits for return of the requested sonar information from the sonar server. When the requested data arrives, it is processed and forwarded to the rendering system via the rendering system interface. For responsiveness threads are used extensively in the processing of the interactive model.

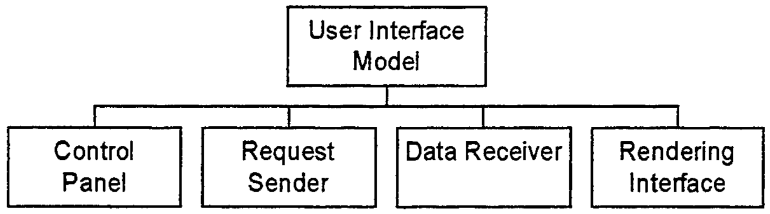


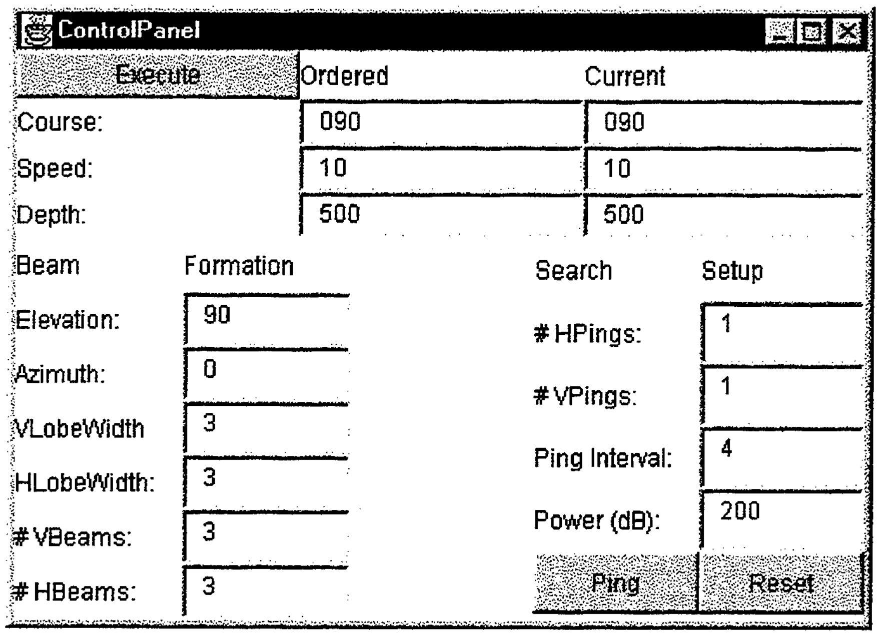
Figure 6.2 General user interface model hierarchy

Threads are processes that can be run in parallel and are used when multiple concurrent processes are required. Thus the platform controller has a timer thread that updates the vehicle position every tenth of a second. Each of the interfaces to the battle scene manager, request sender, data receiver, rendering interface and the control panel, are also threads. This allows the battle scene manager to execute each phase of the management process in parallel and only control the timing of when the results of each interface are added to the current state of the system.

b. Example Execution

This section describes how the individual components in Figure 6.1 are executed. The interactive sonar server is executed on a dedicated remote computer, which allows the server to be located on the most computationally capable computer. The proxy server, a bridge between the user interface client and the sonar server, is typically executed on the client computer where the user determines the visualization is to be run. This is done because most web browsers have security features that prevent them from communicating directly with other networked computers. Finally, the local web browser is started and the virtual world is loaded. The loading of the virtual world performs two functions; the rendering system is loaded and the user interface, which is linked to the virtual world, is started in the browser. At this point the entire interactive simulation is initialized and the control panel is ready to accept user input.

As an example of simulated sonar operation suppose a single sonar ping with a three degree horizontal and vertical lobe width is desired. Each lobe is divided into nine sub-lobes called beams. Thus each beam has a one degree horizontal and vertical beam width. Figure 6.3 shows a control panel with all the choices filled in and thus defines the requested ping. When the 'ping' button is pressed the control panel thread passes the pulse specifications to methods in the battle scene manager. The battle scene manager invokes the request sender thread and the data receiver thread. The ping sender thread communicates with the proxy server which in turn communicates with the sonar server that is running on a separate machine. The sonar server calculates the positions of the sonar beams over time and detects any collisions with objects in the virtual world. While this computation is occurring, the data receiver thread is waiting for information and the ping sender thread finishes execution and terminates. When RRA calculations are finished, information on the sonar beam positions and target detections is delivered over the local area network to the proxy server, which forwards the information to the data receiver thread. The data receiver thread returns the information to the battle scene manager and then terminates execution. The battle scene server then executes a VRML generation thread that converts the raw data into a valid VRML node structure. This node is then sent to the VRML scene. When the node is processed the target information appears on the screen as well as a representation of the beam propagation through the water. This process can be repeated over and over again to explore the entire virtual world. Additionally, if the battle scene manager is modified to eliminate user input and an artificial intelligence module is added to control the sonar and platform, an autonomous virtual vehicle can be created.



	Ordered	Current
Course:	090	090
Speed:	10	10
Depth:	500	500

Beam	Formation
Elevation:	90
Azimuth:	0
VLobeWidth	3
HLobeWidth:	3
#VBeams:	3
#HBeams:	3

Search	Setup
#HPings:	1
#VPings:	1
Ping Interval:	4
Power (dB):	200

Figure 6.3 Interactive server control panel

2. Stand-alone Server

Figure 6.1 also serves as a depiction of the stand-alone server model. As with any useful program it must accept input, access data and return information to the user.

a. *Data Flow*

Data flow in this model is extremely simple. Appendix B shows source code for a typical example. In this example the virtual world is constructed and filled with targets. The platform position and sonar characteristics are established in the lobe class. The lobe class then constructs the beams that comprise it. The beam class then constructs the rays that form it and the simulation is ready to run. The lobe position (and thus the beam and ray positions) is calculated over time and the subsequent positional data is returned as valid VRML 97 code. Then the VRML code representing the surface, bottom and lobe is printed to the console.

b. Example Execution

The only action that the user has to perform once the stand-alone model is coded and compiled is to execute the program. When the program is executed, the surface, bottom, sound speed profile, target and lobe model implementations are initialized. The lobe parameters (and thus the beam and ray parameters) are initialized and the lobe is calculated by calculating the time behavior of each beam in the lobe. Each beam is then calculated by calculating how each ray interacts with the surface, bottom, water column and the targets in the virtual world. Upon completion of the calculations, methods in each model class are called that produce VRML 97 compliant code that allow visualization of the most recent lobe calculation. The simple examples provided in this thesis produce compelling results. Much more sophisticated applications and exciting research in real-time 3D sonar visualization now appears possible.

D. CONCLUSIONS

With current sonar data collection and processing abilities continuing to accelerate, something needs to be done with visualization to ease the information burden on the person evaluating the incoming data. Creative and ingenious ways of displaying data in three dimensions must be devised if even higher information density is to be processed by a human. This new level of abstraction will provide enhanced understanding of the problem under consideration and enhanced ability to make more rapid decisions. A series of example programs which use the RRA API are presented. Initial results provide useful insight and are extremely encouraging. Real-time 3D sonar visualization is an important new field which now appears computationally feasible. A great deal of research and experimentation is needed to capitalize on this new tool.

VII. MODEL IMPLEMENTATION AND INTEGRATION

A. INTRODUCTION

This chapter discusses the practical factors involved in implementing a simulation. Choosing the computer language with which the model is implemented into a simulation is a critical task. The speed of execution of the code, the ease of integration with some form of visualization system and the network interface methods are discussed as they pertain to the RRA simulation. Implementation of the sonar server model (RRA model) and the visualization model is summarized and the integration of these two models is discussed in the last section.

B. CHOICE OF PROGRAMMING LANGUAGE

Using the models specified in the preceding chapters, it is time now to turn to implementation of the simulation. As with most simulations in the recent past, this simulation is implemented on a general purpose digital computer. The primary question is choosing the best computer language to use to accomplish the goal. One goal is that the simulation model components might be distributed as widely as possible. This is a daunting task, since any two computers are rarely configured compatibly. The design criteria for this thesis are provision for cross-platform compatibility, computational speed, real-time 3D visualization and network programming.

1. Cross-platform Compatibility

With the creation of the Internet, cross-platform compatibility has become a rallying cry. In response many private sector companies have developed tools with the network and inter-connectivity in mind. Due to the nature of the RRA model and its multiple potential uses, choosing a programming language designed for network programming is a major requirement. In the next three sections, two complimentary programming languages and two networking protocols are discussed that have recently come into existence that allow all goals to be met.

2. Number-Crunching Speed

As with any inherently physical simulation of a natural process, computational speed is essential if real-time interaction is required. Even if real-time interaction is not required, computational speed is still an important consideration. However, the source of speed is no longer restricted to highly optimized, near machine language, already-compiled code running on supercomputers. The source of speed is the networked interconnection of many computers. In a sense, the network is the computer. With this in mind, Java, developed by Sun Microsystems, is a sensible choice for a programming language. Even though compiled Java code is 5-15% slower on a given single computer compared to optimized FORTRAN or C/C++, Java from the start has been tightly integrated with the Internet and network computing. This tight coupling leverages the power of the Java language to be faster in completing a distributed task than either of the two standard scientific languages. Although C/C++ has seen good success in network computing, networking in C/C++ is not standardized completely and not designed into the language from the very beginning. Another factor weighing heavily is that the Java development environment is completely free as compared to most full-featured FORTRAN and C/C++ compilers. The free aspect of Java allows a person developing an application to know that money will not stop a potential user from either not using it or from trying to use a different version of a given type of compiler that is already owned. When a potential user wants to modify a RRA application, he must merely retrieve the Java runtime environment via the Internet, install it and run the application. Since an unlimited speed potential and widespread availability of the common operating environment exist in the form of Java, it is the logical choice for implementation.

3. Visualization

Though Java does provide some facilities for data visualization, the goal for this simulation was three-dimensional (3D) visualization. Java does not yet supply all the needed constructs to form a truly 3D virtual world and is not used for scientific visualization (although the Java 3D libraries have recently become available). Once again another freely available software language came to the forefront. This product is the Virtual Reality Modeling Language (VRML), which typically runs and is rendered inside a web browser. It is usually a plug-in program to the standard web browsers provided by Microsoft and Netscape. Again, this combination of VRML rendering engine and web browser comes at the right price: no cost. VRML gives all the constructs to form an immersive 3D environment as well as constructs to make the environment change over time with user input. As an added bonus, due to the wide acceptance of Java as the language of the web, direct interface from Java to the VRML web browser is provided. Other graphics-rendering programs exist and are free to distribute as well, such as openGL, but none are so tightly interfaced, as widely available or as popular as VRML. This browser interface allows the interconnection of standalone Java programs to VRML scenes through the web browser. Clearly the choice of visualization language is VRML since it integrates so tightly with the Java compiler already chosen and can immediately render RRA results on any personal computer. More information on integration of Java and VRML is found in [Brutzman, 1998].

4. Networking

Java already has many fine networking features. Typically the data that can be passed through network links is formatted in a way that only reader/writer programs written together can decode the information. In highly structured simulations, entire state vectors need to be passed. Typically state vector changes render all previous versions of a program useless. Two protocols are available which address standardization of data passing. These protocols are Common Object Request Broker API (CORBA) and Distributed Information Simulation (DIS) protocol. These protocols are robustly designed and freely available, but also are complex and difficult to learn. Another candidate is the dial-a-behavior protocol work in progress at NPS which simplifies formal specification of over-the-wire data formats [Brutzman, August 1997]. Thus for the purposes of this thesis the networking facilities provided by Java proved to be sufficient. In the future, as this simulation is integrated into a larger distributed simulation, integrating one of these data-oriented protocols will be useful and necessary. More information on graphics internetworking is found in [Brutzman, August 1997].

C. MODEL IMPLEMENTATION

Now that the necessary software tools have been chosen to transform the many models into a simulation, it is time to present some aspects of the implementation that shaped the final Java class library for the RRA simulation. The following two sections will discuss some of the particulars of the implementation of the RRA and visualization models into a full-fledged API.

1. RRA Implementation

Implementation of the RRA model was straightforward since Java is an object-oriented programming language. Each sub-model of the RRA model was developed as a separate class, since classes in Java are an encapsulation of variables and methods in one common object. Member variables contained in the object keep track of the state of the object and methods contained in the object allow changing of or accessing of the data in the class. This correlates well to the model concept of state vectors and operations. A completed class then becomes a building block for other higher-level classes. As an example, suppose the ray model has been implemented as the ray class. This class itself becomes the definition of a ray object template. As with any object that can be produced from a template, many exact duplicates of the object can be made from the template. This object creation from the class definition is termed instantiation. Thus, with the ray class, many individual rays can be created and as with all objects the state vector describing each object can have no influence on the state vector of any other object. It is seen that the formation of the beam class requires the creation of four individual ray objects. In addition to the ray objects, methods must be made that manipulate the ray objects, in accordance with the allowed operations on ray objects, and that produce the desired effect on the beam object. This object-oriented layering continues through the lobe model to the pinger and pingserver models.

The only methods yet to add to all of these models are those that allow each object to sense interactions and perform actions based on the perceived interaction. For example, when a sound wave strikes a rigid surface, the wave bounces off the surface in accordance with Snell's law. Thus a method is added to the bottom model that allows a ray object to communicate with the bottom object. This communication allows the bottom object to retrieve data from the ray in regards to its position and direction of motion. With that retrieved information the bottom model is then able to tell if an interaction has occurred and can report its findings back to the ray object. The ray object is then able to alter its state with the reflection information provided from the bottom object. The full RRA model is then simulated following these rules of layering and object interaction. The full RRA code is found in Appendix A. Several stand-alone visualization example programs are found in Appendix B. The source code for the pinger and pingserver programs are found in Appendix C. Excerpts from the very useful auto-generated Javadoc software documentation are found in Appendix F.

2. Visualization Implementation

Implementation of the visualization model requires implementation of each of the visualization sub-models just as with the RRA implementation. Whereas implementation of each of the RRA sub-models is quite similar, implementation of each of the visualization sub-models is not.

The user interface sub-model is implemented in two different ways, depending on whether or not the model is stand alone or interactive. The stand alone model uses the web browser user interface. Each desired data set to be viewed is retrieved via a web page or the web browser's file selection routines. At this point the interactive model requires that user input be supplied, however the standard web browser does not provide this kind of input. The VRML world can be programmed to perform user input, but it is difficult compared to functionality already built into Java. Thus, a Java class called the battle scene server provides a control panel for the user to enter detailed information about the desired sonar parameters. This class is much the same as the classes in the RRA class package, but differs in the ability to be executed in a VRML web browser as a program script file.

The data source sub-model also has its implementation determined on the basis of the form of user interaction. In standalone visualization, the source of data is from a file provided by the local operating system as requested via the web browser user interface. In interactive mode the source of data is read via sockets across the Internet by the battle scene server object.

The data from the two different sources is provided to the graphics rendering model. This model is implemented via the VRML browser operating inside the web browser. In standalone mode information from the data source is passed to the VRML browser by the web browser. This data is parsed and rendered by the VRML browser and appears on the computer screen as a virtual world. In the interactive mode the data is parsed by the battle scene server via a method call to the VRML browser. This parsed data is directly inserted into the VRML browser scene graph via another method call to the VRML browser. The data then appears in the currently open virtual world.

D. INTEGRATION

1. Standalone

Integration of the RRA and visualization model implementations into a complete simulation is the final task. The point of interaction and thus the point of integration in the RRA implementation is the pinger class. The pinger class provides for the text being written to a system file. This text is in the form of VRML compliant code that specifies the requested 3D visualization of the desired data, which is stored to a file on the computer. This file can then be viewed with the VRML browser in a semi-interactive manner; the user can explore the data from any perspective, but cannot interactively modify the sonar beam parameters.

2. Interactive

In interactive mode, data transfer occurs across the Internet with the pingserver of the RRA class package acting as the server and the battle scene server of the visualization class package acting as the client. A VRML world is created, and an object in it has sonar capability and thus has a connection out of the world to the web browser. The battle scene server interfaces with the user, the object in the VRML world and the pingserver. When the user requests a sonar pulse, it tells the pingserver the specifics of the request and the pingserver starts calculating. When the calculations are complete the information is forwarded from the pingserver to the battle scene server in the form of VRML compliant text strings. The battle scene server then converts the text to compiled VRML code. This compiled code is sent to the object in the VRML world where it is automatically added to the existing scene and then rendered.

E. CONCLUSIONS

The choice of the programming language depended on many requirements. Java from Sun Microsystems met each of the requirements and was selected. It was selected for its cross-platform compatibility, quick execution, visualization capabilities and network capabilities. The choice of Java allowed the RRA model to be converted to an API for use in simulation. The highly structured object-oriented features of Java allowed the API to be built in a modular fashion. The ability of VRML web browsers to execute Java code created a tight integration between the VRML browser and the Java RRA API. The ability to integrate well with Java is the main reason that VRML was chosen as the rendering language. The integration of the API and the visualization model implementation developed into two programs. One that allowed interactive control of the RRA API parameters and VRML parameters and the other only the VRML parameters.

VIII. SIMULATION RESULTS

A. INTRODUCTION

This chapter presents the simulation results. The accuracy of the RRA API is shown to be nearly identical to the original RRA program as presented in [Ziomek, 1993]. In addition, the accuracy of energy propagation is compared to the normal mode model. This comparison proves that the RRA API is reasonably accurate in energy propagation. Detection of virtual world objects provided expected results with a two percent error rate for randomly placed mines. Visualization results were very good, but they barely scratch the surface of the abilities of real-time 3D sonar visualization.

B. RRA TEMPORAL AND POSITIONAL ACCURACY

In [Ziomek,1993] a comparison of the output of the RRA code to that of a set of four coupled ordinary differential equations (ODEs) was shown. In summary it showed that the outputs down to several decimal places were identical for many different bottom depths and sound speed profiles. Thus it was shown that the RRA solution was not erroneous in its development. Also the RRA algorithm is a full 3D propagation algorithm that allows for range-dependent environments. These features of RRA allow it to be a very general high resolution and high accuracy solution for a wide variety of ocean conditions.

C. SIMULATION ACCURACY: JAVA VERSUS FORTRAN

Ziomek's original RRA code was written in FORTRAN while the code for this thesis is written in Java. Since Ziomek's RRA code was validated against an accepted set of coupled ODE's, showing that the implementation in this thesis compares well to his data set provides validation of the Java implementation. Below are several tables that show the comparison of the RRA simulation in both FORTRAN and Java. Because differences between the solutions are negligible, no plots of these results are provided. Thus the Java implementation produced by this thesis provides valid results for Ziomek's recommended test cases.

Table VIII.1 Sound Speed Profile 1: constant speed of sound

$c(y) = 1500$ m/s $\quad 0 \le y \le 200$ m

Method	y_0 (m)	β_0 (deg)	r (km)	y (m)	τ (s)	s (km)
FORTRAN	100	45	10.0	99.95	9.428091	14.1421
Java	100	45	10.0	100.00	9.428090	1.41421
FORTRAN	100	85	10.0	174.88	6.692132	10.0382
Java	100	85	10.0	174.87	6.692132	10.0382
FORTRAN	100	135	10.0	100.05	9.428091	14.1421
Java	100	135	10.0	100.00	9.428090	14.1421

Table VIII.2 Sound Speed Profile 2: Linear SSP with a positive gradient

$c(y) = 1500$ m/s $+ (0.017/\text{s})y \quad 0 \le y \le 200$ m

Method	y_0 (m)	β_0 (deg)	r (km)	y (m)	τ (s)	s (km)
FORTRAN	100	45	10.0	99.95	9.417422	14.1421
Java	100	45	10.0	100.36	9.417592	14.1424
FORTRAN	100	85	10.0	162.57	6.683496	10.0374
Java	100	85	10.0	162.74	6.683505	10.0374
FORTRAN	100	135	10.0	100.05	9.417422	14.1421
Java	100	135	10.0	100.03	9.417403	1.41421

Table VIII.3 Sound Speed Profile 3: Linear SSP with a negative gradient

$c(y) = 1500$ m/s $+ (-0.017/\text{s})y \quad 0 \le y \le 200$ m

Method	y_0 (m)	β_0 (deg)	r (km)	y (m)	τ (s)	s (km)
FORTRAN	100	45	10.0	99.95	9.438795	14.1421
Java	100	45	10.0	100.40	9.438980	14.1424
FORTRAN	100	85	10.0	168.96	6.699409	10.0380
Java	100	85	10.0	168.86	6.699402	10.0380
FORTRAN	100	135	10.0	100.05	9.438794	14.1421
Java	100	135	10.0	100.44	9.438583	14.1418

Table VIII.4 Sound Speed Profile 4: Parabolic SSP

$c(y) = 1490$ m/s $+ (4\text{x}10^{-5}/\text{ms})(y\text{-}500\text{m})^2 \quad 0 \le y \le 1000$ m

Method	y_0 (m)	β_0 (deg)	r (km)	y (m)	τ (s)	s (km)
FORTRAN	500	85	10.0	777.48	6.705893	10.0150
Java	500	85	10.0	777.45	6.705896	10.0150

FORTRAN	500	95	10.0	222.52	6.705893	10.0150
Java	500	95	10.0	222.55	6.705896	10.0150
FORTRAN	500	135	10.0	545.00	9.448829	14.1104
Java	500	135	10.0	543.48	9.449543	14.1115

Table VIII.5 Sound Speed Profile 5: Classic SSP

$$c(y) = 1500\ \text{m/s} + (0.016/\text{s})y \qquad 0 \le y < 100\ \text{m}$$

$$c(y) = 1501.6\text{m/s} + (-0.02956/\text{s})(y-100\text{m}) \qquad 100 \le y < 1000\ \text{m}$$

$$c(y) = 1475\text{m/s} + (0.017/\text{s})(y-1000\text{m}) \qquad y>1000\ \text{m}$$

Method	y_0 (m)	β_0 (deg)	r (km)	y (m)	τ (s)	s (km)
FORTRAN	50	45	10.0	1760.21	9.604047	14.2771
Java	50	45	10.0	1758.87	9.604941	14.2781
FORTRAN	50	85	10.0	1595.33	6.806873	10.1264
Java	50	85	10.0	1596.28	6.807168	10.1266
FORTRAN	50	125	10.0	840.08	8.295966	12.3284
Java	50	125	10.0	839.14	8.296582	12.3290

where y in the Ziomek coordinate system is identical to depth of water. Figure 8.1 shows a representative deep-water SSP. Each profile also reflects the effects of a pressure release surface and rigid bottom.

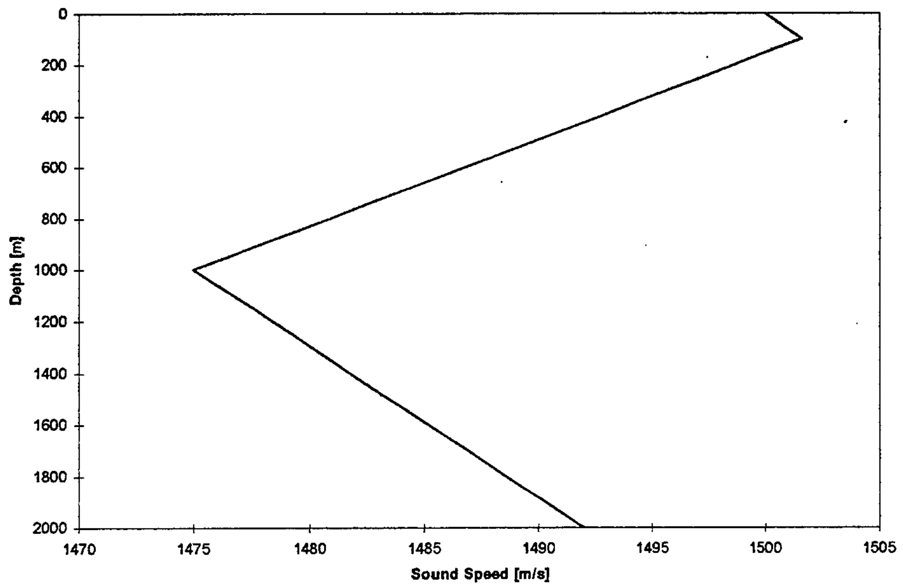


Figure 8.1 Representative deep-water sound speed profile

Minor differences between the solution implemented in Java and FORTRAN are noted in the above tables. These differences were made smaller by choosing a smaller step size of time in the simulation. The reason these results were presented and not the more accurate values was execution time related. The smaller step-size runs had run times on the order of several minutes while the larger step-size had run times on the order of tens of seconds. Since the main use of the RRA API is intended to be a real-time simulation, execution time does become a consideration as well as accuracy. The compromise was to choose the quickest time that had no more than 1% error, compared to the FORTRAN implementation of RRA. This resulted in a time step size of 6ms.

These simulation results show that the RRA algorithm and the specific simulation of it presented in this paper are accurate from a time and positional sense. The next step is to consider the accuracy of energy transport in the RRA API.

D. RRA ENERGY TRANSPORT ACCURACY

In a complex environment, especially with ocean bottoms that are not smooth when compared to the wavelength of the sonar pulse being used, it is a fair assumption that the interference patterns that one expects will be effected by strong interaction with the rough ocean surface and bottom. This interaction with somewhat random surfaces will in essence result in the sound waves becoming incoherent. This break down in coherency allows the pressures of individual waves to be superimposed on one another without regard to phase. This is not necessarily true if the wavelength of the sonar is comparable to the acoustic length of environmental objects. Since the design of this thesis was based on finding mines as the smallest objects, the wavelength of the sonar must be less than the size of a mine. Assuming that the size of a mine is one meter in diameter and the speed of sound in sea water is 1500 m/s, the minimum frequency to be able to resolve the mine is 1500 Hz. This and higher frequencies are typically used for this type of identification work. Most ocean structures tend to have acoustic lengths that are much greater than about one meter, but these larger structures tend to have smaller structures on them that can be near the size of the acoustic wavelength of the sonar. Therefore, with extensive acoustic interaction with the ocean bottom and its structure, it is a safe assumption to use incoherent summation of pressure waves. However, for the sake of comparison purposes, it is necessary to compare the energy transport of the RRA algorithm to that of another accepted accurate model.

To validate the RRA energy transport results, a highly simplified environment is used. This environment has a pressure release surface and a rigid bottom at 100 meters of water depth. Both surface and bottom are infinite in extent and sound speed in the water is 1500 m/s. Since the sound speed is constant, frequencies much lower than the typical lower limit (500 Hz, [Etter 1996]) produce accurate output in the RRA algorithm. Thus a frequency of 100 Hz was used as the frequency of operation for energy transport calculations. This low frequency was also used in order to speed up processing in the normal mode model, since it performs most efficiently at low frequencies. From the set up of this environment, it is seen that coherency must be adhered to in order to obtain accurate results. Thus for the time being, the real world parameters that result in incoherency are neglected and the ideal RRA model energy transport is compared to the normal mode model energy transport. The normal mode model was chosen since it is viewed as being very accurate in predicting energy transport for water columns of very simple geometry.

Figure 8.1 compares the energy transport of the RRA model with the energy transport of the normal modes model. Notice how closely the two data sets correlate to each other. Thus coherent energy transport of the RRA model is similar to energy transport in the normal modes model. Now if we assume incoherency and can agree that this assumption has been used for years with close correlation between simulation and reality, then the RRA output is accurate. That is to say as accurate as assuming incoherency can be. Figure 8.2 shows a logarithmic trendline of the RRA model using incoherent summation of the pressure waves. This trendline shows the expected behavior. It is now seen that not only is the RRA algorithm accurate in a temporal and spatial sense, it is also reasonably accurate in calculating the energy transport in a simple geometry and when incoherency can be assumed in a more complex geometry.

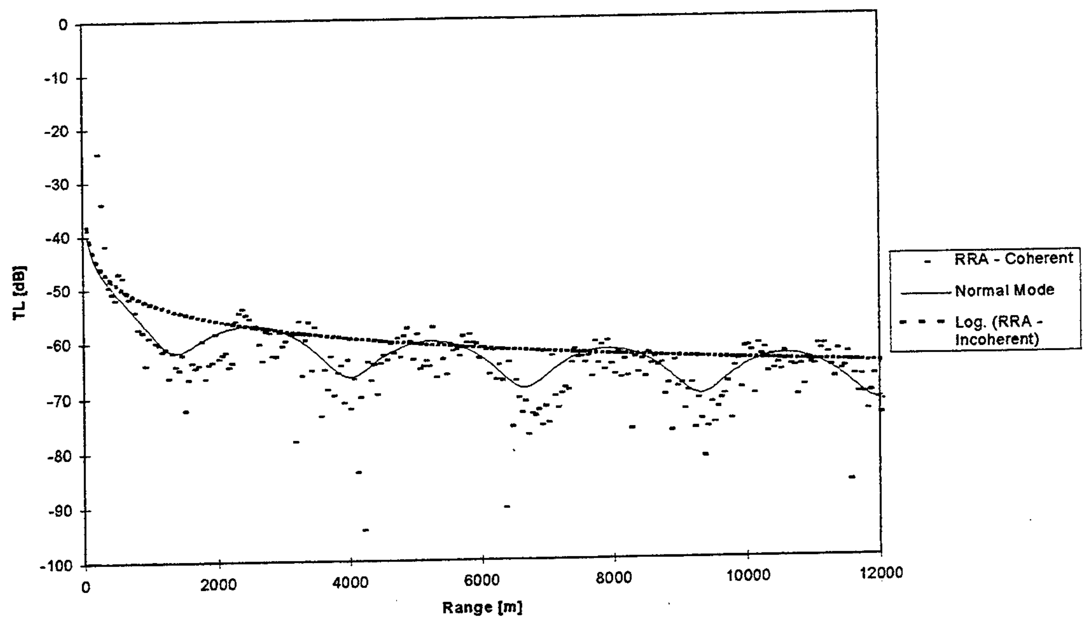


Figure 8.2 Transmission Loss of RRA model versus Normal Mode Model

E. DETECTION OF VIRTUAL OBJECTS

To complete the RRA API validation the simulation needs to be used to perform some good task other than showing itself to produce sound pressure fields comparable to other algorithms. It needs to be used in a task for which it has been designed. That task being, the searching of a complex minefield in order to ascertain whether or not the detection capabilities of the RRA model function as desired.

1. Mine Detection Scenario

In this scenario a single vehicle will traverse a shallow water area of 100 meters in depth. There are 50 mines placed in a 1 kilometer by 1 kilometer minefield. The mines are randomly placed in all three dimensions. The water column is characterized by a constant speed of sound, the surface is a smooth pressure release surface and the bottom is a smooth horizontal rigid surface. Additionally, any acoustic energy leaving the 1 kilometer square is considered lost. In this test scenario, the RRA-based simulated sonar system detected 49 of the 50 mines in the area. One expects that all mines are detected since the conditions are ideal and the only energy losses are when the sound leaves the 1 kilometer square area. Interestingly the reason that one of the mines was missed was due to the handling of reflections. Since a beam is defined as the area enclosed by four rays, when some of the rays encounter the surface sooner than others, there is a period of time that the beam is not traveling as it does in the real world; the wave front is not perpendicular to the directions of propagation of the four rays. The magnitude of this effect can be minimized by keeping the beam width and ping interval small which keeps the area of the advancing beam front small. Ideally this effect can be removed as the beam widths become arbitrarily small. This however introduces a massive time overhead and thus another compromise is made, this time between beam width and computational time. Improved algorithms and faster computers may someday make this small error disappear. A promising topic for future work is adaptively combining higher resolution, more sophisticated collision detection and smaller step size when in the vicinity of reflections. The code for this scenario (Pinger.java) is found in Appendix B.

F. VISUALIZATION RESULTS

The last two pages of this chapter contain four example sonar renderings. These renderings begin to show the potential of high-dimensional plots. All of the figures show the area swept out by a sonar beam with four degrees beam width in both the horizontal and vertical directions. The beam is directed at twenty degrees west of north with an eighty degree elevation from straight down. It is propagated for eight seconds into the sloped ocean terrain. Figure 8.3 shows the beam with transmission loss mapped to the color field of the VRML beam object. Figure 8.4 shows the same sonar beam with time mapped to color and transmission loss mapped to intensity. This mapping has the effect of drawing attention to only the low transmission loss portions of the beam, while giving a good feel for time and hence distance of the ping as it traverses the environment. Figure 8.5 shows presumed detectability and counter-detectability limits mapped to color and time mapped to intensity. This mapping has the effect of showing that the reflections coming back from the slope have little chance of counter detection by an enemy platform. Figure 8.6, which shows the same detectability limits, has its transmission loss mapped to intensity. This figure more clearly shows where the area of counter detection is by allowing only the most intense parts of the beam be seen in the later stages of propagation. Taken together, these example figures serve to show that with some ingenuity and several mapping parameters, much more information can be displayed in the same volume.

In conclusion, these simulation results were exactly those desired when the problem started out. One unexpected result was the large number of rays that need to be calculated to achieve accurate phase-coherent results. If incoherent summation can be assumed then the number of rays necessary drops considerably. However, with the continued increase in speed of general purpose computers and the advances in massively parallel networks, even coherent summation will be rapid.

G. CONCLUSIONS

The accuracy of the RRA API is shown to be nearly identical to the original RRA program as presented in Ziomek [1993]. In addition, the accuracy of energy propagation is compared to the normal mode model. This comparison proves that the RRA API is reasonably accurate in energy propagation. Detection of virtual world objects provided expected results with a two percent error rate for randomly placed mines. Visualization results were very good, but they barely scratch the surface of the abilities of real-time 3D sonar visualization. For information on obtaining a video of important simulation results from this thesis see Appendix G and for information on obtaining a CD ROM of the thesis contents and accompanying code see Appendix H.

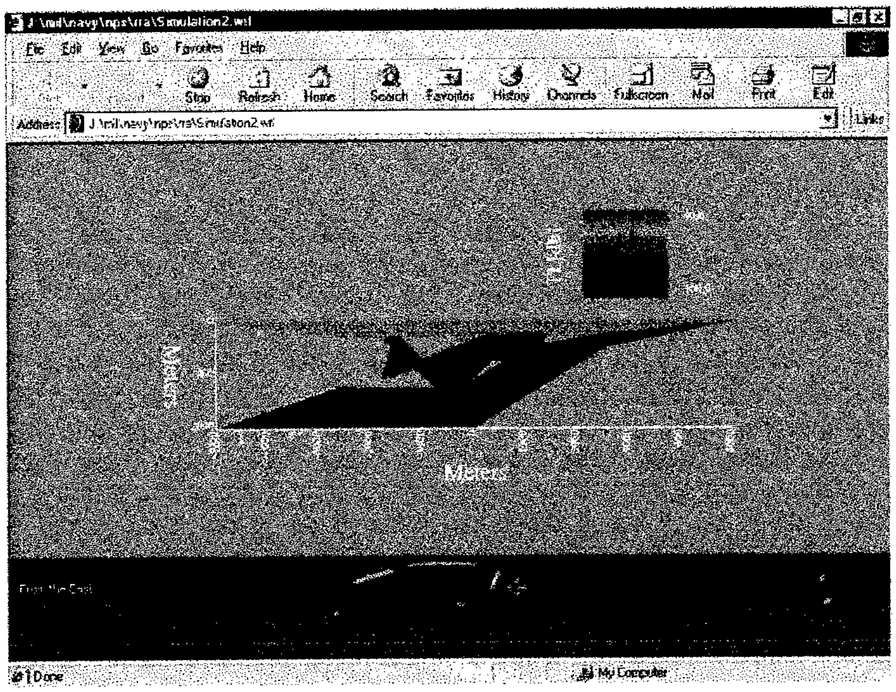


Figure 8. 3 Sonar beam showing transmission loss mapped to color

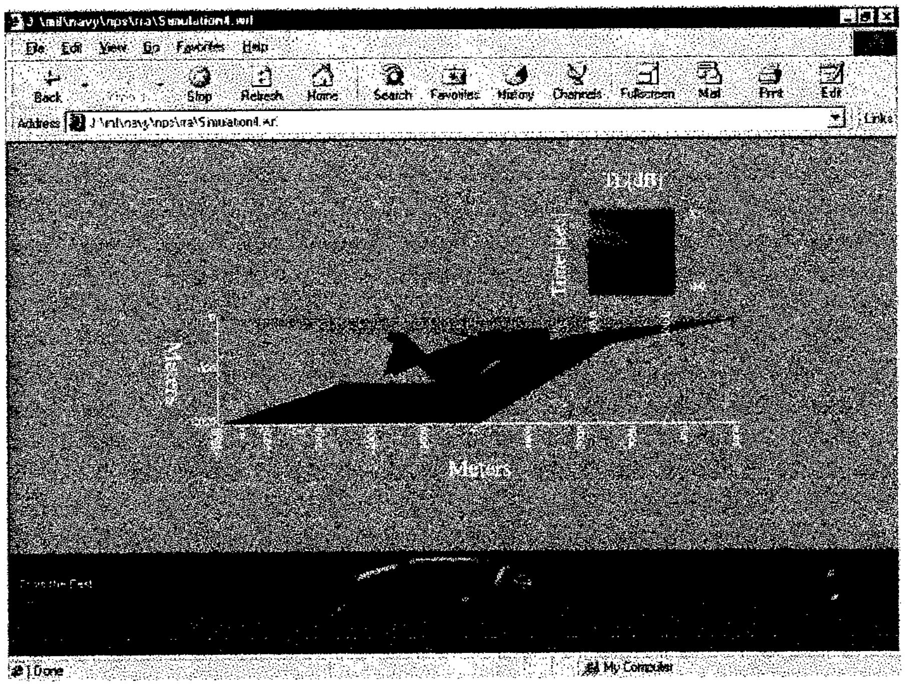


Figure 8. 4 Sonar beam showing time mapped to color and transmission loss to intensity

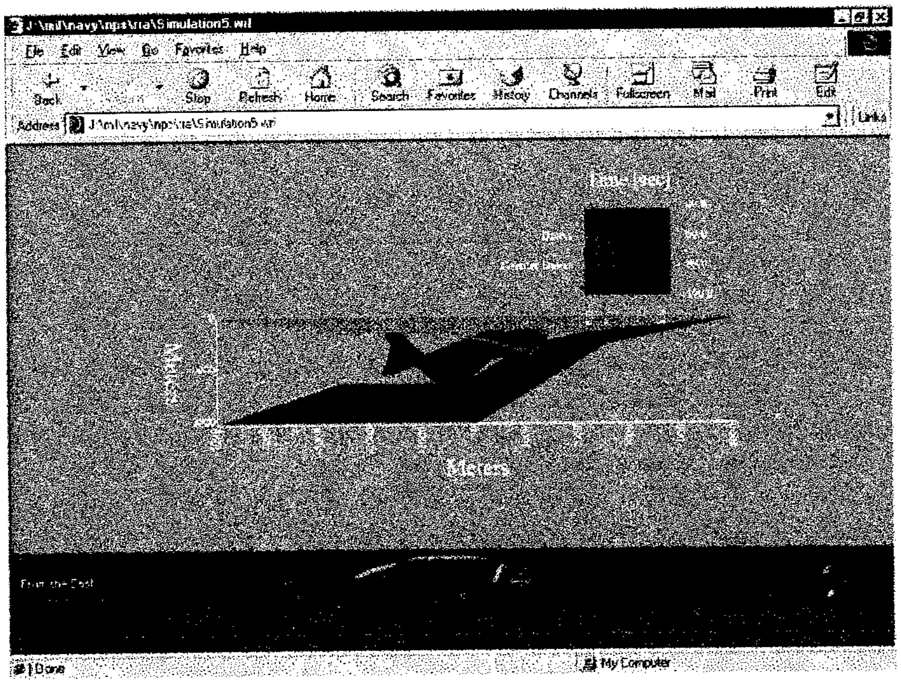

J:\ml\navy\nps\ra\Simulation5.wrl
File Edit View Go Favorites Help
Back Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print Edit
Address J:\ml\navy\nps\ra\Simulation5.wrl
Links
Time [sec]
Meters
Meters
Done
My Computer

Figure 8. 5 Sonar beam showing detection thresholds mapped to color and time to intensity

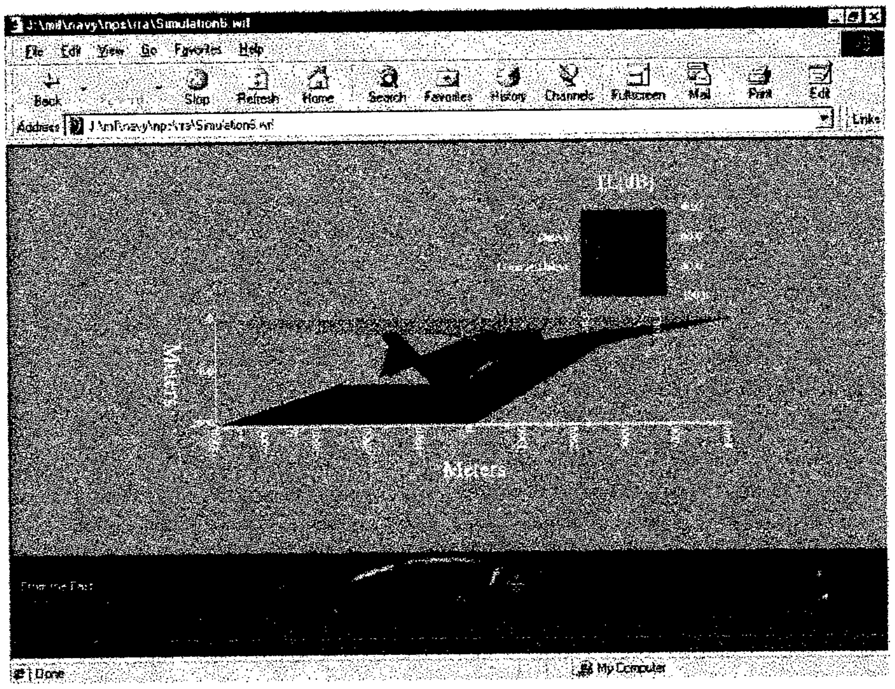

J:\ml\navy\nps\ra\Simulation6.wrl
File Edit View Go Favorites Help
Back Stop Refresh Home Search Favorites History Channels Fullscreen Mail Print Edit
Address J:\ml\navy\nps\ra\Simulation6.wrl
Links
TL [dB]
Meters
Meters
Done
My Computer

Figure 8. 6 Sonar beam showing detection thresholds mapped to color and transmission loss to intensity

IX. CONCLUSIONS AND RECOMMENDATIONS

A. PRINCIPAL THESIS CONCLUSIONS

This chapter gathers together the many conclusions that have been reached throughout the thesis. The principal conclusion of the entire thesis is that realistic real-time 3D sonar simulations and visualizations can be constructed with current computer hardware. Through the combination of strong network programming languages, Java and VRML, and the Internet, almost unlimited processing power can be supplied to the RRA sonar server. The sonar server coupled with a proxy server is highly scaleable. Adding more processors to a LAN can provide more sonar pulses per second or a higher degree of accuracy as needed. Widely available and inexpensive 3D graphics rendering software allow a deeper understanding of sonar data, as well as the ability to create immersive virtual world battle space. The advent of real-time 3D sonar visualization is a major breakthrough in sonar system design and employment.

B. SPECIFIC CONCLUSIONS

1. Tactical Simulation

Tactical simulations improve significantly when using more accurate sonar information. The goal of tactical simulations is that of accurate prediction of real-world tactics performance, for use in preliminary tactics evaluation or operator training. These performance evaluations are only as good as the simulation that they are running on. Likewise, the simulation is only as good as the components from which it is composed. Evaluating tactics with only geographic constraints in mind is limited in value when compared to a simulation that takes in to account geographic constraints and water column properties that affect sonar, the primary underwater sensor. Tactical simulation results are further enhanced by advanced information presentation paradigms. Advanced presentations allow for more intuitive and interactive, high-density information representations that lends more insight into the tactical performance.

2. Sonar Simulation

Ray theory solutions such as RRA can produce fast real-time sonar data information. These solutions can be characterized as three-dimensionally accurate in position, time and energy transport. The energy transport accuracy is verified for short-range shallow-water conditions to be nearly the same as is produced by normal mode theory. The same characteristics that make it ideal for accurate sonar information also create the right conditions for accurate detection information on targets in the virtual world. An especially important conclusion drawn about the RRA model is that it is highly scaleable. The number of processors on a given LAN can be increased without bound (as long as the LAN capacity is large enough to handle inter-process communications) and the sonar lobe can be divided into smaller and smaller pieces (limited only by the accuracy of double-precision floating-point numbers) providing for any desired accuracy of sonar information. Astonishingly, when proper care is taken, this can be done in real time (or faster than real time). Currently such capability is generally considered unfeasible on any computer, so this is an important result.

3. Visualization

In addition to tactical advantages 3D visualization of sonar information gives enhanced feel for space, time and intensity interactions, especially in relation to geographic features and variable sound speed profile (SSP). Given the high-dimension data space associated with acoustic signal processing, new and different forms of sonar visualization must be developed. This thesis lays the foundation for what advanced sonar visualization techniques might provide to tactician and researcher alike. With sonar users encountering larger and larger data sets, meaningful and highly condensed information need to be presented in a highly condensed and intuitive way, in order for the human brain to be able to process all of the data. These information sets are even more important to the tactician than the researcher in that properly designed information rendering can allow for quicker identification and correlation, leading to enhanced response time and thereby improving safety of ship.

C. RECOMMENDATIONS FOR FURTHER WORK

1. Tactical Simulation

With the enhanced simulation capabilities of the RRA sonar server API, the *Manta* UUV and *Phoenix* AUV tactical minefield searches ought to be upgraded to get better, more physically based information on tactics evaluations for autonomous minefield searches. Modification of the search area to incorporate realistic boundary conditions, geographic constraints and environmental sound speed profiles will lead to even more realistic tactical evaluations. Such development can lead to an optimal search pattern development *in situ*.

2. Sonar Simulation

The RRA sonar simulation API might benefit from a number of additions that can enhance realism of the model compared to real world processes. The following short paragraphs summarize the needed modifications, some explicitly stated in the body of the thesis and some only implied.

A more complex bottom needs to be modeled that allows for arbitrary complexity in the physical constraints imposed by the bottom. Along with this increased complexity several choices for bottom composition need to be included that model more complex structures such as shale, clay, sand, and gravel. Ideally these structures can be built into a VRML 97 *ElevationGrid* node with additional fields added to indicate bottom composition at the various grid points. The effect of surface reverberation needs to be added to induce the all-too-present clutter in the sonar information that is generated as the acoustic energy interacts with the bottom.

A more complex surface model needs to be created that includes the disruption in the surface induced by wave action and micro-bubble formation. These effects have great influence on reverberation from the surface of the ocean.

The current SSP model incorporates a spatially varying model that must be specified at compilation time. This is a severe limitation. Ideally, a model that can access a historical environment database of values from real-world sampling and merge it in a meaningful way with a stream of recent sampling updates will provide a more realistic world, one that reflects the most up to date SSP information. This in and of itself forms the basis of an entire thesis. To this enhanced model other effects need to be added, volume reverberation and noise, which further the realism of the API. The extreme dependence of sonar on SSP in every respect makes such work a high priority.

Improvement to the target model offers enough of a challenge so as to warrant additional thesis work. The current model assumes the targets are spherical and that more complex targets (e.g. submarines) can be constructed of multiple adjacent spheres. This model has severe limitations when objects much larger or much smaller than the wavelength of the acoustic signal are encountered. Advancements in the processing speed of collision detection of the ping-to-target interaction needs to be made before even larger numbers of objects might co-exist in the virtual world.

A final improvement might be to offer different propagation models in the RRA sonar API. This can make it a more general sonar API for researchers and extend the range to which the API target information is valid. Probable candidates include the finite element and split-step Fourier parabolic equation models. Both models require stepping out in range which can be converted to the time-space domain and can fit into the basic structure of the current RRA sonar API. Nevertheless, it remains important to note that RRA is very general and accurate, so integration of additional sonar models is not a prerequisite for widespread use. The current RRA API appears ready to serve as a computational engine to model most (and perhaps all) sonars currently employed by naval ships.

3. Visualization

The biggest, most important and most exciting task recommended for continued work is the exploration of new and more intuitive 3D sonar data representations. A detailed study of current sonar data representations needs to be joined together with high-dimensional space plotting concepts that utilize all the capabilities of current 3D rendering hardware and software. A study of this type may prove to be of vital interest to the Navy and military in general. Another possibility is to develop a 3D target motion analysis system. This system takes the processed sonar data from a sonar system and through combined use of the RRA sonar API and knowledge of the sonar environment, statistically calculates the position of targets of interest. A system of this sort will have tremendous operational value.

The final improvement to the RRA sonar API will be to separate the visualization routines from the RRA sonar API and to move them to a Visualization API. Many techniques which might be applied to a Visualization API are found in a VRML 97 generation program, [Roehl et al., 1997] Chapter 21. It is expected that sonar operators and researchers alike will take great advantage of the power, extensibility and ubiquity of VRML-based 3D graphics to better visualize and understand 3D sonar environments.

APPENDIX A. RRA CODE

A. BEAM.JAVA

```
/*
 File:          Beam.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;
import java.lang.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Beam.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Beam.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Forms a beam from four rays, propagates the beam and can return
 *  a VRML representation of the beam.
 *
 *<dt><b>Explanation:</b>
 *<dd>Beams of energy can be considered to consist of bundles of rays.  The
 *  energy in a bundle can be shown not to diverge from the bundle.  Thus the energy
 *  in a bundle is constant and thus the product of the intensity and the
 *  area of the bundle perpendicular to direction of propagation is a
 *  constant as well.  Therefore a beam is a fundemental building block for
 *  a lobe of a sonar pattern.<P>
 *
 *<dt><b>History:</b>
 *<dd>          15Nov97 /Timothy M. Holliday       /New
 *<dd>          17Mar98 /Timothy M. Holliday       /Added HTML comment convention
 *<dd>          12Apr98 /Timothy M. Holliday       /Parameterless Constructors
 *<dd>          14Apr98 /Timothy M. Holliday       /Simplified VRML Routines
 *<dd>          21Apr98 /Timothy M. Holliday       /Fixed Problem with calculateSoundPressureLevel
 *
 *@see Ray
 *@see BeamExampleStatic
 *@see BeamExampleDynamic
 *@see Bottom
 *@see Surface
 *@see Vec3d
 */
public class Beam {
```

```
public static final int T_L = 0;
public static final int TIME = 1;
public static final int NONE = 2;

/**
* Constructor for the Beam class.
*
* A beam is defined as the volume swept out by
* four rays as they traverse the ocean environment
*/
public Beam() {
 /*
 * Initialize member variables that require it.
 */
 position = new Vec3d();

 ray1 = new Ray();
 ray2 = new Ray();
 ray3 = new Ray();
 ray4 = new Ray();

 segment1 = new Vec3d();
 segment2 = new Vec3d();
 segment3 = new Vec3d();
 segment4 = new Vec3d();

 TL =   new double[MAX_POINTS];
 detectTime = new double[MAX_POINTS];
 detectEL = new double[MAX_POINTS];
}

/**
* This method resets all of the beam parameters after
* instanciation has occurred since reuse is more time
* efficient than garbage collection and reallocation.
*/
public void reset() {

 ray1.setPosition(position.get(0),position.get(1),position.get(2));
 ray1.setElevation(elevation+halfBeamWidthY);
 ray1.setAzimuth(azimuth+halfBeamWidthX);
 ray1.setDeltaTime(deltaTime);
 ray1.setDuration(pulseDuration);
 ray1.setBottom(bottom);
 ray1.setSurface(surface);
 ray1.setSsp(ssp);
 ray1.reset();
 ray2.setPosition(position.get(0),position.get(1),position.get(2));
 ray2.setElevation(elevation-halfBeamWidthY);
 ray2.setAzimuth(azimuth+halfBeamWidthX);
 ray2.setDeltaTime(deltaTime);
 ray2.setDuration(pulseDuration);
 ray2.setBottom(bottom);
 ray2.setSurface(surface);
 ray2.setSsp(ssp);
 ray2.reset();
 ray3.setPosition(position.get(0),position.get(1),position.get(2));
 ray3.setElevation(elevation-halfBeamWidthY);
 ray3.setAzimuth(azimuth-halfBeamWidthX);
 ray3.setDeltaTime(deltaTime);
 ray3.setDuration(pulseDuration);
 ray3.setBottom(bottom);
 ray3.setSurface(surface);
```

```
  ray3.setSsp(ssp);
  ray3.reset();
  ray4.setPosition(position.get(0),position.get(1),position.get(2));
  ray4.setElevation(elevation+halfBeamWidthY);
  ray4.setAzimuth(azimuth-halfBeamWidthX);
  ray4.setDeltaTime(deltaTime);
  ray4.setDuration(pulseDuration);
  ray4.setBottom(bottom);
  ray4.setSurface(surface);
  ray4.setSsp(ssp);
  ray4.reset();

  //      initialArea = area 1 meter from source
  //           = 1 meter * solid angle (in steradians)
  //           = 1meter*(2*halfBeamWidthX*pi/180)*(2*halfBeamWidthY*pi/180)
  //           = 1.21846972E-003*halfBeamWidthX*halfBeamWidthY
  initialArea = 1.21846972e-3*halfBeamWidthX*halfBeamWidthY;
  count = -1;
  detectCount = 0;
}

/**
* This method sets the azimuthal angle, which is the angle from
* the x-axis to the z-axis rotating about the y-axis.
*/
public void setAzimuth(double phi) {
  azimuth = phi;
}

/**
* This method returns the azimuthal angle.
*/
public double getAzimuth() {
  return azimuth;
}

/**
* This method sets the half beam width of the beam in the
* azimuthal direction.
*/
public void setHalfBeamWidthX(double pHalfBeamWidthX) {
  halfBeamWidthX = pHalfBeamWidthX;
}

/**
* This method returns the half beam width of the beam in the
* azimuthal direction.
*/
public double getHalfBeamWidthX() {
  return halfBeamWidthX;
}

/**
* This method sets the elevation angle, which is the angle from the
* y-axis to the x-axis rotating about the z-axis .
*/
public void setElevation(double beta) {
  elevation = beta;
}

/**
* This method returns the elevation angle.
*/
```

```
public double getElevation() {
 return elevation;
}

/**
* This method sets the half beam width of the beam in the
* elevation direction.
*/
public void setHalfBeamWidthY(double pHalfBeamWidthY) {
 halfBeamWidthY = pHalfBeamWidthY;
}

/**
* This method returns the half beam width of the beam in the
* azimuthal direction.
*/
public double getHalfBeamWidthY() {
 return halfBeamWidthY;
}

/**
* This method sets the position of the beam.
*/
public void setPosition(double x, double y, double z) {
 position.set(x,y,z);
}

/**
* This method returns the position of the beam.
*/
public Vec3d getPosition() {
 return position;
}

/**
* This method sets the time step in the simulation.
*/
public void setDeltaTime(double pDeltaTime) {
 deltaTime = pDeltaTime;
}

/**
* This method returns the simulation time step.
*/
public double getDeltaTime() {
 return deltaTime;
}

/**
* This method sets the simulation end time.
*
* This value is reletive to the start time which is 0.0.
*/
public void setEndTime(double pEndTime) {
 endTime = pEndTime;
}

/**
* This method returns the simulation end time.
*/
public double getEndTime() {
 return endTime;
}
```

```
/**
* This method sets the duration of the sonar pulse.
*
* The integer refers to the number of deltaTime increments.
* currently only 1 or 2 are allowed.
*/
public void setDuration(int duration) {
 pulseDuration = duration;
}

/**
* This method sets the duration of the sonar pulse.
*
* The integer refers to the number of deltaTime increments.
*/
public int getDuration() {
 return pulseDuration;
}

/**
* This method sets the handle to the bottom object.
*/
public void setBottom(Bottom pBottom) {
 bottom = pBottom;
}

/**
* This method returns the handle to the bottom object.
*/
public Bottom getBottom() {
 return bottom;
}

/**
* This method sets the handle to the surface object.
*/
public void setSurface(Surface pSurface) {
 surface = pSurface;
}

/**
* This method returns the handle to the surface object.
*/
public Surface getSurface() {
 return surface;
}

/**
* This method sets the handle to the sound speed profile object.
*/
public void setSsp(SSP pSsp) {
 ssp = pSsp;
}

/**
* This method returns the handle to the sound speed profile object.
*/
public SSP getSsp() {
 return ssp;
}

/**
```

```
* This method sets BeamNumber. It is used to uniquely
* identify each beam for ROUTEing in VRML.
*/
public void setBeamNumber(int number) {
 beamNum = number;
}

/**
* This returns BeamNumber. Which is used to uniquely
* identify each beam for ROUTEing in VRML.
*/
public int getBeamNumber() {
 return beamNum;
}

/**
* This method writes to the console a VRML shape that is
* the three dimensional representation of the beam that is
* propagated.
*/
public String staticVRML(int colorChoice, int intensityChoice) {
 String temp = "";

 // Prepend the header information
 temp += "Transform {" + appendage ;
 temp += "  rotation 1 0 0 3.14" + appendage ;
 temp += "  children" + appendage ;

 temp += "Shape {" + appendage ;
 temp += "  geometry IndexedFaceSet {" + appendage;
 temp += "    coord Coordinate {" + appendage;
 temp += "      point [ " + appendage;
 j = 0;

 // Print the positions stored in each ray
 for (j=0; j<ray1.getCount(); j += 1) {
  count += 4;
  temp += ray1.position(j) + appendage;
  temp += ray2.position(j) + appendage;
  temp += ray3.position(j) + appendage;
  temp += ray4.position(j) + appendage;
 }

 temp += "      ]" + appendage;
 temp += "    }" + appendage;
 temp += "    coordIndex [" + appendage;

 // Define the faces of the beam
 for(j=3; j<count; j += 4) {
  temp += (j-3) + " " + j + " " + (j+4) + " " + (j+1) + " -1" + appendage ;
  temp += (j-3) + " " + (j+1) + " " + (j+2) + " " + (j-2) + " -1" + appendage ;
  temp += (j-2) + " " + (j+2) + " " + (j+3) + " " + (j-1) + " -1" + appendage ;
  temp += (j-1) + " " + (j+3) + " " + (j+4) + " " + j + " -1" + appendage ;
 }
 temp += "    ]" + appendage;

 temp += "    color Color{" + appendage;
 temp += "      color[" + appendage;

 // Specify a color for each group of ray positions for each time step
 // that corresponds to the Transmission Loss in dB.
 for (j=0; j<ray1.getCount(); j += 1) {
  switch (colorChoice) {
    case T_L:   PrintVRML.setColorValue(TL[j]); break;
    case TIME:  PrintVRML.setColorValue(ray1.getTime(j)); break;
    case NONE:  PrintVRML.setColorValue(1.0); break;
   }
   switch (intensityChoice) {
    case T_L:   PrintVRML.setIntensityValue(TL[j]); break;
    case TIME:  PrintVRML.setIntensityValue(ray1.getTime(j)); break;
    case NONE:  PrintVRML.setIntensityValue(1.0); break;
   }

   temp += PrintVRML.getColor();
  }
  temp += "      ]" + appendage;
  temp += "    }" + appendage;
  temp += "    colorIndex[" + appendage;

  // Specify the vertices and their colors for each face specifed above.
  for(j=1; j<ray1.getCount(); j++) {
   temp +=  (j-1) + " " + (j-1) + " " + (j) + " " + (j) + " -1" + appendage ;
   temp += (j-1) + " " + (j) + " " + (j) + " " + (j-1) + " -1" + appendage ;
   temp += (j-1) + " " + (j) + " " + (j) + " " + (j-1) + " -1" + appendage ;
   temp += (j-1) + " " + (j) + " " + (j) + " " + (j-1) + " -1" + appendage ;
  }

  temp += "    ]" + appendage;
  temp += "    solid FALSE" + appendage;
  temp += "    colorPerVertex TRUE" + appendage;
  temp += "  }" + appendage;
  temp += "}" + appendage;
  temp += "}" + appendage;
  return temp;
 }

 /**
 * This method creates a dynamic VRML string shape that is
 * the three dimensional representation of the beam pulse that is
 * propogated.
 */
 public String dynamicVRML() {
  String beamString;

  beamString = "Transform {" + appendage +
          " rotation 1 0 0 3.14" + appendage +
          " children" + appendage +
          "Shape {" + appendage +
          " appearance Appearance {" + appendage +
          "   material DEF PingColor"+beamNum+" Material {" + appendage +
          "     emissiveColor 1.0 0.0 0.0" + appendage +
          "     diffuseColor 0 0 0" + appendage +
          "     shininess 0" + appendage +
          "     ambientIntensity 0.2" + appendage +
          "     transparency .5" + appendage +
          "   }" + appendage +
          " }" + appendage +
          " geometry IndexedFaceSet {" + appendage +
          "   coord DEF Ping"+beamNum+" Coordinate {" + appendage +
          "     point [ " + appendage;
  j = 0;


 // Print the first position stored in each ray and make a box corresponding
 // to the pulse duration by printing the trailing edge of the pulse
  beamString = beamString + ray1.position(j) + " ";
```

```
beamString = beamString + ray2.position(j) + " ";
beamString = beamString + ray3.position(j) + " ";
beamString = beamString + ray4.position(j) + appendage;
beamString = beamString + ray1.trailingPosition(j) + " ";
beamString = beamString + ray2.trailingPosition(j) + " ";
beamString = beamString + ray3.trailingPosition(j) + " ";
beamString = beamString + ray4.trailingPosition(j) + appendage;

beamString = beamString + "      ]" + appendage +
                    "    }" + appendage +
                    "    coordIndex [" + appendage +

// Define the faces of the pulse
                    "0 3 7 4 -1" + appendage +
                    "0 4 5 1 -1" + appendage +
                    "1 5 6 2 -1" + appendage +
                    "2 5 7 3 -1" + appendage +
                    "0 1 2 3 -1" + appendage +
                    "4 5 6 7 -1" + appendage +
                    "    ]" + appendage +

                    "    solid FALSE" + appendage +
                    "  }" + appendage +
                    "}" + appendage +
                    "}" + appendage +

                    "DEF Propagation"+beamNum+" CoordinateInterpolator {" + appendage +
                    "  key[" + appendage;

 for(j=0; j<ray1.getCount(); j++) {
  beamString = beamString + ray1.normalizedTime(j,endTime*2) + appendage;
 }
 beamString = beamString + "  0.51 1.0 ]" + appendage +
                     "  keyValue[" + appendage;
 for(j=0; j<ray1.getCount(); j++) {
  beamString = beamString + ray1.position(j) + " ";
  beamString = beamString + ray2.position(j) + " ";
  beamString = beamString + ray3.position(j) + " ";
  beamString = beamString + ray4.position(j);
  beamString = beamString + ray1.trailingPosition(j) + " ";
  beamString = beamString + ray2.trailingPosition(j) + " ";
  beamString = beamString + ray3.trailingPosition(j) + " ";
  beamString = beamString + ray4.trailingPosition(j);

 }
 j--;
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
 beamString = beamString + ray1.position(j) + " ";
```

```
    beamString = beamString + "  ]" + appendage +
                          "}" + appendage +

                          "DEF TLColor"+beamNum+" ColorInterpolator {" + appendage +
                          "  key[" + appendage;

    for(j=0; j<ray1.getCount(); j++) {
      beamString = beamString + ray1.normalizedTime(j,endTime*2);
    }
    beamString = beamString + "  0.51 1.0 ]" + appendage +
                          "  keyValue[" + appendage;

    // Specify a color for each group of ray positions for each time step
    // that corresponds to the Transmission Loss in dB.
    for (j=0; j<ray1.getCount(); j += 1) {
      if (TL[j] < 0 ) {
        beamString = beamString + "1.0 0 0" + appendage;
      }
      else {
        beamString = beamString + (1.0-TL[j]*(.8/125)) + " 0 0" + appendage;
      }
    }

    beamString = beamString + "  0 0 0 0 0 0 ]" + appendage +
                          "}" + appendage +
                          "ROUTE PingInterval.fraction_changed TO Propagation"+beamNum+".set_fraction" + appendage
+
                          "ROUTE Propagation"+beamNum+".value_changed TO Ping"+beamNum+".set_point" +
appendage +
                          "ROUTE PingInterval.fraction_changed TO TLColor"+beamNum+".set_fraction" + appendage +
                          "ROUTE TLColor"+beamNum+".value_changed TO
PingColor"+beamNum+".set_emissiveColor" + appendage;
    return beamString;
  }

  /**
  * This method calculates the trajectory of the beam of energy
  * enclosed by the defining rays of the beam tube. It also determines
  * when there is a detection.
  */
  public void calculateBeam(Targets targets) {

    durationCount = 0;
    j = 0;

    // Record first point always
    ray1.recordPoint();
    ray2.recordPoint();
    ray3.recordPoint();
    ray4.recordPoint();
    TL[j] = calculateTL();
    j++;

    while (ray1.getTime() < endTime) {

      // Propagate all rays through one time step
      ray1.Propagate(.006);
      ray2.Propagate(.006);
      ray3.Propagate(.006);
      ray4.Propagate(.006);

      // detect and record target information
      if (targets.isCollision(ray1,ray2,ray3,ray4)) {
```

```
    detectTime[detectCount] = ray1.getTime();
    detectEL[detectCount] = -2*calculateTL()+10*Math.log(targets.getCollisionArea()/4/3.1415);
  }

  durationCount++;

  // If the trailing edge has left the source record all points
  if (durationCount == pulseDuration) {
    ray1.recordPoint();
    ray2.recordPoint();
    ray3.recordPoint();
    ray4.recordPoint();
    TL[j] = calculateTL();
    j++;
  }

  //   If any ray has reflected then record all of the
  //   points in the beam
  if ( ray1.reflected() ||
       ray2.reflected() ||
       ray3.reflected() ||
       ray4.reflected()) {
     ray1.recordPoint();
     ray2.recordPoint();
     ray3.recordPoint();
     ray4.recordPoint();
     TL[j] = calculateTL();
     j++;
  }

  //   If the curvature sum reaches the limit, record
  //   all of the points in the beam
  if ( ray1.totalCurvature() > CURVATURE_LIMIT ||
       ray2.totalCurvature() > CURVATURE_LIMIT ||
       ray3.totalCurvature() > CURVATURE_LIMIT ||
       ray4.totalCurvature() > CURVATURE_LIMIT) {
     ray1.recordPoint();
     ray2.recordPoint();
     ray3.recordPoint();
     ray4.recordPoint();
     TL[j] = calculateTL();
     j++;
   }
  }

  // Record last point always
  ray1.recordPoint();
  ray2.recordPoint();
  ray3.recordPoint();
  ray4.recordPoint();
  TL[j] = calculateTL();

  targets.resetTargets();
}

/**
* This method calculates the trajectory of the beam of energy
* enclosed by the defining rays of the beam tube.
*/
public void calculateSoundPressureLevel(double field[][][],
                                double deltaRange,
                                double deltaDepth,
                                double frequency) {
```

```
double averageX, averageY, averageReflectionPhase, deltaTime, deltaPath;
double nearestGridX,nearestGridY;
Vec3d  averageNormal = new Vec3d();
int count;

if (deltaRange>deltaDepth) {
  deltaTime = deltaDepth/2/1500;
  deltaPath = deltaDepth/2;
}
else {
  deltaTime = deltaRange/2/1500;
  deltaPath = deltaRange/2;
}

averageNormal.set(ray1.getNormal());
averageNormal.add(ray2.getNormal());
averageNormal.add(ray3.getNormal());
averageNormal.add(ray4.getNormal());
averageNormal.scale(.25);

if (averageNormal.get(1) == 0.0) {
  count = (int)(deltaRange/deltaPath);
}
else if (Math.abs(averageNormal.get(0)/averageNormal.get(1))>deltaRange/deltaDepth) {
  count = Math.abs((int)(deltaRange/deltaPath/averageNormal.get(0)));
}
else {
  count = Math.abs((int)(deltaDepth/deltaPath/averageNormal.get(1)));
}

// Propagate a constant x distance for all rays
while (ray1.getPositionX() < endTime*1500) {

deltaTime = 0.002;

  //   Propagate all rays through the proper number of time steps

  for (j=0;j<1;j++) {
    ray1.Propagate(deltaTime);
    ray2.Propagate(deltaTime);
    ray3.Propagate(deltaTime);
    ray4.Propagate(deltaTime); .
  }

  averageX = (ray1.getPositionX()+ray2.getPositionX()+ray3.getPositionX()+ray4.getPositionX())/4;
  averageY = (ray1.getPositionY()+ray2.getPositionY()+ray3.getPositionY()+ray4.getPositionY())/4;
  segment1.set(averageX,averageY,0);
  nearestGridX = (double)Math.round(averageX/deltaRange)*deltaRange;
  nearestGridY = (double)Math.round(averageY/deltaDepth)*deltaDepth;
  segment2.set(nearestGridX,nearestGridY,0);

  if (inTheBeam(nearestGridX,nearestGridY,0)) {
    averageNormal.set(ray1.getNormal());
    averageNormal.add(ray2.getNormal());
    averageNormal.add(ray3.getNormal());
    averageNormal.add(ray4.getNormal());
    averageNormal.scale(.25);

    // calculate distance from wavefront to storage point for phase correction
    segment2.sub(segment1);
    a = segment2.dot(averageNormal);
```

```
      // calculate phase correction
      averageReflectionPhase =
(ray1.getReflectionPhase()+ray2.getReflectionPhase()+ray3.getReflectionPhase()+ray4.getReflectionPhase())/4;
      averageReflectionPhase += 2*Math.PI*frequency*(ray1.getTime()+a/ssp.C(ray1.getPosition()));

      // calculate the cos and sin of the phase
      b = Math.cos(averageReflectionPhase);
      c = Math.sin(averageReflectionPhase);

      // calculate the magnitude of the pressure
      d = Math.sqrt(initialArea/calculateArea());

      // calculate the x component of the pressure field
      field[(int)Math.round(averageX/deltaRange)][(int)Math.round(averageY/deltaDepth)][0] += d*b;
      field[(int)Math.round(averageX/deltaRange)][(int)Math.round(averageY/deltaDepth)][1] += d*c;

      }
    }
  }

  /**
  * This method returns the total number of detects by the beam
  */
  public int getDetectCount() {
    return detectCount;
  }

  /**
  * This method returns the detect time for a given detect in
  * the beam
  */
  public double getDetectTime (int N) {
    return detectTime[N];
  }

  /**
  * This method returns the echo level of the detected target
  */
  public double getDetectEchoLevel (int N) {
    return detectEL[N];
  }

  /**
  * This method returns a VRML timer string that contains
  * the appropriate timing information for the beam
  */
  public String pingTimerVRML() {
    return "DEF PingInterval TimeSensor{" + appendage +
        " cycleInterval "+(endTime*2) + appendage +
        " loop TRUE" + appendage +
        "}" + appendage;
  }

  /**
  * This is a static method used to indicate whether a line feed is desired
  * at the end of every line.
  *
  * 'true' indicates a linefeed is desired and 'false' indicates that a space
  * is desired"
  */
  public static void setAppendLineFeed(boolean pAppendLineFeed) {
    appendLineFeed = pAppendLineFeed;
    Ray.setAppendLineFeed(appendLineFeed);
  if (appendLineFeed) {
   appendage = LINE_FEED;
  }
  else {
   appendage = SPACE;
  }
 }

 /**
 * This is a static method that returns the current line appendage.
 */
 public static boolean getAppendLineFeed(){
  return appendLineFeed;
 }

/**************************************************************
**          Private Section
**************************************************************/
 private static final String LINE_FEED = "\n";
 private static final String SPACE = " ";
 private static boolean appendLineFeed = true;
 private static String appendage = LINE_FEED;

 /**
 * This method determines if a grid point is in the beam in the
 * current step. This is used in calculateSoundPressureLevel.
 * If the given point is in the beam then true is returned.
 */
 private boolean inTheBeam(double x, double y, double z) {
  Vec3d gridPoint = new Vec3d(x,y,z);
  return inBetween(ray1.getPosition(),gridPoint,ray3.getTrailingPosition())&&
       inBetween(ray2.getPosition(),gridPoint,ray4.getTrailingPosition());
 }

 /**
 * This method returns a boolean indicating whether or not the grid
 * point in question is in the sonar pulse.
 */
 private boolean inBetween(Vec3d a, Vec3d b, Vec3d c) {
  if ( ((a.get(0) > b.get(0) && b.get(0) > c.get(0)) ||(c.get(0) > b.get(0) && b.get(0) > a.get(0)))&&
     ((a.get(1) > b.get(1) && b.get(1) > c.get(1)) ||(c.get(1) > b.get(1) && b.get(1) > a.get(1)))&&
     ((a.get(2) > b.get(2) && b.get(2) > c.get(2)) ||(c.get(2) > b.get(2) && b.get(2) > a.get(2)))) {
   return true;
  }
  else {
   return false;
  }
 }

 /**
 * This method calculates the transmission loss at the current point.
 *
 * The algorithm uses the fact that
 *     initialArea x initialIntensity = finalArea x finalIntensity
 *     is true for a beam tube.
 */
 private double calculateTL(){
  double Attenuation = 0;
  double Area;

  Attenuation = (ray1.getAbsorption() + ray2.getAbsorption() +
          ray3.getAbsorption() + ray4.getAbsorption())/4;
```

```
// Calculate area of half of quadralateral
segment1.sub(ray1.getPosition(),ray2.getPosition());
segment2.sub(ray2.getPosition(),ray3.getPosition());
segment1.cross(segment2);
Area = .5*segment1.length();

// Calculate area of other half of quadralateral
segment1.sub(ray4.getPosition(),ray1.getPosition());
segment2.sub(ray3.getPosition(),ray4.getPosition());
segment1.cross(segment2);
Area += .5*segment1.length();

if (Area < .01){
  Area = .01;
}

Attenuation += 10*Math.log(Area/initialArea)/Math.log(10);

return Attenuation;
}

/**
* This method calculates the area of the beam front at the current time.
* It is used exclusively by calculateSoundPressureLevel.
*/
private double calculateArea(){
 double Area;

 // Calculate area of half of quadralateral
 segment1.sub(ray1.getPosition(),ray2.getPosition());
 segment2.sub(ray2.getPosition(),ray3.getPosition());
 segment1.cross(segment2);
 Area = .5*segment1.length();

 // Calculate area of other half of quadralateral
 segment1.sub(ray4.getPosition(),ray1.getPosition());
 segment2.sub(ray3.getPosition(),ray4.getPosition());
 segment1.cross(segment2);
 Area += .5*segment1.length();

 if (Area < .000001){
   Area = .000001;
 }
 return Area;
}

/*
* Curvature of beam that is allowed to elapse before a
* point is recorded. This is for rendering purposes.
*/
private double CURVATURE_LIMIT =.004;

/*
* The position of the ray
*/
private Vec3d position = null;

/*
* The azimuth of the beam
*/
private double azimuth = 0.0;

/*
```

```
* The elevation of the beam
*/
private double elevation = 90.0;

/*
* The azimuthal beam half width
*/
private double halfBeamWidthX = 1.0;

/*
* The elevation beam half width
*/
private double halfBeamWidthY = 1.0;

/**
* Number of iterations in length that the ray
* segment is
*/
private int   pulseDuration = 1;

/*
* Handle to the bottom object
*/
private Bottom bottom = null;

/*
* Handle to the surface object
*/
private Surface surface = null;

/*
* Handle to the sound speed profile object
*/
private SSP ssp = null;

/*
*The simulation time step
*/
private double deltaTime = 0.006;

/**
* Ray one for the current beam
*/
private Ray   ray1 = null;

/**
* Ray two for the current beam
*/
private Ray   ray2 = null;

/**
* Ray three for the current beam
*/
private Ray   ray3 = null;

/**
* Ray four for the current beam
*/
private Ray   ray4 = null;

/**
* Segment one of the wavefront formed by
* the four rays
```

```
*/
private Vec3d segment1 = null;

/**
* Segment two of the wavefront formed by
* the four rays
*/
private Vec3d segment2 = null;

/**
* Segment three of the wavefront formed by
* the four rays
*/
private Vec3d segment3 = null;

/**
* Segment four of the wavefront formed by
* the four rays
*/
private Vec3d segment4 = null;

/**
* Number of the beam
*/
private int   beamNum;

/**
* Area of the wavefront one meter from the source
*/
private double initialArea = 0;   // area 1m from array in sq. meters

/**
* Number of points of information that can be saved along
* the beam path
*/
private static final int MAX_POINTS = 200;

/**
* Array for storing the transmission loss along the beam path
*/
private double[] TL =   null;

/**
* Array for storing time that a sonar detect occurs
*/
private double[] detectTime = null;

/**
* Array for storing the echo level of a sonar detect
*/
private double[] detectEL = null;

/**
* Number of sonar detects that occur
*/
private int detectCount = 0;

/**
* The pulse duration in quanta of deltaTime
*/
private int durationCount;

/**
```

```
 * End time of the simulation
 */
private double endTime = 1.0;

 /**
 * Local counters
 */
private int count = -1;
private int j = 0;

 /**
 * Local doubles
 */
private double a,b,c,d;

}
```

B. BOTTOM.JAVA

```
/*
 File:           Bottom.java
 Compiler:       jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Bottom.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Bottom.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>This class acoustically simulates characteristics of
 *  the ocean bottom.  Currently only a rigid bottom is modeled.
 *
 *<dt><b>Explanation:</b>
 *<dd>Since a ray is the normal vector to a plane wave, rays interact with
 *  surfaces the same way that plane waves do.  Rays obey Snell's law and
 *  the equations of reflection and transmission of wave energy.  This model
 *  uses vector algebra to determine when the end of a ray has passed through
 *  the bottom and to reflect any rays that have penetrated.<P>
 *
 *<dt><b>History:</b>
 *<dd>          8Nov97  /Timothy M. Holliday       /New
 *<dd>          18Mar98 /Timothy M. Holliday       /Added HTML comment convention
 *
 *@see Surface
 *@see Vec3d
 */
public class Bottom {

  /**
  * Constructor for the bottom class.  The first argument is the bottom type,
  * which has two choices, 'noslope' and 'slope'.  The first choice is a flat
  * horizontal bottom and the second is a flat horizontal bottom with a steeply
  * rising shelf and a gentle slope to shore.  The second argument applies
  * only to the no slope case and is the depth of the bottom.
  */
  public  Bottom(String bottomType, double depth)  {
   if  (bottomType.equalsIgnoreCase("noslope")) {
    environment = 1;
    bottomDepth = depth;
   }
   else if (bottomType.equalsIgnoreCase("slope")) {
    environment = 2;
    shallowSlope = (10 - shelfDepth)/(5000 - shelfLine);
    shelfSlope = (shelfDepth - bottomDepth)/(shelfLine - bottomLine);
    planeVector = new Vec3d();
    temporaryVector = new Vec3d();
    reflectedRay = new Vec3d();
    zone1Normal = new Vec3d(-1,1/shallowSlope,0);
    zone2Normal = new Vec3d(-1,1/shelfSlope,0);
    zone3Normal = new Vec3d(0, -1, 0);
    zone1Normal.normalize();
    zone2Normal.normalize();
   }
   else {
     System.out.println("User defined not implemented yet, defaulting to constant");
     environment = 1;
   }
  }

  /**
  * This procedure checks to see if the ray path has crossed the bottom
  * during the current time step and returns true if it did.
  */
  public boolean intersected(Vec3d Pos) {

    switch (environment) {
     case 1:
      if (Pos.get(1) > bottomDepth) {
       intersected = true;
      }
      else {
       intersected = false;
      }
      break;
     case 2:
      if (Pos.get(0) > shelfLine) {
       if (Pos.get(1) > shelfDepth + (Pos.get(0)- shelfLine)*shallowSlope) {
        intersected = true;
         zone = 1;
       }
       else {
        intersected = false;
       }
      }
      else if (Pos.get(0) > bottomLine) {
        if (Pos.get(1) > bottomDepth + (Pos.get(0)- bottomLine)*shelfSlope) {
        intersected = true;
```

```
         zone = 2;
        }
        else {
         intersected = false;
        }
      }
      else {
        if (Pos.get(1) > bottomDepth) {
         intersected = true;
          zone = 3;
        }
        else {
         intersected = false;
        }
      }
      break;
  }
  return intersected;
}

/**
* This method causes a Snell's Law reflection to occur.
*/
public double reflect(Vec3d Pos,Vec3d normal) {
  switch (environment) {
    case 1:
      double diff;
      diff = Pos.get(1) - bottomDepth;
      Pos.set(1,bottomDepth-diff);
      normal.set(1,-normal.get(1));
      break;
    case 2:
      switch (zone) {
        case 1:
          planeVector.set(shelfLine,shelfDepth,0);
          planeVector.sub(Pos);
          intersectedDistance = -zone1Normal.dot(planeVector)/zone1Normal.dot(normal);
          reflectedRay.set(normal);
          temporaryVector.scale(2*zone1Normal.dot(normal),zone1Normal);
          normal.sub(normal,temporaryVector);
          normal.normalize();
          reflectedRay.sub(normal);
          reflectedRay.scale(intersectedDistance);
          Pos.sub(reflectedRay);
          break;
        case 2:
          planeVector.set(bottomLine,bottomDepth,0);
          planeVector.sub(Pos);
          intersectedDistance = -zone2Normal.dot(planeVector)/zone2Normal.dot(normal);
          reflectedRay.set(normal);
          temporaryVector.scale(2*zone2Normal.dot(normal),zone2Normal);
          normal.sub(normal,temporaryVector);
          normal.normalize();
          reflectedRay.sub(normal);
          reflectedRay.scale(intersectedDistance);
          Pos.sub(reflectedRay);
          break;
        case 3:
          planeVector.set(bottomLine,bottomDepth,0);
          planeVector.sub(Pos);
          intersectedDistance = -zone3Normal.dot(planeVector)/zone3Normal.dot(normal);
          reflectedRay.set(normal);
          temporaryVector.scale(2*zone3Normal.dot(normal),zone3Normal);
        normal.sub(normal,temporaryVector);
        normal.normalize();
        reflectedRay.sub(normal);
        reflectedRay.scale(intersectedDistance);
        Pos.sub(reflectedRay);
        break;
    }
    break;
  }
 return 0.0;
}

/**
* This method returns the bottom depth at the given
* (x,z) coordinate. (Ziomek Coordinate System)
*/
public double depth(double x, double z) {
 switch (environment) {
  case 1:
   return bottomDepth;
  case 2:
   if (x < bottomLine ) {
    return bottomDepth;
   }
   else if ((x > bottomLine) && (x < shelfLine)) {
    return bottomDepth + shelfSlope*x;
   }
   else {
    return shelfDepth + shallowSlope*(x-shelfLine);
   }
 }
 return -1;
}

/**
* This method returns the VRML string representing the bottom of the ocean.
*/
public String VRMLBottom() {

 int minX = -5000;
 int minZ = -5000;
 int maxX = 5000;
 int maxZ = 5000;

 String bottomString = "Transform {" + appendage +
                 "  rotation 1 0 0 3.14" + appendage +
                 "  children[" + appendage +
                 "Shape {" + appendage +
                 "  appearance Appearance {" + appendage +
                 "    material Material {" + appendage +
                 "      diffuseColor .3 .15 .15" + appendage +
                 "      transparency 0" + appendage +
                 "    }" + appendage +
                 "  }" + appendage +
                 "  geometry IndexedFaceSet {" + appendage +
                 "    solid FALSE" + appendage +
                 "    coord Coordinate {" + appendage +
                 "      point [" + appendage;

 switch (environment) {
  case 1:
   bottomString = bottomString + "        "+ maxX +" "+ bottomDepth +" "+ maxZ +"," + appendage +
                      "        "+ maxX +" "+ bottomDepth +" "+ minZ +"," + appendage +
                          "     "+ minX +" "+ bottomDepth +" "+ minZ +"," + appendage +
                          "     "+ minX +" "+ bottomDepth +" "+ maxZ +"" + appendage +
                          "    ]" + appendage +
                          "   }" + appendage +
                          "   coordIndex [  0,1,2,3 ]" + appendage;
   break;
  case 2:
    bottomString = bottomString + "       "+ maxX +" "+ "0" +" "+ maxZ +"," + appendage +
                          "     "+ maxX +" "+ "0" +" "+ minZ +"," + appendage +
                          "     "+ shelfLine +" "+ shelfDepth +" "+ minZ +"," + appendage +
                          "     "+ shelfLine +" "+ shelfDepth +" "+ maxZ +"," + appendage +
                          "     "+ bottomLine +" "+ bottomDepth +" "+ maxZ +"," + appendage +
                          "     "+ bottomLine +" "+ bottomDepth +" "+ minZ +"," + appendage +
                          "     "+ minX +" "+ bottomDepth +" "+ minZ +"," + appendage +
                          "     "+ minX +" "+ bottomDepth +" "+ maxZ+ "" + appendage +
                          "    ]" + appendage +
                          "   }" + appendage +
                          "   coordIndex [  0,1,2,3,-1,3,2,5,4,-1,4,5,6,7,-1 ]" + appendage;
   break;
 }
 bottomString = bottomString + "  }" + appendage +
                        "}" + appendage +
                        "]" + appendage+
                        "}" + appendage+
                        VRMLScales();
 return bottomString;
}

/**
* This method applies scales to the VRML bottom
* for the x, y and z directions.
*/
public String VRMLScales() {
 String ts = "";
 ts += PrintVRML.scale("Meters",
                 5000,-5000,
                 0, -bottomDepth, 5000,
                 0, 0, 1, 0,
                 10);
 ts += PrintVRML.scale("Meters",
                 bottomDepth, 0,
                 -5000, 0, 5000,
                 0, 0, 1, -1.57,
                 2);
 ts += PrintVRML.scale("Meters",
                 -5000, 5000,
                 -5000, -bottomDepth, 0,
                 0, 1, 0, -1.57,
                 10);
 return ts;
}

/**
* This is a static method used to indicate whether a line feed is desired
* at the end of every line.
*
* 'true' indicates a linefeed is desired and 'false' indicates that a space
* is desired"
*/
public static void setAppendLineFeed(boolean pAppendLineFeed) {
 appendLineFeed = pAppendLineFeed;
 if (appendLineFeed) {
   appendage = LINE_FEED;
 }
 else {
  appendage = SPACE;
 }
}

/**
* This is a static method that returns the current line appendage.
*/
public static boolean getAppendLineFeed(){
 return appendLineFeed;
}

/****************************************************************
**              Private Section
****************************************************************/
private static final String LINE_FEED = "\n";
private static final String SPACE = " ";
private static boolean appendLineFeed = true;
private static String appendage = LINE_FEED;

/** standard environment that was initialized */
private int   environment = 1;

/** zone within the environment where the ray currently is */
private int   zone = 0;

/** depth of the ocean floor in a simple sloped model */
private double bottomDepth = 2000;

/** depth of the ocean shelf in a simple sloped model */
private double shelfDepth = 500;

/** slope of the plane from the shelf to the ocean bottom */
private double shelfSlope = 0.0;

/** slope of the plane from the shore to the ocean shelf */
private double shallowSlope = 0.0;

/** location of the line in the x direction where the ocean shelf ends */
private double shelfLine = 2500.0;

/** location of the line in the x direction where the ocean floor ends */
private double bottomLine = 0.0;

/** distance from ray intersection with the bottom to the current ray position */
private double intersectedDistance = 0.0;

/** vector from a point on the reflection plane to the current ray position */
private Vec3d   planeVector = null;

/** temporary vector used in calculating the reflection */
private Vec3d   temporaryVector = null;

/** the vector expressing the ray that is reflected from the bottom */
private Vec3d   reflectedRay = null;

/** the normal to the bottom in zone 1 in the simple sloped model*/
private Vec3d   zone1Normal = null;

/** the normal to the bottom in zone 2 in the simple sloped model*/
private Vec3d   zone2Normal = null;
```

```
 /** the normal to the bottom in zone 3 in the simple sloped model*/
 private Vec3d  zone3Normal = null;

 /** temporary boolean variable indicating if a reflection has occured*/
 private boolean intersected;
}
```

C. LOBE.JAVA

```
/*
 File:          Lobe.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;
import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Lobe.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Lobe.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>This class forms a lobe by defining multiple beams.
 *
 *<dt><b>Explanation:</b>
 *<dd>Lobes are emitted in a pattern from an source.  In a
 *  properly designed active sonar, there is only one main
 *  lobe.  This class simulates that lobe.  Lobes are characterized
 *  by a direction that they are pointed in and typically a
 *  horizonatal and vertical beamwidth.  Within this lobe resides
 *  the acoustic energy used for detection of distant objects.
 *  This class subdivides the lobe into beams.  This is done to
 *  keep the wave front of each individual bundle of rays from
 *  getting significantly larger than the wavelength of the acoustic
 *  energy.  This constraint is so that accuracy is maintained and
 *  so that many special problems with large area bundle reflection
 *  can be ignored.  Thus, the lobe class divides the lobe into a
 *  matrix of beams and divides the energy in the lobe among the
 *  beams.  Each beam is then calculated individually using the beam
 *  class.<P>
 *
 *<dt><b>Future Work:</b>
 *<dd>To optimize the execution time, repeated calculation of duplicate rays
 *   needs to be eliminated.
 *
 *<dt><b>History:</b>
```

```
*<dd>        29Nov97 /Timothy M. Holliday     /New
*<dd>        17Mar98 /Timothy M. Holliday     /Added HTML comment convention
*<dd>        12Apr98 /Timothy M. Holliday     /Parameterless Constructors
*<dd>        14Apr08 /Timothy M. Holliday     /Simplified VRML Routines
*
*@see Beam
*@see ExampleLobeStatic
*@see ExampleLobeDynamic
*@see Bottom
*@see Surface
*@see Vec3d
*/
public class Lobe {

 /**
 * Constructor for the lobe class.
 *
 * A lobe is defined as an mxn array of beams. Due to memory and
 * initialization constraints the current limit is 5 by 5.
 */
 public Lobe() {
  for (i=0;i<MAX_X_PARTITIONS;i++) {
   for (j=0;j<MAX_Y_PARTITIONS;j++) {
    mainLobe[i][j] = new Beam();
   }
  }
 }


 /**
 *  This method resets all of the lobe parameters after
 *  instantiation has occurred since reuse is more time
 *  efficient than garbage collection and reallocation.
 *
 */
 public void reset() {
  beamWidthX = lobeWidthX/(double)numberXPartition;
  beamWidthY = lobeWidthY/(double)numberYPartition;
  double halfLobeWidthX = lobeWidthX/2.0;
  double halfLobeWidthY = lobeWidthY/2.0;

  for (i=0; i<numberXPartition; i++) {
   for (j=0; j<numberYPartition; j++) {
    mainLobe[i][j].setPosition(position.get(0), position.get(1), position.get(2));
    mainLobe[i][j].setElevation(elevation-halfLobeWidthY+(double)(2*i+1)/2.0*beamWidthY);
    mainLobe[i][j].setAzimuth(azimuth-halfLobeWidthX+(double)(2*j+1)/2.0*beamWidthX);
    mainLobe[i][j].setHalfBeamWidthY(beamWidthY/2.0);
    mainLobe[i][j].setHalfBeamWidthX(beamWidthX/2.0);
    mainLobe[i][j].setDuration(pulseDuration);
    mainLobe[i][j].setEndTime(endTime);
    mainLobe[i][j].setBottom(bottom);
    mainLobe[i][j].setSurface(surface);
    mainLobe[i][j].setSsp(ssp);
    mainLobe[i][j].setBeamNumber(i*MAX_X_PARTITIONS+j);
    mainLobe[i][j].reset();
   }
  }
 }

 /**
 * This method sets the azimuthal angle, which is the angle from
 * the x-axis to the z-axis rotating about the y-axis.
 */
```

```
public void setAzimuth(double phi) {
 azimuth = phi;
}

/**
* This method to gets the Azimuthatl Angle.
*/
public double getAzimuth() {
 return azimuth;
}

/**
* This method to sets the total lobe width
* in the azimuthal direction reletive to the Ziomek
* Coordinate System. Argument is in degrees.
*/
public void setLobeWidthX(double pLobeWidthX) {
 lobeWidthX = pLobeWidthX;
}

/**
* This is an accessor method to get the total lobe width
* in the azimuthal direction reletive to the Ziomek Coordinate
* System. Value is in degrees.
*/
public double getLobeWidthX() {
 return lobeWidthX;
}

/**
* This method sets the elevation angle, which is the angle from the
* y-axis to the x-axis rotating about the z-axis .
*/
public void setElevation(double beta) {
 elevation = beta;
}

/**
* This method returns the elevation angle.
*/
public double getElevation() {
 return elevation;
}

/**
* This is an accessor method to set the total lobe width
* in the elevation direction reletive to the Ziomek
* Coordinate System. Argument is in degrees.
*/
public void setLobeWidthY(double pLobeWidthY) {
 lobeWidthY = pLobeWidthY;
}

/**
* This is an accessor method to get the total lobe width
* in the elevation direction reletive to the Ziomek Coordinate
* System. Value is in degrees.
*/
public double getLobeWidthY() {
 return lobeWidthY;
}

/**
```

```
* This method sets the lobe positon.
*/
public void setPosition(Vec3d pPosition) {
 position.set(pPosition);
}

/**
* This method returns the lobe position.
*/
public Vec3d getPosition() {
 return position;
}

/**
* This method sets the time step in the simulation.
*/
public void setDeltaTime(double pDeltaTime) {
 deltaTime = pDeltaTime;
}

/**
* This method returns the simulation step time.
*/
public double getDeltaTime() {
 return deltaTime;
}

/**
* This method sets the number of beams in the azimuthal
* direction that there are in the lobe.
*/
public void setNumberXPartition(int number) {
 numberXPartition = number;
}

/**
* This method returns the number of beams in the azimuthal
* direction that there are in the lobe.
*/
public int getNumberXPartition() {
 return numberXPartition;
}

/**
* This method sets the number of beams in the elevation
* direction that there are in the lobe.
*/
public void setNumberYPartition(int number) {
 numberYPartition = number;
}

/**
* This method returns the number of beams in the elevation
* direction that there are in the lobe.
*/
public int getNumberYPartition() {
 return numberYPartition;
}

/**
* This method sets the simulation end time.
*
* This value is reletive to the start time which is 0.0.
```

```
*/
public void setEndTime(double pEndTime) {
  endTime = pEndTime;
}

/**
* This method returns the simulation end time.
*/
public double getEndTime() {
  return endTime;
}

/**
* This is an accessor method. Duration is currently 1 or 2.
* The integer refers to the number of deltTime increments.
*/
public void setDuration(int duration) {
  pulseDuration = duration;
}

/**
* This is an accessor method. Duration is currently 1 or 2.
* The integer refers to the number of deltTime increments.
*/
public int getDuration() {
  return pulseDuration;
}

/**
* This method sets the handle to the bottom object.
*/
public void setBottom(Bottom pBottom) {
  bottom = pBottom;
}

/**
* This method returns the handle to the bottom object.
*/
public Bottom getBottom() {
  return bottom;
}

/**
* This method sets the handle to the surface object.
*/
public void setSurface(Surface pSurface) {
  surface = pSurface;
}

/**
* This method returns the handle to the surface object.
*/
public Surface getSurface() {
  return surface;
}

/**
* This method sets the handle to the sound speed profile object.
*/
public void setSsp(SSP pSsp) {
  ssp = pSsp;
}
```

```
/**
 * This method returns the handle to the sound speed profile object.
 */
public SSP getSsp() {
  return ssp;
}

/**
*  This method calculates the Lobe by calling the
*  calculateBeam method for each beam in the lobe.
*/
public void calculateLobe(Targets targets) {

  for (i=0; i<numberYPartition; i++) {
    for (j=0; j<numberXPartition; j++) {
      mainLobe[i][j].calculateBeam(targets);
    }
  }
}

/**
 * This method returns VRML objects that signify detections that
 * were made in the virtual world.
 *
 * The objects are collections of red spheres indicating a detect.
 */
public String detectionVRML() {

  double lAzimuth, lElevation, x, y, z, range, EL;

  String temporaryString =
    " Group {" + appendage+
    " children [" + appendage;

  for (i=0; i<numberYPartition; i++) {
    for (j=0; j<numberXPartition; j++) {
      for(k=0;k<mainLobe[i][j].getDetectCount();k++) {
        lAzimuth = azimuth + beamWidthX/2.0*(2*(double)j+1-(double)numberXPartition);
        lAzimuth *= (3.1415/180);
        lElevation = elevation + beamWidthY/2.0*(2*(double)i+1-(double)numberYPartition);
        lElevation *= (3.1415/180);
        range = mainLobe[i][j].getDetectTime(k)*1500;

        // Convert from Ziomek to VRML coordinate system
        x = position.get(0) + range*Math.sin(lElevation)*Math.cos(lAzimuth);
        y = -(position.get(1) + range*Math.cos(lElevation));
        z = -(position.get(2) + range*Math.sin(lElevation)*Math.sin(lAzimuth));

        EL = mainLobe[i][j].getDetectEchoLevel(k)/-200;
        if (EL > 0.9) {
          EL = 0.9;
        }
        else if (EL < 0) {
          EL = 0;
        }

        temporaryString +=
          " Transform {" + appendage+
          " translation "+x+" "+y+" "+z+ appendage +
          " children [" + appendage+
          " Shape {" + appendage+
          " appearance Appearance {" + appendage+
          " material Material {" + appendage+
          " emissiveColor 1 0 0" + appendage+
          " transparency "+EL+ appendage +
          " }" + appendage+
          " }" + appendage+
          " geometry Sphere { radius 20 }" + appendage+
          " }" + appendage+
          " ]" + appendage+
          " }" + appendage;
     }
   }
  }
 return temporaryString + "] } " + appendage;
}

/**
* This method writes the dynamic VRML representation of the Lobe by calling the
* VRMLBeam routine for each beam.
* Also Writes out a Viewpoint node.
*/
public String dynamicVRML() {

  String lobeString = "Viewpoint {" + appendage +
            " position " + (position.get(0)-100) +" "+ -position.get(1) +" "+ (position.get(2)-20)+ "" + appendage +
            " orientation 0 -1 0 " + ((90-azimuth)*3.14/180)+"" + appendage +
            " description \"Beam\" " + appendage+
            "}" + appendage;

  lobeString = lobeString + mainLobe[0][0].pingTimerVRML();

  Date timecheck = new Date();
  for (i=0; i<numberYPartition; i++) {
    for (j=0; j<numberXPartition; j++) {
      lobeString = lobeString + mainLobe[i][j].dynamicVRML();
    }
  }
  Date timecheck2 = new Date();
  lobeString = lobeString + "# " + timecheck + "" + appendage;
  lobeString = lobeString + "# " + timecheck2 + "" + appendage;
  return lobeString;

 }

/**
* This method writes the static VRML representation of the Lobe by calling the
* VRMLBeam routine for each beam.
*/
 public String staticVRML(int colorMap, int intensityMap) {

   String temp = "";

   for (i=0; i<numberYPartition; i++) {
     for (j=0; j<numberXPartition; j++) {
       temp += mainLobe[i][j].staticVRML( colorMap, intensityMap);
     }
   }
   return temp;
 }

 /**
 * This is a static method used to indicate whether a line feed is desired
 * at the end of every line.
 *
 * 'true' indicates a linefeed is desired and 'false' indicates that a space
* is desired"
*/
public static void setAppendLineFeed(boolean pAppendLineFeed) {
  appendLineFeed = pAppendLineFeed;
  Beam.setAppendLineFeed(appendLineFeed);
  if (appendLineFeed) {
    appendage = LINE_FEED;
  }
  else {
    appendage = SPACE;
  }
}

 /**
 * This is a static method that returns the current line appendage.
 */
 public static boolean getAppendLineFeed(){
  return appendLineFeed;
 }

/*****************************************************************
**              Private Section
*****************************************************************/
 private static final String LINE_FEED = "\n";
 private static final String SPACE = " ";
 private static boolean appendLineFeed = true;
 private static String appendage = LINE_FEED;

 private static final int MAX_X_PARTITIONS = 5;
 private static final int MAX_Y_PARTITIONS = 5;

 private Vec3d position = new Vec3d();
 private double azimuth = 0.0;
 private double elevation = 90.0;
 private double lobeWidthX = 1.0;
 private double lobeWidthY = 1.0;
 private double beamWidthX = 1.0;
 private double beamWidthY = 1.0;
 private int numberXPartition = 1;
 private int numberYPartition = 1;
 private int   pulseDuration = 1;
 private Bottom bottom = null;
 private Surface surface = null;
 private SSP ssp = null;
 private double deltaTime = 0.006;
 private double endTime;

 private Beam[][]      mainLobe = new Beam[MAX_X_PARTITIONS][MAX_Y_PARTITIONS];
 private int           i, j, k;

}
```

D. PRINTVRML.JAVA

```
/*
 File:          PrintVRML.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;
```

```
import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/PrintVRML.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/PrintVRML.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Contains static methods that return standard VRML strings.
 *
 *<dt><b>Explanation:</b>
 *<dd>Creates strings in VRML for header, directionalLight and
 *   navigationInfo.<P>
 *
 *<dt><b>History:</b>
 *<dd>           15Apr98 /Timothy M. Holliday      /New
 *<dd>           14May98 /Timothy M. Holliday      /Added color and intensity schemes
 *<dd>       18May98 /Timothy M. Holliday   /Added VRML colorBarView and textNode
 *
 */
public class PrintVRML {

  public static final int RAINBOW = 1;
  public static final int RED = 2;
  public static final int GREEN = 3;
  public static final int BLUE = 4;
  public static final int TRICOLOR = 5;
  public static final int CONSTANT = 6;
  public static final int LINEAR = 7;
  public static final String HORIZONTAL = "0 0 1 0";
  public static final String VERTICAL = "0 0 1 1.57";
  public static final String RIGHT = "\"END\"";
  public static final String CENTER = "\"MIDDLE\"";
  public static final String LEFT = "\"BEGIN\"";

  public static String header() {
   return "#VRML V2.0 utf8" + appendage;
  }

  public static String directionalLight() {
   return "DirectionalLight {"+appendage+
        "  ambientIntensity 0"+appendage+
        "  color 1 1 1"+appendage+
        "  direction 0 -1 0"+appendage+
        "  intensity 1"+appendage+
        "  on TRUE"+appendage+
        "}"+appendage;
  }

  public static String navigationInfo() {
   return "NavigationInfo { "+appendage+
        " type        [\"FLY\" \"EXAMINE\"]"+appendage+
        " speed       50"+appendage+
        " headlight TRUE"+appendage+
        " visibilityLimit 50000"+appendage+
        "}"+appendage;
}

public static String printLegendView(String title,
                         String info1,
                         String info2,
                         String info3,
                         String info4,
                         String info5,
                         double x,
                         double y,
                         double z) {
  String temp = appendage;
  temp += "EXTERNPROTO protoLegendViewpoint[" +appendage;
  temp += "eventIn      SFBool     set_bind" +appendage;
  temp += "exposedField SFFloat    fieldOfView   " +appendage;
  temp += "exposedField SFBool     jump          " +appendage;
  temp += "exposedField SFRotation orientation " +appendage;
  temp += "exposedField SFVec3f    position      " +appendage;
  temp += "field        SFString   description" +appendage;
  temp += "eventOut     SFTime     bindTime" +appendage;
  temp += "eventOut     SFBool     isBound" +appendage;
  temp += "field        MFString   Title   " +appendage;
  temp += "field        MFString   info   " +appendage;
  temp += "] \"protoLegendViewpoint.wrl\"" +appendage;

  temp += "protoLegendViewpoint{" +appendage;
  temp += "Title [\""+ title +"\"]" +appendage;
  temp += "position "+x+" "+y+" "+z+"" +appendage;
  temp += "info  [\"" +info1+ "\" \"" +info2+ "\" \"" +info3+ "\"" +appendage;
  temp += " \"" +info4+ "\" \"" +info5+ "\"]" +appendage;
  temp += "}" +appendage;
  return temp;
}

public static String protoColorBar(String vLabel,
                           String hLabel,
                           String vSubLabel1,
                           String vSubLabel2) {
  String temp = appendage;
  temp += "PROTO protoColorBar[" +appendage;
  temp += "]  {"+appendage;

  temp += "Transform {"+appendage;
  temp += "translation 2 1 4.5"+appendage;
  temp += "children ["+appendage;
  temp += colorBar(vLabel,hLabel,vSubLabel1,vSubLabel2);
  temp += "]" +appendage;
  temp += "}" +appendage;
  temp += "}" +appendage;
  return temp;
}

public static String colorBarView(String title,
                         double x,
                         double y,
                         double z,
                         double xAxis,
                         double yAxis,
                    double zAxis,
                    double theta) {

  String temp = "Transform {" +appendage;
  temp += "translation "+x+" "+y+" "+z+"" +appendage;
  temp += "rotation "+xAxis+" "+yAxis+" "+zAxis+" "+theta +appendage;
  temp += "children [" +appendage;
  temp += "Viewpoint{" +appendage;
  temp += "description \""+ title +"\"" +appendage;
  temp += "}" +appendage;
  temp += "protoColorBar {}";
  temp += "]" +appendage;
  temp += "}" +appendage;
  return temp;
}

public static void setAppendLineFeed(boolean pAppendLineFeed) {
  appendLineFeed = pAppendLineFeed;
  Ray.setAppendLineFeed(appendLineFeed);
  if (appendLineFeed) {
    appendage = LINE_FEED;
  }
  else {
    appendage = SPACE;
  }
}

public static boolean getAppendLineFeed(){
  return appendLineFeed;
}

public static void setColorScheme(int scheme) {
  colorMaximum = 1;
  colorMinimum = 0;
  colorScheme = scheme;
  colorDelta = 1;
  colorSubDelta = colorDelta/3;
}

public static void setColorScheme(int scheme, double maximum, double minimum) {
  colorMaximum = maximum;
  colorMinimum = minimum;
  colorScheme = scheme;
  colorDelta = maximum - minimum;
  colorSubDelta = colorDelta/3;
}

public static void setColorScheme(int scheme,
                    double maximum,
                    double detection,
                    double counterDetection,
                    double minimum) {
  colorScheme = scheme;
  colorMaximum = maximum;
  colorDetection = detection;
  colorCounterDetection = counterDetection;
  colorMinimum = minimum;
  colorDelta = maximum - minimum;
}

public static void setColorValue(double value) {
  if ((colorScheme == RAINBOW)||(colorScheme == TRICOLOR)) {
    colorValue = (value-colorMinimum)/colorDelta;
```

```
      if (colorValue<0.0) {
        colorValue = 0.0;
      }
      else if(colorValue>1.0) {
        colorValue = 1.0;
      }
    }
    //else if (colorScheme == TRICOLOR) {
    //  colorValue = value;
    //}
    else {
//      System.out.println("#PrintVRML:  setColorValue: not a valid scheme");
      colorValue = 1;
    }
  }

  public static void setIntensityScheme(int scheme) {
    intensityScheme = scheme;
    intensityMaximum = 1;
    intensityMinimum = 0;
    intensityDelta = 1;
  }
  public static void setIntensityScheme(int scheme, double maximum, double minimum) {
    intensityScheme = scheme;
    intensityMaximum = maximum;
    intensityMinimum = minimum;
    intensityDelta = maximum - minimum;
  }

  public static void setIntensityValue(double value) {
    if (intensityScheme == LINEAR ) {
      intensityValue = (value-intensityMinimum)/intensityDelta;
      if (intensityValue < 0.0) {
        intensityValue = 0.0;
      }
      else if(intensityValue > 1.0) {
        intensityValue = 1.0;
      }
    }
    else if (intensityScheme == CONSTANT) {
      intensityValue = 1.0;
    }
    else {
      System.out.println("#PrintVRML:  setIntensityValue: not a valid scheme");
      intensityValue = 1.0;
    }
  }

  public static String getColor() {
    double red = 0;
    double green = 0;
    double blue = 0;

    if (colorScheme == RAINBOW) {
      if ( colorValue > .66667) {
        red = (colorValue - .66667)/.33333;
        green = 1-red;
        blue = 0.0;
      }
      else if (colorValue > .33333) {
        red = 0.0;
        green = (colorValue - .33333)/.33333;
        blue = 1-green;
    }
    else {
      green = 0.0;
      blue = colorValue/.33333;
      red = 1-blue;
    }
  }
  else if (colorScheme == RED) {
    red = 1.0;
    green = 0.0;
    blue = 0.0;
  }
  else if (colorScheme == GREEN) {
    red = 0.0;
    green = 1.0;
    blue = 0.0;
  }
  else if (colorScheme == BLUE) {
    red = 0.0;
    green = 0.0;
    blue = 1.0;
  }
  else if (colorScheme == TRICOLOR) {
    if (colorValue > (colorDetection-colorMinimum)/colorDelta) {
      red = 1.0;
      green = 0.0;
      blue = 0.0;
    }
    else if (colorValue > (colorCounterDetection-colorMinimum)/colorDelta) {
      red = 0.0;
      green = 1.0;
      blue = 0.0;
    }
    else {
      red = 0.0;
      green = 0.0;
      blue = 1.0;
    }
  }


  red *= intensityValue;
  green *= intensityValue;
  blue *= intensityValue;

  return (String)(red+" "+green+" "+blue+appendage);
}

public static String colorBar(String vLabel,
                              String hLabel,
                              String detectionLabel,
                              String counterDetectionLabel) {
  double i = 0;
  double j = 0;
  String ts = "";

  switch (colorScheme) {
    case RAINBOW:
      ts += textNode(VERTICAL,-.75, 0, 0, vLabel, .25,.25,CENTER);
      ts += textNode(HORIZONTAL,.6, .4, 0, ""+colorMaximum, .15,.15,LEFT);
      ts += textNode(HORIZONTAL,.6, -.4, 0, ""+colorMinimum, .15,.15,LEFT);
      break;
```

```
 case TRICOLOR:
  ts += textNode(HORIZONTAL,-.6, .5+(colorDetection-colorMaximum)/colorDelta,
           0, detectionLabel, .15,.15,RIGHT);
  ts += textNode(HORIZONTAL,-.6, .5+(colorCounterDetection-colorMaximum)/colorDelta,
           0, counterDetectionLabel, .15,.15,RIGHT);
  ts += textNode(HORIZONTAL,.6, .5, 0, ""+colorMaximum, .15,.15,LEFT);
  ts += textNode(HORIZONTAL,.6, .5+(colorDetection-colorMaximum)/colorDelta,
           0, ""+colorDetection, .15,.15,LEFT);
  ts += textNode(HORIZONTAL,.6, .5+(colorCounterDetection-colorMaximum)/colorDelta,
           0, ""+colorCounterDetection, .15,.15,LEFT);
  ts += textNode(HORIZONTAL,.6, -.5, 0, ""+colorMinimum, .15,.15,LEFT);
  break;

 case RED:
  ts += textNode(VERTICAL,-.75, 0, 0, vLabel, .25,.25,CENTER);
  break;

 case BLUE:
  ts += textNode(VERTICAL,-.75, 0, 0, vLabel, .25,.25,CENTER);
  break;

 case GREEN:
  ts += textNode(VERTICAL,-.75, 0, 0, vLabel, .25,.25,CENTER);
  break;

}

switch (intensityScheme) {
 case LINEAR:
  ts += textNode(HORIZONTAL,0 , .75, 0, hLabel, .25,.25,CENTER);
  ts += textNode(VERTICAL,-.4, -.6, 0, ""+intensityMaximum, .15,.15,RIGHT);
  ts += textNode(VERTICAL,.4, -.6, 0, ""+intensityMinimum, .15,.15,RIGHT);
  break;

 case CONSTANT:
  ts += textNode(HORIZONTAL,0 , .75, 0, hLabel, .25,.25,CENTER);
  break;
}

ts += "Shape {"+appendage;
ts += "geometry IndexedFaceSet {"+appendage;
ts += "coord Coordinate {"+appendage;
ts += "point ["+appendage;
for (i=0;i<20;i++) {
 for (j=0;j<20;j++) {
  ts += (-.5+(double)i/20) +" ";
  ts += (.5-(double)j/20) +" 0.0"+appendage;
 }
}
ts += "]"+appendage;
ts += "}"+appendage;

ts += "coordIndex ["+appendage;
for (i=0;i<19;i++) {
 for (j=0;j<19;j++) {
  ts += (int)(j*20+i)+" "+(int)((j+1)*20+i)+" "+(int)((j+1)*20+i+1)+" "+(int)(j*20+i+1)+" -1"+appendage;
 }
}
ts += "]"+appendage;

ts += "color Color {"+appendage;
ts += "color ["+appendage;
for (i=0;i<20;i++) {
   for (j=0;j<20;j++) {
     setIntensityValue(intensityMaximum-(double)i/20*intensityDelta);
     setColorValue(colorMaximum-(double)j/20*colorDelta);
     ts += getColor();
   }
  }
  ts += "]"+appendage;
  ts += "}"+appendage;

  ts += "colorIndex ["+appendage;
  for (i=0;i<19;i++) {
   for (j=0;j<19;j++) {
     ts += (int)(j*20+i)+" "+(int)((j+1)*20+i)+" "+(int)((j+1)*20+i+1)+" "+(int)(j*20+i+1)+" -1"+appendage;
   }
  }
  ts += "]"+appendage;
  ts += "solid FALSE"+appendage;
  ts += "colorPerVertex TRUE"+appendage;

  ts += "}"+appendage;
  ts += "}"+appendage;
  return ts;
 }
 public static String scale(String label,
                            double maximum,
                            double minimum,
                            double x,
                            double y,
                            double z,
                            double xAxis,
                            double yAxis,
                            double zAxis,
                            double theta,
                            int divisions) {
  int i=0;
  String value = "";
  String ts = "";
  ts +="Transform {"+appendage;
  ts +="translation "+x+" "+y+" "+z+appendage;
  ts +="rotation "+xAxis+" "+yAxis+" "+zAxis+" "+theta+appendage;
  ts +="children ["+appendage;
  ts +="Shape {"+appendage;
  ts +="appearance Appearance {"+appendage;
  ts +="material Material {"+appendage;
  ts +="emissiveColor 1 1 1"+appendage;
  ts +="}"+appendage;
  ts +="}"+appendage;
  ts +="geometry IndexedLineSet{"+appendage;
  ts +="coord Coordinate{"+appendage;
  ts +="point ["+appendage;
  ts +=""+minimum+" 0 0"+appendage;
  ts +=""+maximum+" 0 0"+appendage;
  ts +="]"+appendage;
  ts +="}"+appendage;

  ts +="coordIndex ["+appendage;
  ts +="0 1 -1"+appendage;
  ts +="]"+appendage;
  ts +="}"+appendage;
  ts +="}"+appendage;

  for (i=0; i<=divisions;i++) {
   if (minimum > maximum) {
    value = ""+(int)(maximum+i*((minimum-maximum)/divisions));
   }
   else {
    value = ""+(int)(minimum+i*((maximum-minimum)/divisions));
   }
   ts +=textNode(VERTICAL,
              minimum+(maximum-minimum)*(double)i/(double)divisions,
              0,
              0,
              value,
              Math.abs(((maximum-minimum)/4.0)/divisions),
              Math.abs(((maximum-minimum)/4.0)/divisions),
              RIGHT);
  }
   ts +=textNode(HORIZONTAL,
              (minimum+maximum)/2.0,
              -Math.abs((minimum-maximum)/divisions),
              0,
              label,
              Math.abs(((maximum-minimum)/2.0)/divisions),
              Math.abs(((maximum-minimum)/2.0)/divisions),
              CENTER);

  ts +="]"+appendage;
  ts +="}"+appendage;
  return ts;
 }

/**************************************************************
**          Private Section
**************************************************************/
 private static String textNode(String orientation,
                        double x,
                        double y,
                        double z,
                        String text,
                        double size,
                        double space,
                        String justify) {

  String ts = "Transform {"+appendage;
  ts += "translation "+x+" "+y+" "+z+appendage;
  ts += "rotation "+orientation+appendage;
  ts += "children Shape {"+appendage;
  ts += "appearance Appearance{material Material{ emissiveColor 1 1 1}}"+appendage;
  ts += "geometry Text {"+appendage;
  ts += "string \" "+text+" \" "+appendage;
  ts += "fontStyle FontStyle {"+appendage;
  ts += "size "+size+appendage;
  ts += "spacing "+space+appendage;
  ts += "justify "+justify+appendage;
  ts += "} "+appendage;
  ts += "} "+appendage;
  ts += "}}"+appendage;
  return ts;
 }

 private static final String LINE_FEED = "\n";
 private static final String SPACE = " ";
 private static boolean appendLineFeed = true;
 private static String appendage = LINE_FEED;
 private static int colorScheme = 0;
 private static double colorMaximum = 0.0;
 private static double colorMinimum = 0.0;
 private static double colorDelta = 0.0;
 private static double colorSubDelta = 0.0;
 private static double colorDetection = 0.0;
 private static double colorCounterDetection = 0.0;
 private static double colorValue = 0.0;
 private static int intensityScheme = 0;
 private static double intensityMaximum = 0.0;
 private static double intensityMinimum = 0.0;
 private static double intensityDelta = 0.0;
 private static double intensityValue = 0.0;

}
```

E. RAY.JAVA

```
/*
 File:          Ray.java
 Compiler:      jdk1.1.3
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;
import java.lang.Math;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A
HREF="http://dubhe.cc.nps.navy.mil/~tmhollid">http://dubhe.cc.nps.navy.mil/~tmhollid</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Ray.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Ray.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>This class incorporates the RRA (recursive ray acoustics)
 *  algorithm from Professor Ziomek.
 *
 *<dt><b>Explanation:</b>
 *<dd>Any wave front can be decomposed into a set of plane waves
 *  with the plane waves travelling in the direction of the normal
 *  to the wave front.  The normal for each plane wave is defined by
 *  the normal to the wave front at the point where the front
 *  and the plane wave touch.  This class uses the ray theory solution
 *  to the linear acoustic wave equation to simulate one ray from
 *  the surface of a wavefront as it propagetes through the ocean.<P>
 *
 *<dt><b>History:</b>
 *<dd>          1Nov97  /Timothy M. Holliday       /New
 *<dd>          17Mar98 /Timothy M. Holliday       /Added HTML comment convention
 *<dd>          12Apr98 /Timothy M. Holliday       /Parameterless Constructors
```

```
*<dd>            14Apr08 /Timothy M. Holliday        /Simplified VRML Routines
*
*@see Surface
*@see Bottom
*@see Beam
*@see ExampleRay
*@see Vec3d
*/
public class Ray {


 /**
 * Constructor for the Ray class.  Constructs a fixed size structure to trace a
 * ray of acoustic energy through the ocean.
 */
 public Ray () {
  /*
  * Initialize the member variables that require it
  */
  position = new Vec3d();
  normal = new Vec3d();
  startPosition = new Vec3d();
  prevPosition = new Vec3d();
  prevNormal = new Vec3d();
  deltaPosition = new Vec3d();
  starPosition = new Vec3d();
  trailingPosition = new Vec3d();

  int i;
  for (i=0;i<MAX_POINTS;i++) {
   rayPath[i] = new Vec3d();
   trailingPath[i] = new Vec3d();
  }
 }

 /**
 * This method resets all of the ray parameters after
 * instanciation has occurred since reuse is more time
 * efficient than garbabage collection and reallocation.
 */
 public void reset() {

  /**
  * Initialize the member variables that require it
  */
  prevPosition.set(position);
  setNormal();
  prevNormal.set(normal);
  deltaPosition.set(position);
  starPosition.set(position);
  trailingPosition.set(position);
  time =          0.0;
  pathLength =    0.0;
  reflectionPhase = 0.0;
  pointCount =    0;
  reflection =    false;
  absorbtion =    0.0;
  totalCurve =    0.0;
 }

 /**
 * This method takes the direction cosines of the ray
 * and forms the vector normal to the wavefront.
```

```
  */
  private void setNormal() {
    double u,v,w;

    u = (double)(Math.sin(elevation*Math.PI/180) * Math.cos(azimuth*Math.PI/180));
    v = (double)(Math.cos(elevation*Math.PI/180));
    w = (double)(Math.sin(elevation*Math.PI/180) * Math.sin(azimuth*Math.PI/180));

    normal.set(u,v,w);
  }

  /**
  * This method returns the normal vector to the wavefront.
  */
  public Vec3d getNormal() {
    return normal;
  }

  /**
  * This method sets the azimuthal angle, which is the angle from the x-axis
  * to the z-axis rotating about the y-axis.
  */
  public void setAzimuth(double phi) {
    azimuth = phi;
  }

  /**
  * This method returns the azimuthal angle.
  */
  public double getAzimuth() {
    return azimuth;
  }

  /**
  * This method sets the elevation angle, which is the angle from the y-axis
  * to the x-axis rotating about the z-axis.
  */
  public void setElevation(double beta) {
    elevation = beta;
  }

  /**
  * This method returns the elevation angle.
  */
  public double getElevation() {
    return elevation;
  }

  /**
  * This method sets the position of the ray.
  */
  public void setPosition(double x, double y, double z) {
    startPosition.set(x,y,z);
    position.set(x,y,z);
    prevPosition.set(x,y,z);
    time = deltaTime;
  }

  /**
  * This method gets the position of the ray.
  */
  public Vec3d getPosition() {
    return position;
```

```
}

/**
* This method is used to change the time step of the ray after
* instanciation.
*/
public void setDeltaTime(double deltaTime) {
 time = deltaTime;
}

/**
* This method is returns the time step of the ray.
*/
public double getDeltaTime() {
 return time;
}

/**
* This method sets the ping duration. Duration is currently 1 or 2.
* The integer refers to the number of deltaTime increments.
*/
public void setDuration(int duration) {
 pulseDuration = duration;
}

/**
* This returns the ping duration. Duration is currently 1 or 2.
* The integer refers to the number of deltTime increments.
*/
public int getDuration() {
 return pulseDuration;
}

/**
* This method sets the handle to the bottom object.
*/
public void setBottom(Bottom pBottom) {
 bottom = pBottom;
}

/**
* This method returns the handle to the bottom object.
*/
public Bottom getBottom() {
 return bottom;
}

/**
* This method sets the handle to the surface object.
*/
public void setSurface(Surface pSurface) {
 surface = pSurface;
}

/**
* This method returns the handle to the surface object.
*/
public Surface getSurface() {
 return surface;
}

/**
* This method sets the handle to the sound speed profile object.
```

```
*/
public void setSsp(SSP pSsp) {
  ssp = pSsp;
}

/**
* This method returns the handle to the sound speed profile object.
*/
public SSP getSsp() {
  return ssp;
}


/**
* This method returns the phase change of the wavefront
*/
public double getReflectionPhase() {
  return reflectionPhase;
}

/**
* This method returns the x-component of the position
* of the wavefront
*/
public double getPositionX() {
  return position.get(0);
}

/**
* This method returns the y-component of the position
* of the wavefront
*/
public double getPositionY() {
  return position.get(1);
}

/**
* This method returns the z-component of the position
* of the wavefront
*/
public double getPositionZ() {
  return position.get(2);
}

/**
* This method returns the position of the trailing edge of
* the wavefront
*/
public Vec3d getTrailingPosition() {
  return trailingPosition;
}


/**
* This method returns the current simulation time.
*/
public double getTime() {
  return time;
}

/**
* This method returns a saved simulation time.
*/
```

```
public double getTime(int index) {
 return rayTime[index];
}

/**
* This method causes the ray to propagate one timestep into the future.
*/
public void Propagate(double timeStep) {
 deltaTime = timeStep;
 nextPosition();
 nextNormal();
 nextTimeandpathLength();

}

/**
* This method returns the current number of points stored for the ray.
*/
public int getCount() {
 return pointCount;
}

/**
* This method returns the current total relaxation absorbtion in dB.
*/
public double getAbsorption() {
 return absorbtion;
}

/**
* This method returns whether or not the ray has been reflected in
* the most recent time step.
*/
public boolean reflected() {
 return reflection;
}

/**
* This method forces a recording of the current ray position. It also forces the
* total curvature of the ray since the last recorded point to 0.0.
*/
public void recordPoint() {
 rayPath[pointCount].set(position);

 if (pulseDuration == 2) {
  trailingPath[pointCount].set(trailingPosition);
 }
 else {
  trailingPath[pointCount].set(prevPosition);
 }

 rayTime[pointCount] = time;
 pointCount++;
 totalCurve = 0.0;
}

/**
* This returns the total curvature of the ray path since the last recorded point.
*
* The curvature of a small segment of curve can be approximated by the curvature
* of a circle which is the change in angular position around the circle divided
* by the pathlength change. K = deltaTheta/deltaPathLength. Thus total curvature
* is the sum of the curvatures of the small segments along a ray trajectory.
```

```
*/
public double totalCurvature() {
  return totalCurve;
}

/**
* This method returns the requested position as a String.
*/
public String position(int N) {
  return (int)rayPath[N].get(0) + " " +
       (int)rayPath[N].get(1) + " " +
       (int)rayPath[N].get(2) + appendage;
}

/**
* This method returns the requested trailing edge of the ray as a String.
*/
public String trailingPosition(int N) {
  return (int)trailingPath[N].get(0) + " " +
       (int)trailingPath[N].get(1) + " " +
       (int)trailingPath[N].get(2) + appendage;
}

/**
* This method returns the requested normalized time as a string.
*/
public String normalizedTime(int N, double endtime) {
  return Math.floor(100*rayTime[N]/endtime)/100 + appendage;
}

/**
* This is a static method used to indicate whether a line feed is desired
* at the end of every line.
*
* 'true' indicates a linefeed is desired and 'false' indicates that a space
* is desired"
*/
public static void setAppendLineFeed(boolean pAppendLineFeed) {
  appendLineFeed = pAppendLineFeed;
  if (appendLineFeed) {
    appendage = LINE_FEED;
  }
  else {
    appendage = SPACE;
  }
}

/**
* This is a static method that returns the current line appendage.
*/
public static boolean getAppendLineFeed(){
  return appendLineFeed;
}

/*************************************************************************
**
**          Beginning of private section
**
*************************************************************************/
private static final String LINE_FEED = "\n";
private static final String SPACE = " ";
private static boolean appendLineFeed = true;
private static String appendage = LINE_FEED;
```

```
/*
* This method calculates the position of the ray after the time step.
*/
private void nextPosition() {

 // save trailing position depending on duration
 if (pulseDuration == 2) {
  trailingPosition.set(prevPosition);
 }
 else {
  trailingPosition.set(position);
 }

 prevPosition.set(position);
 prevdeltaArc = deltaArc;
 prevNormal.set(normal);
 position.add(deltaPosition());
 reflection = false;

 if (bottom.intersected(position)) {
  reflectionPhase += bottom.reflect(position, prevNormal);
  reflection = true;
 }

 if (surface.intersected(position)) {
  reflectionPhase += surface.reflect(position,prevNormal);
  reflection = true;
 }
}

/*
* This method calculates the normal to the wavefront at the position of the ray
* after the time step using the ziomek formula.
*/
private void nextNormal() {
 updatestarPosition();
 deltaArc = deltaTime*ssp.C(position);
 term1.scale(ssp.C(position)/ssp.C(prevPosition),prevNormal);
 term2.scale(deltaTime,ssp.gradC(starPosition));
 totalCurve += term2.length();
 normal.sub(prevNormal,term2);
}

/*
* This method the calcultes the total elapsed time and length of path travelled.
*/
 private void nextTimeandpathLength() {
 time += deltaTime;
 pathLength += deltaTime*ssp.C(position);
 absorbtion = pathLength * 1e-3;
}

/*
* This method calcultes the vector that expresses the change from current position
* to the next.
*/
private Vec3d deltaPosition() {
 temp.scale(prevdeltaArc, prevNormal);
 return temp;
}

/*
```

```
* This method calculates the midpoint between the two most recent points in
* the ray path.
*/
private void updatestarPosition() {
 temp.scale(prevdeltaArc/2.0,prevNormal);
 starPosition.add(prevPosition,temp);
}

/*
* This method sets the x position of the ray
*/
private void setPositionX(double x) {
 position.set(0,x);
}

/*
* This method sets the y position of the ray
*/
private void setPositionY(double y) {
 position.set(1,y);
}

/*
* This method sets the z position of the ray
*/
private void setPositionZ(double z) {
 position.set(2,z);
}

/*
* Maximum number of points of of information along the ray
* path.
*/
private static final int MAX_POINTS = 100;


/*
* Handle to the sound speed profile object
*/
private SSP     ssp =           null;

/*
* Handle to the bottom object
*/
private Bottom  bottom =        null;

/*
* Handle to the surface object
*/
private Surface  surface =      null;

/*
* Starting position of the ray
*/
private Vec3d   startPosition =    null;

/*
* Position of the ray after the last iteration
*/
private Vec3d   prevPosition =     null;

/*
* Position of the ray in this iteration
```

```
*/
private Vec3d    position =          null;

/*
* Change in position in this iteration
*/
private Vec3d    deltaPosition =     null;

/*
* Position halfway between position and
* position+deltaPosition
*/
private Vec3d    starPosition =      null;

/*
* Position of the trailing edge of the ray segment, not
* necessarily the prevPosition
*/
private Vec3d    trailingPosition =  null;

/*
* Direction of ray propagation in the last iteration
*/
private Vec3d    prevNormal =        null;

/*
* Direction of ray propagation in the current iteration
*/
private Vec3d    normal =            null;

/*
* Change in path length in the current iteration
*/
private double   deltaArc =          2.0;

/*
* Change in path length in the previous iteration
*/
private double   prevdeltaArc =      0.0;

/*
* Simulation time for the ray
*/
private double   time =              0.0;

/*
* Number of points saved for later use
*/
private int      pointCount =        0;

/*
* Total curavature of the ray path, used to
* decide when a point should automatically be saved
*/
private double   totalCurve =        0.0;

/*
* Indicates if a surface or bottom reflection has occurred
*/
private boolean  reflection =        false;

/*
* Angular elevation of the ray from downward (Y-direction)
```

```
 * toward the (X-Z plane). In the RRA coordinate system
 */
private double   elevation =        85.0;

/*
 * Angular displacement of the ray in the (X-Z plane)
 * about the (+Y-axis)
 */
private double   azimuth =          0.0;

/*
 * Change in time in one iteration
 */
private double   deltaTime =        0.006;

/*
 * Number of deltaTime segments from the beginning
 * of the ray segment to the trailing edge
 */
private int      pulseDuration =    0;

/*
 * Total relaxation absorbtion along the ray path
 */
private double   absorbtion =       0.0;

/*
 * Total path length that the ray traverses
 */
private double   pathLength =       0.0;

/*
 * Total phase changes due to reflection
 */
private double   reflectionPhase =  0.0;

/*
 * Storage of the positions of the ray at the desired points
 * along the ray path
 */
private Vec3d[]  rayPath =      new Vec3d[MAX_POINTS];

/*
 * Storage of the trailing edge positions at the desired
 * points along the ray path
 */
private Vec3d[]  trailingPath = new Vec3d[MAX_POINTS];

/*
 * Storage of simulation time at the desired points along
 * the ray path
 */
private double[] rayTime =      new double[MAX_POINTS];

/*
 * Variable allocated at instanciation to speed execution
 * so that we do not have to wait for dynamic allocation
 * at every iteration
 */
private Vec3d    gradc =        new Vec3d();

/*
 * Temporary variable allocated at instanciation so that
* time is not wasted allocating it dynamically.
*/
private Vec3d term1 =          new Vec3d();

/*
* Temporary variable allocated at instanciation so that
* time is not wasted allocating it dynamically.
*/
private Vec3d term2 =          new Vec3d();

/*
* Temporary variable allocated at instanciation so that
* time is not wasted allocating it dynamically.
*/
private Vec3d temp =           new Vec3d();

/*
* Temporary variable allocated at instanciation so that
* time is not wasted allocating it dynamically.
*/
private double temp3;



}
```

F. SSP.JAVA

```
/*
 File:          SSP.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/SSP.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/SSP.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>*  This class simulates the characteristics of the sound speed
 *  of the volume of ocean under simulation.
 *
 *<dt><b>Explanation:</b>
 *<dd>  This class simulates five standard profiles:
 *    1) no gradient - constant sound velocity of 1500 m/s
*   2) positive gradient - constantly increasing sound velocity
*      starting with 1500 m/s and having a slope of .017 m/s/m
*   3) negative gradient - constantly decreasing sound velocity
*      starting with 1500 m/s and having a slope of .017 m/s/m
*   4) parabolic - velocity profile obeying
*               1490 + 4e-5 *(500 - depth)^2
*   5) traditional - velocity profile obeying
*                    at 0m    1500 m/s
*                  0m -  100m  + slope of .016 m/s/m
*               1000m - 2000m  - slope of .02956 m/s/m
*                   > 2000m    + slope of .017 m/s/m<P>
*      this profile froms the typical deep sound channel
*
*<dt><b>History:</b>
*<dd>             25Oct97  /Timothy M. Holliday        /New
*<dd>             17Mar98 /Timothy M. Holliday         /Added HTML comment convention
*
*/
public class SSP {

 /**
  * Constructor for the SSP class.  Its argument is one the the following
  * five sound speed profile types:  'constant', '+gradient', '-gradient',
  * 'parabolic' and 'traditional'
  */
 public SSP (String sspType)  {
  if  (sspType.equalsIgnoreCase("constant")) {
    environment = 1;
  }
  else if (sspType.equalsIgnoreCase("+gradient")) {
    environment = 2;
  }
  else if (sspType.equalsIgnoreCase("-gradient")) {
    environment = 3;
  }
  else if (sspType.equalsIgnoreCase("parabolic")) {
    environment = 4;
  }
  else if (sspType.equalsIgnoreCase("traditional")) {
    environment = 5;
  }
  else {
    System.out.println("User defined not implemented yet, defaulting to constant");
    environment = 1;
  }
 }

 /**
  * Method used to calculate the gradient of sound speed for each of the
  * five standard profiles
  */
 public Vec3d gradC(Vec3d starPosition) {

  switch (environment) {
   case 1:  gradient = gradC1(starPosition); break;
   case 2:  gradient = gradC2(starPosition); break;
   case 3:  gradient = gradC3(starPosition); break;
   case 4:  gradient = gradC4(starPosition); break;
   case 5:  gradient = gradC5(starPosition); break;
  }
  return gradient;
 }
```

```
/**
* Method used to calculate the gradient of sound speed from any sound
* speed profile.
*/
public Vec3d generalgradC(Vec3d starPosition) {

  position2.add(starPosition,deltax);
  position1.sub(starPosition,deltax);
  xgradient = (C(position2)-C(position1))/delta;
  position2.add(starPosition,deltay);
  position1.sub(starPosition,deltay);
  ygradient = (C(position2)-C(position1))/delta;
  position2.add(starPosition,deltaz);
  position1.sub(starPosition,deltaz);
  zgradient = (C(position2)-C(position1))/delta;

  gradient.set(xgradient,ygradient,zgradient);

  return gradient;
}

/**
* Method used to calculate the speed of sound for each of the five
* standard profiles.
*/
public double C(Vec3d position) {

  switch (environment) {
    case 1:  soundSpeed = SSP1(position); break;
    case 2:  soundSpeed = SSP2(position); break;
    case 3:  soundSpeed = SSP3(position); break;
    case 4:  soundSpeed = SSP4(position); break;
    case 5:  soundSpeed = SSP5(position); break;
  }
  return soundSpeed;
}

/*******************************************************
**
** Private setion begins here
**
*******************************************************/
private double SSP1(Vec3d position) {
  return 1500.0;
}

private double SSP2(Vec3d position) {
  return (1500.0+0.017*position.get(1));
}

private double SSP3(Vec3d position) {
  return (1500.0-0.017*position.get(1));
}

private double SSP4(Vec3d position) {
  return (1490.0 + 4.0e-5 * (500.0 - position.get(1))*(500.0 - position.get(1)));
}

private double SSP5(Vec3d position) {

  if ( position.get(1) <= 100) {
    soundSpeed = 1500.0+0.016*position.get(1);
  }
  else if ( position.get(1) <= 1000) {
   soundSpeed = 1501.6 - 0.02956 * (position.get(1) - 100);
  }
  else
  {
   soundSpeed = 1475.0 + 0.03 * (position.get(1) -1000);
  }
  return soundSpeed;
 }

 private Vec3d gradC1(Vec3d position) {
  gradient.set(0,0,0);
  return gradient;
 }

 private Vec3d gradC2(Vec3d position) {
  gradient.set(0,+0.017,0);
  return gradient;
 }

 private Vec3d gradC3(Vec3d position) {
  gradient.set(0,-0.017,0);
  return gradient;
 }

 private Vec3d gradC4(Vec3d position) {
  gradient.set(0,4.0e-5 * -2 *(500.0 - position.get(1)),0);
  return gradient;
 }

 private Vec3d gradC5(Vec3d position) {

  if ( position.get(1) <= 100) {
   gradient.set(0,0.016,0);
  }
  else if ( position.get(1) <= 1000) {
   gradient.set(0,-0.02956,0);
  }
  else
  {
   gradient.set(0,0.017,0);
  }
  return gradient;
 }

 /** standard environment that was initialized */
 private int   environment = 1;

 /** class constant defining incrementally small variation in x direction*/
 private final Vec3d  deltax = new Vec3d(0.01, 0.0, 0.0);
 /** class constant defining incrementally small variation in y direction*/
 private final Vec3d  deltay = new Vec3d(0.0, 0.01, 0.0);
 /** class constant defining incrementally small variation in z direction*/
 private final Vec3d  deltaz = new Vec3d(0.0, 0.0, 0.01);
 /** class constant defining incrementally small variation in magnitude in any direction*/
 private final double delta = 0.02;

 /** class variable used in calculating speed of sound */
 private double soundSpeed;

 /** class variable used in calculating the gradient of sound speed*/
 private Vec3d  position1 = new Vec3d();
 /** class variable used in calculating the gradient of sound speed*/
```

```
  private Vec3d  position2 = new Vec3d();
  /** class variable used in calculating the gradient of sound speed*/
  private Vec3d  gradient = new Vec3d();
  /** class variable used in calculating the gradient of sound speed*/
  private double xgradient;
  /** class variable used in calculating the gradient of sound speed*/
  private double ygradient;
  /** class variable used in calculating the gradient of sound speed*/
  private double zgradient;
}
```

G. SURFACE.JAVA

```
/*
 File:          Surface.java
 Compiler:      jdk1.1.6
 */
package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A
HREF="http://www.stl.nps.navy.mil/~auv/holliday">http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Surface.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Surface.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>*  This class acoustically simulates the characteristics of
 *  the ocean's surface.  It is simulated as being a pressure release
 *  surface with no roughness
 *
 *<dt><b>Explanation:</b>
 *<dd>Since a ray is the normal vector to a plane wave, rays interact with
 *  surfaces the same way that plane waves do.  Rays obey Snell's law and
 *  the equations of reflection and transmission of wave energy.  This model
 *  uses vector algebra to determine when the end of a ray has passed through
 *  the bottom and to reflect any rays that have penetrated.  Since the
 *  surface is being approximated, to a good degree of accuracy, as a perfect
 *  pressure release surface any wave striking it will have a 180 degree
 *  phase shift and a reflection coefficient of one.<P>
 *
 *<dt><b>History:</b>
 *<dd>          8Nov97  /Timothy M. Holliday       /New
 *<dd>          18Mar98 /Timothy M. Holliday       /Added HTML comment convention
 *
 *@see Ray
 *@see Bottom
 *@see Vec3d
 */
```

```
public class Surface {

 /**
 * Constructor for the surface class. Its argument has one choice, smooth.
 * Smooth indicates a perfect pressure release surface.
 */
 public Surface (String surfaceType) {
  if (surfaceType.equalsIgnoreCase("smooth")) {
   environment = 1;
  }
  else {
    System.out.println("User defined not implemented yet, defaulting to smooth");
    environment = 1;
  }
 }

 /**
 * This procedure checks to see if the ray path has crossed the surface
 * during the current time step and returns true if it did.
 */
 public boolean  intersected(Vec3d Pos) {

  intersected = false;
  switch (environment) {
   case 1:
    if (Pos.get(1) < surfaceDepth) {
     intersected = true;
    }
    break;
  }
  return intersected;
 }

 /**
 * This method causes a Snell's Law reflection to occur at the surface of the ocean.
 */
 public double reflect(Vec3d Pos,Vec3d normal) {
  switch (environment) {
   case 1:
    double diff;
    diff = Pos.get(1) - surfaceDepth;
    Pos.set(1,surfaceDepth-diff);
    normal.set(1,-normal.get(1));
    break;
  }
  return Math.PI;
 }

/**
* This method returns a VRML string representing the Surface of the ocean.
*/
 public String VRMLSurface() {
  int minX = -5000;
  int minZ = -5000;
  int maxX = 5000;
  int maxZ = 5000;

  return "Transform {" + appendage +
            "  rotation 1 0 0 3.14" + appendage +
            "  children" + appendage +

            "Shape {" + appendage +
            "  appearance Appearance {" + appendage +
                    "   material Material {" + appendage +
                    "     emissiveColor 0 0 .4" + appendage +
                    "     diffuseColor 0 0 0" + appendage +
                    "     transparency .9" + appendage +
                    "   }" + appendage +
                    "  }" + appendage +
                    "  geometry IndexedFaceSet {" + appendage +
                    "   solid FALSE" + appendage +
                    "   coord Coordinate {" + appendage +
                    "     point [" + appendage +
                    "       "+ maxX +" "+ surfaceDepth +" "+ maxZ +"," + appendage +
                    "       "+ maxX +" "+ surfaceDepth +" "+ minZ +"," + appendage +
                    "       "+ minX +" "+ surfaceDepth +" "+ minZ +"," + appendage +
                    "       "+ minX +" "+ surfaceDepth +" "+ maxZ +"" + appendage +
                    "     ]" + appendage +
                    "   }" + appendage +
                    "   coordIndex [  0,1,2,3 ]" + appendage +
                    "  }" + appendage +
                    "}" + appendage +
                    "}" + appendage;
  }

  /**
  * This is a static method used to indicate whether a line feed is desired
  * at the end of every line.
  *
  * 'true' indicates a linefeed is desired and 'false' indicates that a space
  * is desired"
  */
  public static void setAppendLineFeed(boolean pAppendLineFeed) {
    appendLineFeed = pAppendLineFeed;
    if (appendLineFeed) {
      appendage = LINE_FEED;
    }
    else {
      appendage = SPACE;
    }
  }

  /**
  * This is a static method that returns the current line appendage.
  */
  public static boolean getAppendLineFeed(){
    return appendLineFeed;
  }

/*****************************************************************
**                    Private Section
*****************************************************************/
  private static final String LINE_FEED = "\n";
  private static final String SPACE = " ";
  private static boolean appendLineFeed = true;
  private static String appendage = LINE_FEED;

  /** standard environment that was initialized */
  private int   environment = 1;

  /** seems useless, but allows for variations due to sea state*/
  private double surfaceDepth = 0;

  /**  temporary boolean variable indicating if a reflection has occured*/
  private boolean intersected;
```

```
}
```

H. TARGETS.JAVA

```
/*
 File:          Targets.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Targets.java">
 * http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Targets.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>This class tracks the positions of targets in the VRML scene.
 * It provides for random target generation and a method to see if
 * a sonar ping has collided with a target.
 *
 *<dt><b>Explanation:</b>
 *<dd>As acoustic waves traverse the ocean environment they impinge
 * on objects. The bottom and suface are constraints on the water
 * colomn extent and as such they are handled in special classes
 * by themselves. All other objects, those in the water column,
 * are considered targets. When the sonar hits the target, energy
 * is scattered off the target in a mainly isotropic manner. This
 * scattered energy is what returns to the sending sonar giving
 * away the location of the target. In the simulation the plane
 * forming the front of the wave front is compared to the distance
 * of the target awar from the plane by means of a vector
 * algebra clculation. When this distance changes sign, there
 * is a possibility that the sonar pulse hit the object, and
 * continued on its way. Further checks are then done to verify
 * whether or not a detection event has occurred. The corresponding
 * information is then made available to the calling method as
 * the return value of a method. This class also provides a method
 * for randomly generating sample targets and is the logical place
 * to add code to receive position information from external sources,
 * like the internet via DIS.<P>
 *
 *<dt><b>History:</b>
 *<dd>          30Jan98 /Timothy M. Holliday        /New
 *<dd>          17Mar98 /Timothy M. Holliday        /Added HTML comment convention
 *<dd>          13May98 /Timothy M. Holliday        /New Constructor for placing a
 *                       minefield. New statistic method.
 *
 *@see Beam
 */
public class Targets {

 /**
 * Constructor for the targets class.  The first two arguments supply the surface
 * and bottom types.  The second two indicate how many mines  and submarines to add
 * to the target data base.  The mines+submarines are limited to 100 and they are
 * randomly placed.
 */
 public Targets (Surface s,Bottom b, int numberMines, int numberSubs)  {
  int counter;
  double x,y,z;

  skipCount = 0;
  surfaceNormal = new Vec3d();
  position = new Vec3d();
  segment1 = new Vec3d();
  segment2 = new Vec3d();

  numberofMines = numberMines;
  numberofSubs = numberSubs;

  if ((numberofMines+numberofSubs)>=MAX_TARGETS) {
   System.out.println("Too many Targets: " + (numberofMines+numberofSubs));
  }
  else {
   for (counter=0;counter<numberofMines/2;counter++) {
    x = Math.random()*2500; //in ziomek coordinate system
    z = Math.random()*4800+200;
    y = Math.random()*b.depth(x,z);
    target[counter] = new Vec3d(x,y,z);
    targetSize[counter][0] = 1.0;  // x dimension
    targetSize[counter][1] = 1.0;  // y dimension
    targetSize[counter][2] = 1.0;  // z dimension
    targetArea[counter] = 3.1415;
    targetDistance[counter] = 0.0;
   }
   for (;counter<numberofMines;counter++) {
    x = Math.random()*2500; //in ziomek coordinate system
    z = Math.random()*-4800-200;
    y = Math.random()*b.depth(x,z);
    target[counter] = new Vec3d(x,y,z);
    targetSize[counter][0] = 1.0;  // x dimension
    targetSize[counter][1] = 1.0;  // y dimension
    targetSize[counter][2] = 1.0;  // z dimension
    targetArea[counter] = 3.1415;
    targetDistance[counter] = 0.0;
   }
   for (;counter<numberofSubs+numberofMines;counter++) {
    x = Math.random()*-1000;
    z = Math.random()*800-400;
    y = Math.random()*b.depth(x,z);
    target[counter] = new Vec3d(x,y,z);
    targetSize[counter][0] = 100.0;  // x dimension
    targetSize[counter][1] = 10.0;  // y dimension
    targetSize[counter][2] = 10.0;  // z dimension
    targetArea[counter] = 3.1415*100;
    targetDistance[counter] = 0.0;
   }
  }
 }
```

```
/**
* Constructor for the tartgets class.  The first two arguments are
* the bottom and surface objects.  The next two are the max and min
* x-direction boundaries for a minefield and the next two are for
* the z-direction.  The last is the number of mines.  The maaximum
* number of mines is 100 and they are randomly placed.
*/
public Targets (Surface s,
              Bottom b,
              double maximumX,
              double minimumX,
              double maximumZ,
              double minimumZ,
              int numberMines)  {
  int counter;
  double x,y,z;

  skipCount = 0;
  surfaceNormal = new Vec3d();
  position = new Vec3d();
  segment1 = new Vec3d();
  segment2 = new Vec3d();

  numberofMines = numberMines;
  numberofSubs = 0;

  if ((numberofMines+numberofSubs)>=MAX_TARGETS) {
    System.out.println("Too many Targets: " + (numberofMines+numberofSubs));
  }
  else {
    for (counter=0;counter<numberofMines;counter++) {
      x = Math.random()*(maximumX-minimumX)+minimumX; //in ziomek coordinate system
      z = Math.random()*(maximumZ-minimumZ)+minimumZ;
      y = Math.random()*b.depth(x,z);
      target[counter] = new Vec3d(x,y,z);
      targetSize[counter][0] = 1.0;  // x dimension
      targetSize[counter][1] = 1.0;  // y dimension
      targetSize[counter][2] = 1.0;  // z dimension
      targetArea[counter] = 3.1415;
      targetDistance[counter] = 0.0;
      numberOfDetects[counter] = 0;
      estimatedPosition[counter] = new Vec3d();
    }
  }
}

/**
* This method resets the parameters that speed up target collision detection.
* It is used whenever a new beam is calculated.
*/
public void resetTargets() {
   skipCount = 0;
   for (counter=0;counter<numberofSubs+numberofMines;counter++) {
     targetDistance[counter] = 0.0;
   }
 }

/**
* This method returns a VRML string representing the targets in the ocean.
*/
public String VRMLTargets() {
  int counter;
  String temporaryString;

  temporaryString = "EXTERNPROTO ContactMine [\n" +
              "  exposedField SFVec3f translation ]\"ContactMine.wrl\"\n" +

              "EXTERNPROTO DieselSub [\n" +
              "  exposedField SFVec3f translation ]\"DieselSub.wrl\"\n" +

              "Transform {\n" +
              "  children [\n";
  for(counter=0;counter<numberofMines;counter++){
    temporaryString += "    ContactMine {translation " + (int)target[counter].get(0) + " "
                                    + (-(int)target[counter].get(1)) + " "
                                    + (-(int)target[counter].get(2)) + " "
                                    + "}\n";
  }
  for(;counter<numberofMines+numberofSubs;counter++){
    temporaryString += "    DieselSub {translation " + (int)target[counter].get(0) + " "
                                    + (-(int)target[counter].get(1)) + " "
                                    + (-(int)target[counter].get(2)) + " "
                                   + "}\n" ;
  }
  return temporaryString += "  ]\n" +
                    "}\n";
}

/**
* This method returns a boolean indicating whether or not a detection as occurred.
* True indicates a detection and false indicates no detection.
*/
public boolean isCollision(Ray ray1,
                    Ray ray2,
                    Ray ray3,
                    Ray ray4) {
  collisionCount = 0;
  collision = false;

  // speed up detection by skipping distance between targets
  if (skipCount > 0) {
    skipCount--;
  }
  else {
    //reset the minimum distance to a large value
    minDistance = 100000;

    //calculate the surface normal
    segment1.sub(ray2.getPosition(),ray1.getPosition());
    segment2.sub(ray4.getPosition(),ray1.getPosition());
    surfaceNormal.cross(segment1,segment2);
    surfaceNormal.normalize();
    collisionArea = 0.0;


    for (counter=0;counter<(numberofMines+numberofSubs);counter++) {

      // determine the distance of the pulse from the target
      position.sub(target[counter],ray1.getPosition());
      distance = position.dot(surfaceNormal);

      // if the product of the distance in the last step and the
      // current step is negative then the beam may have passed
      // through the object
      if (targetDistance[counter]*distance < 0) {
        collision = possibleCollision(ray1,
                                      ray2,
                                      ray3,
                                      ray4);
        if (collision) {
          numberOfDetects[counter]++;
          estimatedPosition[counter].add(ray1.getPosition(),ray3.getPosition());
          estimatedPosition[counter].scale(.5);
          collisionArea += targetArea[counter];
        }
        else {
        }

      }
      targetDistance[counter] = distance;

      // save the distance if it is the minimum for this time step
      if (Math.abs(distance) < minDistance) {
        minDistance = Math.abs(distance);
      }
    }

    // calculate the number of time steps that looking for detections can
    // skipped from the minimum distance to the next targer,time step and
    //average sound speed
    skipCount = (int)(minDistance/12.0);
  }

  return collision;
}

/**
* This method returns the cross sectional area of the object that was detected.
*/
public double getCollisionArea() {
  return collisionArea;
}

/**
* This method prints the mine number, position, estimated position and
* number of detects on each mine in the data base. The numbers are
* printed to the standard console.
*/
public void printMineStatistics() {
  int i;
  for(i=0;i<numberofMines;i++) {
    System.out.println(
      "Mine "+i+
      " Position "+target[i].get(0)+" "+
            target[i].get(1)+" "+
            target[i].get(2)+
      " Estimate "+estimatedPosition[i].get(0)+" "+
            estimatedPosition[i].get(1)+" "+
            estimatedPosition[i].get(2)+
      " Detects "+numberOfDetects[i]);
  }
}

/*****************************************************************
**
**              Private Section
**
```

```
*****************************************************************/

/*
 * This method returns a boolean indicating that a possible interaction between
 * the sonar pulse and another registered object has occurred.
 */
private boolean possibleCollision(Ray ray1,
                                  Ray ray2,
                                  Ray ray3,
                                  Ray ray4) {

  if ( inBox(ray1.getTrailingPosition(),ray3.getPosition()) &&
      inBox(ray2.getTrailingPosition(),ray4.getPosition()) &&
      inBox(ray3.getTrailingPosition(),ray1.getPosition()) &&
      inBox(ray4.getTrailingPosition(),ray2.getPosition())) {
      collisionCount++;
      return true;
  }
  else {
    return false;
  }
}

/*
 * This method returns a boolean indicating whether or not the any part
 * of the object is within the parallelpiped formed by the sonar pulse
 */
private boolean inBox(Vec3d corner1, Vec3d corner2) {
  if ( inInterval(corner1.get(0),corner2.get(0), target[counter].get(0)) &&
      inInterval(corner1.get(1),corner2.get(1), target[counter].get(1)) &&
      inInterval(corner1.get(2),corner2.get(2), target[counter].get(2))) {
    return true;
  }
  else {
    return false;
  }
}

/*
 * This method returns a boolean indicating whether or not the any part
 * of the object is within the extent of the pulse in one of the three
 * cartesian directions.
 */
private boolean inInterval(double pos1, double pos2, double target) {
  if ( ((pos1 < target) && (target < pos2)) ||
      ((pos2 < target) && (target < pos1))) {
    return true;
  }
  else {
    return false;
  }
}

private static final int MAX_TARGETS = 100;
private Vec3d target[] =  new Vec3d[MAX_TARGETS];
private double targetSize[][] = new double[MAX_TARGETS][3];
private int numberOfDetects[] = new int[MAX_TARGETS];
private Vec3d estimatedPosition[] = new Vec3d[MAX_TARGETS];

private double collisionArea = 0.0;
private double targetArea[] = new double[MAX_TARGETS];
```

```
    private int numberofMines;
    private int numberofSubs;
    private boolean collision = true;
    private int counter;

    private int collisionCount = 0;
    private Vec3d surfaceNormal = null;
    private Vec3d position = null;
    private Vec3d segment1 =  null;
    private Vec3d segment2 =  null;
    private double distance = 0.0;
    private double targetDistance[] = new double[MAX_TARGETS];
    private int skipCount;
    private double minDistance;

}
```

APPENDIX B. EXAMPLE APPLICATIONS

A. EXAMPLEBEAMDYNAMIC.JAVA

```
/*
 File:          ExampleBeamDynamic.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;
import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleBeamDynamic.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleBeamDynamic.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>ExampleBeamStatic:</b>
 *<dd><a href="ExampleBeamDynamic.wrl"><IMG SRC="images/ExampleBeamDynamic.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Produces an example VRML scene showing a beam.
 *
 *<dt><b>Explanation:</b>
 *<dd>Shows a VRML scene with a surface, bottom and beam. The beam
 *   show the volume traced out by the pulse during its transmission.<P>
 *
 *<dt><b>History:</b>
 *<dd>          15Apr97 /Timothy M. Holliday        /New
 *
 *@see Beam
 */
public class ExampleBeamDynamic {

/**
* This method initializes the environment and the sonar Lobe and
* has the VRML reprsentation of each written to the console.
*/
  public static void main(String[] args) {

    SSP  ssp   = new SSP("traditional");
    Bottom bott = new Bottom("slope",2000);
    Surface surf = new Surface("smooth");
    Beam beam = new Beam();
    beam.setElevation(80);
    beam.setAzimuth(20);
    beam.setHalfBeamWidthY(2);
    beam.setHalfBeamWidthX(2);
    beam.setPosition(-1000, 50, 0);
    beam.setDuration(1);
    beam.setBottom(bott);
    beam.setSurface(surf);
    beam.setSsp(ssp);
    beam.setEndTime(8);
    beam.reset();

    // Create object with no targets for methods requiring a target
    // as a parameter
    Targets targets = new Targets(surf,bott,0,0);

    // record start time
    Date timecheck = new Date();

    // print out a VRML header
    System.out.println("#VRML V2.0 utf8");
    System.out.println("\nInline{url[\"Header.wrl\"]}\n");

    // call each class' VRML print routine
    System.out.println(bott.VRMLBottom());
    System.out.println(surf.VRMLSurface());

    System.out.println(beam.pingTimerVRML());
    beam.calculateBeam(targets);
    beam.dynamicVRML();

    // record stop time
    Date timecheck2 = new Date();

    System.out.println( "# " + timecheck );
    System.out.println( "# " + timecheck2 );

  }
}
```

B. EXAMPLEBEAMSTATIC.JAVA

```
/*
 File:          ExampleBeamStatic.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;
import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleBeamStatic.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleBeamStatic.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>ExampleBeamStatic:</b>
 *<dd><a href="ExampleBeamStatic.wrl"><IMG SRC="images/ExampleBeamStatic.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Produces an example VRML scene showing a beam.
 *
 *<dt><b>Explanation:</b>
 *<dd>Shows a VRML scene with a surface, bottom and beam. The beam
 *   show the volume traced out by the pulse during its transmission.<P>
 *
 *<dt><b>History:</b>
 *<dd>            15Apr97 /Timothy M. Holliday        /New
 *
 *@see Beam
 */
public class ExampleBeamStatic {

/**
* This method initializes the environment and the sonar Beam and
* has the VRML representation of each written to the console.
*/
 public static void main(String[] args) {

   Vec3d pos = new Vec3d( -1000, 50, 0);
   SSP  ssp  = new SSP("traditional");
   Bottom bott = new Bottom("slope",2000);
   Surface surf = new Surface("smooth");

   Beam beam = new Beam();
   beam.setElevation(80);
   beam.setAzimuth(20);
   beam.setHalfBeamWidthY(2);
   beam.setHalfBeamWidthX(2);
   beam.setPosition(-1000, 50, 0);
   beam.setDuration(1);
   beam.setBottom(bott);
   beam.setSurface(surf);
   beam.setSsp(ssp);
   beam.setEndTime(8);
   beam.reset();

   // record start time
   Date timecheck = new Date();


   PrintVRML.setColorScheme(PrintVRML.RAINBOW,40,100);
   PrintVRML.setIntensityScheme(PrintVRML.CONSTANT);

   // print out a VRML header
   System.out.println(PrintVRML.header());
   System.out.println(PrintVRML.navigationInfo());
   System.out.println(PrintVRML.directionalLight());
   System.out.println(PrintVRML.protoColorBar("TL[db]","","",""));
   System.out.println(PrintVRML.colorBarView("From the East",
                                        -500, -500, 14500,
                                        0, 0, 1, 0));
    System.out.println(PrintVRML.colorBarView("From the South",
                                        -14500, -500, 0,
                                        0, 1, 0, -1.57));
    System.out.println(PrintVRML.colorBarView("From the West",
                                        -500, -500, -14500,
                                        0, 1, 0, 3.14));
    System.out.println(PrintVRML.colorBarView("From the Air",
                                        -500, 14500, 0,
                                        1, 0, 0, -1.57));


    // call each class' VRML print routine
    System.out.println(bott.VRMLBottom());
    System.out.println(surf.VRMLSurface());
    PrintVRML.setColorScheme(PrintVRML.RAINBOW,8,0);
    PrintVRML.setIntensityScheme(PrintVRML.CONSTANT);
    System.out.println(PrintVRML.protoColorBar("TL [dB]","","",""));
    System.out.println(beam.staticVRML(Beam.T_L,Beam.NONE));
    System.out.println(PrintVRML.printLegendView("ExampleBeamStatic",
                                    "  This is an example beam.",
                                    "It has an initial elevation",
                                    "of 80 degrees from",
                                    "vertical and 20 degrees",
                                    "west of north.",
                                    -500, -500, 15000) );


    // record stop time
    Date timecheck2 = new Date();

    System.out.println( "# " + timecheck );
    System.out.println( "# " + timecheck2 );

  }
}
```

C. EXAMPLELOBEDYNAMIC.JAVA

```
/*
 File:           ExampleLobeDynamic.java
 Compiler:       jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleLobeDynamic.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleLobeDynamic.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>ExampleLobeDynamic:</b>
 *<dd><a href="ExampleLobeDynamic.wrl"><IMG SRC="images/ExampleLobeDynamic.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Produces an example VRML scene showing a lobe.
 *
 *<dt><b>Explanation:</b>
 *<dd>Shows a VRML scene with a surface, bottom and lobe. The scene
 *   show the pulse during its transmission as it travels.
 *   A lobe is composed of several individual beams.<P>
 *
 *<dt><b>History:</b>
 *<dd>              15Apr97 /Timothy M. Holliday          /New
 *
 *@see Lobe
 */
public class ExampleLobeDynamic {

/**
* This method initializes the environment and the sonar Lobe and
* has the VRML reprsentation of each written to the console.
*/
 public static void main(String[] args) {

  Vec3d pos = new Vec3d( -1000, 50, 0);
  SSP  ssp   = new SSP("traditional");
  Bottom bott = new Bottom("slope",2000);
  Surface surf = new Surface("smooth");

  Lobe lobe = new Lobe();
  lobe.setElevation(80);
  lobe.setAzimuth(20);
  lobe.setLobeWidthY(4);
  lobe.setLobeWidthX(4);
  lobe.setNumberYPartition(2);
  lobe.setNumberXPartition(2);
  lobe.setPosition(pos);
  lobe.setDuration(1);
  lobe.setBottom(bott);
  lobe.setSurface(surf);
  lobe.setSsp(ssp);
  lobe.setEndTime(8);
  lobe.reset();

  // Create object with no targets for methods requiring a target
  // as a parameter
  Targets targets = new Targets(surf,bott,0,0);

  // record start time
  Date timecheck = new Date();

  // print out a VRML header
  System.out.println(PrintVRML.header());
  System.out.println(PrintVRML.navigationInfo());
  System.out.println(PrintVRML.directionalLight());
```

```
    System.out.println("\nInline{url[\"Header.wrl\"]}\n");

    // call each class' VRML print routine
    System.out.println(bott.VRMLBottom());
    System.out.println(surf.VRMLSurface());
    lobe.calculateLobe(targets);
    System.out.println(lobe.dynamicVRML());
    System.out.println(PrintVRML.printLegendView("ExampleLobeDynamic",
                                    "  This is an example lobe.",
                                    "It is made of 4 beams",
                                    "Elevation - 80 degrees",
                                    "from vertical and 20",
                                    "degrees west of north.",
                                    -500, -500, 15000) );

    // record stop time
    Date timecheck2 = new Date();

    System.out.println( "# " + timecheck );
    System.out.println( "# " + timecheck2 );

  }
}
```

D. EXAMPLELOBESTATIC.JAVA

```
/*
 File:          ExampleLobeStatic.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleLobeStatic.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleLobeStatic.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>ExampleLobeStatic:</b>
 *<dd><a href="ExampleLobeStatic.wrl"><IMG SRC="images/ExampleLobeStatic.jpg" ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Produces an example VRML scene showing a lobe.
 *
 *<dt><b>Explanation:</b>
```

```
*<dd>Shows a VRML scene with a surface, bottom and lobe. The lobe
*   show the volume traced out by the pulse during its transmission.
*   A lobe is composed of several individual beams.<P>
*
*<dt><b>History:</b>
*<dd>              15Apr97 /Timothy M. Holliday          /New
*
*@see Lobe
*/
public class ExampleLobeStatic {

/**
* This method initializes the environment and the sonar Beam and
* has the VRML representation of each written to the console.
*/
 public static void main(String[] args) {

   Vec3d pos = new Vec3d( -1000, 50, 0);
   SSP  ssp   = new SSP("traditional");
   Bottom bott = new Bottom("slope",2000);
   Surface surf = new Surface("smooth");
   Targets targets = new Targets(surf,bott,0,0);

   Lobe lobe = new Lobe();
   lobe.setElevation(80);
   lobe.setAzimuth(20);
   lobe.setLobeWidthY(4);
   lobe.setLobeWidthX(4);
   lobe.setNumberYPartition(5);
   lobe.setNumberXPartition(5);
   lobe.setPosition(pos);
   lobe.setDuration(1);
   lobe.setBottom(bott);
   lobe.setSurface(surf);
   lobe.setSsp(ssp);
   lobe.setEndTime(8);
   lobe.reset();
   lobe.calculateLobe(targets);

   // record start time
   Date timecheck = new Date();


   // print out a VRML header
   System.out.println(PrintVRML.header());
   System.out.println(PrintVRML.navigationInfo());
   System.out.println(PrintVRML.directionalLight());
   System.out.println(bott.VRMLBottom());
   System.out.println(surf.VRMLSurface());

/*  System.out.println("#Scheme 1");
   PrintVRML.setColorScheme(PrintVRML.RAINBOW,8,0);
   PrintVRML.setIntensityScheme(PrintVRML.CONSTANT);
   System.out.println(PrintVRML.protoColorBar("Time [sec]","","",""));
   System.out.println(lobe.staticVRML(Beam.TIME,Beam.NONE));

   System.out.println("#Scheme 2");
   PrintVRML.setColorScheme(PrintVRML.RAINBOW,40,100);
   PrintVRML.setIntensityScheme(PrintVRML.CONSTANT);
   System.out.println(PrintVRML.protoColorBar("TL[dB]","","",""));
   System.out.println(lobe.staticVRML(Beam.T_L,Beam.NONE));
*/
   System.out.println("#Scheme 3");
   PrintVRML.setColorScheme(PrintVRML.RAINBOW,40,100);
   PrintVRML.setIntensityScheme(PrintVRML.LINEAR,0,8);
   System.out.println(PrintVRML.protoColorBar("TL[dB]","Time [sec]","",""));
   System.out.println(lobe.staticVRML(Beam.T_L,Beam.TIME));
/*
   System.out.println("#Scheme 4");
   PrintVRML.setColorScheme(PrintVRML.RAINBOW,8,0);
   PrintVRML.setIntensityScheme(PrintVRML.LINEAR,40, 100);
   System.out.println(PrintVRML.protoColorBar("Time [sec]","TL[dB]","",""));
   System.out.println(lobe.staticVRML(Beam.TIME,Beam.T_L));

   System.out.println("#Scheme 5");
   PrintVRML.setColorScheme(PrintVRML.TRICOLOR,40,60,80,100);
   PrintVRML.setIntensityScheme(PrintVRML.LINEAR,0,8);
   System.out.println(PrintVRML.protoColorBar("","Time [sec]","Detect","Counter Detect"));
   System.out.println(lobe.staticVRML(Beam.T_L,Beam.TIME));

   System.out.println("#Scheme 6");
   PrintVRML.setColorScheme(PrintVRML.TRICOLOR,40,60,80,100);
   PrintVRML.setIntensityScheme(PrintVRML.LINEAR,40,100);
   System.out.println(PrintVRML.protoColorBar("","TL[dB]","Detect","Counter Detect"));
   System.out.println(lobe.staticVRML(Beam.T_L,Beam.T_L));

   System.out.println("#Scheme 7");
   PrintVRML.setColorScheme(PrintVRML.RED,40,100);
   PrintVRML.setIntensityScheme(PrintVRML.LINEAR,40,100);
   System.out.println(PrintVRML.protoColorBar("","TL[db]","",""));
   System.out.println(lobe.staticVRML(Beam.NONE,Beam.T_L));
*/
   System.out.println(PrintVRML.colorBarView("From the East",
                                   -500, -500, 14500,
                                   0, 0, 1, 0));
   System.out.println(PrintVRML.colorBarView("From the South",
                                   -14500, -500, 0,
                                   0, 1, 0, -1.57));
   System.out.println(PrintVRML.colorBarView("From the West",
                                   -500, -500, -14500,
                                   0, 1, 0, 3.14));
   System.out.println(PrintVRML.colorBarView("From the Air",
                                   -500, 14500, 0,
                                   1, 0, 0, -1.57));
   System.out.println(PrintVRML.printLegendView("ExampleLobeStatic",
                               "  This is an example lobe.",
                               "It is made of 4 beams",
                               "Elevation - 80 degrees",
                               "from vertical and 20",
                               "degrees west of north.",
                               -500, -500, 15000) );


   // record stop time
   Date timecheck2 = new Date();

   System.out.println( "# " + timecheck );
   System.out.println( "# " + timecheck2 );

  }
}
```

E. EXAMPLERAY.JAVA

```
/*
 File:            ExampleRay.java
 Compiler:        jdk1.1.6
 */

package mil.navy.nps.rra;

import mil.navy.nps.rra.*;

import java.util.Date;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleRay.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/ExampleRay.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>ExampleRay:</b>
 *<dd><a href="ExampleRay.wrl"><IMG SRC="images/ExampleRay.jpg" ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Produces an example VRML scene showing a ray.
 *
 *<dt><b>Explanation:</b>
 *<dd>Shows a VRML scene with a surface, bottom and ray.<P>
 *
 *<dt><b>History:</b>
 *<dd>            15Apr97 /Timothy M. Holliday         /New
 *
 *@see Ray
 */
public class ExampleRay {

/**
* This method initializes the environment and the sonar Ray and
* has the VRML reprsentation of each written to the console.
*/
 public static void main(String[] args) {


   SSP  ssp   = new SSP("traditional");
   Bottom bott = new Bottom("slope",2000);
   Surface surf = new Surface("smooth");

   Ray ray = new Ray();
   ray.setPosition( -1000, 50, 0);
   ray.setElevation(80);
   ray.setAzimuth(20);
   ray.setDeltaTime(.006);
ray.setDuration(1);
ray.setBottom(bott);
ray.setSurface(surf);
ray.setSsp(ssp);
ray.reset();

int j = 0;

// record start time
Date timecheck = new Date();

// print out a VRML header
System.out.println(PrintVRML.header());
System.out.println(PrintVRML.navigationInfo());
System.out.println(PrintVRML.directionalLight());
System.out.println("\nInline{url[\"Header.wrl\"]}\n");


// call each class' VRML print routine
System.out.println(bott.VRMLBottom());
System.out.println(surf.VRMLSurface());

// propagate the ray and print it
System.out.println("Transform {" );
System.out.println("  rotation 1 0 0 3.14" );
System.out.println("  children" );

System.out.println("Shape {" );
System.out.println("  appearance Appearance {");
System.out.println("    material Material {");
System.out.println("      emissiveColor 1.0 0.0 0.0");
System.out.println("      diffuseColor  0.0 0.0 0.0");
System.out.println("    }");
System.out.println("  }");
System.out.println("  geometry IndexedLineSet {");
System.out.println("    coord Coordinate {");
System.out.println("      point [ ");
j = 0;

// Record first point always
ray.recordPoint();
j++;

while (ray.getTime() < 8.0) {

  //  Propagate all rays through one time step
  ray.Propagate(.006);


  //  If any ray has reflected then record all of the
  //  points in the beam
  if ( ray.reflected()) {
     ray.recordPoint();
     j++;
  }

  //  If the curvature sum reaches the limit, record
  //  all of the points in the beam
  if ( ray.totalCurvature() > .022 ) {
     ray.recordPoint();
     j++;
  }
}
```

```
  // Record last point always
  ray.recordPoint();

  // Print the positions stored in each ray
  for (j=0; j<ray.getCount(); j += 1) {
    System.out.print(ray.position(j));
  }

  System.out.println("      ]");
  System.out.println("    }");
  System.out.println("    coordIndex [");

  // Define the segments of the ray
  for(j=0; j<ray.getCount(); j++) {
    System.out.println(j);
  }
  System.out.println("    ]");
  System.out.println("  }");
  System.out.println("}");
  System.out.println("}");

  System.out.println(PrintVRML.printLegendView("ExampleRay",
                        "  This is an example ray.",
                        "It has an initial elevation",
                        "of 80 degrees from",
                        "vertical and 20 degrees",
                        "west of north.",
                        -500, -500, 15000) );


  // record stop time
  Date timecheck2 = new Date();

  System.out.println( "# " + timecheck );
  System.out.println( "# " + timecheck2 );

 }
}
```

F. PINGER.JAVA

```
/*
 File:          Pinger.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import java.util.Date;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Pinger.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Pinger.java</a>
```

```
*
*<dt><b>Hierarchy Diagram:</b>
*<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
*
*<dt><b>Coordinate System Diagram:</b>
*<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
*
*<dt><b>Summary:</b>
*<dd>This stand alone program initializes the environment (bottom, surface and
* SSP) and the sonar array radiation lobe.
*
*<dt><b>Explanation:</b>
*<dd>This program provides a template for running a stand-alone simulation
*   that can run without user interaction to generate sonar pings and returns for a
*   virtual world.<P>
*
*<dt><b>History:</b>
*<dd>          15Jan98  /Timothy M. Holliday       /New
*<dd>          17Mar98 /Timothy M. Holliday       /Added HTML comment convention
*
*@see Lobe
*/
public class Pinger {

  public static final double MINIMUMX = -5000;
  public static final double MINIMUMZ = 4000;
  public static final double MAXIMUMZ = 5000;
  public static final double MAXIMUMX = -4000;
  public static final double NORTH = 000.0;
  public static final double SOUTH = 180.0;
  public static final double EAST  = 090.0;
  public static double deltaTime = .8; // in seconds
  public static double lastZLane = 4950;
  public static Vec3d position = null;
  public static Vec3d deltaPosition = null;
  public static Vec3d velocity = null;
  public static double orientation = 0.0;


/**
* This method initializes the environment and the sonar Lobe and
* has the VRML reprsentation of each written to the console.
*/
  public static void main(String[] args) {


   // Position velocity and orientation in Ziomek Coordinate System
   position = new Vec3d( -5000, 50, lastZLane); //in meters
   deltaPosition = new Vec3d( 0, 0, 0); //in meters
   velocity = new Vec3d( 2.8, 0, 0);  // in m/s which is 5 kts
   orientation = NORTH; // in degrees

   SSP ssp   = new SSP("constant");
   Bottom bott = new Bottom("noslope",100);
   Surface surf = new Surface("smooth");
   Lobe lobe = new Lobe();
   Targets targets = new Targets(surf,bott,MAXIMUMX,MINIMUMX,MAXIMUMZ,MINIMUMZ,50);

   lobe.setElevation(90);
   lobe.setAzimuth(0);
   lobe.setLobeWidthY(40);
```

```
    lobe.setLobeWidthX(52);
    lobe.setNumberYPartition(4);
    lobe.setNumberXPartition(4);
    lobe.setDuration(1);
    lobe.setBottom(bott);
    lobe.setSurface(surf);
    lobe.setSsp(ssp);
    lobe.setEndTime(.133); // 200m/1500m/s

    // record start time
    Date timecheck = new Date();

    while (notDoneSearching()) {
      lobe.setPosition(position);
      lobe.setAzimuth(orientation);
      lobe.reset();
//System.out.println("calculating");
      lobe.calculateLobe(targets);
      updateVehiclePosition();
    }

    targets.printMineStatistics();
    // record stop time
    Date timecheck2 = new Date();

    System.out.println( "# " + timecheck );
    System.out.println( "# " + timecheck2 );

  }

  public static boolean notDoneSearching() {
    if (position.get(2) <= MINIMUMZ) {
      return false;
    }
    else {
      return true;
    }
  }

  public static void updateVehiclePosition () {
    if (orientation == NORTH) {
      if (position.get(0) >= MAXIMUMX) {
        orientation = EAST;
        velocity.set(0,0,-2.8);
        position.print();
      }
    }
    else if (orientation == SOUTH) {
      if (position.get(0) <= MINIMUMX) {
        orientation = EAST;
        velocity.set(0,0,-2.8);
        position.print();
      }
    }
    else if (orientation == EAST) {
      if (position.get(2) <= (lastZLane-100.0)) {
        if (position.get(0) >= MAXIMUMX) {
          orientation = SOUTH;
          position.print();
          velocity.set(-2.8,0,0);
        }
        else {
          orientation = NORTH;
          velocity.set(2.8,0,0);
          position.print();
        }
        lastZLane = position.get(2);
      }
    }
    deltaPosition.set(velocity);
    deltaPosition.scale(deltaTime);
    position.add(deltaPosition);
  }

}
```

APPENDIX C. CLIENT SERVER CODE

A. BATTLESCENE.JAVA

```
/*
 File:           BattleScene.java
 Compiler:       jdk1.1.6, Netscape 3.0, WorldView 2.0
 */

import java.net.*;
import java.io.*;
import java.util.*;
import vrml.*;
import vrml.node.*;
import vrml.field.*;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/BattleScene.java">
 * http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/BattleScene.java</a>
 *
 *<dt><b>Summary:</b>
 *<dd>Interfaces between the browser and pinserver. This script also provides for
 *   platform mobility and controls all timing of the interactions between ping
 *   requests, VRML returns and platform movement.
 *
 *<dt><b>Explanation:</b>
 *<dd>A VRML world is in essence a static world, with some basic movements. To be able
 *   to emulate more real world processes the VRML world needs to have a script
 *   connection to a program running outside of the VRML rendering engine, that is
 *   running in the browsers virtual machine. This program is that extension of a
 *   particular VRML world. It also provides network connections to a sonar ping
 *   server runing on a separate computer. This network connection is provided since
 *   both rendering VRML and calculating real-time sonar are computationally intensive.
 *   Thus, the two functions are distributed two two machines over the net.<P>
 *
 *<dt><b>History:</b>
 *<dd>          15Jan98  /Timothy M. Holliday        /New
 *<dd>          17Mar98 /Timothy M. Holliday         /Added HTML comment convention
 *
 *@see BridgeServer
 *@see PingServer
 */
public class BattleScene extends Script implements Timed{

  // Browser Specific Stuff
  private Browser browser;
  ControlPanel panel;
  MFNode addChildren, removeChildren;
  MFNode addBeamChildren, removeBeamChildren;
  SFVec3f beamTranslation;
  SFFloat yaw;
  SFVec3f translation;
```

```
SFFloat beamYaw;
BaseNode nodes[] = null;
private BaseNode oldBeamNodes[][] = {null,null,null,null,null};
private BaseNode oldTargetsNodes[][] = {null,null,null,null,null};
private int activeBeamNode = 0;
private int activeTargetsNode = 0;


//Network specific stuff
public final static int PORT = 4129;
public final static String HOST = "localhost"; // server host name
public final static float SET_SONAR_PARAMETERS = (float)0.0;
public final static float DETECT_TARGETS = (float)1.0;
public final static float DETECT_BEAM = (float)2.0;
public final static float DETECT_BOTH = (float)3.0;
Socket socket = null;
DataInputStream in = null;    // input stream from server to client
DataOutputStream out = null;  // output sream from client to server

// Gereral Stuff specific to the vehicle in question
private final static boolean DEBUG = true;
private final static boolean LOCAL = false;
private final static float START_DEPTH = 500;
private final static float START_X_POS = -5000;
private final static float START_Y_POS = 0;
private float orderedSpeed = 0;
private float orderedDepth = 0;
private float orderedCourse = 0;
private float position[] = new float[3];  // naval coordinate system
private float drPosition[] = new float[3]; // naval coordinate system
private float heading = 0;
private float drHeading = 0;
private float speed = 0;
private float drSpeed = 0;
private float depth = START_DEPTH;
private float drDepth = 0;

//Timing specific stuff
private final static int interval= 100;
private int tickCount = 0;
private int pingOrderTick = 0;
public boolean createVRMLThreadRunning = false;
private float pingInterval = (float)4.0;
private CreateVRMLBeamThread beamThread = null;
private CreateVRMLTargetsThread targetsThread = null;

/**
 * This method initializes the Java script.  It establishes connections
 * to the browser, the control panel, the timer and to BridgeServer.
 */
public void initialize(){

if (DEBUG)
System.out.println("BattleScene:  initialize:  starting");

   // Connects to the browser and various VRML nodes in the browser
   browser = getBrowser();
   // get the reference to the target node.
   Node root = (Node)((SFNode)getField("root")).getValue();
   Node manta = (Node)((SFNode)getField("manta")).getValue();
   Node beam = (Node)((SFNode)getField("beam")).getValue();

   // Initiates the control panel
```

```
        panel = new ControlPanel(this);        // start the panel.

        // get the references to exposed fields, event-in/outs in the VRML nodes.
        addChildren = (MFNode)root.getEventIn("addChildren");
        removeChildren = (MFNode)root.getEventIn("removeChildren");
        addBeamChildren = (MFNode)beam.getEventIn("addChildren");
        removeBeamChildren = (MFNode)beam.getEventIn("removeChildren");
        beamTranslation = (SFVec3f)beam.getExposedField("translation");
        beamYaw = (SFFloat)beam.getEventIn("yaw");
        translation = (SFVec3f)manta.getExposedField("translation");
        position[0]= translation.getX();
        position[1]= translation.getZ(); // translate from VRML to NAVAL system
        position[2]= -translation.getY();
        yaw = (SFFloat)manta.getEventIn("yaw");

    if (!LOCAL) {
        // open socket connections and establish data streams with BridgeServer
        try {
          // open network and input/output stream
          socket = new Socket("localhost", PORT);
          in = new DataInputStream(socket.getInputStream());
          out = new DataOutputStream(socket.getOutputStream());
        } catch (UnknownHostException e) {
          browser.setDescription("Unknown host: " + HOST);
        } catch (Exception e) {
          browser.setDescription("Connection error");
        }
    }

        // initiate a callback timer
        new TimerThread(this,interval).start();

        depth = position[2];
        orderedDepth = depth;

    if (DEBUG)
    System.out.println("BattleScene:  initialize:  ending");
      }

     /**
      * This method receives ticks from TimerThread and updates vehicle
      * position in the VRML scene.
      */
     public void tick(TimerThread t) {

       if (orderedSpeed-speed > .05) {
        speed += 0.1;
       }
       else if (orderedSpeed-speed < -.05) {
        speed -= 0.1;
       }


       if (orderedCourse-heading > .05) {
        heading += 0.1;
       }
       else if (orderedCourse-heading < -.05) {
        heading -= 0.1;
       }

       position[0] += Math.cos(heading*3.14/180.0)*.1*speed;
       position[1] += Math.sin(heading*3.14/180.0)*.1*speed;
```

```
  if (orderedDepth-depth > .05) {
   depth += 0.1;
  }
  else if (orderedDepth-depth < -.05) {
   depth -= 0.1;
  }

  position[2] = depth;

  if (tickCount++%10 == 0) {
   panel.updateShipState(heading, speed, depth);
   browser.setDescription(" X=" + (int)position[0] +
                          " Y=" + (int)position[1] +
                          " Z=" + (int)position[2]);
   updateVehiclePosition();
  }
 }

 /**
 * This method calculates the position in the future that the vehicle
 * will be at when the pulse is sent out.
 *
 * Since it takes a finite time to calculate sonar pings, all requests
 * made to the server are future requests. This method calculates the
 * vehicles position at that future position so that it matches the
 * sonar pings starting position
 */
 private void deadReckon(float interval) {
  float timer = 0;
  drPosition[0] = position[0];
  drPosition[1] = position[1];
  drPosition[2] = position[2];
  drSpeed = speed;
  drHeading = heading;
  drDepth = depth;

  while (timer < interval) {
   if (orderedSpeed-drSpeed > .05) {
    drSpeed += 0.1;
   }
   else if (orderedSpeed-drSpeed < -.05) {
    drSpeed -= 0.1;
   }

   if (orderedCourse-drHeading > .05) {
    drHeading += 0.1;
   }
   else if (orderedCourse-drHeading < -.05) {
    drHeading -= 0.1;
   }

   drPosition[0] += Math.cos(drHeading*3.14/180.0)*.1*drSpeed;
   drPosition[1] += Math.sin(drHeading*3.14/180.0)*.1*drSpeed;

   if (orderedDepth-drDepth > .05) {
    drDepth += 0.1;
   }
   else if (orderedDepth-drDepth < -.05) {
    drDepth -= 0.1;
   }

   drPosition[2] = drDepth;
   timer += 0.1;
   }
  }

  /**
  * This method processes events generated by the VRML Browser.
  * When the VRML world is entered, an 'entered' event is sent out
  * and this event handler processes it and starts the whole
  * simulation in motion.
  */
  public void processEvent(Event ev){

if (DEBUG)
System.out.println("BattleScene: processEvent: starting");

    if(ev.getName().equals("entered")){
      ConstSFBool v = (ConstSFBool)ev.getValue();
    String receivedString = "#VRML V2.0 utf8\n";

// get VRML data from whichever source it is necessary
if (!LOCAL) {
    try {
      boolean start = true;

      // send start Order
      out.writeBoolean(start);

      // receive new VRML from server
      receivedString += in.readLine();

      browser.setDescription("Got a String!");
    } catch (IOException e) {
      browser.setDescription("IOException: " + e);
    }

    // create VRML and add it to the appropriate VRML node
    try {
      nodes = browser.createVrmlFromString(receivedString);
      if (null!= nodes) {
        addChildren.setValue(nodes);
      }
    }
    catch (Exception e){
      browser.setDescription("can not create VRML node");
      e.printStackTrace();
    }
} // end DEBUG

      // initialize the control panel and draw it
      if(v.getValue()){ panel.map(); }
      else { panel.hide(); }
    }

if (DEBUG)
System.out.println("BattleScene: processEvent: ending");

   }

   /**
   * This method cleans up memory and closes sockets at shutdown
   */
   public void shutdown(){
    panel.dispose();
    try {
```

```
    out.close();
    in.close();
    socket.close();
  } catch (Exception e) {
     browser.setDescription("Connection close error");
  }
 }

 /**
 * This method is called by the control panel to set the
 * requested maneuver in BattleScene.
 */
 public void setManeuver(float d,
                         float s,
                         float c) {
   orderedDepth = d;
   orderedSpeed = s;
   orderedCourse = c;
 }

 /**
 * This method is called by the control panel to set the
 * requested sonar ping inn BattleScene.
 */
 public void setSonarParameters(float elevation,
                                float azimuth,
                                float verticalBeamWidth,
                                float horizontalBeamWidth,
                                float verticalBeamConfiguration,
                                float horizontalBeamConfiguration,
                                float verticalSearchWidth,
                                float horizontalSearchWidth,
                                float pingI,
                                float powerLevel){

if (DEBUG)
System.out.println("BattleScene:  setSonarParameters:  starting");

    pingInterval = pingI;

if (!LOCAL) {
    browser.setDescription("Sending Ping");
    try {
      out.writeFloat(SET_SONAR_PARAMETERS);
      out.writeFloat(position[0]);
      out.writeFloat(position[2]);
      out.writeFloat(-position[1]);
      out.writeFloat(elevation);
      out.writeFloat(-azimuth);
      out.writeFloat(verticalBeamWidth);
      out.writeFloat(horizontalBeamWidth);
      out.writeFloat(verticalBeamConfiguration);
      out.writeFloat(horizontalBeamConfiguration);
      out.writeFloat((pingI/(float)2.0));
      out.writeFloat(powerLevel);
    }
    catch (Exception e) {
      browser.setDescription("IOException:  " + e);
    }
}

 if (DEBUG)
System.out.println("BattleScene:  setSonarParameters:  ending");
```

```
  }

  /**
  * This method is called by sonarDetection to decide if another
  * sonar request can be made.
  *
  * Another sonar request cannot be made if the threads for
  * adding VRML beams and targets are still alive.
  * When the threads die this indicates that the VRML has been
  * added to the scene graph and another request can be made.
  */
  public void waitOnThreads(){
   if (targetsThread != null && targetsThread.isAlive()){
    try {
     targetsThread.join();
    }
    catch (InterruptedException e) {
     System.out.println("BattleScene:  waitOnThreads: targetsThread "+e);
    }
   }
   if (beamThread != null && beamThread.isAlive()){
    try {
     beamThread.join();
    }
    catch (InterruptedException e) {
     System.out.println("BattleScene:  waitOnThreads: beamThread: "+e);
    }
   }
  }

  /**
  * This method is called from the control panel to initiate a
  * sonar ping.  When a ping is recieved and the required threads
  * die a new sonar request is made and the corresponding VRML is
  * received.
  */
  public void sonarDetection(float order) {

if (DEBUG)
System.out.println("BattleScene:  sonarDetection:  starting");

   waitOnThreads();
   String receivedTargets = "";
   String receivedBeam = "";
if (!LOCAL) {
   try {
        // send Ping Order
    pingOrderTick = tickCount;
    System.out.println("BattleScene:  sonarDetection: order = "+order);
    out.writeFloat(order);

    // receive desired vrml strings;
    if (order == DETECT_TARGETS) {
     receivedTargets = in.readLine();
    }
    else if (order == DETECT_BEAM) {
     receivedBeam = in.readLine();
    }
    else if (order == DETECT_BOTH) {
     receivedTargets = in.readLine();
     receivedBeam = in.readLine();
    }
```

```
      }
      catch (IOException e) {
        browser.setDescription("IOException:  " + e);
      }
  }
  else {
   // fill with fake VRML so that when the sonar server is being run
   // the local machine, the simulation won't bog down. Used for debugging.
   pingOrderTick = tickCount;
   receivedTargets = targetsVRML();
   receivedBeam = beamVRML();
  }

      targetsThread = new CreateVRMLTargetsThread(this,browser);
      beamThread = new CreateVRMLBeamThread(this,browser,targetsThread);

      // perform a small wait if the sonar server is operating faster than
      // real time
      while (tickCount-pingOrderTick < (int)pingInterval*10) {
        int i;
        for (i=0;i<10;i++){}
      }


      // start the threads to produce the VRML;
      if (order == DETECT_TARGETS) {
        targetsThread.start(receivedTargets,oldTargetsNodes[(activeTargetsNode+1)%5]);
        updateBeamPosition();
      }
      else if (order == DETECT_BEAM) {
        beamThread.start(receivedBeam,oldBeamNodes[(activeBeamNode+1)%1]);
      }
      else if (order == DETECT_BOTH) {
        targetsThread.start(receivedTargets,oldTargetsNodes[(activeTargetsNode+1)%5]);
        beamThread.start(receivedBeam,oldBeamNodes[(activeBeamNode+1)%1]);
        updateBeamPosition();
      }


  if (DEBUG)
  System.out.println("BattleScene:  sonarDetection:  ending");

    }

    /**
    * This method adds the beam VRML to the scene graph
    */
    public void AddBeamChildren(BaseNode nodes[]){

  if (DEBUG)
  System.out.println("BattleScene:  AddBeamChildren:  starting");

      activeBeamNode++;
      oldBeamNodes[activeBeamNode%1] = nodes;
      addBeamChildren.setValue(nodes);

  if (DEBUG)
  System.out.println("BattleScene:  AddBeamChildren:  ending");

    }

    /**
    *  This method decides whiche beam VRML to remove from the scene graph
  */
  public void removeBeam() {
   waitOnThreads();
   while (tickCount-pingOrderTick < (int)pingInterval*20) {
     int i;
     for (i=0;i<10;i++){}
   }
    RemoveBeamChildren(oldBeamNodes[activeBeamNode%1]);
  }

  /**
  *  This method removes the beam VRML from the scene graph
  */
  public void RemoveBeamChildren(BaseNode nodes[]) {

if (DEBUG)
System.out.println("BattleScene:  RemoveBeamChildren:  starting");

     removeBeamChildren.setValue(nodes);

if (DEBUG)
System.out.println("BattleScene:  RemoveBeamChildren:  ending");

  }

  /**
  * This method adds the target VRML to the scene graph
  */
  public void AddTargetsChildren(BaseNode nodes[]){

if (DEBUG)
System.out.println("BattleScene:  AddTargetsChildren:  starting");

    activeTargetsNode++;
    oldTargetsNodes[activeTargetsNode%5] = nodes;
    addChildren.setValue(nodes);

if (DEBUG)
System.out.println("BattleScene:  AddTargetsChildren  ending");

  }

  /**
  *  This method removes the target VRML from the scene graph
  */
  public void RemoveTargetsChildren(BaseNode nodes[]) {

if (DEBUG)
System.out.println("BattleScene:  RemoveTargetsChildren:  starting");

     removeChildren.setValue(nodes);

if (DEBUG)
System.out.println("BattleScene:  RemoveTargetsChildren:  ending");

  }

  private void updateVehiclePosition() {
    float transformPosition[] = new float[3];
    transformPosition[0] = position[0];
    transformPosition[1] = -position[2];
    transformPosition[2] = position[1];
   translation.setValue(transformPosition);
   yaw.setValue(heading*(float)(-3.14/180));
   return;
  }

  private void updateBeamPosition() {
   float transformPosition[] = new float[3];
   transformPosition[0] = position[0];
   transformPosition[1] = -position[2];
   transformPosition[2] = position[1];

   beamTranslation.setValue(transformPosition);
   beamYaw.setValue(drHeading*(float)(-3.14/180));
   return;
  }


 private String targetsVRML() {
  return "Transform {"+
       "  translation "+ (drPosition[0]+Math.cos(-drHeading)*50) +" "+
                   (-drPosition[2]) +" "+
                   (drPosition[1]+Math.sin(-drHeading)*50) +
       "  children ["+
       "    Shape {"+
       "     appearance Appearance {"+
       "       material Material {"+
       "        emissiveColor 1.0 0 0"+
       "       }"+
       "     }"+
       "     geometry Sphere { radius 20}"+
       "    }"+
       "  ]"+
       "}";
 }

 private String beamVRML() {
  return "Transform {"+
       "  translation 50 50 0"+
       "  children ["+
       "    Shape {"+
       "     appearance Appearance {"+
       "       material Material {"+
       "        emissiveColor 0 1.0 0"+
       "       }"+
       "     }"+
       "     geometry Sphere { radius 20}"+
       "    }"+
       "  ]"+
       "}";
 }

}
```

B. BRIDGESERVER.JAVA

```
/*
File:           BridgeServer.java
Compiler:       jdk1.1.6
*/

package mil.navy.nps.rra;
```

```
import java.net.*;
import java.io.*;
import java.util.*;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/BridgeServer.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/BridgeServer.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Takes requests for sonar pings and passes them to PingServer and receives VRML
 *   from the PingServer and returns it to BattleScene.
 *
 *<dt><b>Explanation:</b>
 *<dd>Since calculation of numerous ray trajectories is extremely time intensive, rendering and
 *  ray tracing need to be on separate computers.  This provides an interface from the rendering methods
 *  to the ray tracing methods.  The use of this intermediate bridge avoids the security
 *  issues that are present in most browsers.<P>
 *
 *<dt><b>History:</b>
 *<dd>          15Jan98  /Timothy M. Holliday        /New
 *<dd>          17Mar98 /Timothy M. Holliday         /Added HTML comment convention
 *
 *@see PingServer
 *@see BattleScene
 */

public class BridgeServer{
  public final static int PORT = 4129;
  public final static int REMOTE_PORT = 4130;
  public final static float SET_SONAR_PARAMETERS = (float)0.0;
  public final static float DETECT_TARGETS = (float)1.0;
  public final static float DETECT_BEAM = (float)2.0;
  public final static float DETECT_BOTH = (float)3.0;
  public static DataInputStream remoteIn = null;
  public static DataOutputStream remoteOut = null;
  public static DataInputStream in = null;
  public static PrintStream out = null;
  public final static String HOST = "electric.stl.nps.navy.mil"; // remote server host name

  /**
  * The main loop of the bridge program.  It accepts input and output until its
  * network connections are dropped or the program is killed by the user.  No arguments
  * are expected.
  /**
  public static void main(String[] args){

    // Declare Network specific variables to browser
```

```
ServerSocket server_socket = null;
Socket client_socket = null;
boolean r = true;
float serviceRequest = (float)-1.0;

// Network specific variables to remote host
Socket remoteSocket = null;

// Sonar Ping specific variables
int i = 0;
int j = 0;
Vec3d pos = new Vec3d();

System.out.println("Start server: " + PORT);

// Declare the socket to connect to BattleScene
try{
  server_socket = new ServerSocket(PORT);
} catch(IOException e){
  System.out.println("Could not create socket on: " + PORT + ", " + e);
  System.exit(1);
}

System.out.println("Socket created: " + PORT);

// request to open socket and data streams to the PingServer
try {
 // open network and input/output stream
 remoteSocket = new Socket(HOST, REMOTE_PORT);
 remoteIn = new DataInputStream(remoteSocket.getInputStream());
 remoteOut = new DataOutputStream(remoteSocket.getOutputStream());
} catch (UnknownHostException e) {
 System.out.println("Unknown host: " + HOST);
} catch (Exception e) {
 System.out.println("Connection error");
}

System.out.println("Waiting for client...");

// accept BattleScene request for establishing a socket connection
try{
 client_socket = server_socket.accept();
} catch(IOException e) {
 System.out.println("Accept failed: " + PORT + ", " + e);
 System.exit(1);
}

System.out.println("Connection established: "
                 + client_socket.getInetAddress());
System.out.println("Open input/output stream...");

// establish data streams to and from the BattleScene
try{
 in = new DataInputStream(client_socket.getInputStream());
 out = new PrintStream(client_socket.getOutputStream());
} catch (IOException e) {
 System.out.println("Could not create input/output stream on: "
                 + PORT + ", " + e);
 System.exit(1);
}

// wait for a signal from client to begin
System.out.println("Reading data from client...");
```

```
try {
 r = in.readBoolean();
 remoteOut.writeBoolean(r);
 System.out.println("Relayed a Start Request");
}
catch (IOException e) {
 System.out.println("Could not read data.");
 System.exit(1);
}

// returning data Environment data from PingServer to BattleScene
try {
 String tempString = remoteIn.readLine();
 System.out.println("  Relaying new VRML environment");
 out.println(tempString);
}
catch (IOException e) {
 System.out.println("Could not read data.");
 System.exit(1);
}

// Start an infinite loop of accepting ping requests and returning VRML objects
while(true){

  System.out.println("Reading data from client...");
  try {
   serviceRequest = in.readFloat();
   remoteOut.writeFloat(serviceRequest);
   if (serviceRequest == SET_SONAR_PARAMETERS) {
    setParameters();
   }
   else if (serviceRequest == DETECT_TARGETS) {
    System.out.println("Relaying a Ping Request...");
    String  tempString = remoteIn.readLine();
    System.out.println("  Relaying detection field VRML...");
    out.println(tempString);
   }
   else if (serviceRequest == DETECT_BEAM) {
    System.out.println("Relaying a Ping Request...");
    String  tempString = remoteIn.readLine();
    System.out.println("  Relaying VRML Beam...");
    out.println(tempString);
   }
   else if (serviceRequest == DETECT_BOTH) {
    System.out.println("Relaying a Ping Request...");
    String  tempString = remoteIn.readLine();
    System.out.println("  Relaying detection field VRML...");
    out.println(tempString);
    tempString = remoteIn.readLine();
    System.out.println("  Relaying VRML Beam...");
    out.println(tempString);
   }

  } catch (IOException e) {
   System.out.println("Could not read data");
   System.exit(1);
  }
 }
}

/*
* This method sends the pertainant data to PingServer
* from BattleScene
```

```
  */
  private static void setParameters () {
    try {
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
      remoteOut.writeFloat(in.readFloat());
    }
    catch (Exception e) {
      System.out.println("Could not read data");
      System.exit(1);
    }

  }
}
```

C. PINGSERVER.JAVA

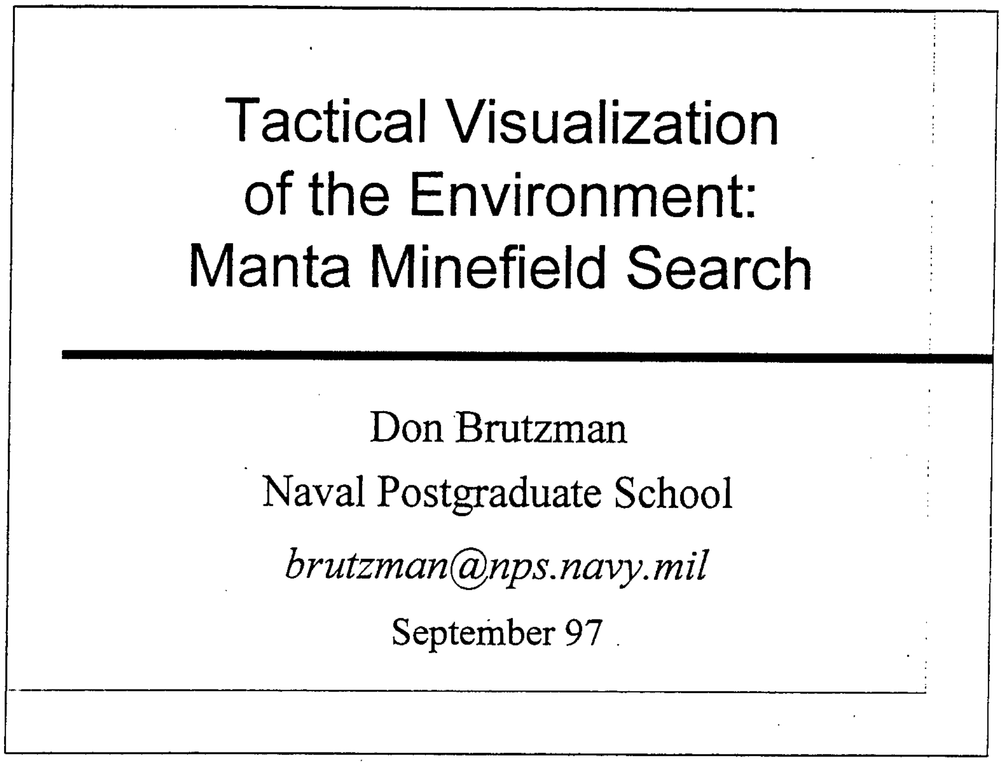
```
/*
 File:          PingServer.java
 Compiler:      jdk1.1.6
 */

package mil.navy.nps.rra;

import java.net.*;
import java.io.*;

import mil.navy.nps.rra.*;

/**
 *@version 1.0
 *@author LT Timothy M. Holliday (<A HREF="http://www.stl.nps.navy.mil/~auv/holliday">
http://www.stl.nps.navy.mil/~auv/holliday</A>)
 *
 *<dt><b>Location:</b>
 *<dd><a href="http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/PingServer.java">
 *  http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/PingServer.java</a>
 *
 *<dt><b>Hierarchy Diagram:</b>
 *<dd><a href="images/RRAClassHierarchy.gif"><IMG SRC="images/RRAClassHierarchyButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Coordinate System Diagram:</b>
 *<dd><a href="images/CoordinateSystem.gif"><IMG SRC="images/CoordinateSystemButton.jpg"
ALIGN=ABSCENTER></a>
 *
 *<dt><b>Summary:</b>
 *<dd>Takes requests for sonar pings and returns VRML representations of the sonar ping and
 *  any detections made.
 *
 *<dt><b>Explanation:</b>
 *<dd>Since calculation of numerous ray trajectories is extremely time intensive, rendering and
 *  ray tracing often need to be on separate computers to improve performance times.  This provides an interface
from the rendering methods
 *  to the ray tracing methods.<P>
 *
 *<dt><b>History:</b>
 *<dd>            15Jan98  /Timothy M. Holliday        /New
 *<dd>            17Mar98 /Timothy M. Holliday         /Added HTML comment convention
 *
 *@see Lobe
 */
class PingServer{

  public final static int PORT = 4130;
  public final static float SET_SONAR_PARAMETERS = (float)0.0;
  public final static float DETECT_TARGETS = (float)1.0;
  public final static float DETECT_BEAM = (float)2.0;
  public final static float DETECT_BOTH = (float)3.0;

  public static void main(String[] args){

//  Network specific variables
    ServerSocket server_socket = null;
    Socket client_socket = null;
    DataInputStream in = null;
    PrintStream out = null;
    boolean r = true;
    float serviceRequest = (float)-1.0;

//  Sonar Ping specific variables
   int i = 5;
   int j = 110;
   Vec3d pos = new Vec3d( -5000, 500, 0);
   SSP  ssp   = new SSP("traditional");
   Bottom bott = new Bottom("slope",2000);
   Surface surf = new Surface("smooth");
   Lobe lobe1 = new Lobe();
   lobe1.setDuration(1);
   lobe1.setBottom(bott);
   lobe1.setSurface(surf);
   lobe1.setSsp(ssp);

//(double)j, 10, (double)i, 1.0, 10, 10,
//                     pos, 2, bott, surf, ssp,
//                     4);
   Targets targets = new Targets(surf,bott,70,0);

   // call each class' VRML print routine
    System.out.println("Start server: " + PORT);

    // open socket on PORT
    try{
      server_socket = new ServerSocket(PORT);
    } catch(IOException e){
      System.out.println("Could not create socket on: " + PORT + ", " + e);
      System.exit(1);
    }

    System.out.println("Socket created: " + PORT);
    System.out.println("Waiting for client...");

    // accept client's request
    try{
      client_socket = server_socket.accept();
  } catch(IOException e) {
    System.out.println("Accept failed: " + PORT + ", " + e);
    System.exit(1);
  }

  System.out.println("Connection established: "
                  + client_socket.getInetAddress());
  System.out.println("Open input/output stream...");

  try{
    in = new DataInputStream(client_socket.getInputStream());
    out = new PrintStream(client_socket.getOutputStream());
  } catch (IOException e) {
    System.out.println("Could not create input/output stream on: "
          + PORT + ", " + e);
    System.exit(1);
  }
// wait for a signal from client to begin
  System.out.println("Reading data from client...");
  try {
   r = in.readBoolean();
   System.out.println("Received a Start Request");
  }
  catch (IOException e) {
   System.out.println("Could not read data.");
   System.exit(1);
  }

  bott.setAppendLineFeed(false);
  surf.setAppendLineFeed(false);
  System.out.println("  Sending new VRML environment");
  String tempString = bott.VRMLBottom()+surf.VRMLSurface();
  out.println(tempString);

  lobe1.setAppendLineFeed(false);

  while(true){
    System.out.println("Reading data from client...");
    try {
          serviceRequest = in.readFloat();
      if (serviceRequest == SET_SONAR_PARAMETERS) {
       setParameters(lobe1, in, pos);
      }
      else if (serviceRequest == DETECT_TARGETS) {
       System.out.println("Received a Ping Request...");
       lobe1.calculateLobe(targets);
       System.out.println("  Sending detection field VRML...");
       out.println(lobe1.detectionVRML());
      }
      else if (serviceRequest == DETECT_BEAM) {
       System.out.println("Received a Ping Request...");
       lobe1.calculateLobe(targets);
       System.out.println("  Sending VRML Beam...");
       out.println(lobe1.dynamicVRML());
      }
      else if (serviceRequest == DETECT_BOTH) {
       System.out.println("Received a Ping Request...");
       lobe1.calculateLobe(targets);
       System.out.println("  Sending detection field VRML...");
       out.println(lobe1.detectionVRML());
       System.out.println("  Sending VRML Beam...");
       lobe1.setNumberYPartition(1);
       lobe1.setNumberXPartition(1);
```

```
          lobe1.reset();
          lobe1.calculateLobe(targets);
          out.println(lobe1.dynamicVRML());
        }

      } catch (IOException e) {
        System.out.println("Could not read data");
        System.exit(1);
      }
    }
  }

private static void setParameters (Lobe lobe1,
                                   DataInputStream in,
                                   Vec3d pos) {
  try {
    pos.set((double)in.readFloat(),
        (double)in.readFloat(),
        (double)in.readFloat());
    float elevation = in.readFloat();
    float azimuth = in.readFloat();
    float verticalBeamWidth = in.readFloat();
    float horizontalBeamWidth = in.readFloat();
    float verticalBeamConfiguration = in.readFloat();
    float horizontalBeamConfiguration = in.readFloat();
    float pingInterval = in.readFloat();
    float powerLevel = in.readFloat();
    lobe1.setElevation(elevation);
    lobe1.setAzimuth(azimuth);
    lobe1.setLobeWidthY(verticalBeamWidth);
    lobe1.setLobeWidthX(horizontalBeamWidth);
    lobe1.setNumberYPartition((int)verticalBeamConfiguration);
    lobe1.setNumberXPartition((int)horizontalBeamConfiguration);
    lobe1.setPosition(pos);
    lobe1.setEndTime(pingInterval);
    lobe1.reset();
    System.out.println("Beam azimuth angle "+azimuth);
    System.out.println("Beam elevation angle "+elevation);
  }
  catch (Exception e) {
    System.out.println("Could not read data");
    System.exit(1);
  }
 }
}
```

APPENDIX D. TACTICAL VISUALIZATION POWER POINT SLIDE SET

A. INTRODUCTION

This appendix presents the annotated slide set for a Manta tactical visualization scenario. This slide set gives an overview of the Manta design, a presumed tactical scenario and how the scenario develops. This slide set is integrated with the Hyper Text Mark-up Language (HTML) and Virtual Reality Modeling Language (VRML) web pages shown in Appendix E.



Tactical Visualization of the Environment: Manta Minefield Search

Don Brutzman

Naval Postgraduate School

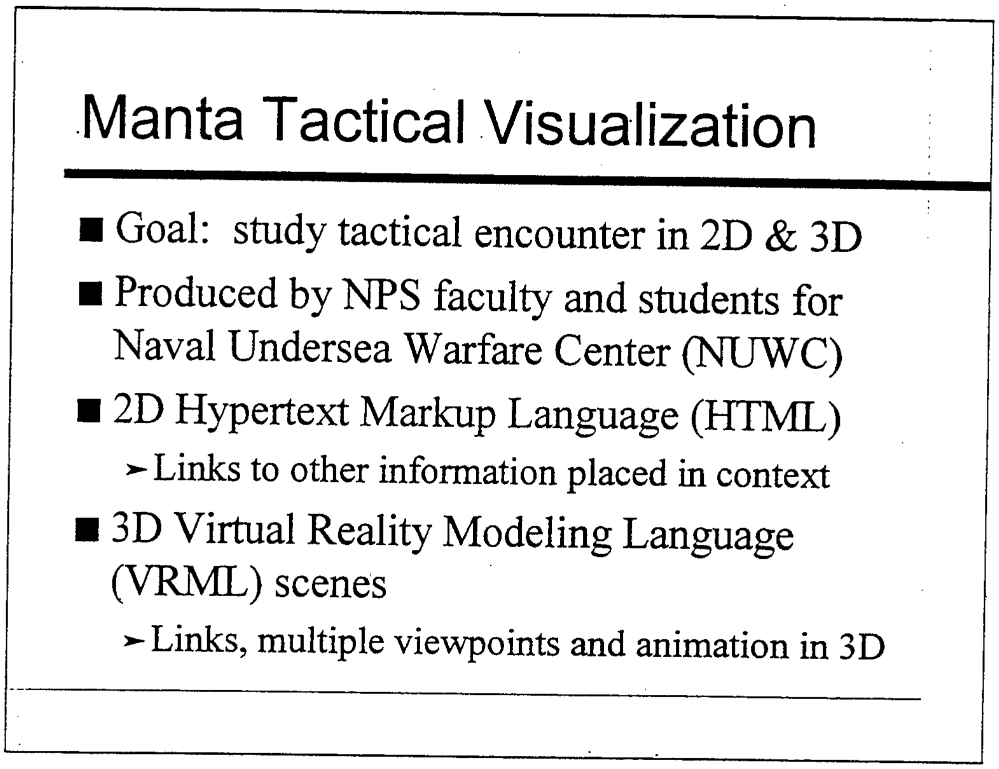
brutzman@nps.navy.mil

September 97

The purpose of this project is to examine a Manta minefield search mission. We will show how tactical visualization of the environment improves mission planning and understanding.

You are reading a PowerPoint presentation. It corresponds to the 2D/3D slide show written in HTML (the Hypertext Markup Language) and VRML (the Virtual Reality Modeling Language).

Annotations for each slide describe MPEG video dialog and VRML scene viewpoints.



Manta Tactical Visualization

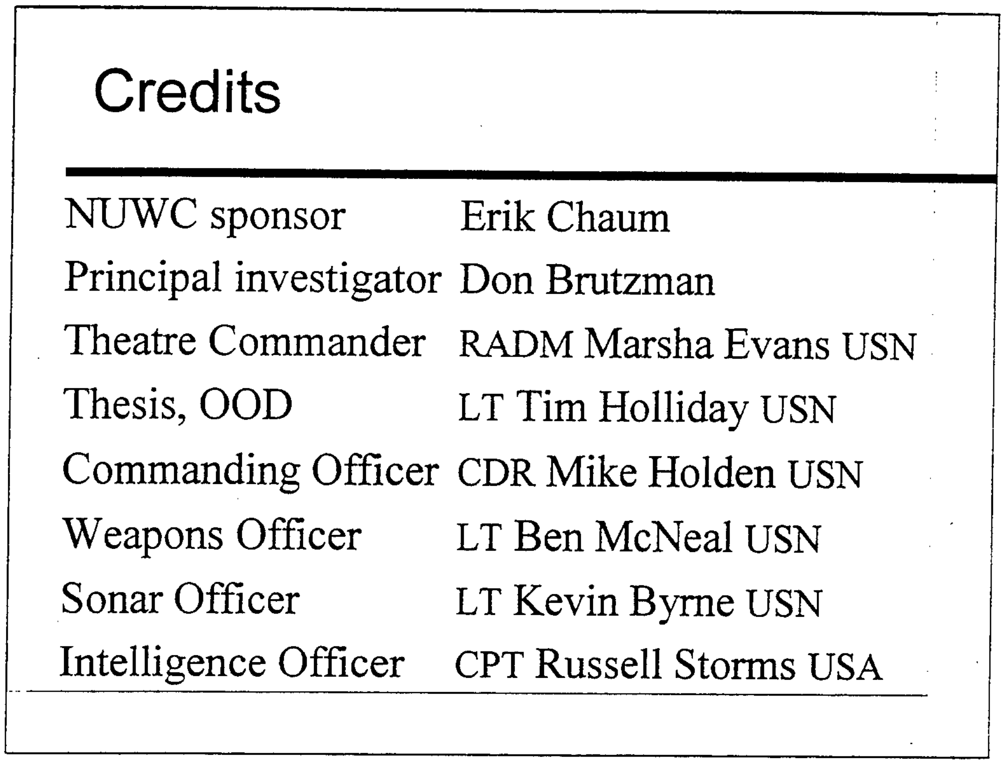
- Goal: study tactical encounter in 2D & 3D
- Produced by NPS faculty and students for Naval Undersea Warfare Center (NUWC)
- 2D Hypertext Markup Language (HTML)
  - Links to other information placed in context
- 3D Virtual Reality Modeling Language (VRML) scenes
  - Links, multiple viewpoints and animation in 3D

Left video: Principal investigator.

"This is a realistic scenario, both for submarines and for multiple Mantas."

VRML scene: Simple Manta model plus moving sonar beam and spatialized sound. Green is detection, red is counterdetection.

Viewpoints: Manta close-up, over-the-shoulder view, looking back towards Manta sonar beam, Manta plus sonar from 1000m off track.



Credits

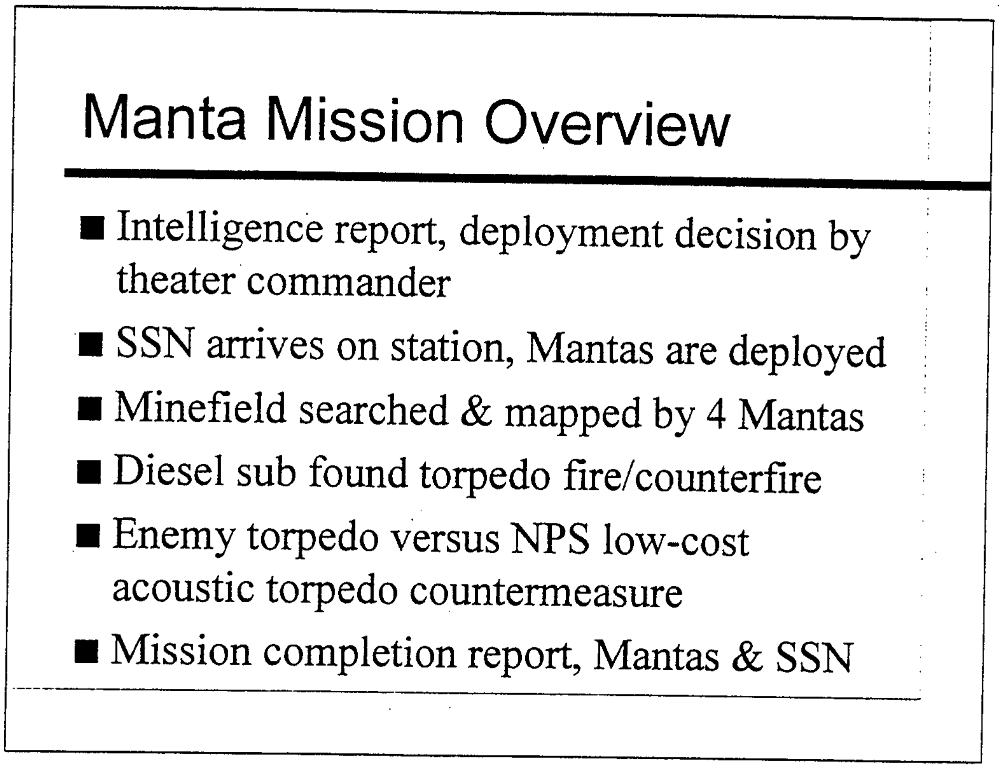
NUWC sponsor	Erik Chaum
Principal investigator	Don Brutzman
Theatre Commander	RADM Marsha Evans USN
Thesis, OOD	LT Tim Holliday USN
Commanding Officer	CDR Mike Holden USN
Weapons Officer	LT Ben McNeal USN
Sonar Officer	LT Kevin Byrne USN
Intelligence Officer	CPT Russell Storms USA

Erik Chaum works in NUWC Code 22.
He supervised the design and construction of the Warfare Systems Presentation Facility (WSPF).

Don Brutzman is a computer scientist and assistant professor at NPS. His research interests include 3D graphics, underwater robotics and internetworking.

RADM Marsha Evans USN is NPS Superintendent.

(This is an optional link from the web page.
There is no corresponding VRML scene.)



Manta Mission Overview

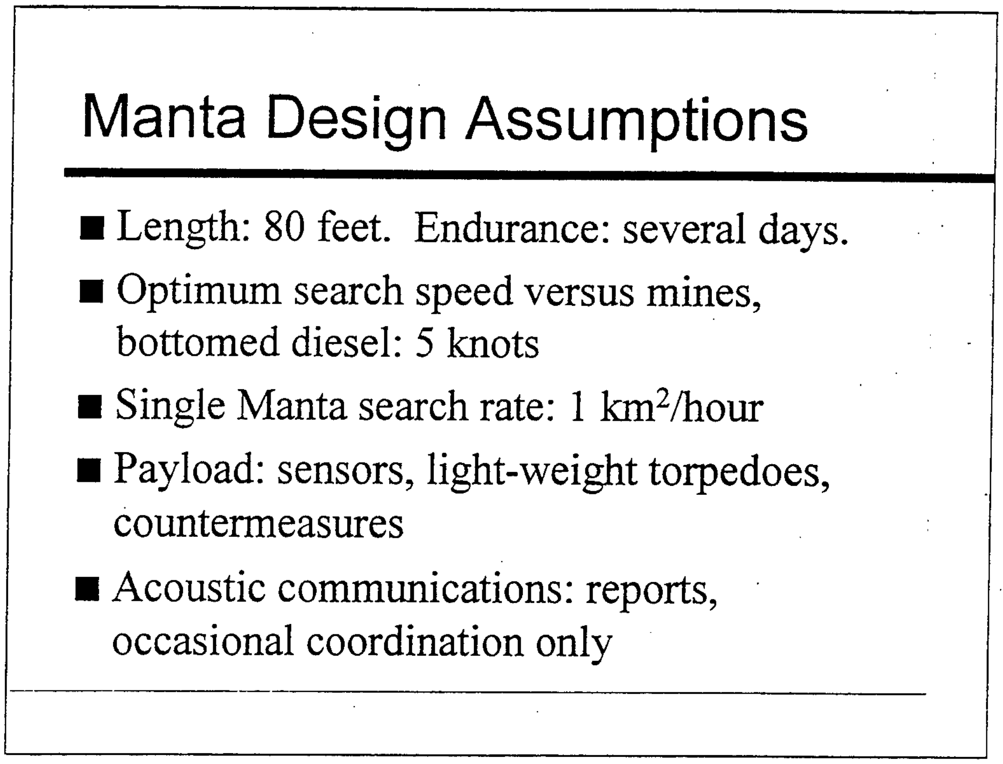
- Intelligence report, deployment decision by theater commander
- SSN arrives on station, Mantas are deployed
- Minefield searched & mapped by 4 Mantas
- Diesel sub found torpedo fire/counterfire
- Enemy torpedo versus NPS low-cost acoustic torpedo countermeasure
- Mission completion report, Mantas & SSN

Left video: Principal investigator.

"All of the robot searches and software shown in this mission can be demonstrated today. We are using existing capabilities of the NPS Phoenix AUV."

VRML scene: A 25-meter Manta searching for a much smaller 1-meter mine. A white vertical marker helps viewers locate the mine.

Viewpoints: Over-the-shoulder view, Manta plus sonar beam 1000m off search track, Manta close-up, looking back towards Manta sonar beam.



Manta Design Assumptions

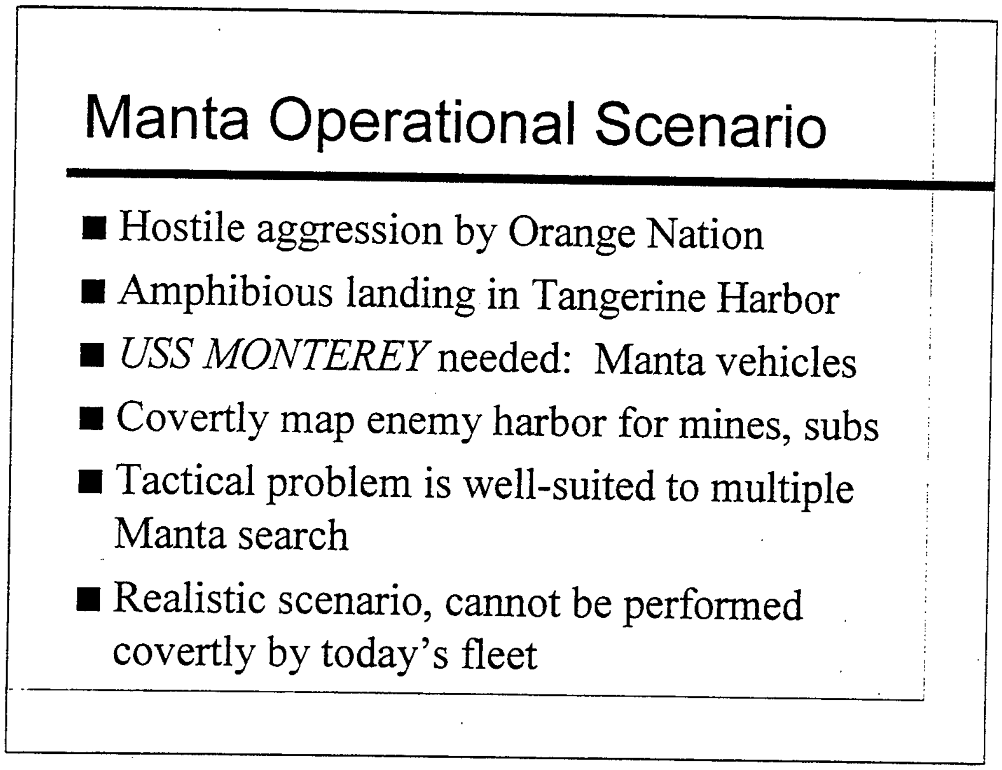
- Length: 80 feet. Endurance: several days.
- Optimum search speed versus mines, bottomed diesel: 5 knots
- Single Manta search rate: 1 km^2/hour
- Payload: sensors, light-weight torpedoes, countermeasures
- Acoustic communications: reports, occasional coordination only

Left video: Principal investigator.

"A variety of potential Manta designs are being evaluated by NUWC engineers."

VRML scene: Simple Manta model plus moving sonar beam and spatialized sound.

Viewpoints: Manta Close-up, Manta plus sonar beam, Manta Aft Port View, Manta Fore Port View.



Manta Operational Scenario

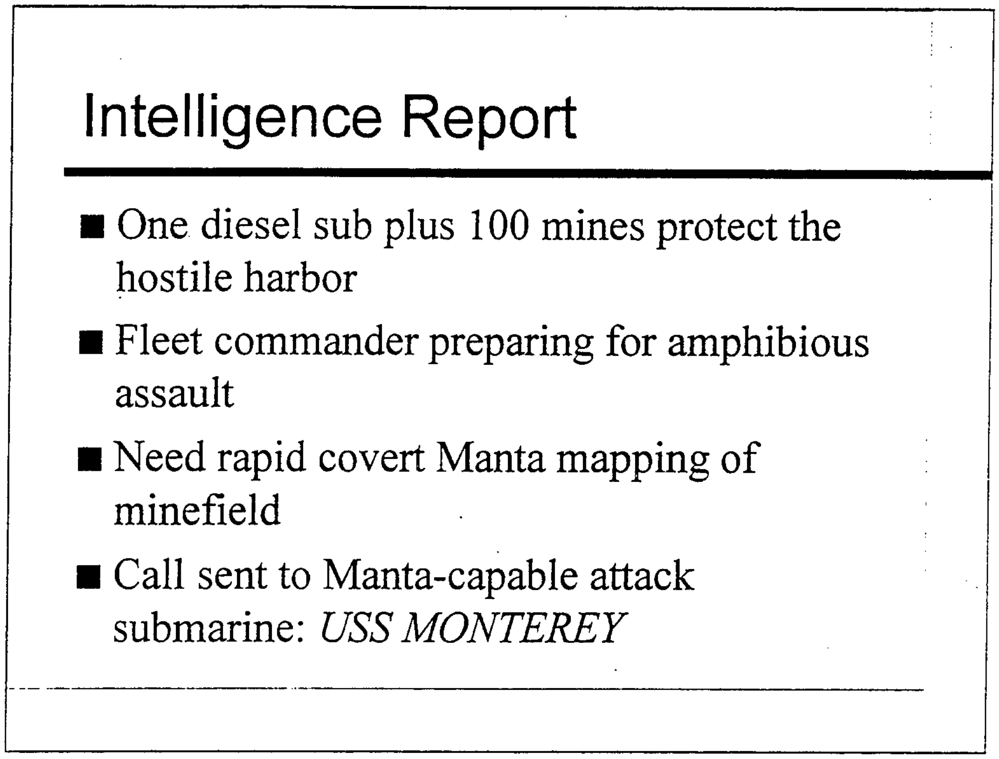
- Hostile aggression by Orange Nation
- Amphibious landing in Tangerine Harbor
- *USS MONTEREY* needed: Manta vehicles
- Covertly map enemy harbor for mines, subs
- Tactical problem is well-suited to multiple Manta search
- Realistic scenario, cannot be performed covertly by today's fleet

Left video: Principal investigator.

"The slides, scenes & videos in the rest of this presentation walk you through the entire Manta tactical encounter."

VRML scene: A simple VRML sphere wrapped with an image texture of the Earth. Geographic maps of Tangerine Harbor and SUBLANT Norfolk are hidden inside.

Viewpoints: Click Earth to see maps appear.



Intelligence Report

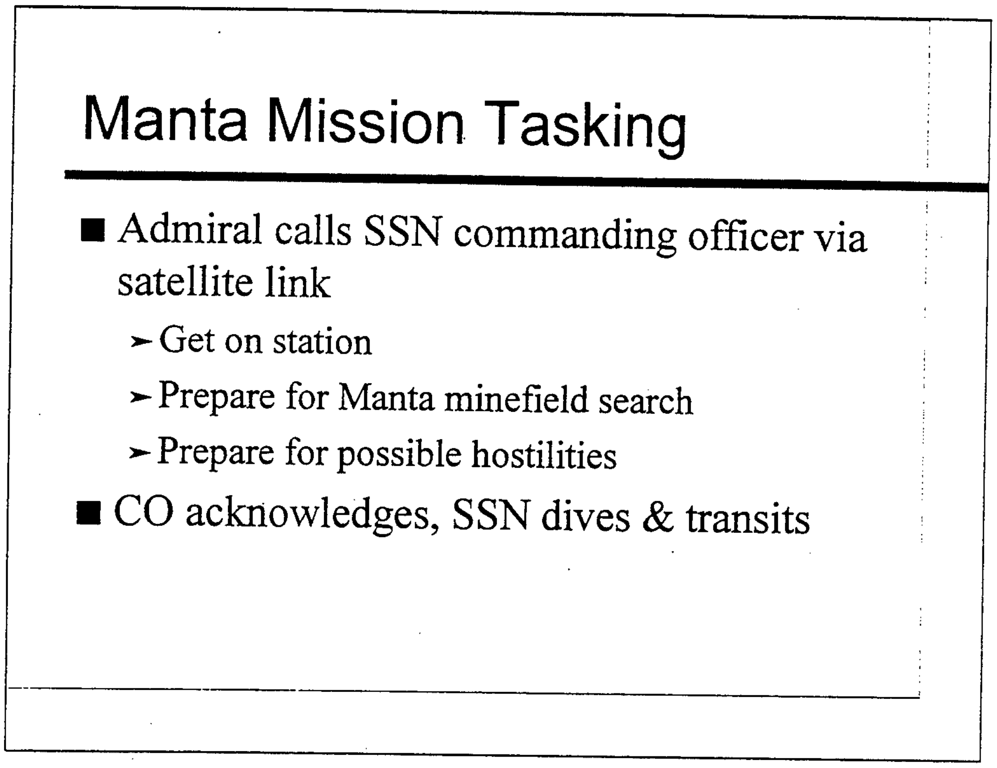
- One diesel sub plus 100 mines protect the hostile harbor
- Fleet commander preparing for amphibious assault
- Need rapid covert Manta mapping of minefield
- Call sent to Manta-capable attack submarine: *USS MONTEREY*

Left video: Intelligence officer, reporting to Admiral. *"Admiral, intelligence shows a diesel sub and 100 mines have been laid in Tangerine Harbor."*

Right video: Theater commander.

"That could delay the amphibious landing. Get me the skipper of the Monterey on the satellite link."

VRML scene: Simple Manta scene.


Manta Mission Tasking

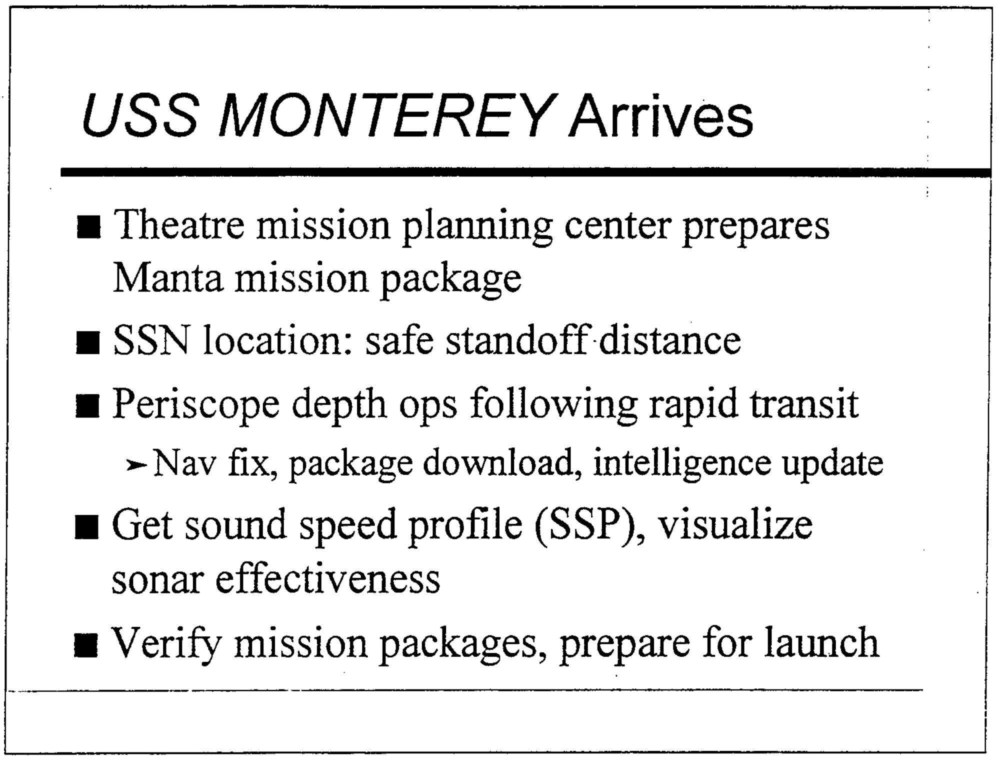
- Admiral calls SSN commanding officer via satellite link
  - ➢ Get on station
  - ➢ Prepare for Manta minefield search
  - ➢ Prepare for possible hostilities
- CO acknowledges, SSN dives & transits

Left video: Theater commander. *"Skipper, we need you to search Tangerine Harbor with your onboard Mantas. We can't afford any delay in the amphibious invasion."*

Right video: Commanding officer. *"We're on our way, Admiral... Officer of the Deck, take us down and proceed at flank speed to the landing zone at Tangerine Harbor."*

VRML scene: Highly detailed submarine at periscope depth. Mountains in background.

Viewpoints: Multiple viewpoints: use PgUp/PgDn to walk camera around the sub. Click on periscope to lower it.



USS MONTEREY Arrives

- Theatre mission planning center prepares Manta mission package
- SSN location: safe standoff distance
- Periscope depth ops following rapid transit
  - ➢ Nav fix, package download, intelligence update
- Get sound speed profile (SSP), visualize sonar effectiveness
- Verify mission packages, prepare for launch

Left video: Intelligence officer.

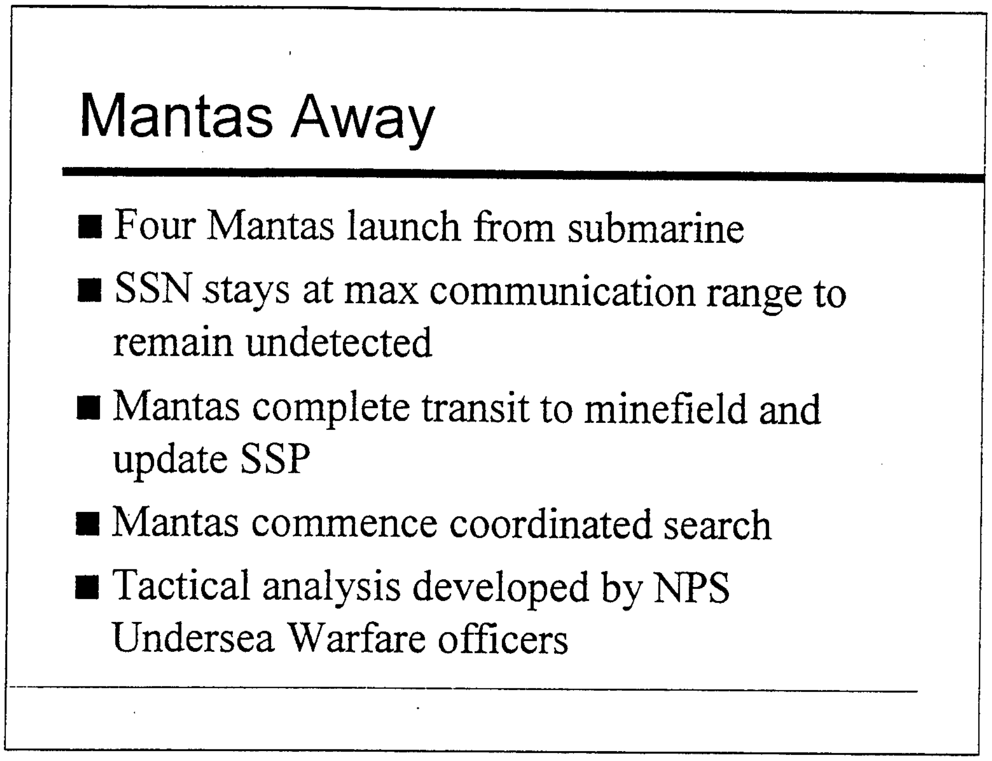
"The MONTEREY will only have 24 hours to map Tangerine Harbor. We'll evaluate search tactics using the virtual world."

VRML scene:

Submarine proceeds to periscope depth (PD).

Viewpoints:

Sub 500m away, scope up at periscope depth.
Close-up: sub approach to periscope depth.



Mantas Away

- Four Mantas launch from submarine
- SSN stays at max communication range to remain undetected
- Mantas complete transit to minefield and update SSP
- Mantas commence coordinated search
- Tactical analysis developed by NPS Undersea Warfare officers

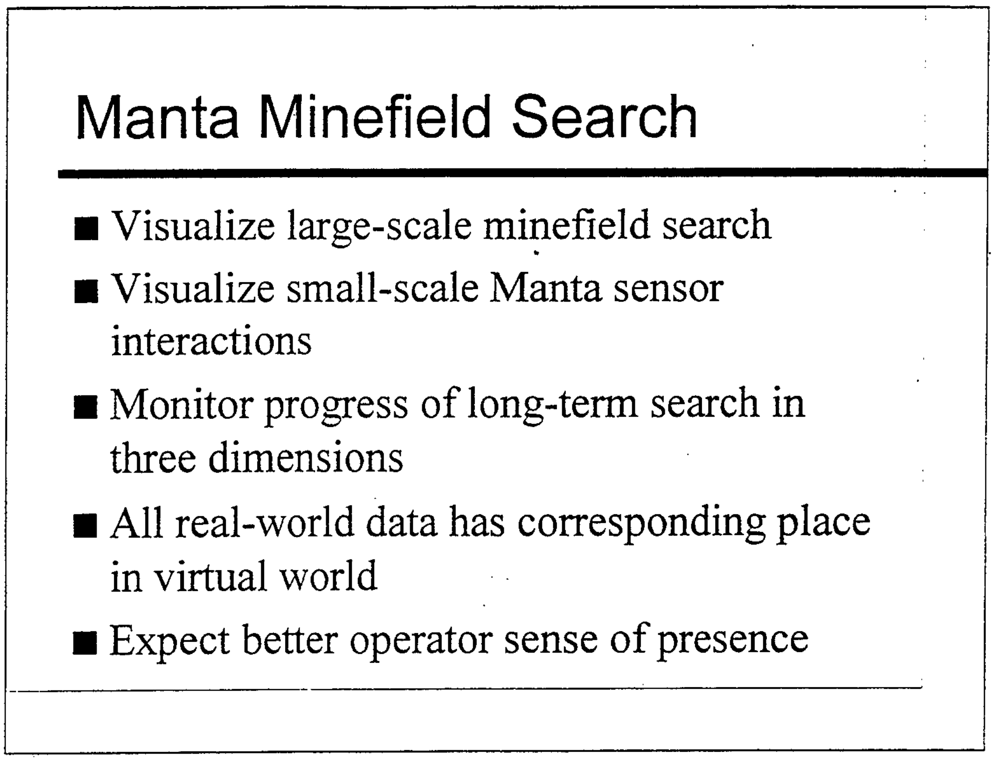
Left video: Officer of the Deck (OOD). *"Captain, we're on station, and the Manta mission package download is in progress."*

Right video: Commanding Officer (CO). *"Weps, you may launch Mantas when ready."*

Weapons Officer (Weps). *"The boat is maintaining constant depth and speed... Mantas away."*

VRML scene: Two of four Mantas are shown separating from the forward hull and proceeding to the search area.

Viewpoints: Various camera angles.



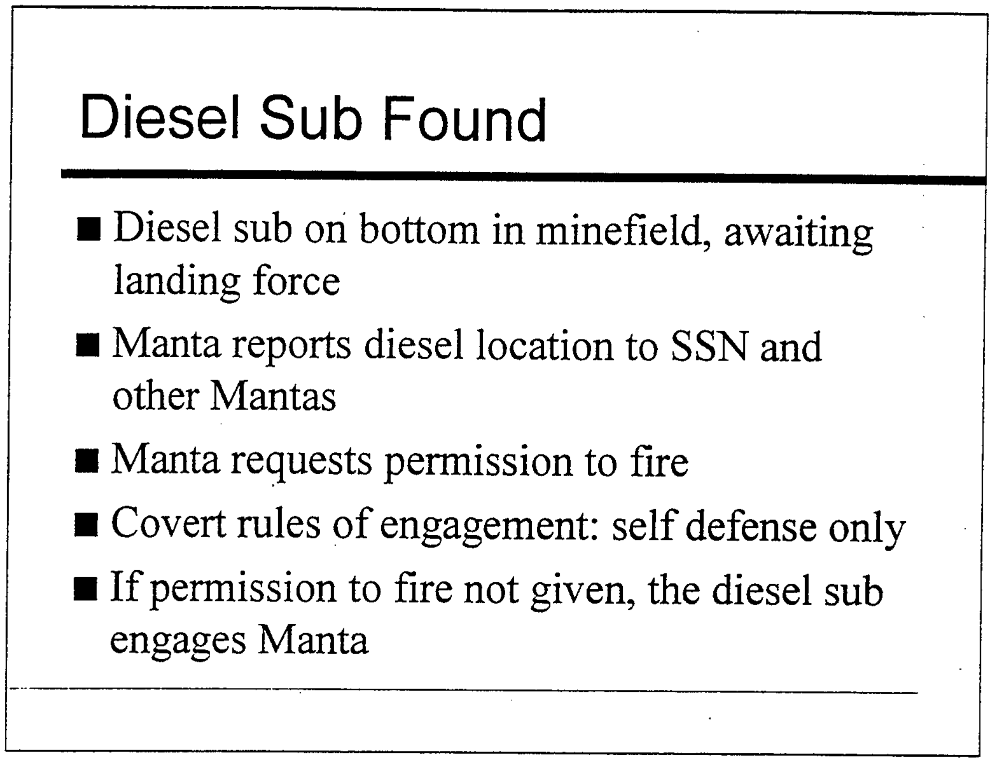
Manta Minefield Search

- Visualize large-scale minefield search
- Visualize small-scale Manta sensor interactions
- Monitor progress of long-term search in three dimensions
- All real-world data has corresponding place in virtual world
- Expect better operator sense of presence

Left video: Weapons Officer (Weps). *"All Mantas are away, all communication links are up, and the Mantas are finding mines."*

VRML scene: This is an extended (24 hour) minefield search, reenacted by playing back the Manta mission scripts in the 3D VRML scene. After 100 mine boxes appear, Manta will start.

Viewpoints: A variety of camera angles allow big-picture visualization of a large (10 kilometer by 10 kilometer) harbor, as well as close-up views of Manta-mine interactions. Red bounding boxes & spheres help viewers visualize the search and localize small mines.



Diesel Sub Found

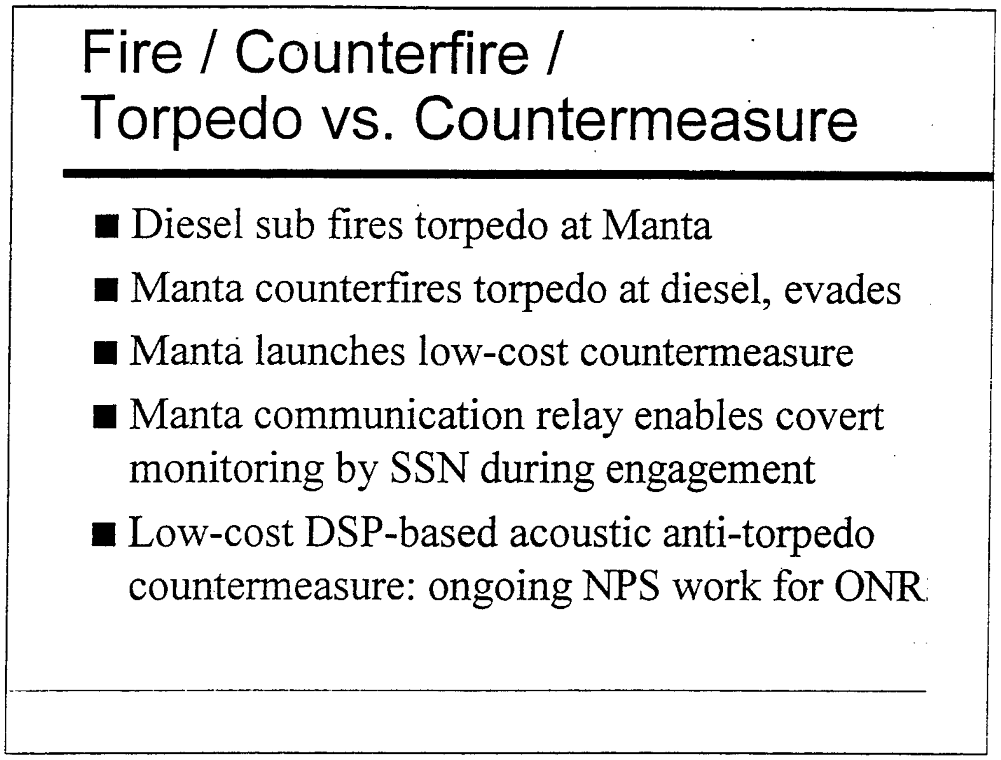
- Diesel sub on bottom in minefield, awaiting landing force
- Manta reports diesel location to SSN and other Mantas
- Manta requests permission to fire
- Covert rules of engagement: self defense only
- If permission to fire not given, the diesel sub engages Manta

Left video: Officer of the Deck (OOD). *"Captain, Manta Alpha has found a diesel submarine and requests permission to fire."*

Commanding Officer (CO). *"Permission denied. Our current rules of engagement only allow firing in self defense."*

VRML scene: The Manta passes over a hostile diesel submarine hiding on the bottom.

Viewpoints: Various aspects relative to submarine and Manta enable visualizing the encounter.



Fire / Counterfire / Torpedo vs. Countermeasure

- Diesel sub fires torpedo at Manta
- Manta counterfires torpedo at diesel, evades
- Manta launches low-cost countermeasure
- Manta communication relay enables covert monitoring by SSN during engagement
- Low-cost DSP-based acoustic anti-torpedo countermeasure: ongoing NPS work for ONR

Left video: Sonar: *"Torpedo in the water, launched from the diesel submarine."*

Weps: *"Second torpedo counterfired by Manta."*

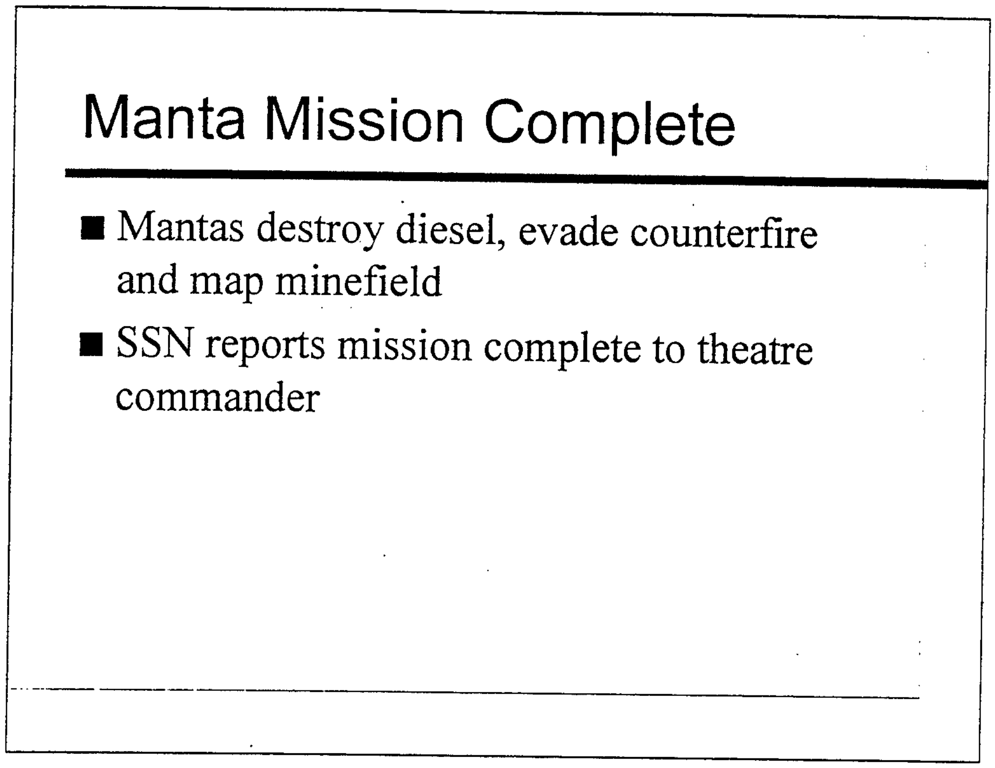
Sonar: *"Manta is evading with countermeasures."*

OOD: *"Manta's torpedo is in terminal homing, on the diesel."*

Sonar: *"Loud explosion down the bearing of the diesel submarine."*

VRML scene: This one-quarter project shows embedded sound in a torpedo-countermeasure interaction. Click on the torpedo to start the run.

Viewpoints: Note NTDS symbols and equipment labels embedded in the environment.



Manta Mission Complete

- Mantas destroy diesel, evade counterfire and map minefield
- SSN reports mission complete to theatre commander

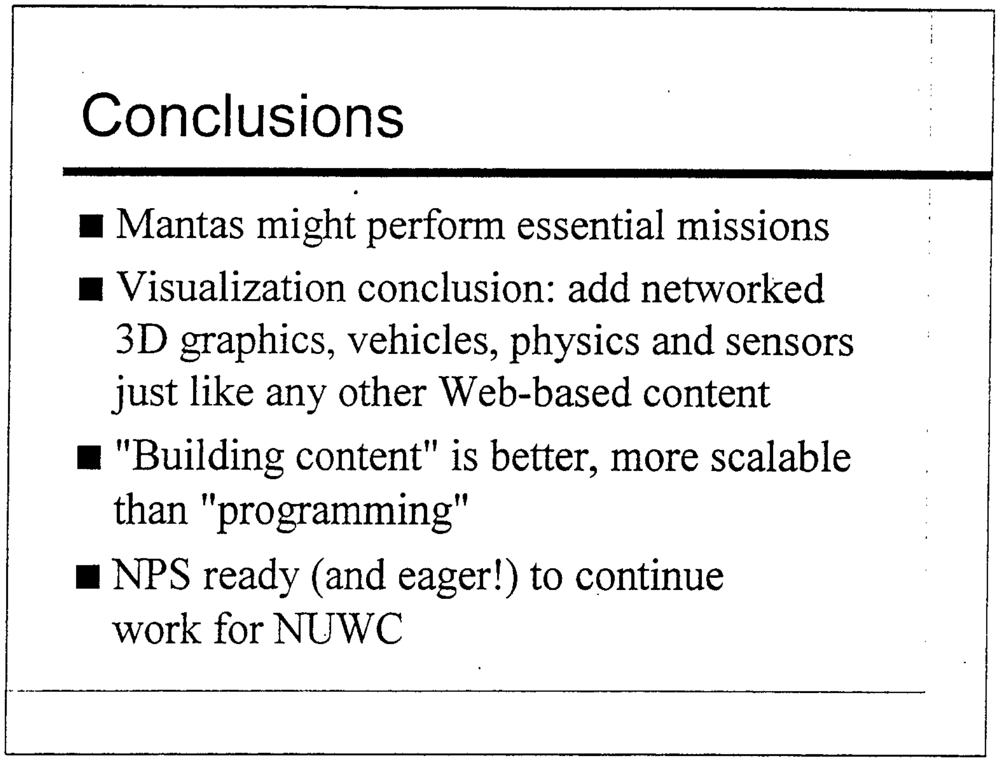
Left video: Commanding Officer (CO).
"Admiral, the minefield at Tangerine Harbor has been mapped, one diesel submarine has been destroyed, and all four Mantas are recovered & back on board."

Right video: Theater commander.

"Naval Postgraduate School student officers and faculty can make big contributions to real fleet challenges."

VRML scene: Manta search pings.

Viewpoints: Over-the-shoulder viewpoint.



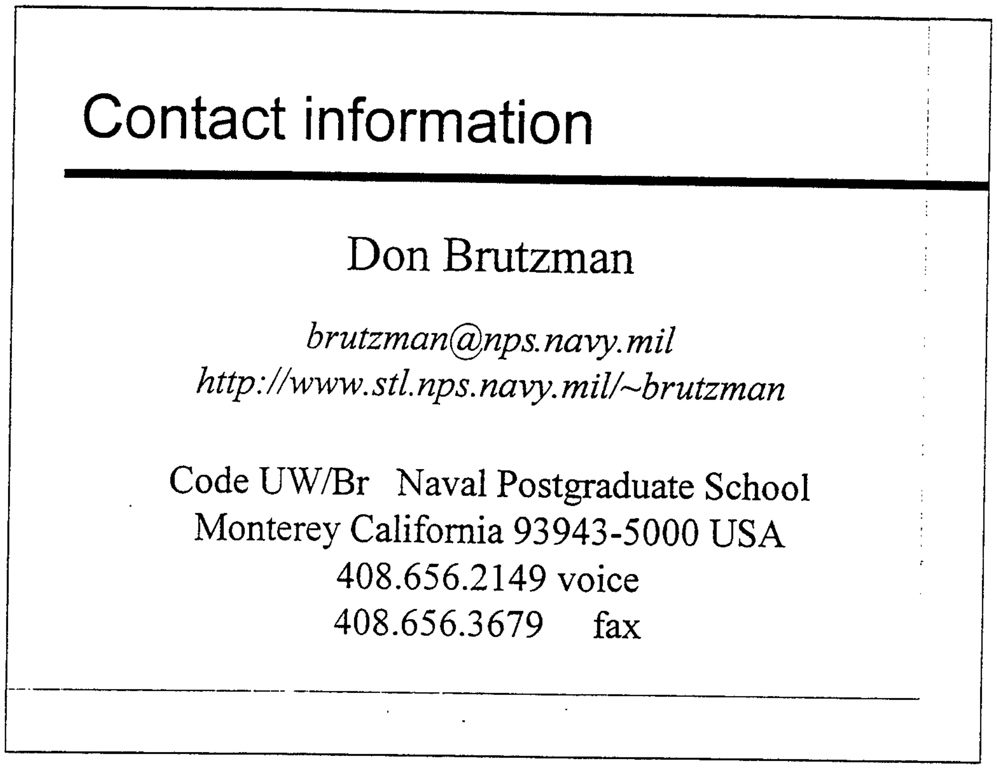
Conclusions

- Mantas might perform essential missions
- Visualization conclusion: add networked 3D graphics, vehicles, physics and sensors just like any other Web-based content
- "Building content" is better, more scalable than "programming"
- NPS ready (and eager!) to continue work for NUWC

Our conclusions match the expected results of the NUWC-NPS research proposal.

These dramatic results show that construction of composable physics-based 3D scenes is possible using VRML. Tactical visualization of the environment improves understanding for forces afloat and the engineering community ashore.

The star of this show is 3D visualization. We have demonstrated the viability of 2D/3D tactical visualization of the environment. A lot of promising work lies ahead.



Contact information

Don Brutzman

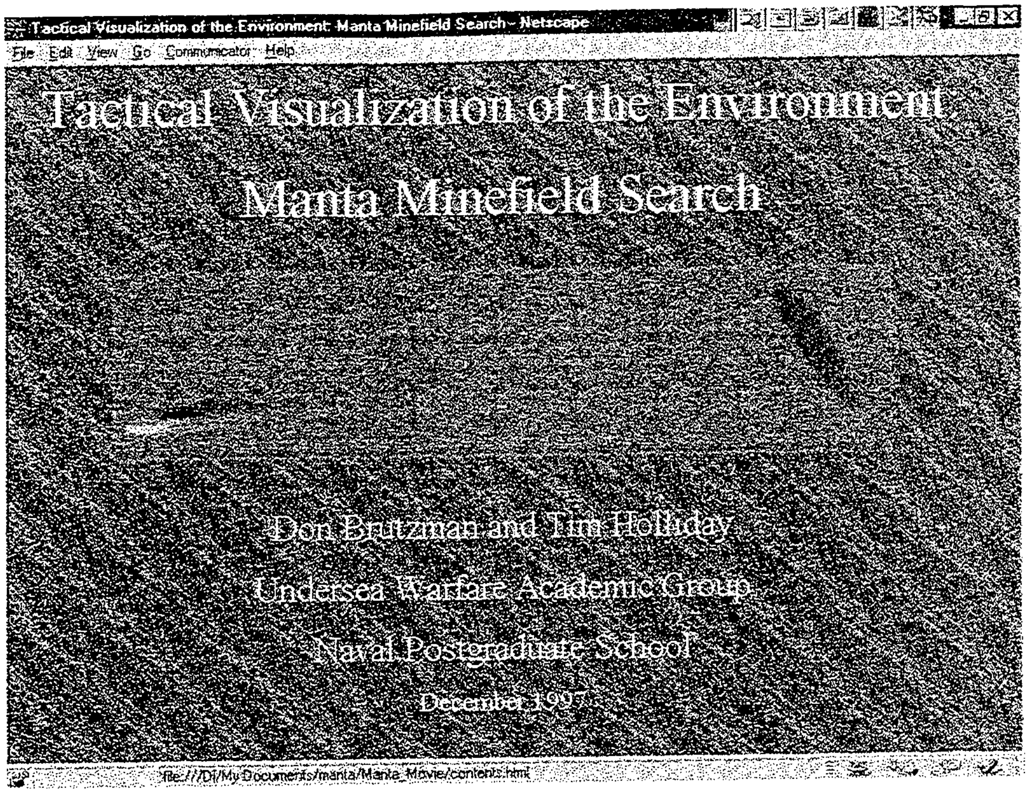
brutzman@nps.navy.mil
http://www.stl.nps.navy.mil/~brutzman

Code UW/Br   Naval Postgraduate School
Monterey California 93943-5000 USA
408.656.2149 voice
408.656.3679   fax

APPENDIX E. TACTICAL VISUALIZATION HTML DOCUMENTS

A. INTRODUCTION

This appendix presents Hyper Text Mark-up Language (HTML), MPEG-2 digitized video and Virtual Reality Modeling Language (VRML) web pages which show how HTML and VRML may be combined to form a tactical scenario presentation system. One advantage to using VRML in the presentation system is that behaviors can be given to the objects in the scene. This allows the scene to have a time dependent nature, instead of being just a static picture.


Tactical Visualization of the Environment: Manta Minefield Search - Netscape

File Edit View Go Communicator Help

Tactical Visualization of the Environment:

Manta Minefield Search

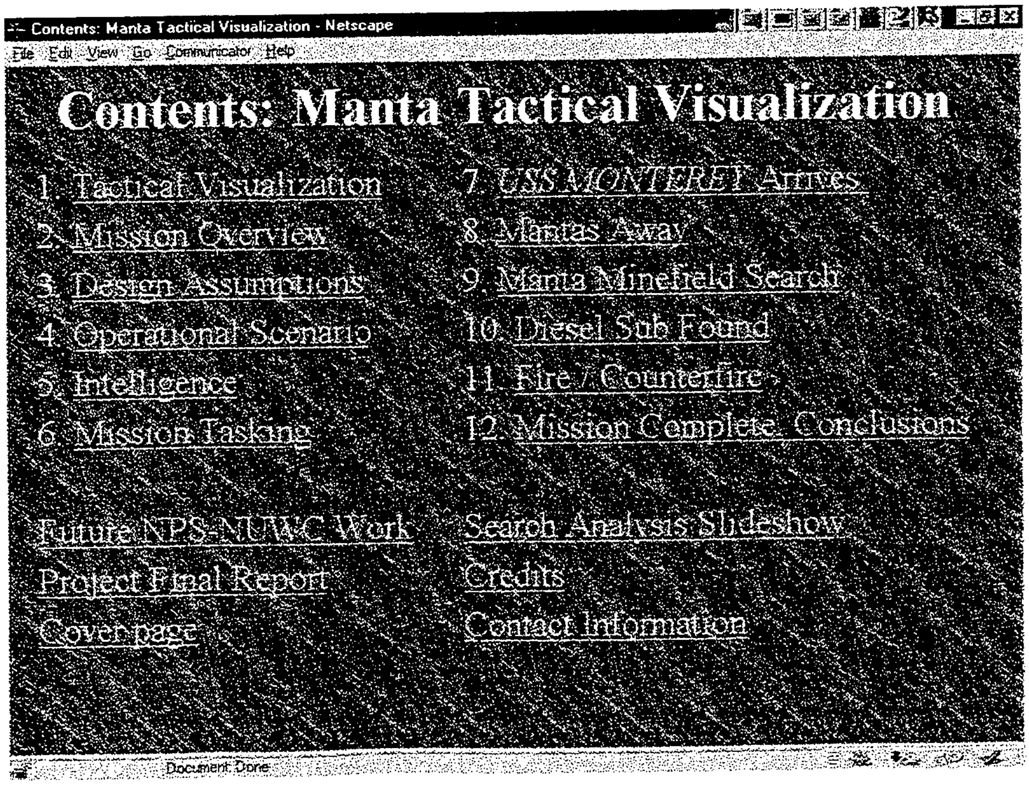
Don Brutzman and Tim Holliday

Undersea Warfare Academic Group

Naval Postgraduate School

December 1997

file:///D|/My Documents/manta/Manta_Movie/contents.html


Contents: Manta Tactical Visualization - Netscape

File Edit View Go Communicator Help

Contents: Manta Tactical Visualization

1. Tactical Visualization
2. Mission Overview
3. Design Assumptions
4. Operational Scenario
5. Intelligence
6. Mission Tasking
7. *USS MONTEREY* Arrives
8. Mantas Away
9. Manta Minefield Search
10. Diesel Sub Found
11. Fire / Counterfire
12. Mission Complete, Conclusions

Future NPS-NUWC Work

Project Final Report

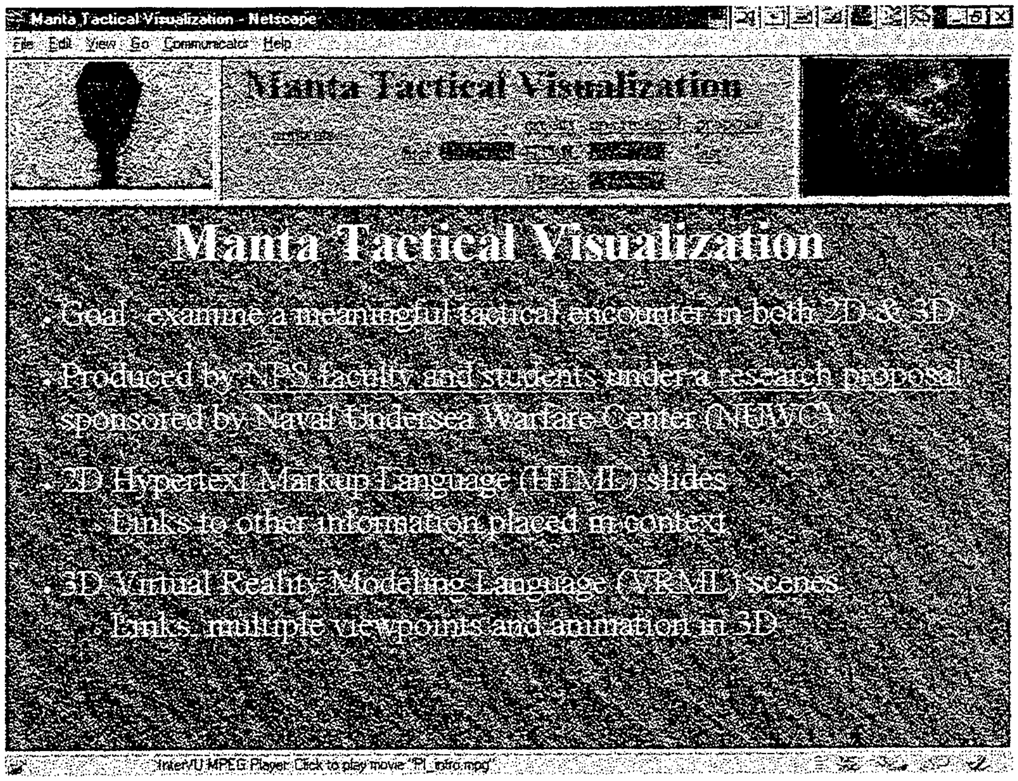
Cover page

Search Analysis Slideshow

Credits

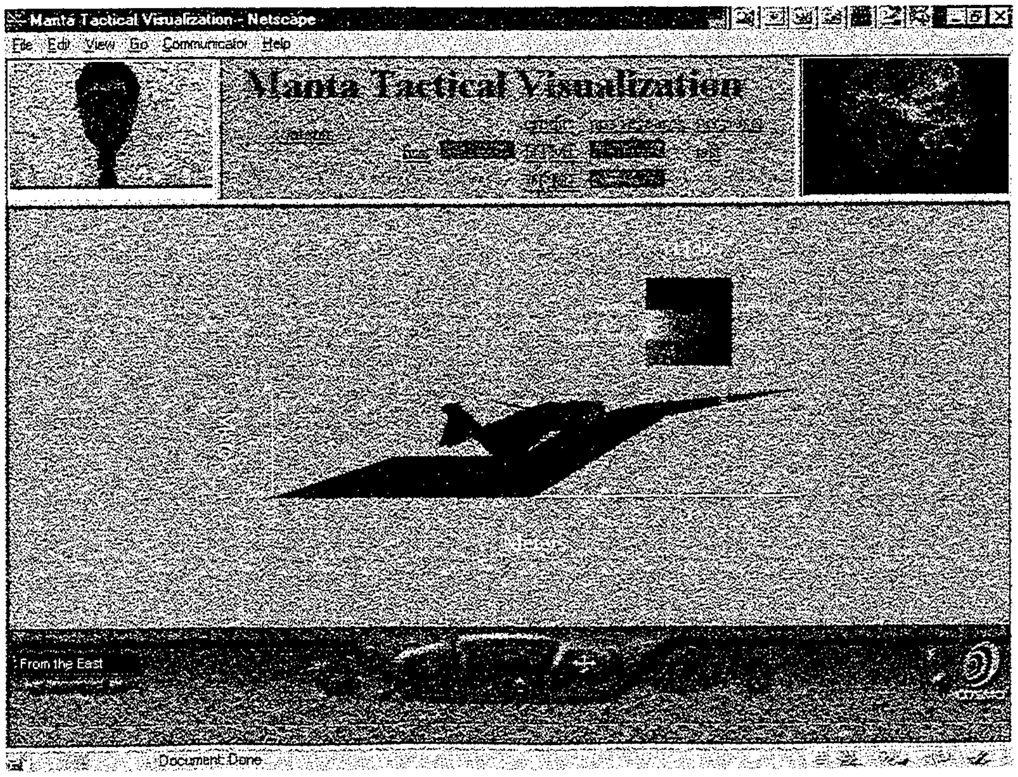
Contact Information

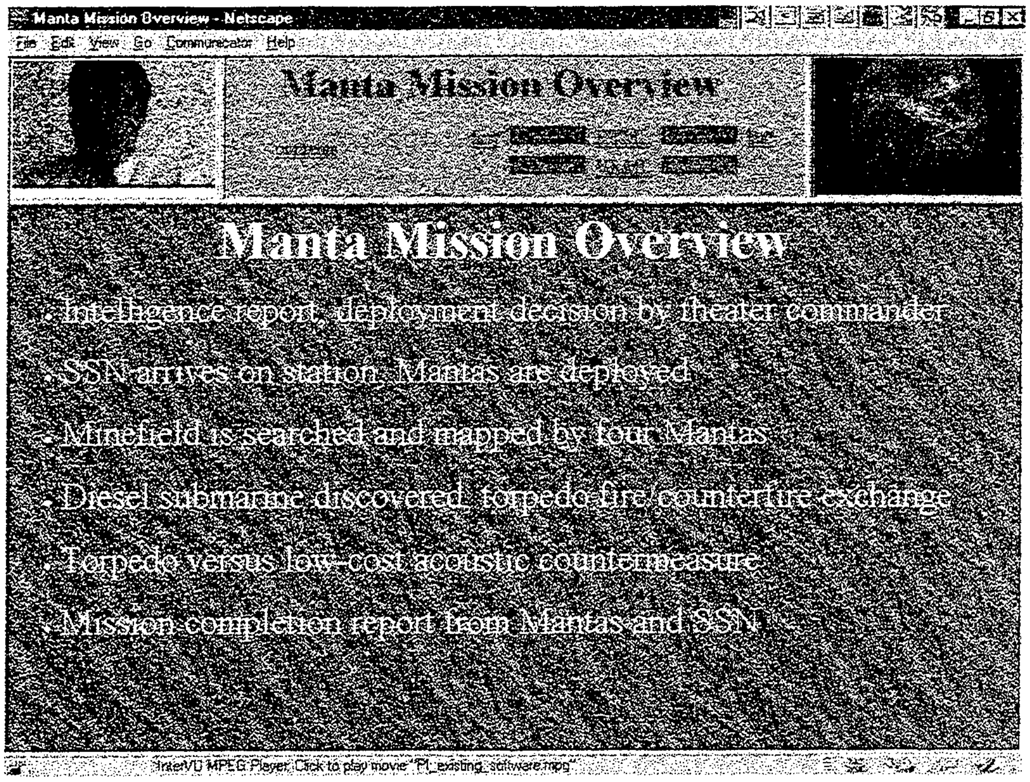
Document: Done


Manta Tactical Visualization - Netscape
File Edit View Go Communicator Help
Manta Tactical Visualization
InterVU MPEG Player: Click to play movie "P1_intro.mpg"

Manta Tactical Visualization

- Goal: examine a meaningful tactical encounter in both 2D & 3D
- Produced by NPS faculty and students under a research proposal sponsored by Naval Undersea Warfare Center (NUWC)
- 2D Hypertext Markup Language (HTML) slides
  - Links to other information placed in context
- 3D Virtual Reality Modeling Language (VRML) scenes
  - Links, multiple viewpoints and animation in 3D


Manta Tactical Visualization - Netscape
File Edit View Go Communicator Help
Manta Tactical Visualization
From the East
Document: Done

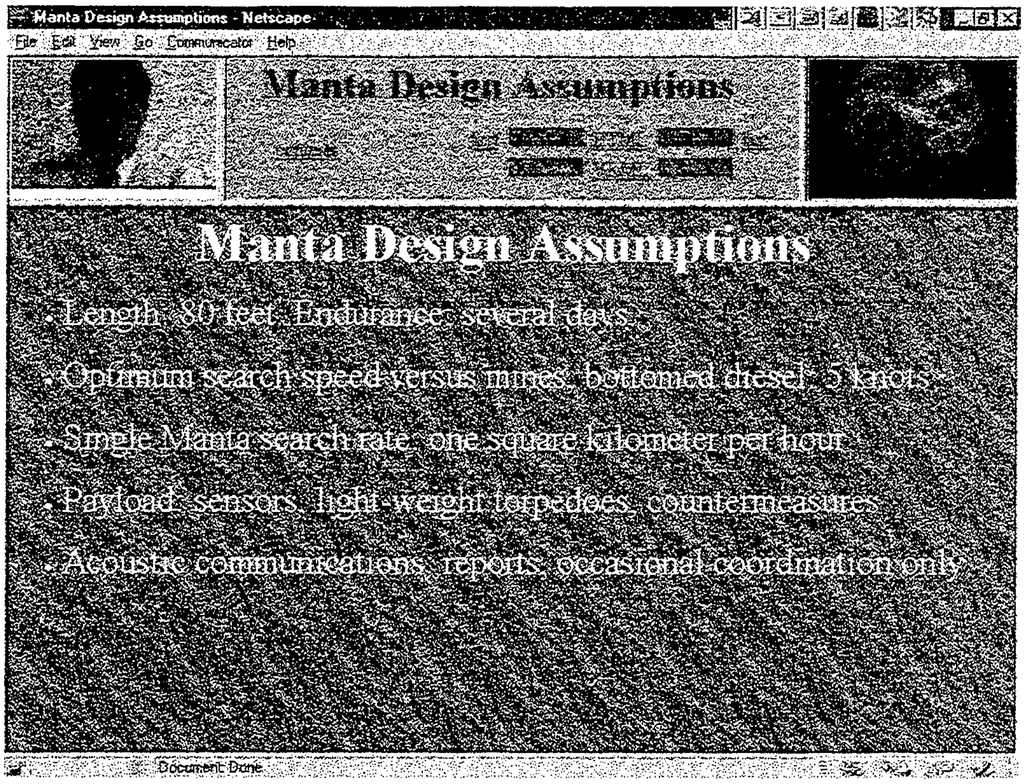

Manta Mission Overview - Netscape
File Edit View Go Communicator Help
Manta Mission Overview
overview

Manta Mission Overview

- Intelligence report, deployment decision by theater commander
- SSN arrives on station, Mantas are deployed
- Minefield is searched and mapped by four Mantas
- Diesel submarine discovered, torpedo fire/counterfire exchange
- Torpedo versus low-cost acoustic countermeasure
- Mission completion report from Mantas and SSN

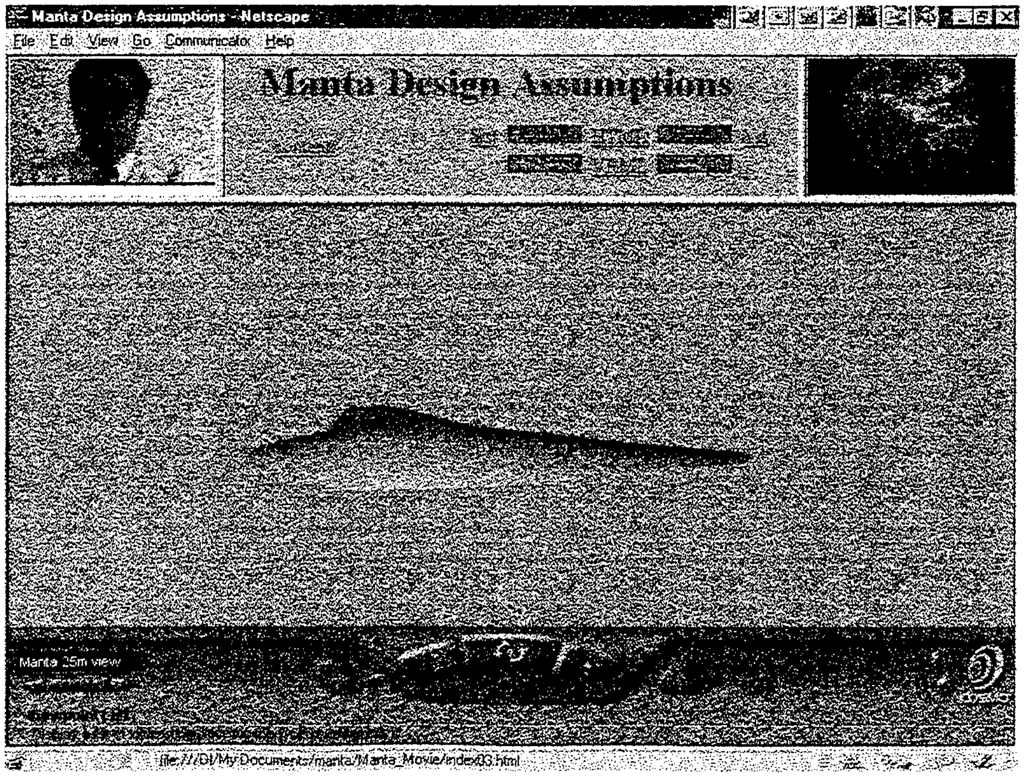
InterVU MPEG Player: Click to play movie "P1_existing_software.mpg"

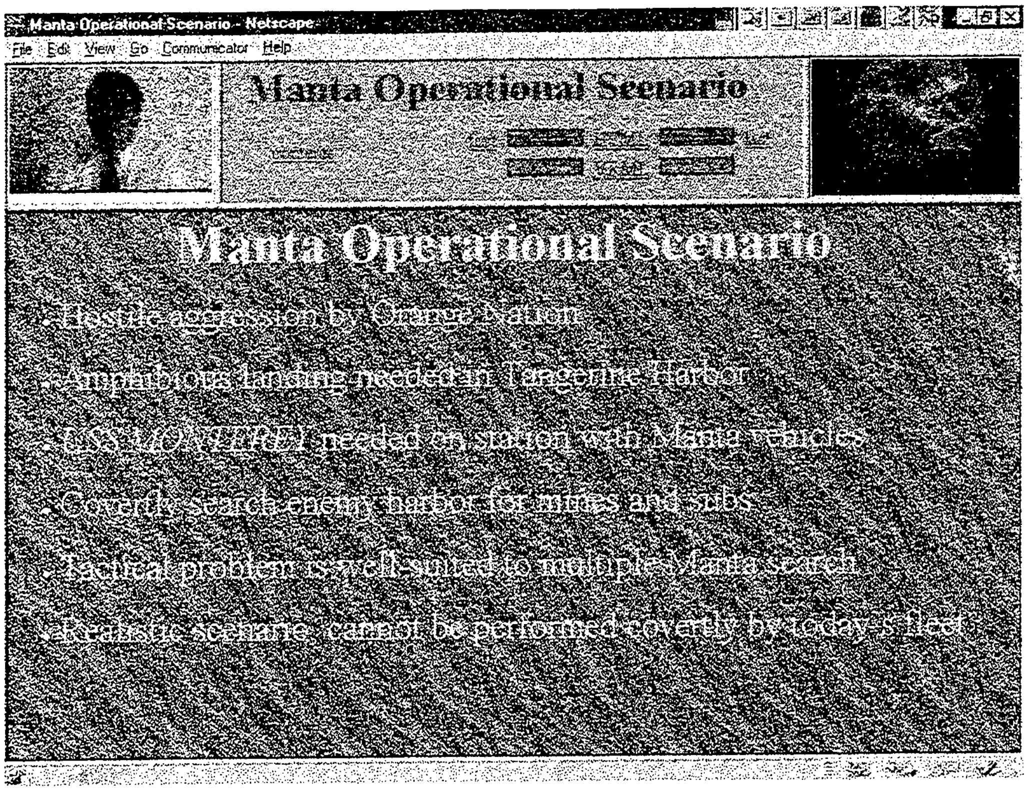
Manta Mission Overview - Netscape
File Edit View Go Communicator Help
Manta Mission Overview
Single mine-Mant
Document: Done


Manta Design Assumptions - Netscape
File Edit View Go Communicator Help
Manta Design Assumptions
Document: Done

Manta Design Assumptions

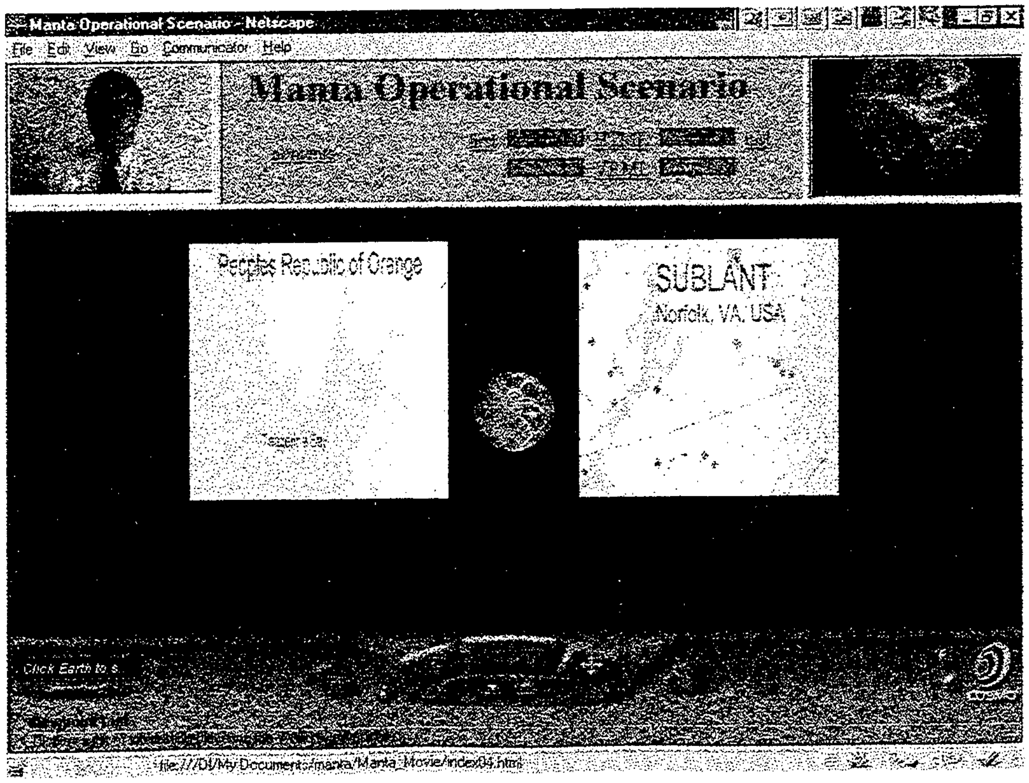
- Length: 80 feet. Endurance: several days
- Optimum search speed versus mines, bottomed diesel: 5 knots
- Single Manta search rate: one square kilometer per hour
- Payload: sensors, light-weight torpedoes, countermeasures
- Acoustic communications: reports, occasional coordination only

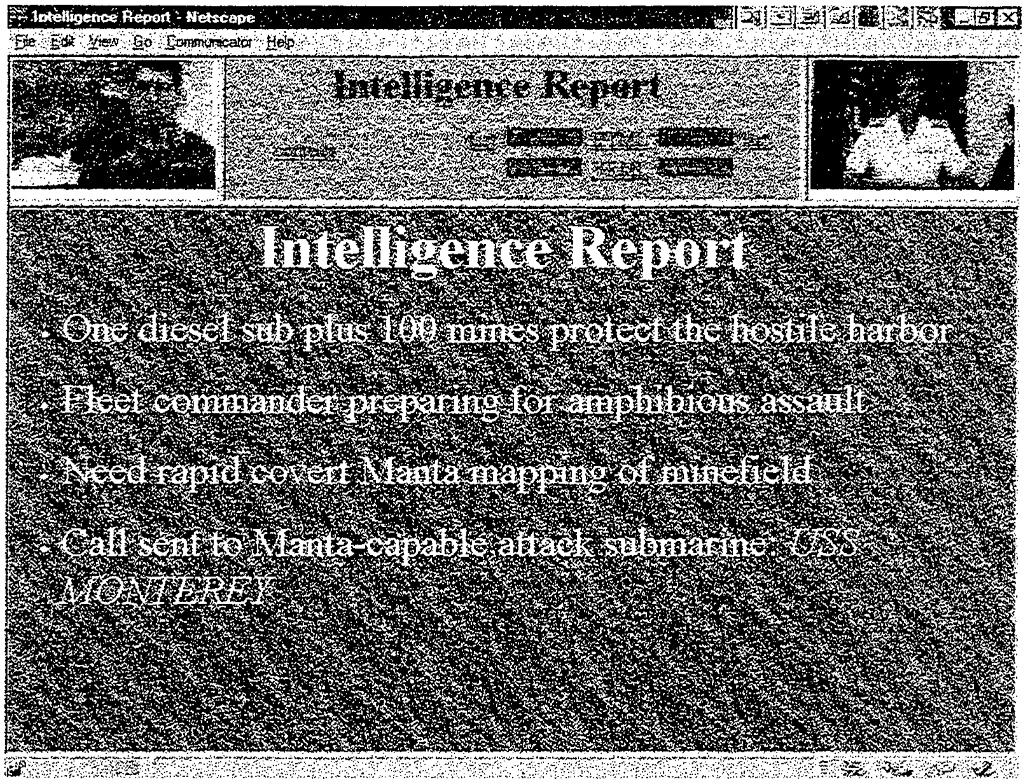

Manta Design Assumptions - Netscape
File Edit View Go Communicator Help
Manta Design Assumptions
Manta 25m view
file:///D|/My Documents/manta/Manta_Movie/index03.html


Manta Operational Scenario - Netscape
File Edit View Go Communicator Help
Manta Operational Scenario

Manta Operational Scenario

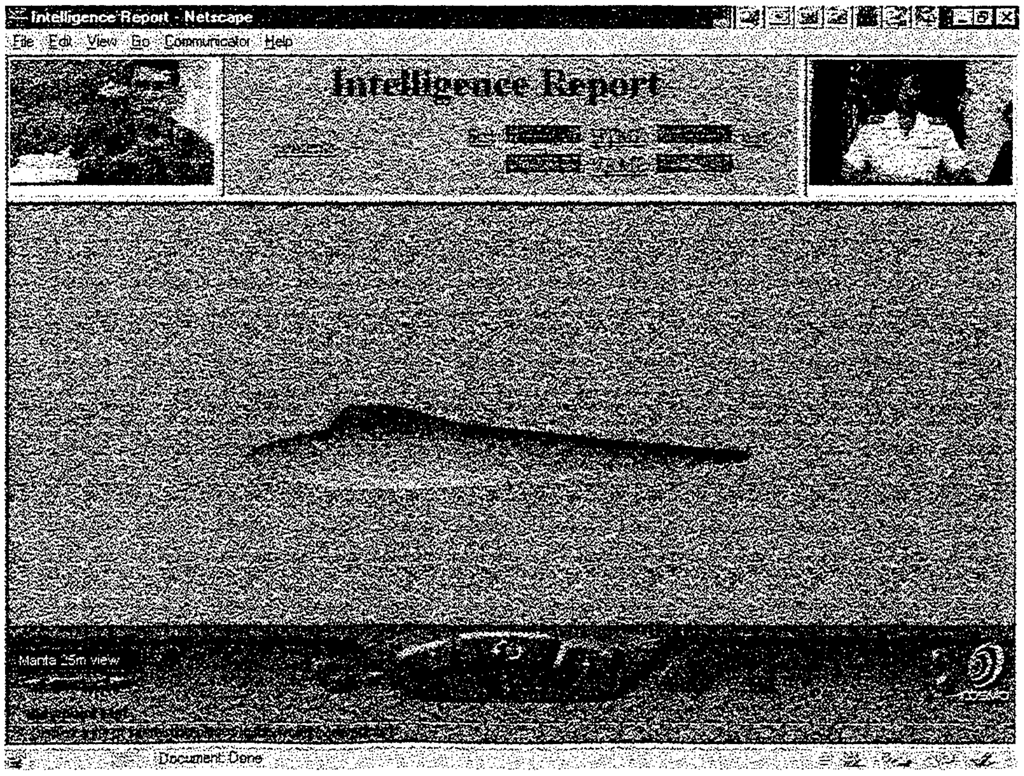
- Hostile aggression by Orange Nation
- Amphibious landing needed in Tangerine Harbor
- USS *MONTEREY* needed on station with Manta vehicles
- Covertly search enemy harbor for mines and subs
- Tactical problem is well-suited to multiple Manta search
- Realistic scenario: cannot be performed covertly by today's fleet

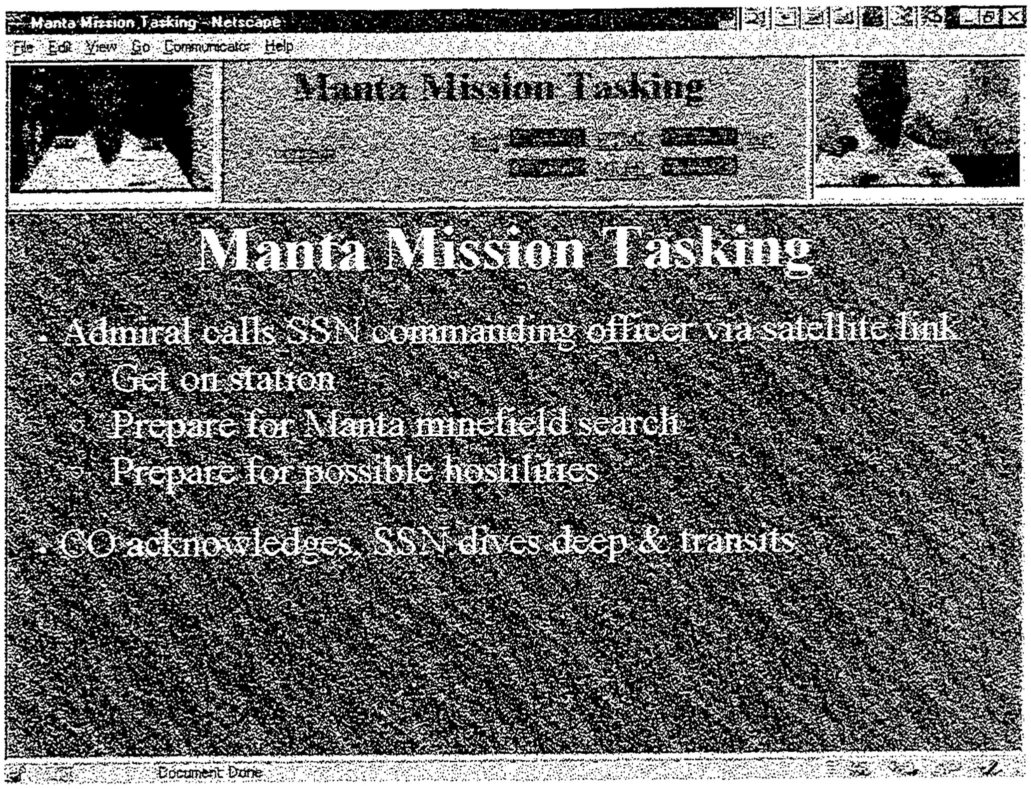

Manta Operational Scenario - Netscape
File Edit View Go Communicator Help
Manta Operational Scenario
Peoples Republic of Orange
Tangerine Bay
SUBLANT
Norfolk, VA, USA
Click Earth to s
file:///D|/My Documents/manta/Manta_Movie/index04.html


Intelligence Report - Netscape
File Edit View Go Communicator Help
Intelligence Report

Intelligence Report

- One diesel sub plus 100 mines protect the hostile harbor
- Fleet commander preparing for amphibious assault
- Need rapid covert Manta mapping of minefield
- Call sent to Manta-capable attack submarine *USS MONTEREY*

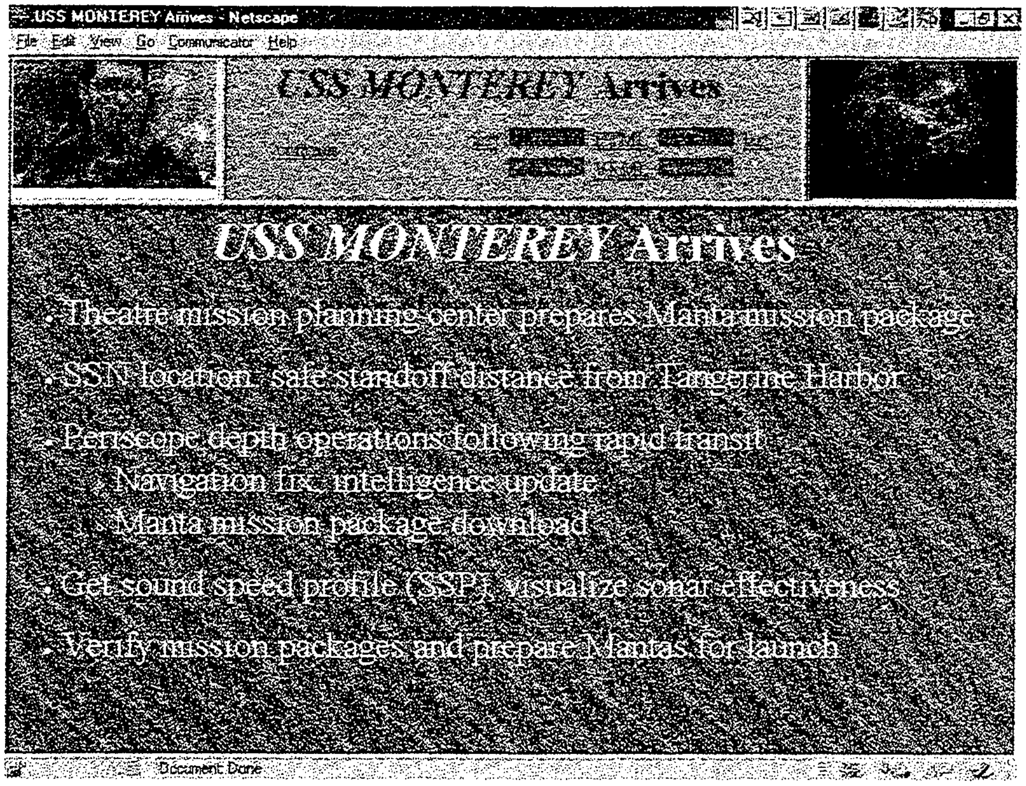

Intelligence Report - Netscape
File Edit View Go Communicator Help
Intelligence Report
Manta 25m view
Document: Done


Manta Mission Tasking - Netscape
File Edit View Go Communicator Help
Manta Mission Tasking
Document: Done

Manta Mission Tasking

- Admiral calls SSN commanding officer via satellite link
  - Get on station
  - Prepare for Manta minefield search
  - Prepare for possible hostilities
- CO acknowledges, SSN dives deep & transits

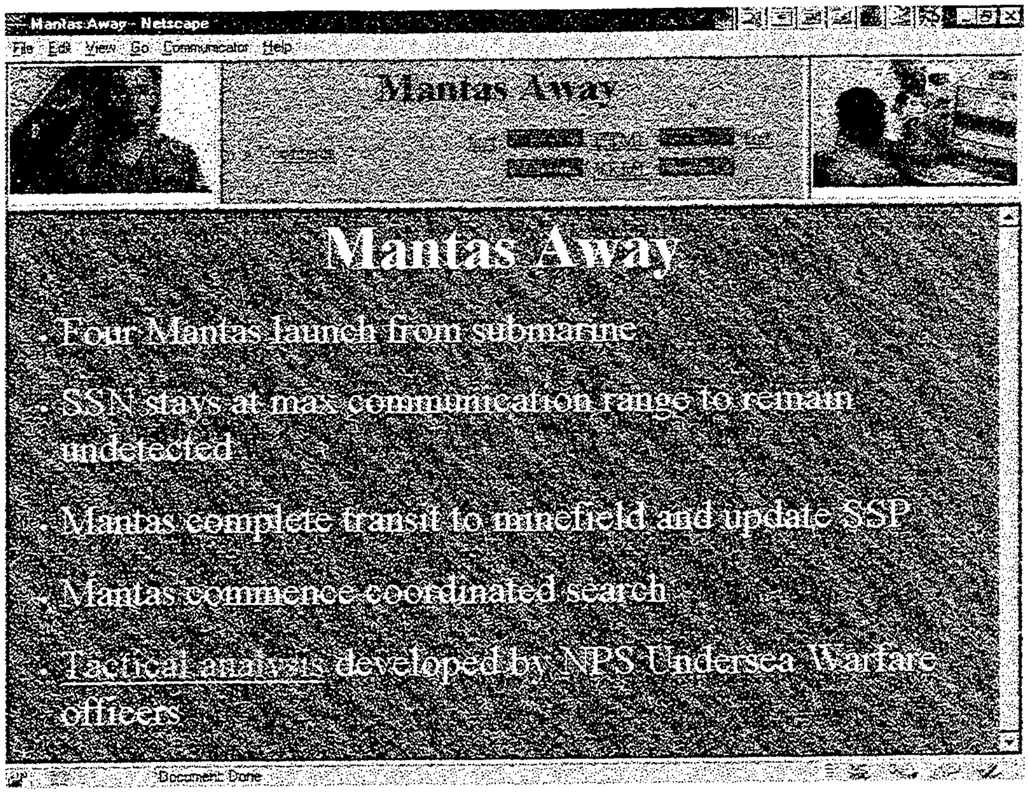
Manta Mission Tasking - Netscape
File Edit View Go Communicator Help
Manta Mission Tasking
Document: Done


USS MONTEREY Arrives - Netscape
File Edit View Go Communicator Help
USS MONTEREY Arrives
Document: Done

USS MONTEREY Arrives

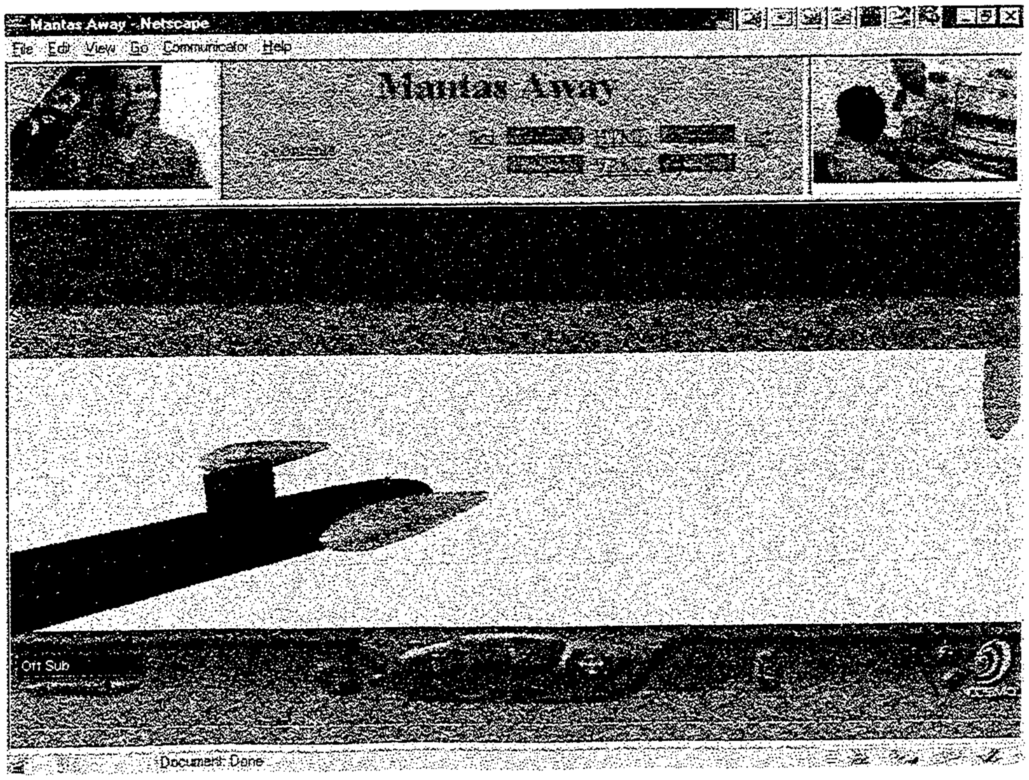
- Theatre mission planning center prepares Manta mission package
- SSN location: safe standoff distance from Tangerine Harbor
- Periscope depth operations following rapid transit
  - Navigation fix, intelligence update
  - Manta mission package download
- Get sound speed profile (SSP), visualize sonar effectiveness
- Verify mission packages and prepare Mantas for launch

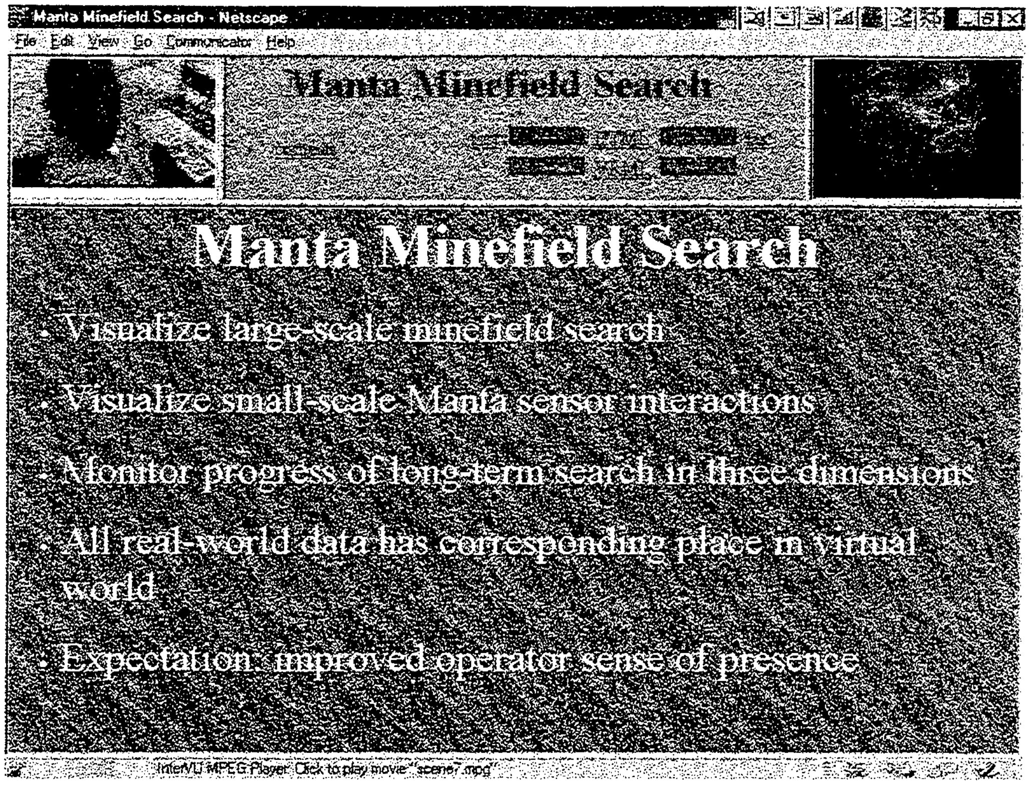
USS MONTEREY Arrives - Netscape
File Edit View Go Communicator Help
USS MONTEREY Arrives
Sub 500m away...
Document: Done



Mantas Away

- Four Mantas launch from submarine
- SSN stays at max communication range to remain undetected
- Mantas complete transit to minefield and update SSP
- Mantas commence coordinated search
- Tactical analysis developed by NPS Undersea Warfare officers




Manta Minefield Search - Netscape

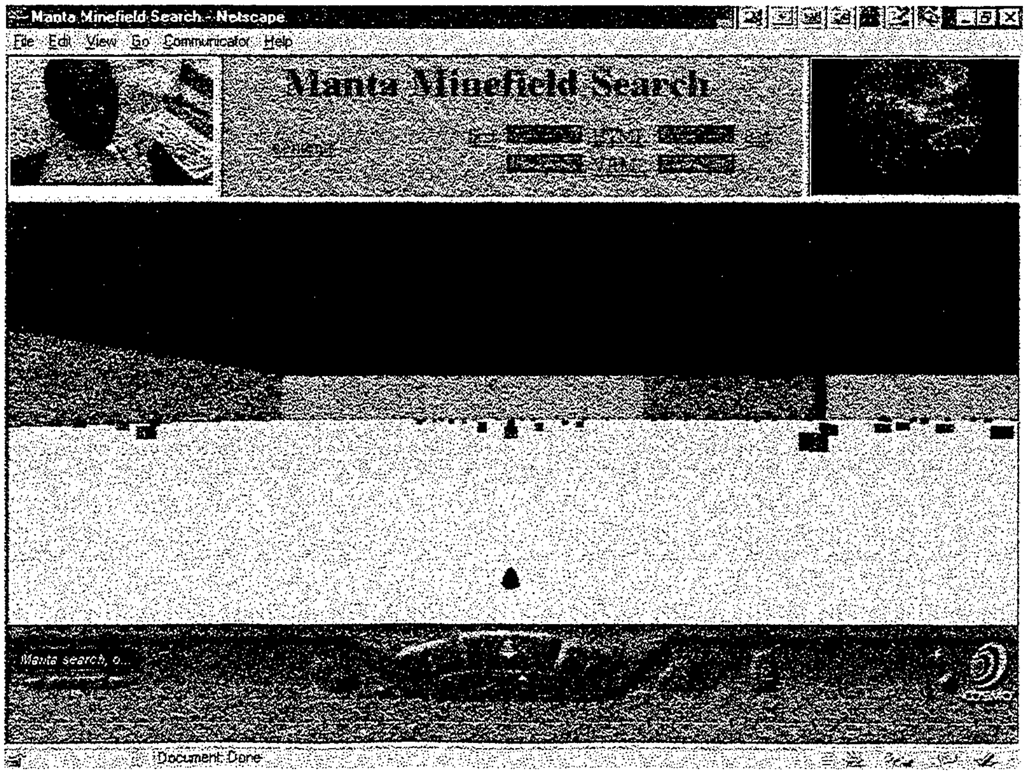
File Edit View Go Communicator Help

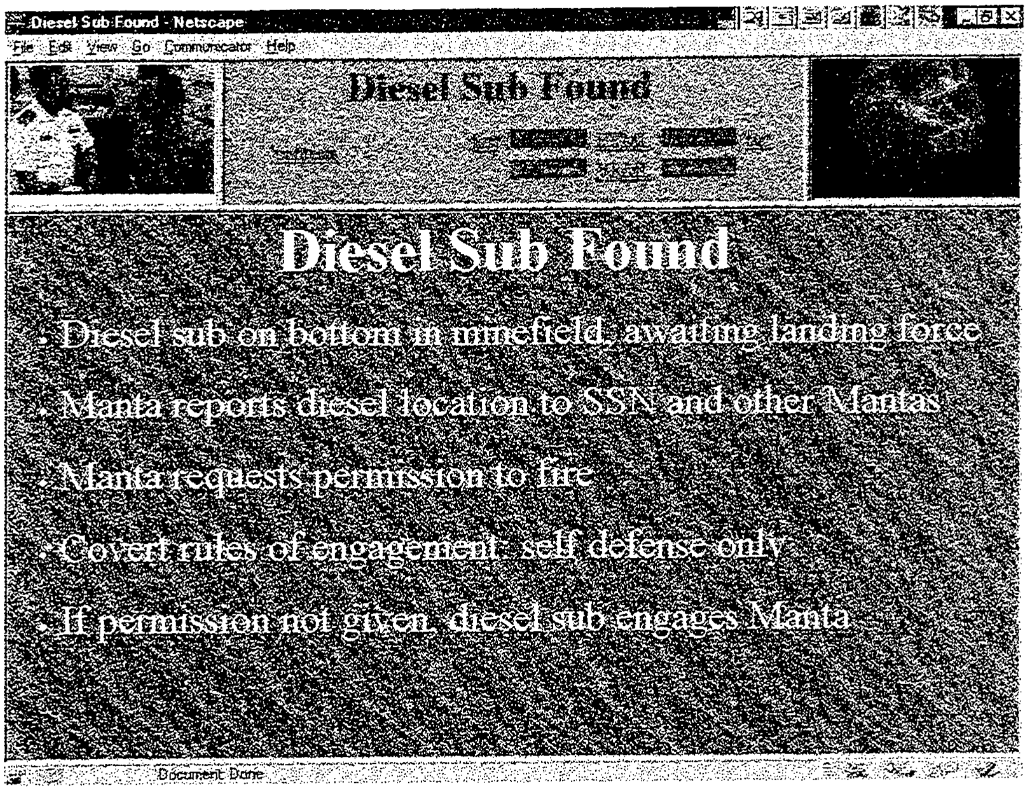
Manta Minefield Search

Manta Minefield Search

- Visualize large-scale minefield search
- Visualize small-scale Manta sensor interactions
- Monitor progress of long-term search in three dimensions
- All real-world data has corresponding place in virtual world
- Expectation: improved operator sense of presence

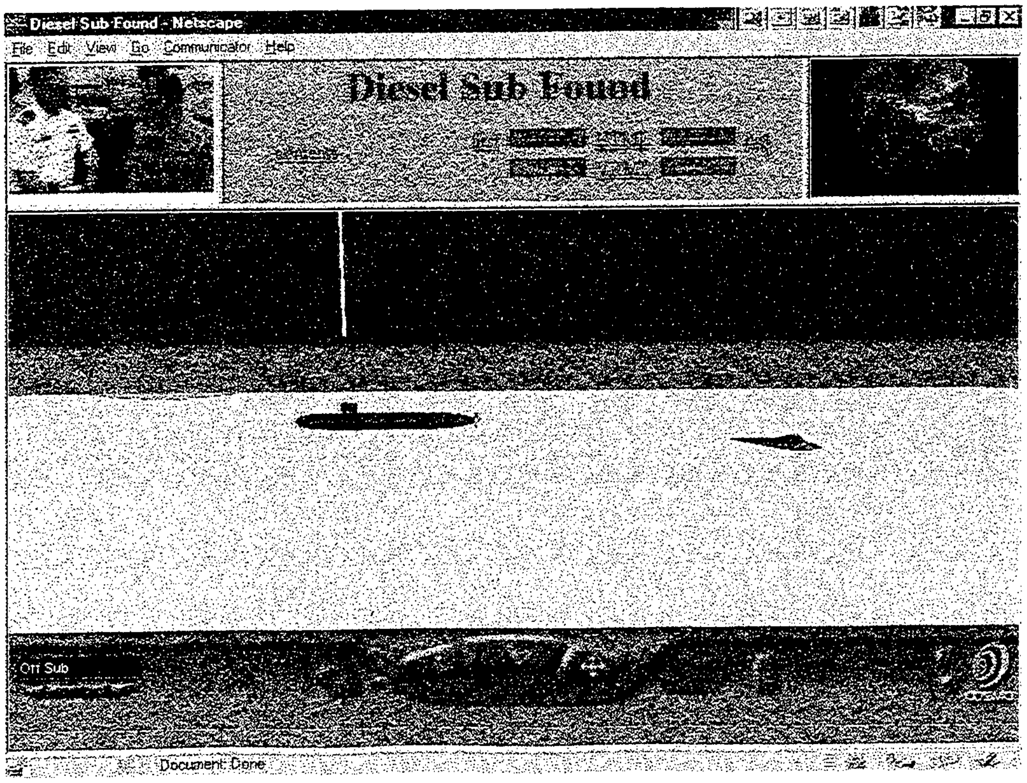
InterVU MPEG Player: Click to play movie "scene1.mpg"

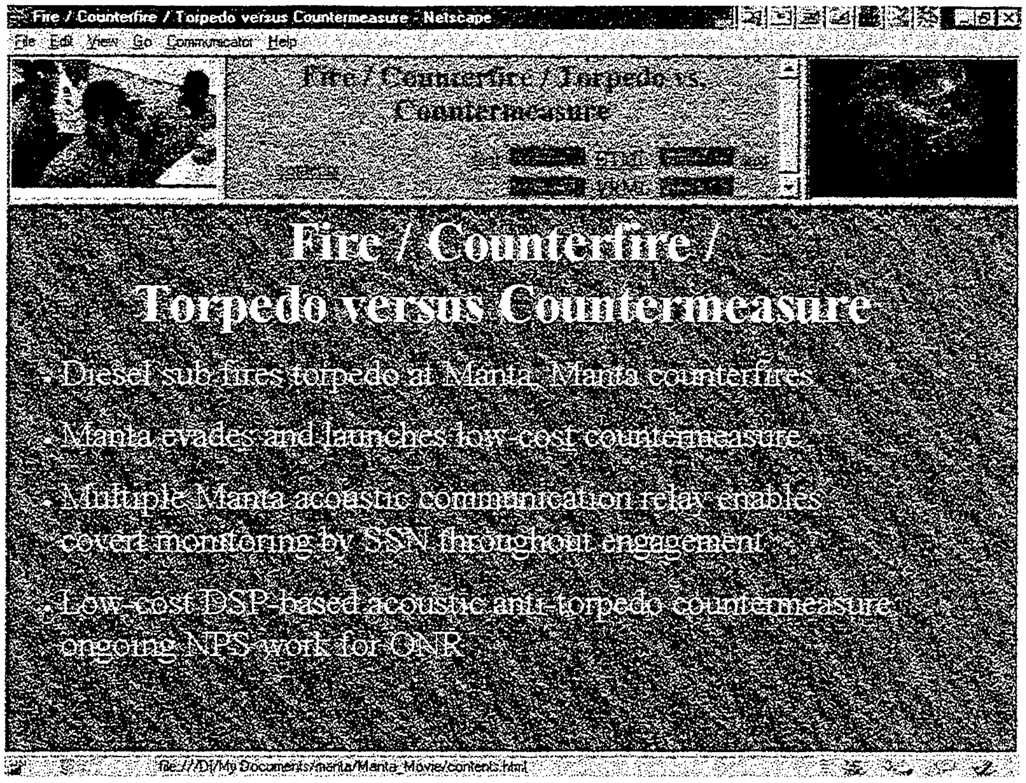

Manta Minefield Search - Netscape
File Edit View Go Communicator Help
Manta Minefield Search
Manta search, o...
Document: Done


Diesel Sub Found - Netscape
File Edit View Go Communicator Help
Diesel Sub Found
Document: Done

Diesel Sub Found

- Diesel sub on bottom in minefield, awaiting landing force
- Manta reports diesel location to SSN and other Mantas
- Manta requests permission to fire
- Covert rules of engagement: self defense only
- If permission not given, diesel sub engages Manta


Diesel Sub Found - Netscape
File Edit View Go Communicator Help
Diesel Sub Found
Off Sub
Document: Done


Fire / Counterfire / Torpedo versus Countermeasure - Netscape

File Edit View Go Communicator Help

Fire / Counterfire / Torpedo vs. Countermeasure

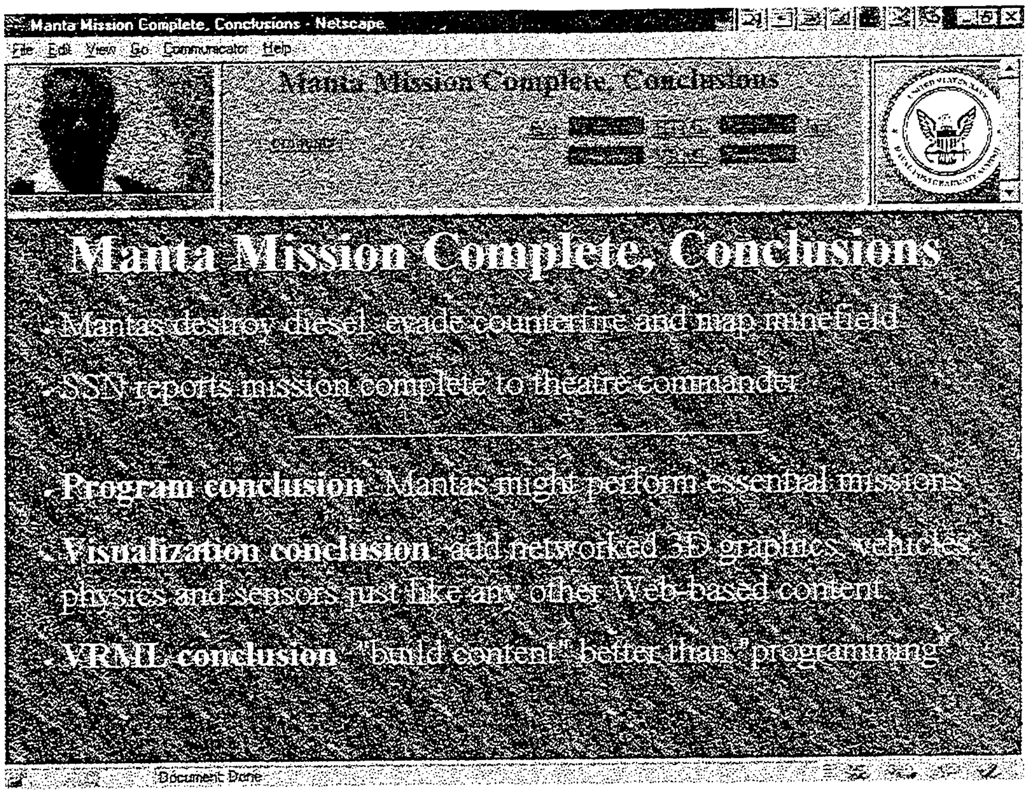
Fire / Counterfire / Torpedo versus Countermeasure

- Diesel sub fires torpedo at Manta; Manta counterfires
- Manta evades and launches low-cost countermeasure
- Multiple Manta acoustic communication relay enables covert monitoring by SSN throughout engagement
- Low-cost DSP-based acoustic anti-torpedo countermeasure: ongoing NPS work for ONR

file:///D|/My Documents/manta/Manta_Movie/contents.html

Fire / Counterfire / Torpedo versus Countermeasure - Netscape

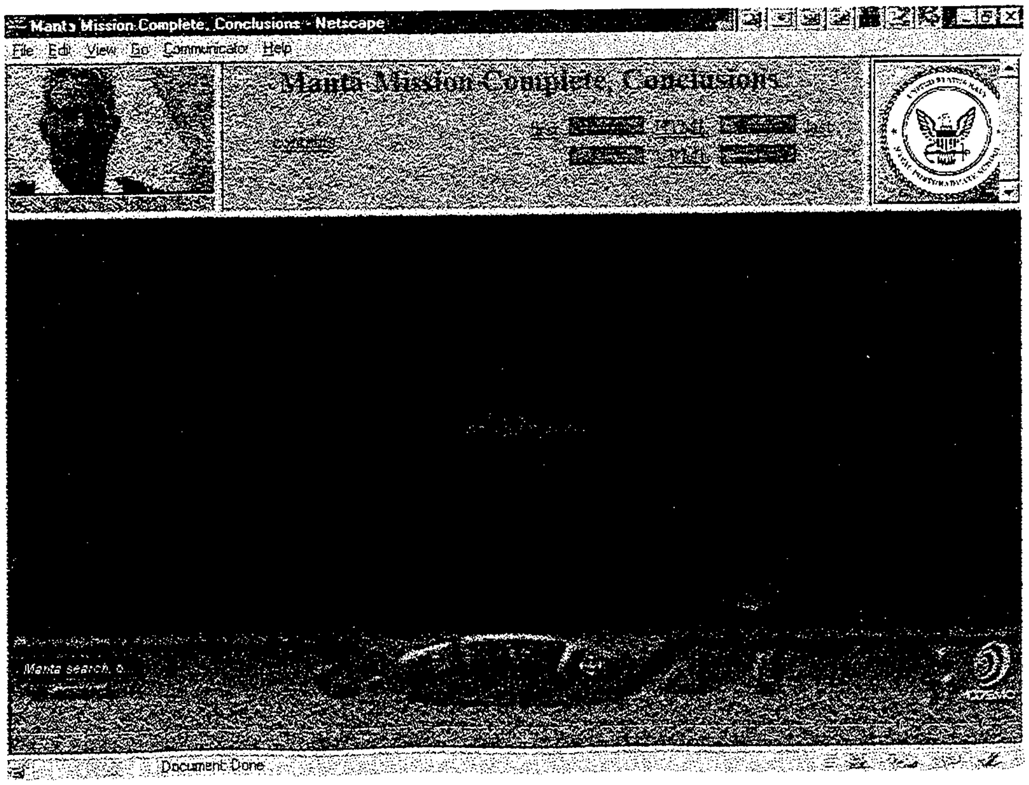
File Edit View Go Communicator Help
Fire / Counterfire / Torpedo vs. Countermeasure
MK-48 Right Side
Document: Done


Manta Mission Complete, Conclusions - Netscape
File Edit View Go Communicator Help
Manta Mission Complete, Conclusions
Document: Done

Manta Mission Complete, Conclusions

- Mantas destroy diesel, evade counterfire and map minefield
- SSN reports mission complete to theatre commander

- **Program conclusion**: Mantas might perform essential missions
- **Visualization conclusion**: add networked 3D graphics, vehicles, physics and sensors just like any other Web-based content
- **VRML conclusion**: "build content" better than "programming"


Manta Mission Complete, Conclusions - Netscape
File Edit View Go Communicator Help
Manta Mission Complete, Conclusions
Manta search, c
Document: Done

APPENDIX F. RRA API JAVADOC DOCUMENTATION

A. INTRODUCTION

This appendix gives a sample of RRA API documentation produced by *javadoc*, a Sun Microsystems product. This program comes with the Java distribution. This program can process a Java package (collection of Java classes) and produce detailed and standardized reference documents. These documents come in the Hyper Text Mark-up Language format and thus can be read by any standard web browser. This documentation generation program allows fast development of a cohesive API, since all parties can have rapid access to the most up-to-date API documentation.

Class Hierarchy

- class java.lang.Object
  - class mil.navy.nps.rra.Beam
  - class mil.navy.nps.rra.Bottom
  - class mil.navy.nps.rra.ExampleBeamDynamic
  - class mil.navy.nps.rra.ExampleBeamStatic
  - class mil.navy.nps.rra.ExampleLobeDynamic
  - class mil.navy.nps.rra.ExampleLobeStatic
  - class mil.navy.nps.rra.ExampleRay
  - class mil.navy.nps.rra.Lobe
  - class mil.navy.nps.rra.PrintVRML
  - class mil.navy.nps.rra.Ray
  - class mil.navy.nps.rra.SVP
  - class mil.navy.nps.rra.Surface
  - class mil.navy.nps.rra.Targets
  - class mil.navy.nps.rra.Vec3d

Class mil.navy.nps.rra.Beam

```
java.lang.Object
   |
   +----mil.navy.nps.rra.Beam
```

public class **Beam**
extends Object

Author:
LT Timothy M. Holliday (http://www.stl.nps.navy.mil/~auv/holliday)

Location:
http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/rra/Beam.java

Hierarchy Diagram:
Class Hierarchy

Coordinate System Diagram:
Coordinate Systems

Summary:
Forms a beam from four rays, propagates the beam and can return a VRML representation of the beam.

Explanation:
Beams of energy can be considered to consist of bundles of rays. The energy in a bundle can be shown not to diverge from the bundle. Thus the energy in a bundle is constant and thus the product of the intensity and the area of the bundle perpendicular to direction of propagation is a constant as well. Therefore a beam is a fundemental building block for a lobe of a sonar pattern.

History:
15Nov97 /Timothy M. Holliday /New
17Mar98 /Timothy M. Holliday /Added HTML comment convention
12Apr98 /Timothy M. Holliday /Parameterless Constructors
14Apr98 /Timothy M. Holliday /Simplified VRML Routines
21Apr98 /Timothy M. Holliday /Fixed Problem with calculateSoundPressureLevel

See Also:
Ray, BeamExampleStatic, BeamExampleDynamic, Bottom, Surface, Vec3d

Variable Index

- **NONE**
- **T_L**
- **TIME**

Constructor Index

- **Beam**()
  Constructor for the Beam class.

Method Index

- **calculateBeam**(Targets)
  This method calculates the trajectory of the beam of energy enclosed by the defining rays of the beam tube.
- **calculateSoundPressureLevel**(double[][][], double, double, double)
  This method calculates the trajectory of the beam of energy enclosed by the defining rays of the beam tube.
- **dynamicVRML**()
  This method creates a dynamic VRML string shape that is the three dimensional representation of the beam pulse that is propogated.
- **getAppendLineFeed**()
  This is a static method that returns the current line appendage.
- **getAzimuth**()
  This method returns the azimuthal angle.
- **getBeamNumber**()
  This returns BeamNumber.
- **getBottom**()
  This method returns the handle to the bottom object.
- **getDeltaTime**()
  This method returns the simulation time step.
- **getDetectCount**()
  This method returns the total number of detects by the beam
- **getDetectEchoLevel**(int)
  This method returns the echo level of the detected target
- **getDetectTime**(int)
  This method returns the detect time for a given detect in the beam
- **getDuration**()
  This method sets the duration of the sonar pulse.
- **getElevation**()

This method returns the elevation angle.

- **getEndTime**()
  This method returns the simulation end time.
- **getHalfBeamWidthX**()
  This method returns the half beam width of the beam in the azimuthal direction.
- **getHalfBeamWidthY**()
  This method returns the half beam width of the beam in the azimuthal direction.
- **getPosition**()
  This method returns the position of the beam.
- **getSsp**()
  This method returns the handle to the sound speed profile object.
- **getSurface**()
  This method returns the handle to the surface object.
- **pingTimerVRML**()
  This method returns a VRML timer string that contains the appropriate timing information for the beam
- **reset**()
  This method resets all of the beam parameters after instanciation has occurred since reuse is more time efficient than garbage collection and reallocation.
- **setAppendLineFeed**(boolean)
  This is a static method used to indicate whether a line feed is desired at the end of every line.
- **setAzimuth**(double)
  This method sets the azimuthal angle, which is the angle from the x-axis to the z-axis rotating about the y-axis.
- **setBeamNumber**(int)
  This method sets BeamNumber.
- **setBottom**(Bottom)
  This method sets the handle to the bottom object.
- **setDeltaTime**(double)
  This method sets the time step in the simulation.
- **setDuration**(int)
  This method sets the duration of the sonar pulse.
- **setElevation**(double)
  This method sets the elevation angle, which is the angle from the y-axis to the x-axis rotating about the z-axis .
- **setEndTime**(double)
  This method sets the simulation end time.
- **setHalfBeamWidthX**(double)
  This method sets the half beam width of the beam in the azimuthal direction.
- **setHalfBeamWidthY**(double)
  This method sets the half beam width of the beam in the elevation direction.
- **setPosition**(double, double, double)
  This method sets the position of the beam.
- **setSsp**(SSP)
  This method sets the handle to the sound speed profile object.
- **setSurface**(Surface)

This method sets the handle to the surface object.

- staticVRML(int, int)

  This method writes to the console a VRML shape that is the three dimensional representation of the beam that is propagated.

Variables

● T_L

```
public static final int T_L
```

● TIME

```
public static final int TIME
```

● NONE

```
public static final int NONE
```

Constructors

● Beam

```
public Beam()
```

Constructor for the Beam class. A beam is defined as the volume swept out by four rays as they traverse the ocean environment

Methods

● reset

```
public void reset()
```

This method resets all of the beam parameters after instanciation has occurred since reuse is more time efficient than garbage collection and reallocation.

● setAzimuth

```
public void setAzimuth(double phi)
```

This method sets the azimuthal angle, which is the angle from the x-axis to the z-axis rotating about the y-axis.

● **getAzimuth**

```
public double getAzimuth()
```

This method returns the azimuthal angle.

● **setHalfBeamWidthX**

```
public void setHalfBeamWidthX(double pHalfBeamWidthX)
```

This method sets the half beam width of the beam in the azimuthal direction.

● **getHalfBeamWidthX**

```
public double getHalfBeamWidthX()
```

This method returns the half beam width of the beam in the azimuthal direction.

● **setElevation**

```
public void setElevation(double beta)
```

This method sets the elevation angle, which is the angle from the y-axis to the x-axis rotating about the z-axis .

● **getElevation**

```
public double getElevation()
```

This method returns the elevation angle.

● **setHalfBeamWidthY**

```
public void setHalfBeamWidthY(double pHalfBeamWidthY)
```

This method sets the half beam width of the beam in the elevation direction.

● **getHalfBeamWidthY**

```
public double getHalfBeamWidthY()
```

This method returns the half beam width of the beam in the azimuthal direction.

● **setPosition**

```
public void setPosition(double x,
                        double y,
                                 double z)
```

This method sets the position of the beam.

- **getPosition**

```
public Vec3d getPosition()
```

This method returns the position of the beam.

- **setDeltaTime**

```
public void setDeltaTime(double pDeltaTime)
```

This method sets the time step in the simulation.

- **getDeltaTime**

```
public double getDeltaTime()
```

This method returns the simulation time step.

- **setEndTime**

```
public void setEndTime(double pEndTime)
```

This method sets the simulation end time. This value is reletive to the start time which is 0.0.

- **getEndTime**

```
public double getEndTime()
```

This method returns the simulation end time.

- **setDuration**

```
public void setDuration(int duration)
```

This method sets the duration of the sonar pulse. The integer refers to the number of deltaTime increments. currently only 1 or 2 are allowed.

- **getDuration**

```
public int getDuration()
```

This method sets the duration of the sonar pulse. The integer refers to the number of deltaTime increments.

● **setBottom**

```
public void setBottom(Bottom pBottom)
```

This method sets the handle to the bottom object.

● **getBottom**

```
public Bottom getBottom()
```

This method returns the handle to the bottom object.

● **setSurface**

```
public void setSurface(Surface pSurface)
```

This method sets the handle to the surface object.

● **getSurface**

```
public Surface getSurface()
```

This method returns the handle to the surface object.

● **setSsp**

```
public void setSsp(SSP pSsp)
```

This method sets the handle to the sound speed profile object.

● **getSsp**

```
public SSP getSsp()
```

This method returns the handle to the sound speed profile object.

● **setBeamNumber**

```
public void setBeamNumber(int number)
```

This method sets BeamNumber. It is used to uniquely identify each beam for ROUTEing in VRML.

● **getBeamNumber**

```
public int getBeamNumber()
```

This returns BeamNumber. Which is used to uniquely identify each beam for ROUTEing in VRML.

- **staticVRML**

```
public String staticVRML(int colorChoice,
                         int intensityChoice)
```

This method writes to the console a VRML shape that is the three dimensional representation of the beam that is propagated.

- **dynamicVRML**

```
public String dynamicVRML()
```

This method creates a dynamic VRML string shape that is the three dimensional representation of the beam pulse that is propogated.

- **calculateBeam**

```
public void calculateBeam(Targets targets)
```

This method calculates the trajectory of the beam of energy enclosed by the defining rays of the beam tube. It also determines when there is a detection.

- **calculateSoundPressureLevel**

```
public void calculateSoundPressureLevel(double field[][][],
                                        double deltaRange,
                                        double deltaDepth,
                                        double frequency)
```

This method calculates the trajectory of the beam of energy enclosed by the defining rays of the beam tube.

- **getDetectCount**

```
public int getDetectCount()
```

This method returns the total number of detects by the beam

- **getDetectTime**

```
public double getDetectTime(int N)
```

This method returns the detect time for a given detect in the beam

● getDetectEchoLevel

```
public double getDetectEchoLevel(int N)
```

This method returns the echo level of the detected target

● pingTimerVRML

```
public String pingTimerVRML()
```

This method returns a VRML timer string that contains the appropriate timing information for the beam

● setAppendLineFeed

```
public static void setAppendLineFeed(boolean pAppendLineFeed)
```

This is a static method used to indicate whether a line feed is desired at the end of every line. 'true' indicates a linefeed is desired and 'false' indicates that a space is desired"

● getAppendLineFeed

```
public static boolean getAppendLineFeed()
```

This is a static method that returns the current line appendage.

All Packages Class Hierarchy This Package Previous Next Index

APPENDIX G. RRA VIDEO INFORMATION

A. INTRODUCTION

This appendix provides information on how to obtain a copy of a video that shows the abilities of the RRA API and the simulations and visualizations obtained from it. The point of contact for obtaining a copy is Dr. Don Brutzman, *brutzman@nps.navy.mil.*

APPENDIX H. RRA CD ROM INFORMATION

A. INTRODUCTION

This appendix provides information on how to obtain a copy of a CD ROM that has this thesis, the RRA API source code, example programs using the RRA API, the simulations and visualizations obtained from using the RRA API and the contents of Appendix D, Appendix E and Appendix F. The point of contact for obtaining a copy is Dr. Don Brutzman, *brutzman@nps.navy.mil.*

LIST OF REFERENCES

Brutzman, Don, *A Virtual World for an Autonomous Underwater Vehicle*, Doctoral Dissertation, Naval Postgraduate School, Monterey, California, December 1994.

Brutzman, Don, "AUV Tactical Minefield Search," *Symposium on Technology and The Mine Problem*, Naval Postgraduate School, Monterey California, February 1997.

Brutzman, Don, "Graphics Internetworking: Bottlenecks and Breakthroughs," chapter four, *Digital Illusions*, Clark Dodsworth editor, Addison-Wesley, Reading Massachusetts, August 1997.

Brutzman, Donald P., Eugene Chan, Mark Evans, Timothy Holliday, Michael Huck, Robert Jezek, BinBing Ma, Steve Murley, Ronald Toland, Young Yee, "Minefield Search Tactic Evaluation using 4 Autonomous Manta UUVs", *Symposium on Technology and The Mine Problem* , April 6-10 1998.

Brutzman, Don, Healey, Tony, Marco, Dave and McGhee, Bob, "The *Phoenix* Autonomous Underwater Vehicle," chapter 13, *AI-Based Mobile Robots*, editors David Kortenkamp, Pete Bonasso and Robin Murphy, MIT/AAAI Press, Cambridge Massachusetts, 1998.

Brutzman, Don, The Virtual Reality Modeling Language and Java, *Communications of the ACM*, to appear 1998.

Clay, Clarence, Medwin, Herman, *Acoustical Oceanography*, First Edition, John Wiley and Sons, New York, NY, 1977.

Davis, Duane, *Precision Maneuvering and Control of the Phoenix Autonomous Underwater Vehicle for Entering a Recovery Tube*, Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1996.

Defanti, Thomas, "Insight Through Images," *A Unix Review*, Vol. 7, No. 3, 1987

Etter, Paul, *Underwater Acoustic Modeling*, Second Edition, University Press, Cambridge, UK, 1996.

Fishwick, Paul, *Simulation Model Design and Execution*, Prentice Hall, Englewood Cliffs, NJ, 1995.

Hardin, R.H., Tappert, F.D., "Applications of the Split-step Fourier Method to the Numerical Solution of Nonlinear and Variable Coefficient Wave Equations," *SIAM* rev. 15, 1973.

Hill, Jim, Warren, Michael, Goda, Pat, "I'm Not Going to Pay a Lot for This Supercomputer!," *Linux Journal*, January 1998.

Kinsler, Lawrence, Fry, Austin, Coppens, Alan, Sanders, James, *Fundamentals of Acoustics*, Third Edition, John Wiley and Sons, New York, NY, 1982.

Karahalios, Margarida, "Underwater Source Localization Using Scientific Data Visualization," Department of Computer Science and Engineering, University of South Florida, July 1991.

Roehl, Bernie, Couch, Justin, Reed-Ballreich, Cindy, Rohaly, Tim, Brown, Geoff, *Late Night VRML 2.0 with Java*, Ziff-Davis Press, Emeryville, CA, 1997

Smith, Kevin, Tappert, F.D., "Ray Chaos in Underwater Acoustics," *Acoustical Society of America Journal*, December 18, 1991.

Stewart, W.K., "Visualization Resources and Strategies for Remote Subsea Exploration," *The Visual Computer International Journal of Computer Graphics*, Vol. 8, No. 5-6, 1992.

Urick, Robert, *Principles of Underwater Sound,* Second Edition, McGraw-Hill, New York, NY, 1975

Ziomek, Lawrence, "The RRA Algorithm: Recursive Ray Acoustics for Three-Dimensional Speeds of Sound," *IEEE Journal of Oceanic Engineering*, Vol. 18, No. 1, January 1993.

Ziomek, Lawrence, *Fundamentals of Acoustic Field Theory and Space-Time Signal Processing*, First Edition, CRC Press, Boca Raton, FL, 1995.

INITIAL DISTRIBUTION LIST

1 . Defense Technical Information Center..2
8725 John J. Kingman Rd., Ste 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3. Dr. Donald P. Brutzman, Code UW/BR 1
Undersea Warfare Academic Group
Naval Postgraduate School
Monterey, CA 93943

4. Dr. Kevin B. Smith, Code PH/SK..1
Physics Department
Naval Postgraduate School
Monterey, CA 93943

5. CDR Mitch Shipley, Code PH/SM..1
Physics Department
Naval Postgraduate School
Monterey, CA 93943

6. Timothy M. Holliday..2
3001 Santa Clara S.E.
Albuquerque, NM 87106

7. Dr. William Maier, Code PH/MW..1
Department of Physics
Naval Postgraduate School
Monterey, CA 93943

8. Dr. Jim Eagle, Code UW 1
Undersea Warfare Academic Group
Naval Postgraduate School
Monterey, CA 93943

9. Dr. Lawrence Ziomek, Code ECE/ZM.. 1
Electrical Engineering Department
Naval Postgraduate School
Monterey, CA 93943

10. Commander, Naval Undersea Warfare Center (NUWC) Division.. 1
1176 Howell Street
Attn: Erik Chaum, Code 2251, Building 1171-3
Combat Systems Engineering and Analysis Laboratory (CSEAL)
Newport, RI 02841-1708

11. Dr. James Bellingham.. 1
Undersea Vehicles Laboratory, MIT Sea Grant College Program
Northeastern University
East Point, Nahant, Massachusetts 01908

12. W. Kenneth Stewart.. 1
Deep Submergence Laboratory
Woods Hole Oceanographic Institution Woods Hole, MA 02534

13. Sam Smith.. 1
Florida Atlantic University
777 Glades Road, Room 176, Bldg. 36
Boca Raton FL, 33431

14. David Zelzer... 1
Sarnoff Corp.
CN 5300
Princeton, NJ 08543-5300

15. Dr. Tony Healey ME/HY..1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, CA 93943