AFIT/GCS/ENG/97D-05

Visualization and Animation of a Missile/Target Encounter

THESIS

Jeffrey T. Bush, Captain, USAF

AFIT/GCS/ENG/97D-05

The views expressed in this thesis are those of the author and do not reflect the official policy of position of the Department of Defense or the U. S. Government.

AFIT/GCS/ENG/97D-05

Visualization and Animation

of a

Missile/Target Encounter

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science.

Jeffrey T. Bush, B.S.C.S.

Captain, USAF

December, 1997

AFIT/MS/ENG/97D-05

VISUALIZATION AND ANIMATION OF A MISSILE/TARGET ENCOUNTER

Jeffrey T. Bush, B.S.C.S.
Captain, USAF

Approved:

Keith A. Shomper, PhD, Major, Committee Chairman
Department of Electrical and Computer Engineering — 2 Dec 97 Date

Martin R. Stytz, PhD, LtCol, Committee Member
Department of Electrical and Computer Engineering — 2 Dec 97 Date

Mark A. Kanko, PhD, Major, Committee Member
Department of Electrical and Computer Engineering — 2 Dec 97 Date

Acknowledgments

I would like to thank 1) my wife, Shani, 2) my advisor, Major Keith Shomper, and 3) my savior, the Lord Jesus Christ for: 1) love, support and understanding, 2) wisdom, guidance and fellowship, and 3) salvation, peace and full life (respectively).

Jeffrey T. Bush

Table of Contents

List of Figures

List of Tables

Abstract

Existing missile/target encounter modeling and simulation systems focus on improving probability of kill models. Little research has been done to visualize these encounters. These systems can be made more useful to the engineers by incorporating current computer graphics technology for visualizing and animating the encounter. Our research has been to develop a graphical simulation package for visualizing both endgame and full fly-out encounters. Endgame visualization includes showing the interaction of a missile, its fuze cone proximity sensors, and its target during the final fraction of a second of the missile/target encounter. Additionally, this system displays dynamic effects such as the warhead fragmentation pattern and the specific skewing of the fragment scattering due to missile yaw at the point of detonation. Fly-out visualization, on the other hand, involves full animation of a missile from launch to target. Animating the results of VisSim fly-out simulations provides the engineer a more efficient means of analyzing his data. This research also involves investigating fly-out animation via the World Wide Web.

Visualization and Animation of a

Missile/Target Encounter

1. Introduction

1.1 Overview

Maximizing the probability of kill (*Pk*) in a missile-target engagement is an important element in the Air Force's primary goal of obtaining and maintaining air superiority. It affects our performance in aerial combat. Therefore, we must pay careful attention to technology which holds promise for improving the design of air-to-air missiles.

Computer simulation has proven effective in providing an economic means of predicting the results of real world events. For example, the Air Force Armament Laboratory uses computer simulation to provide engineers an economical means of evaluating a warhead's effectiveness against its target during and endgame[1] scenario. Computer simulation is also used to model the full fly-out of a missile from launch to detonation. The information obtained from these missile/target simulation tools currently assists engineers in the Armament Laboratory in designing missile guidance, fuzing, and warhead components. Furthermore, these automated tools are helpful in identifying necessary improvements in offensive and defensive weapon systems. Specifically, target damage predictions are useful in analyzing optimal munitions quantity, delivery platform, and tactics [Shirley93:1]. Many of these systems were developed during the 70's and mid 80's and are still being used today. Some provide graphical feedback of the simulation results; however, the graphics are very rudimentary and limit the tools' effectiveness. The existing simulation systems can be made more useful to the engineer by

providing capabilities for graphically visualizing their results.

Humans assimilate graphical information more efficiently than its textual and numerical equivalents. Current trends in usability engineering include information visualization as one element in increased software effectiveness and overall user productivity [Nielson93][Marcus, et al 93]. Also, advances in computer graphic technology make scientific visualization possible. This technology should be applied to the visualization of missile/target encounters. Work to do so began in 1996 by Lt. Joseph Moritz, a graduate student at the Air Force Institute of Technology (AFIT) [Moritz96]. The result of his research was of the AFIT Missile Endgame Simulation program (AMES), a three-dimensional endgame visualization tool built on the Silicon Graphics Inc. workstation platform (SGI). This research advances Moritz's efforts by creating the AFIT Missile Visualization System (AMVS), an improved visualization and animation system for the endgame as well as full fly-out encounters.

1.2 Summary of Current Knowledge

From the 1970's until now, several software packages have been written to simulate a missile/target encounter. Most of these software systems focus on accurate modeling of real-world physics and not on the graphical display of the simulation results. Oftentimes, a lack of computational power combined with the large amount of modeling data made graphical display of the endgame impractical. The following sections summarize previous efforts to visualize endgame simulation. For a more complete discussion of endgame simulation software see [Mortiz96:2-1..2-12].

1.2.1 FASTGEN

FASTGEN, developed in the 1970's, was one of the first endgame simulation systems. It was designed to calculate warhead fragmentation trajectories as they intersect an intricate target

[1] An endgame is defined as the final milliseconds of a missile/target encounter.

model, keeping track of which target components each fragment intersects. Part of FASTGEN's development involved the creation of an intricate target model format for accurately depicting aircraft. The FASTGEN target model involves multiple base primitives including spheres, cylinders, donuts, boxes, wedges, rods, and triangles. Several aircraft (both friendly and hostile) have been modeled using the FASTGEN format. As a result, most of the endgame systems available today use FASTGEN target models directly, or models converted from FASTGEN to a format containing all triangles.

1.2.2 PLOT5

PLOT5 displays FASTGEN files and was written primarily as a tool for debugging target models. It is capable of displaying the set-up encounter of a simulation, the orientation of the missile and target. PLOT5 uses two dimensional cross sections of the target to display component damage [Moritz96:2-3][Cramer85:B-9].

1.2.3 IVAVIEW

IVAVIEW is an X-Windows application capable of running on an SGI. IVAVIEW displays FASTGEN files, showing the entire target and missile models in three dimensions. The wireframe models can be views from various directions, and target component visibility options are provided to the user [Moritz96:2-3,4][SURVICE92].

1.2.4 ENCOUNT

Engineers need a means of visualizing an endgame scenario; specifically, they must know how a missile and target are oriented and see their relative flight paths. ENCOUNT is a FORTRAN program which takes endgame parameters from an ENCOUNT file, along with FASTGEN target and missile models, and produces a third FASTGEN file containing the target and missile oriented according to the endgame parameters along with lines representing relative trajectories. This file is then examined with IVAVIEW.

1.2.5 OPEC

The Ordnance Package Evaluation Code (OPEC) is the most recently developed endgame simulation system. OPEC was written for the IBM PC; its calculations have a number of improvements over previous simulation systems (see [Moritz96:2-7]). Although OPEC provides a more accurate simulation of a missile/target engagement, it's results suffer from a poor graphical display, because its PC platform does not have the computer graphics power to display the abundant model data.

1.2.6 AMES

Engineers like OPEC because it provides a more realistic calculation of target probability of kill (*pk*). However, they need better images of the encounters. The AFIT Missile Endgame Simulation program was implemented to display OPEC simulation results. With AMES, the engineers are able to view the orientation and position of the missile and target for the given encounter. It also displays OPEC's warhead burst point, target component damage, and warhead fragmentation fly-out velocities. Furthermore, AMES displays encounters based on endgame parameters found in an ENCOUNT file and attempts to animate the motion of the missile and target. Finally, AMES displays multiple fixed fuze-cone sensor patterns.

AMES was a good initial attempt at visualizing an endgame scenario; however, several major improvements can be made on existing features, and new capabilities must be added to make it useful for all endgame analysis activities.

Improvements on existing features include complete re-engineering of animation to correct the missile and target flight path, give the user more control over the animation, provide feedback as to current simulation time and target and missile positions, improve rendering performance, and provide more fidelity. AMES is limited in its ability to provide complete fidelity and control due to its underlying implementation.. Second, although AMES properly color codes internal target components based upon *pk* damage values calculated by OPEC, these

cannot be seen due to conclusion from the aircraft skin, plus the target damage color code scheme should be modified to improve visualization [Tuft90:82,91]. Third, a different method for visualizing warhead fragmentation needs to be applied (including animation) [Cunard97]. Fourth, calculations for orienting the target and missile based upon parameters found in OPEC and ENCOUNT files need to be corrected. Fifth, a better technique should be implemented for visualization of inter-object spatial relationships, rather than using multiple one-point perspective projections [Wagner92] [Herndon92]. Lastly, AMES user-interface needs to be modified to increase usability.

At the completion of Moritz's research, some of the original requirements were left incomplete and are listed here [Moritz96:6-3]:

1. Visualize warhead fragmentation pattern skewing.
2. Provide multiple coordinate center feedback options.
3. Provide velocity vector orientation feedback relative to the model.
4. Provide speed control over the simulation.
5. Provide feedback on time passed during the simulation.

In addition, Moritz provides a list of recommendations for future work [Moritz96:6-3]:

1. Modifiable levels of detail.
2. Saving a simulation to disk, allowing the user to save her work and return to it later.
3. Capability to change simulation speed.

In addition, early system prototypes and interviews with the engineers have revealed the needs for the following requirements:

1. Animation of the entire endgame scenario, including fuze-cone sensor target detection, warhead fragmentation fly-out and fragmentation-target intersection based upon SHAZAM[2] simulation results.

[2] SHAZAM is missile endgame simulation tool for calculating *pk* based upon fragmentation hits on a target.

2. Ability to create, edit, visualize and save the information within an ENCOUNT file.
3. Target component-group visibility and skin transparency control.
4. More control over the current view including position and orientation feedback relative to the target, and the ability to save and restore multiple views.
5. Ability to animate full missile fly-out simulations as produced by VisSim[3].

1.3 Thesis Statement

It is clear from the preceding summary of endgame visualization systems that there remains ample opportunity for improvement and a definite need for new for new capabilities. This research explores these areas by discovering and implementing techniques for:

1. Visualizing the fragmentation fly-out skewing phenomenon
2. Animating an endgame encounter from target detection to warhead detonation and the target/fragmentation intersection
3. Animating full fly-out simulations
4. Visualizing three dimensional inter-object spatial relationships on a two-dimensional display.
5. Providing the engineer complete control over the simulation time while being able to view the animation from any angle.

1.4 Scope

1.4.1 Research

This research is limited to providing engineers graphical capabilities for visualizing and

[3] VisSim is a modeling and simulation system for simulating the full path of the missile from launch to target.

animating missile/target encounters. This research will not involve modeling missile flight behavior in full fly-out simulations, or *pk* calculations for endgame scenarios.

1.4.2 Equipment

The primary development environment is a Silicon Graphics (SGI) workstation running UNIX, X-Windows, and Motif. Various graphical libraries are also available and are discussed further in Chapter III. Additional prototypes for animating VisSim results will be developed on both the SGI and the PC. Developed prototypes written will be portable to both platforms (see Section 5.7 for more details).

1.4.3 Methods for Enhanced Software Quality

Because this research includes software development, methodologies for enhancing software quality were incorporated. The intent was to create stable, maintainable, extendible and efficient software. Chapter III discusses these methodologies and how they are used to meet these intentions.

1.5 Thesis Presentation

The remainder of this thesis is divided into five chapters. Chapter II presents background information relevant to this thesis. In particular, it describes the three key components in an air-intercept missile: guidance, fuze, and warhead, and then briefly discusses the relevant missile/target simulation systems which preceded my research. Chapter III discusses the selection of programming language and graphical library, and introduces the software development methodologies I followed during this research. Chapter IV outlines design decisions regarding three-dimensional visualization, user interface, software library selection, and software architecture. In Chapter V, I first present the AMVS' implementation from the user's perspective, presenting its features and capabilities while tying these to the engineer's needs

outlined in Chapter II. The second part of Chapter V highlights some of AMVS' underlying implementation. Finally, Chapter VI summarizes this research, identifying its specific contributions to the field, and proposes recommendations for future work.

2. Background

2.1 Introduction

This chapter begins with a brief description of air intercept missiles which includes a discussion of the guidance, warhead, and fuze components followed by explanation of a phenomenon known as skewed fragmentation fly-out and its effects on air intercept missile performance. Next, I discuss a brief history of computer based simulation of missile/target encounters. This discussion includes a description of Lt. Joseph Moritz' implementation of the AFIT Missile Endgame Simulation (AMES). Finally, I discuss the engineer's need for a briefing and training tool to explain some of the concepts outlined in this chapter.

2.2 Air Intercept Missiles

2.2.1 Missile Guidance Component

The guidance component's primary objective is to navigate the missile to intercept a target directly. A miss occurs when the guidance is unable to turn the missile tightly enough to intercept the target on final approach [Mack87:2,14]. When a direct intercept cannot be achieved, the guidance component's secondary objective is to minimize the miss distance between the missile and target in order to increase the effectiveness of the missile's warhead against the target.

2.2.2 Missile Warhead Component

The missile warhead projects fragmentation perpendicularly from the missile's longitudinal axis in an effort to kill a target during a miss. The warhead plane (centered at the

warhead, perpendicular to the longitudinal axis) defines the fragmentation flight path relative to the missile (see Figure 2-1). The actual fragmentation flight path is found by adding the lateral fly-out velocity vector with the missile's velocity vector. When timed correctly, fragmentation will intersect the target at the point where the target passes the warhead plane [Cunard97].

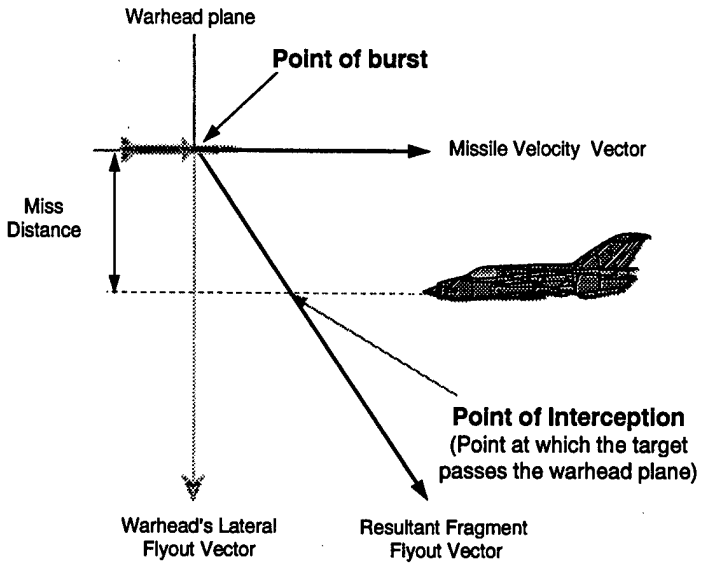


Figure 2-1: Warhead Fragmentation Fly-out

Missile warheads can be directional or isotropic. Directional warheads project fragmentation out one side of the missile, depending upon where the target is at the point of detonation [Cunard97]. Isotropic warheads send an expanding "ring" of fragmentation in all directions from the missile. Over time, this, expanding ring creates a conical pattern due to forward momentum of the warhead at the time of burst. This characteristic will be referred to as the fragmentation pattern cone throughout the rest of this document.

2.2.3 Missile Fuze Component

The fuze component determines when to detonate the warhead. The fuze component

makes this decision based upon information known about the target's proximity, relative velocity, and vulnerable point (aimpoint) along with the predicted fragmentation pattern cone [Kobaz74:9]. The fuze component consists of a proximity sensor to detect proximity, relative velocity and aimpoint of the target, and the fuzing algorithm (or fuzing logic) to calculate the time of burst.

This thesis focuses on fixed-cone proximity sensors. The fixed-cone fuze uses a high gain conical beam antenna with 360 degree azimuth coverage to detect a target. The antenna is fixed at a specified elevation angle. The antenna elevation angle and sensor range define the sensor cone. A target passing through this cone is detected by the fuze, which in turn triggers the fuzing logic. A target approaching at an encounter angle greater than the antenna elevation angle will not be detected by the fuze. As a result, no detonation will take place (see Figure 2-2). Widening the elevation angle can increase the range of detection; however, as seen in the next paragraph, this can adversely effect the ability of the fuzing logic to calculate the warhead detonation time.

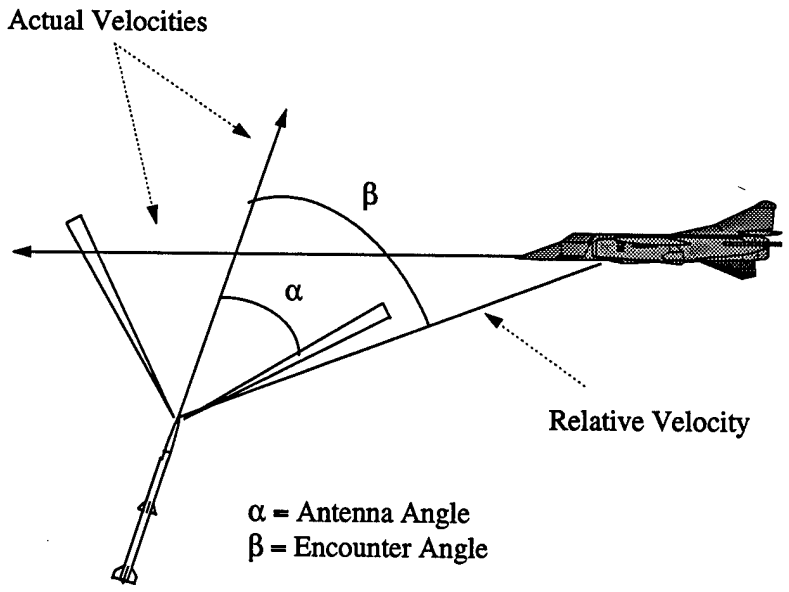


Figure 2-2: Antenna/Encounter Angle

The fuzing logic uses the fragmentation pattern cone and detected target distance, relative velocity, and aim-point to calculate the time-to-burst. The time-to-burst (also called the timing delay) is the delay between target detection and warhead detonation (see Figure 2-3).

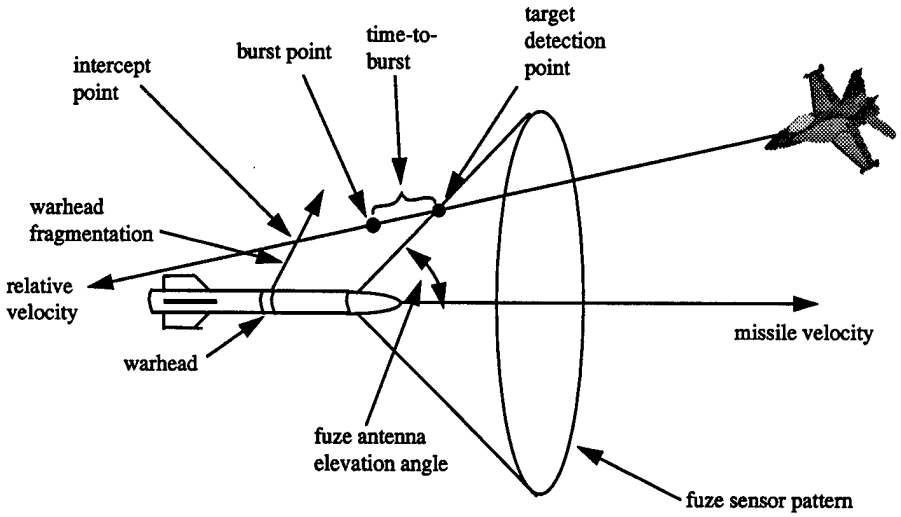


Figure 2-3: Time-to-burst

The objective of the fuzing logic is to calculate the optimal time-to-burst to achieve the highest coverage of fragmentation on the target at its most vulnerable point. If the fuzing logic calculates a *negative* time-to-burst (occurring in high-speed head-on encounters), the warhead will not detonate. A negative time-to-burst is calculated when the target intercepts the fuze cone *after* the optimal point of burst (see Figure 2-4). This can be alleviated by lowering the antenna elevation angle.

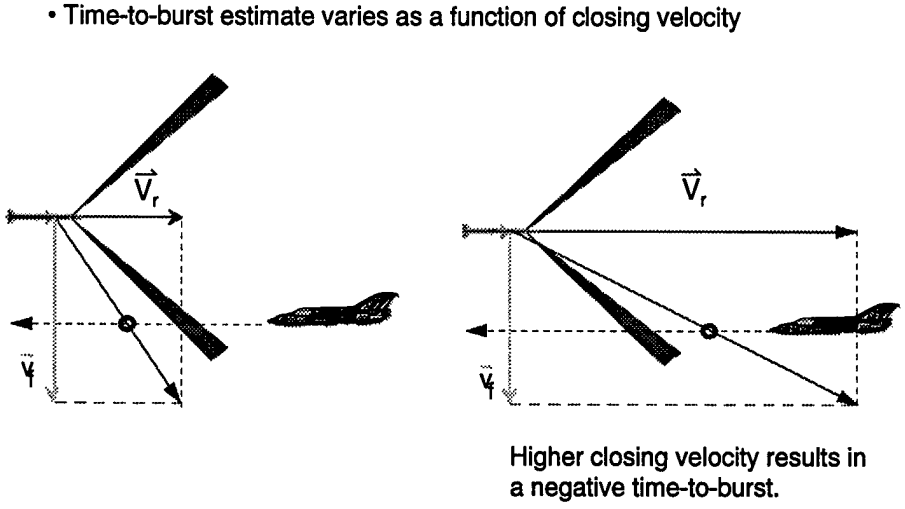


Figure 2-4: Negative time-to-burst

Obvious trade-offs exist when selecting an antenna elevation angle for fixed-cone fuze proximity sensors. Selecting a high elevation angle will allow the fuze cone to detect targets approaching from larger encounter angles, while lower elevation angles allow the fuze cone to detect high speed head-on targets early enough to calculate a positive time-to-burst. Engineers must find a valid setting for fixed-cone antenna, or search for alternative proximity sensing techniques. A technique under consideration is the use of multiple fixed-cone fuzes mounted on a single air-intercept missile.

2.2.4 Skewed Fragmentation Pattern Cone

A phenomenon known fragmentation pattern cone skewing affects the performance of the fuze sensor and warhead. Skewing of the fragmentation pattern cone results when the missile is performing a hard bank at the time of warhead detonation. A missile with a yaw or pitch value at

detonation will not have its longitudinal axis in line with its flight path. Combining warhead fragmentation fly-out velocity (lateral to the missile) with the missile's actual flight path results in a cone that is not centered on the missile's longitudinal axis (see Figure 2-5). The cone is rather "skewed" towards the flight path.

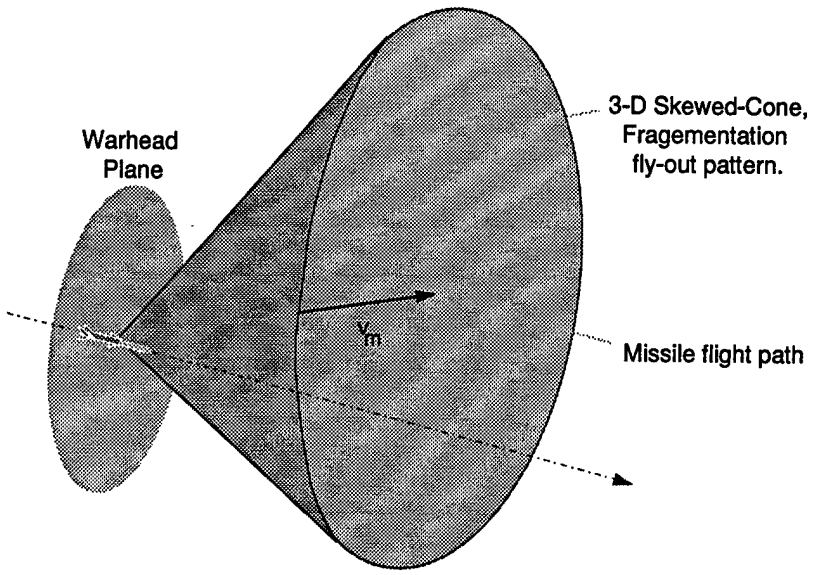


Figure 2-5: Fragmentation fly-out's skewed cone

The fuzing logic does not take into account the skewing of the fragmentation pattern cone when calculating the time of burst. As a result, an error in calculation occurs when the missile has a yaw or pitch at the point of detonation.

2.3 Computer-Based Simulation

The following sections describe a subset of current computer based simulation systems as applicable to my research. For a more in-depth discussion of simulation systems, see [Moritz96].

2.3.1 FASTGEN

FASTGEN is a simulation system for calculating the effects of warhead fragmentation on a target model. FASTGEN derives its name from SHOTGEN, a computerized mathematical model of fragmentation analysis using a "shotline" method to predict damage to a target[Cramer85:1-1]. FASTGEN improves upon SHOTGEN by decreasing processing time. A target model file format has been developed specifically for FASTGEN and is used by other simulation systems as well. The FASTGEN target model uses multiple primitives to define intricate aircraft components. These primitives include spheres, cylinders, donuts, boxes, wedges, rods, and triangles [Sherly93:23]. Aircraft components in the FASTGEN format are numbered according to their structural or functional groupings as seen in Table 2-1.

Table 2-1: Target Component Groups

Component Number	Description
0000-0999	Skin and other external covers
1000-1999	Power Plant and Accessories
2000-2999	Crew
3000-3999	Flight Control System and Hydraulics
4000-4999	Fuel System
5000-5999	Ammunition (include Bombs) and Missiles
6000-6999	Armament
7000-7999	Stringer, Ribs, and Structural Members Airframe
8000-8999	Fire Directional System and Avionics
9000-9999	Miscellaneous

2.3.2 SHAZAM

SHAZAM, created after FASTGEN, is a statistically improved endgame simulation system [Coffield86]. SHAZAM runs its simulations against FASTGEN target files that have

been converted entirely to triangles. Results produced by SHAZAM are viewed in a program called FLYOUT. FLYOUT shows the scenario set-up and displays warhead blast fragmentation patterns as an expanding ring formation from the burst point [Coffield86:1,10-11]. Simple wireframes are used to represent the target and missile. As a result, the graphical display is insufficient in revealing how the fragmentation intersects the target [Moritz96:2-5,6].

2.3.3 IVAVIEW

IVAVIEW is an X-Windows application built for an SGI workstation. IVAVIEW displays FASTGEN target models three-dimensionally. Since Moritz' writing, a few modifications have been made to IVAVIEW. These include transparency settings on selected target components, an "outline" display of the external skin, and ability to edit the target model directly within IVAVIEW.

2.3.4 ENCOUNT

Each endgame scenario is unique in its placement and orientation of the missile, target and relative flight paths and is defined by a series of vectors, velocities, and orientations values. ENCOUNT was written to assist engineers in creating and visualizing an endgame scenario. ENCOUNT takes a file containing endgame parameters and creates a single FASTGEN model file containing both a missile and target oriented according to these parameters. Visible relative velocity vectors are included and are displayed as lines extending through the missile and target. This file is then viewed using IVAVIEW. Examining an endgame scenario requires the engineer to edit an ENCOUNT file, run this file through ENCOUNT, and view the output using IVAVIEW. After viewing the FASTGEN file, the engineer must exit IVAVIEW before he can edit the ENCOUNT file again. This process can be made more efficient by creating a single application for editing, viewing, and saving an endgame file.

2.3.5 OPEC

The Ordnance Package Evaluation Code (OPEC), developed for the IBM PC, is the most recently developed endgame simulation system [PMC]. OPEC also uses FASTGEN target models which have been converted to another format, GEORGE. The GEORGE format contains all triangles. OPEC has a number of improvements over SHAZAM for simulating a missile/target endgame encounter and calculating *pk* [Moritz96:1-6]. Although OPEC provides a more statistically accurate *pk* calculation, its results are poorly displayed. Its three dimensional display shows wireframe target and missile models and target damage displayed as red asterisks. Since it is developed for the PC, it suffers from poor rendering performance and limited resolution.

The results produced by OPEC are numerous and complicated. A graphical tool for visualizing these results would make OPEC more valuable to the engineers. Visualizing this data provides the engineer a more efficient and complete means of assimilating and comprehending simulation results.

2.3.6 VisSim

VisSim is a PC based application for simulating the full flight of an air intercept missile from launch to target (a fly-out). Upon completion of a simulation, VisSim outputs a file containing the full flight path of the missile and target. Currently, only a two dimensional view of these results are available to the engineers. The engineer needs to know the missile's full path as well as orientation at specific points along that path. He must also be able to comprehend timing issues between a moving target and missile. This simulation application can be made more informative by providing a three-dimensional visualization and animating the results. Animating the missile's flight path helps the engineers understand timing issues between the

missile and target as well as missile orientation throughout the flight path. Furthermore, a three-dimensional display of the flight path provides the engineer an efficient means of evaluating simulation results.

The sponsor has expressed interest in viewing this animation on a PC as well as an SGI. This research explores technologies for displaying fly-out results on both platforms. This exploration includes extending AMVS to display VisSim results along with the creation of two prototypes, one written in C++ using OpenGL for graphical rendering, and one written in Java using the Virtual Reality Modeling Language (VRML) for graphical rendering. These prototypes will run on both the SGI and PC platform. Chapter V presents the implementation of these prototypes and discusses the trade-offs between them.

2.3.7 AMES

Previous research by Moritz for the Wright Laboratory Armament Directorate (WL/MNMF) resulted in the AFIT Missile Endgame Simulation (AMES). AMES was written to investigate techniques for improving endgame graphical feedback by creating a three dimensional view of an endgame simulation. It has the ability to load and view endgame parameters found in OPEC and ENCOUNT files. AMES is also the first graphical tool to attempt animating the motion of the missile and target during an endgame. AMES also displays fixed fuze cone sensor coverage patterns. See [Moritz96] for more detail on AMES' capabilities.

AMES was a good initial step toward providing graphical capabilities for endgame visualization; however, there are several areas which need improvement and some necessary capabilities were altogether lacking. The identification, design, implementation, and evaluation of these areas constitute the bulk of my research and are covered in detail in Chapters IV and V. Nevertheless, I've included the following overview for easy reference. The following section

discuss additional major and minor requirements for missile visualization, as well as major and minor improvements on existing features.

2.4 Requirements for Missile Visualization

2.4.1 Major Additional Capabilities

2.4.1.1 Visualizing the Fragmentation Fly-out Skewed Cone

The fragmentation fly-out cone skewing phenomenon, as discussed above, effects the performance of air-intercept missiles. Engineers currently have no graphical tool for dynamically visualizing this phenomenon. A visualization tool would assist the engineers in better understanding fly-out cone skewing and thereby assist in discovering solutions to combat its effects. Also, such a tool would be valuable in briefing others on this problem. As engineers come up with solutions to this problem, they will also need to communicate the problem and their solution to decision makers [Cunard97].

2.4.1.2 Full Animation of an Endgame Scenario

Currently there exists no three-dimensional graphics tools for animating the full endgame scenario. Animation should include visualizing fuze-cone target detection, warhead fragmentation fly-out, and fragmentation-target impacts as produced by SHAZAM. Such an animation tool not only helps the engineers understand timing issues and missile/target interaction during an endgame, it also provides an improved method for communicating endgame concepts to people not familiar with them.

2.4.1.3 Visualization and Animation of Full Fly-out Simulation Results

As stated before, engineers currently have no three-dimensional tool for visualizing and animating VisSim fly-out simulation results. The lack of such a tool makes assimilation of these results difficult. Engineers need to be able to understand the missile's flight path in relation to the target, as well as its orientation throughout the flight.

2.4.1.4 Fixed Fuze-cone Attribute Experimentation

AMES was implemented to allow the engineer to visualize fixed fuze cone sensor pattern coverage based upon antenna azimuth and range attributes specified in a FUZE file. Engineers need an environment for not only visualizing this sensor coverage but also for experimenting with fuze cone antenna azimuth and range settings all within a single graphical application.

2.4.2 Minor Additional Capabilities

2.4.2.1 Endgame Scenario Creation

As stated in Section 2.3.4, engineers need an efficient means of creating an endgame scenario. Although AMES was implemented to load an ENCOUNT file, no effort was made to allow for saving or creating new ones. As a result, engineers are still left editing these files by means of a text editor. The task of creating an endgame scenario would be more efficient by creating a single environment with a graphical display of the results complete with GUI interface for parameter entry. These modifications should then be written out to a user specified ENCOUNT file.

2.4.2.2 Modifiable Levels of Detail

To improve performance while still providing high fidelity models, Moritz suggests the

implementation of user modifiable levels of detail (LOD) with future versions of AMES [Mortiz96:6-3]. This will allow the user to choose between efficiency and accuracy. During some tasks, such as examining target/fuze-cone interaction, lower fidelity in target and missile models may suffice. During briefings, improved rendering may be more desirable. As a result, target and missile level models using various levels of detail are required.

2.4.2.3 Saving a Simulation

AMES could provide a more efficient development environment by allowing the engineer to save her work and return to it later. At a minimum, saving the simulation setup should include such information as the currently selected missile and target models as well as the current ENCOUNT or OPEC file.

2.4.2.4 Visualized Missile and Target Velocity Vectors

AMES displays relative velocities; however it does not provide visual feedback as to the missile and target's actual flight path. Displaying these flight paths assists understanding of missile/target interaction. Displaying the missile's actual flight path is especially necessary when examining fragmentation fly-out skewing. Recall that the fragmentation fly-out cone is centered around the missile's flight path and not around the missile's longitudinal axis. Visualizing the actual flight path is necessary to completely understand the skewing phenomenon.

2.4.2.5 Target component group visibility.

Each target consists of several hundred components. To reduce scene complexity, IVAVIEW provides the user the ability to set component group visibility. This is a useful feature and should be included in any prototype.

2.4.2.6 View Control

To further understand the scenario, engineers should be given current view position and orientation feedback relative to the target. Also, the engineer should be able to save and restore key viewpoints.

2.4.2.7 Rendering Scene Image Capturing

Tools supporting training and briefing should be able to capture their rendering scenes and save them to user-specified image files. These image files can then be incorporated into briefing slides or training manuals.

2.4.3 Major Enhancements to Existing Features

2.4.3.1 Corrected Endgame Scenario Calculations

AMES contains a few calculation errors when displaying an endgame based upon parameters found in ENCOUNT and OPEC files. Incorrect calculations include the miss-placed visible target coordinate system, arbitrary point on the missile, and visual display of velocity vectors as well as the miss-calculated missile center of rotation and direction of flight for missile and target during animation. These need to be corrected.

2.4.3.2 Improved Rendering Performance

AMES' rendering performance falls far below the suggested minimum frame rate of 10 frames per second for smooth animation [Foley92:180]. Not only does this degrade the animation, it limits AMES' ability to show object positions with a sufficient time resolution due to its underlying implementation. Object positions at each frame are based upon a clock value, as a result the poor frame rate leads to loss of fidelity. Additionally, as new features are added in order to animate the full endgame scenario, this problem will be compounded. Therefore,

rendering performance must increase.

2.4.3.3 Improved Animation Control User-Interface

AMES user-interface limits the user. It needs to provide information feedback during the animation and more control over the simulation time. The engineers require feedback including the current simulation time along with current target and missile positions displayed in target, missile, or world coordinate systems. Positions should be displayed in inches or meters. The engineer also requires more control over the animation including animation speed control, target and missile motion relative to the target, missile, or world, and ability to easily move the animation to a specific point in time.

2.4.3.4 Ability to View OPEC Target Damage

During a simulation, OPEC calculates target component damage on internal components. Although AMES reads these results and colors the internal target component damage accordingly, these results are occluded by the aircraft skin and cannot be viewed easily.

2.4.4 Minor Enhancements to Existing Features

2.4.4.1 Inter-object Visualization Techniques

In order to provide insight into inter-object spatial relationships, AMES provides three additional windows showing top, side and front views of the encounter to complement the main rendering window. My sponsors have shared disinterest in this approach. Moreover, better technique exists to highlight these relationships. Also, although using transparent cones to visualize fixed fuze cone sensor coverage patterns effectively shows target/cone relationships when the target is penetrating the cone, it is ineffective at times in revealing this relationship

when the target is positioned before or behind the cone. A separate inter-object visualization technique is required to help users mentally visualize target/cone relationships.

2.4.4.2 Alternative Warhead Fragmentation Visualization

AMES displays warhead fragmentation trajectories as lines emanating from the warhead origin point [Moritz96:4-17]. The sponsors have expressed disinterest in this approach. The preferred visualization technique involves an expanding ring or torus representing a mass of fragmentation emanating from the missile at a specific point in time. The expanding ring or torus more accurately models the actual fragmentation fly-out. This expanding ring should also be animated.

2.4.4.3 Improved Target Damage Coloring Scheme

Component damage produced by OPEC is represented by a scalar value ranging from 0.0 to 1.0, with 1.0 representing complete damage. AMES visualizes this target component damage using a coloring index scheme in 1/10 increments with one color arbitrarily assigned to each of the ten increments. No color index table is provided. Additionally, information is lost when converting the scalar value to an integer ranging from one to ten. An improved coloring technique is required which does not loose information and more logically conveys damage amounts.

2.4.4.4 Need for a briefing and training tool

An additional reason for AMES development was to create a briefing and training tool for people unfamiliar with endgame simulation concepts. AMVS' development should be done keeping this requirment in mind. AMVS will be used to inform upper levels of management on problem areas such as skewed fragmentation fly-out. AMVS will also be used to brief new

techniques in proximity sensing such as incorporating multiple fixed-fuzed sensors on a single missile. Animation of an endgame encounter provides an excellent means of illustrating the problems engineers face as well as fostering the understanding necessary to appreciate their solutions.

2.4.4.5 Improved Usability and Stability.

The sponsors found AMES unstable and difficult to use. Care should be taken on future development to ensure that applications delivered to the sponsor have increased usability and stability.

2.5 Conclusion

Engineers have a variety of systems for simulating missile/target encounters available to them. A great deal of computation is performed by these systems to predict the performance of air intercept missiles. These systems can be made more valuable to the engineers by enhancing the graphical display of simulation results. AMES was a first attempt at meeting this need; however, significantly more needs to be done. The rest of this thesis outlines what I have done. The next chapter introduces methodologies I use during the development process followed by a presentation of my design in Chapter IV. Chapter V shows AMES' implementation; finally Chapter VI presents contributions to the field of visualizing and animating missile/target encounters and makes recommendations for future work.

3. Methodology

3.1 Introduction

Developing high-performance, highly usable, robust three-dimensional graphical applications takes a variety of skills. One must keep abreast of current software technologies, understand graphic rendering and three-dimensional computation, be familiar with users tasks and needs, have a good understanding of human-computer interaction (HCI) and data visualization, and be skillful in programming and software engineering. This chapter summarizes methods I use in each of these areas and how they apply to the development of applications for visualization and animation of missile/target encounters.

3.2 Research Process

My research began with a literature search in the field of interest: missile/target simulation. The result of this study is shown in Chapter 2. In this and subsequent chapters, I turn my focus to the development of a graphical application for missile/target simulation using principles in HCI, data visualization, and software engineering as a guide. Finally, this effort explores some opportunities provided by recent advances in computer graphics and software development.

3.3 Human Computer Interaction and Data Visualization

My goal in developing the AFIT Missile Visualization System (AMVS) is to produce a three-dimensional application with a high degree of usability, and effectiveness for data visualization of a missile/target encounter. My philosophy of design is to create an environment

that provides an engineer with the temporal and spatial freedom to view, analyze and evaluate the performance of air-intercept missiles against hostile targets in a simple to use application. To do so, I apply current trends in HCI and data visualization.

The Air Force Institute of Technology's graduate level graphics sequence includes studies in usability and HCI. Information obtained from course notes are applied in the design and development of AMVS' user interface. In particular, I apply principles and methods from Jakob Nielsen's book Usability Engineering [Nielson93]. The following methods are used:

- User task analysis - getting to know the user [Nielson93:75].
- Vertical prototyping - full implementation of a few chosen features (vs. Horizontal Prototyping, reduced functionality in a more complete system) [Nielson93:18].
- Participatory design - continued feedback from the user [Nielson93:88].
- Interface evaluation and user testing - feedback from test users [Nielson93:165] Test users will include sponsors, fellow students and faculty.

Applying methods of good data visualization techniques can be difficult. For this area, I rely on publications on the subject. One author in particular, Edward Tufte, has produced an excellent series on the subject [Tufte89][Tufte90][Tufte97].

3.4 Evaluation of Software Platforms

Several software platforms are currently available within AFIT's computer graphics lab. For the scope of the projects at hand, I must choose between software development languages, and graphical rendering and user interface libraries. Evaluations of each are made primarily according to performance, capability, and ease of use. Results of this evaluation can be seen in Section 4.4.

3.4.1 Software Development languages

AMVS was developed in C++, no evaluation of software development languages has been performed for its selection. C++ provides the ability to apply object oriented programming techniques [Stroustrup91] as well as providing ease of use and increased performance when working with existing graphical and windowing libraries. Open Inventor is written in C++, while Performer, X-Windows and Motif are written in C [OIAG94][IRIS95] [Nye92] [Heller92].

Introducing PC-based animation of VisSim results into my research, as mentioned in Section 2.4.6, provides me the opportunity to explore new technologies in graphical application development. In my evaluation, I examine the effectiveness of C++ against Java in this environment. Prototypes are written in both languages and an evaluation is made regarding performance, capabilities, ease of use, and portability. Results of this evaluation are shown in Section 5.7.

3.4.2 Graphical libraries and languages

AMVS graphical rendering was done using either Performer or Open Inventor. AMES, currently written in Open Inventor, suffers from poor performance. Performer is known for its ability to maintain high rendering rates, and there for is considered as an alternative. The possibility exists, however, that AMES' poor performance is in its implementation, and not its choice of graphical libraries. To make an unbiased performance comparison, I evaluate simple missile/target animation prototypes using both libraries. While these prototypes are compared primarily by rendering frame rates, the decision includes issues such as the Application Programming Interface (API), and portability.

Research in PC-based animation provides me the opportunity to explore new technologies in three-dimensional computer graphics. In particular, I examine the capability of OpenGL, VRML (Virtual Reality Modeling Language), and Java3D [Woo97][Ames97] to

provide visualization and animation of VisSim results. Although OpenGL is standardized across many platforms, thereby increasing portability, other forms of three-dimensional computer graphic display and interaction are possible. Such options include VRML and Java3D. VRML and Java3D both provide platform independent three dimensional rendering, eliminating portability issues all together. Prototypes are written in all three software platforms with the evaluation results presented in Section 5.7.

3.4.3 User Interface development libraries and tools

AMES user interface was implemented using the IRIS Viewkit as well as Motif. Lt. Joseph Moritz [Moritz96] took advantage of SGI's RapidApp code generation tool for automatic generation of Viewkit and Motif code. This gave Lt. Moritz the advantage of rapid GUI development, thereby avoiding the learning curve associated with X-Windows/Motif programming [Moritz96:4-1]. Tools such as RapidApp, BuilderXcessory and UIMX provided a quick means of creating complicated Motif dialogs, however, the generated code can sometimes be difficult to integrate into a project. IRIS Viewkit builds upon the set of Motif widgets and encapsulates their use into C++ classes. I examine both RapidApp and IRIS Viewkit for their usefulness. Capability and ease of use are my primary consideration for user interface library and tool selection. Section 4.4.2 shows the results of this evaluation.

3.5 Software Development

An initial look at the code for AMES and interviews with its author early in my research gave me the impression that AMES' state of maintainability, extendibility, stability and performance was in question. Before beginning my development I evaluated the existing code to determine its validity as a baseline for AMVS, given the list of requirements outlined in Chapter II and software quality goals presented in Section 1.4.3. Throughout development, I implemented proper software engineering methodologies. AMVS is a sufficient baseline for future AFIT

research and my goals of developing an application with a high degree of stability, maintainability, extendibility and performance, mentioned in Section 1.4.3, are achieved.

3.5.1 Evaluation of AMES

I evaluated AMES' code in the areas of maintainability, extendibility, stability and performance. Stability was evaluated by putting AMES to the test, repeatedly running it to the point of fault or failure. Stability, as well as maintainability, was also evaluated with the use of automated run-time debuggers, personal interviews with its author, and examination of the code. Extendibility was evaluated through interviews with Lt. Moritz, analysis of his design, and examination of the existing code. Performance was evaluated by comparing AMES rendering rates with that of prototypes written in Open Inventor.

In addition, I made an estimation of the cost of reproducing features found in AMES. AMES currently consists of 22k lines of code in 81 source files; therefore, choosing to not use this existing code is a significant one, considering the limited research time available to me. Chapter IV shows the evaluation results and the final decision.

3.5.2 Application of Software Engineering methodologies

To achieve my goals mentioned above, I incorporated software development processes, principles of good object oriented design, use of coding standards and automated tools, and documentation into the development of AMVS.

3.5.2.1 Software Development process

The nature of the problem lends itself nicely to such software development processes as the Spiral Model, Rapid Application Development (RAD), and Rapid Prototyping [Gottisdiener95] [O'Brien96]. The compressed schedule necessitated by AFIT's thesis program together with our need to re-evaluate user requirements during development well match to

Nielson's vertical prototyping method of user interface design. For these reasons, I have chosen rapid prototyping for my software development process.

3.5.2.2 Object Oriented Design

To achieve stable, maintainable, and most of all, extensible code, I implement Object Oriented Design (OOD) and Object Oriented Programming (OOP) principles into AMVS' development [Rumbaugh91][Booch91][Sessions92] [Coad93] [Pohl97]. My design and implementation invokes principles of inheritance, polymorphism, and encapsulation. Encapsulation is used to ensure AMVS' stability and maintainability, while inheritance and polymorphism are used to enhance AMVS' extendability.

3.5.2.3 Coding Standard

Scott Meyers' book *Effective C++* lists 50 specifics ways to improve programs and design [Meyers92]. These methods are particular to the C++ programming language and range from specifics such as proper use of constructors/destructors and overloaded operators to class implementation and design issues. I have opted to use these principles as a coding standard in my development.

3.5.2.4 Automated Tools

In addition to SGI's debugger, *cvd,* I use a powerful runtime debugging tool called Insure++ created by ParaSoft. Insure++ is a very effective tool for ensuring the stability and robustness of a program [ParaSoft96]. Code analyzed with Insure++ is checked for errors such as:

- Memory corruption from attempts to access memory outside the valid areas defined by global, local, shared and dynamically allocated objects
- Operations on illegal, or unrelated pointers

- Reading uninitialized memory
- Memory leaks
- Errors allocating and freeing dynamic memory
- Incompatible variable declarations

ParaSoft has also produced an application called CodeWizard that analyzes code for use of proper design and programming practices. It uses Scott Meyers *Effective C++* as its standard. This tool compares a program written in C++ against a subset of Meyers' coding standards listed in his book. This tool is valuable in ensuring my compliance with this coding standard.

3.5.2.5 Documentation

To improve AMVS' maintainability, I document its implementation throughout my development. This includes file headers and in-line documentation. In addition to this, I keep a *Deviations* file in the *Continuity* directory where I document any deviation from coding standards, good design, and proper OOP principles. The need for such deviations center mostly around performance, implementation time or tailoring to the project. C++ has advantages in program design and development, however, OOP's added overhead at run-time can decrease performance in ways not experienced by pure C code [Ege92:45][Adams88:32]. As a result, I occasionally deviate from proper OOP principles at critical sections of implementation for the sake of performance. In the interest of time, I occasionally avoid implementation of particular coding standards as the scope of the problem allows.

4. Design

4.1 Introduction

This chapter discusses design decisions made during the development of the AFIT Missile Visualization System (AMVS). Design decisions were made regarding three-dimensional visualization, user interface, software library selection, usefulness of existing code, software development and rendering scene graph creation.

4.2 Three Dimensional Visualization

This section outlines rendering and visualization design choices in the areas of inter-object spatial perception, use of transparency, target/fuze cone interaction, warhead fly-out animation, and target component damage coloring.

4.2.1 Spatial Perception

User feedback revealed a need for increasing the AFIT Missile Endgame Simulator's (AMES) usefulness as a tool for perceiving the inter-object spatial relationship of a target and missile during an endgame. Mental modeling of a three-dimensional encounter displayed on a two dimensional screen is non-intuitive. Moritz elected to use additional top, front and side views [Moritz96:4-11] to present the required information; however, this takes up significant display space. Furthermore, it requires the user to mentally combine four separate images into a single coherent understanding of the encounter [Cooper84:106-114] [Herndon92:9.1]. Herndon suggests an alternative approach using shadow projections. Shadow projections of three-dimensional data in scientific visualization has been demonstrated to help users understand their

data [Grotch83]. User tests by Wanger et al. revealed shadow projections rank high in effectively visualizing inter-object spatial relationships [Wanger92:44-58]. This visualization technique has been incorporated into AMVS and can be seen in Figure 4-1. Additionally, shadows soften contrasts (through use of a gray background, darkened target shadows, and lighter grid lines) to enhance the visualization by reducing contrast intensity [Tufte97:21].

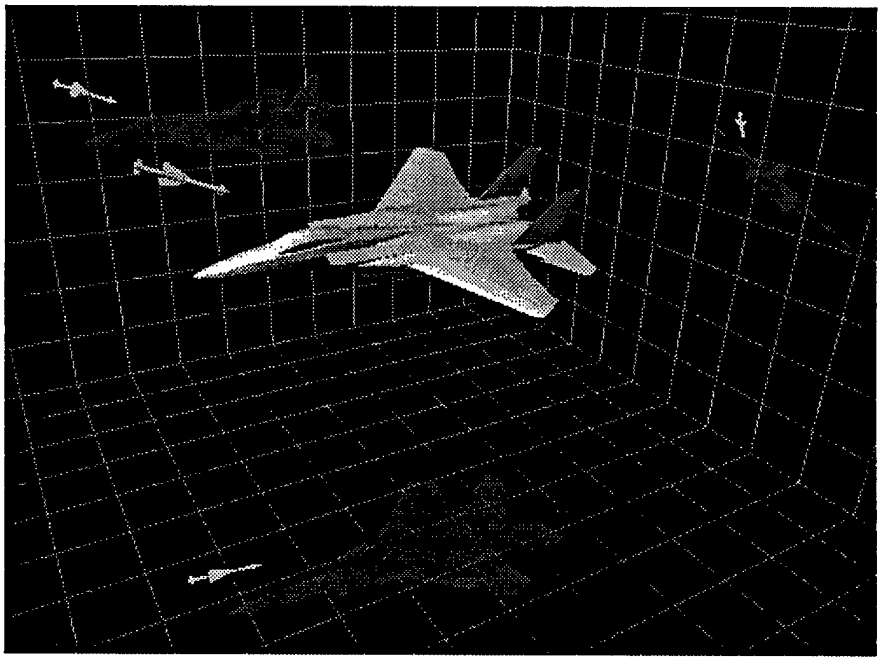


Figure 4-1: Top, Side and Front Shadow Projections

4.2.2 Transparency

Open Inventor's ability to render transparency enhances the visibility of internal target components as well as the usefulness of the fuze sensor and fragmentation fly-out cones. However, transparency can also confuse the image by allowing many layers to appear simultaneously. Therefore, I've given the user transparency intensity control on aircraft skin components (see Figure 4-2) and the sensor and fragmentation cones (see Figure 4-3).

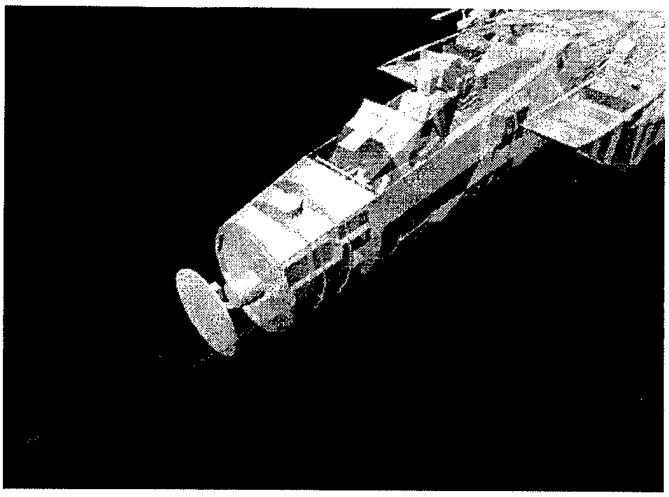


Figure 4-2: Aircraft Skin Transparency

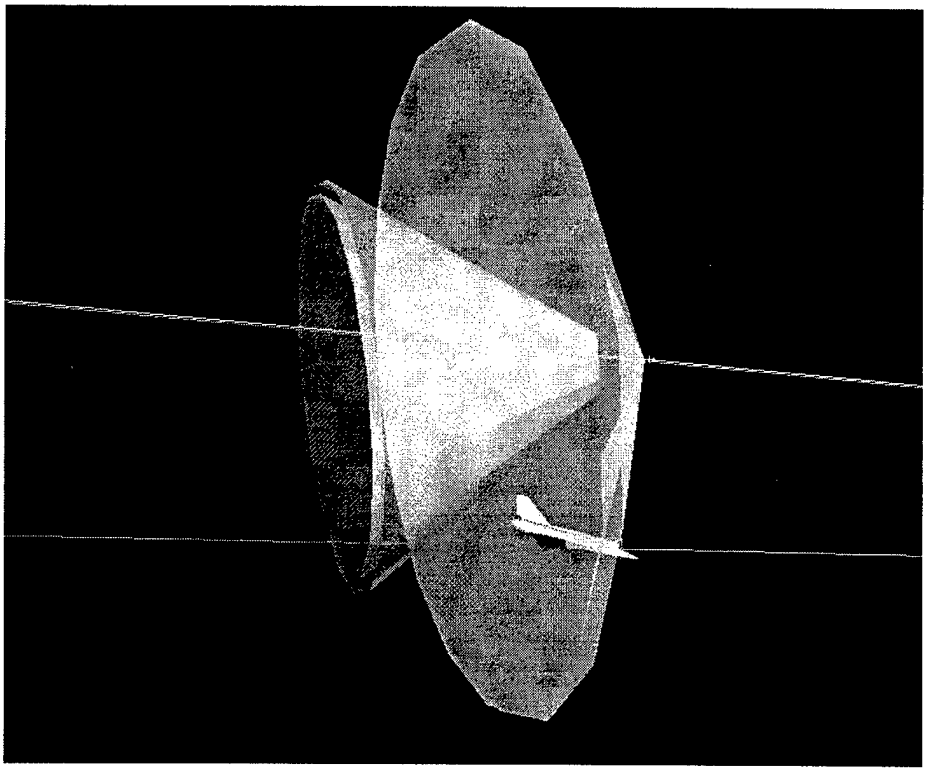


Figure 4-3: Cone Transparency

4.2.3 Target/Fuze Cone Interaction

A separate visualization technique is required to convey the spatial relationships of the target with fuze sensor and the fragmentation fly-out cone pattern. When the target does not

intersect the cone edge, it is difficult to comprehend the relationship between them. In particular, it is difficult to determine how far the target is from penetrating the cone. Recognizing that the two-dimensional drawings, as seen in Chapter 2, are effective in showing this relationship, I incorporate two-dimensional "cross-sections" of all pattern cones into AMVS. Figure 4-4 demonstrates how target/cone spatial awareness is enhanced by this technique.

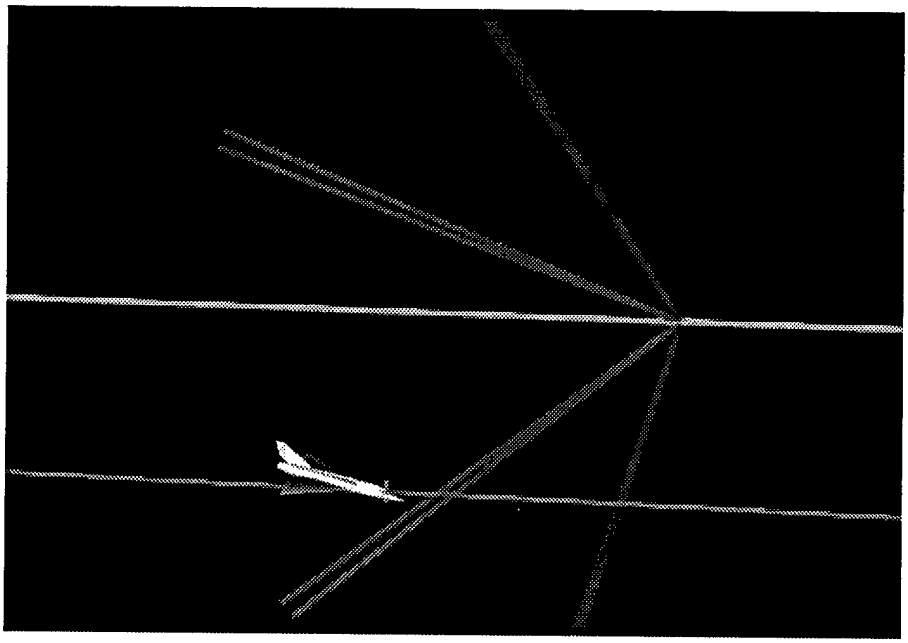


Figure 4-4: Cone Cross Sections

4.2.4 Fly-out Animation

Two techniques have been incorporated to visualize the fragmentation fly-out pattern of the warhead. These are shown in Figure 4-5, parts A and B. Part A shows a ring representing the center of mass. This is valuable in depicting accurate fragmentation/target intersection points and is beneficial in demonstrating warhead detonation timing principles. Part B shows a torus, representing a mass area of fragmentation dispersal over time. Animation of an expanding torus demonstrates the effects of fragmentation dispersal over time. Users may alternate between the two presentations.

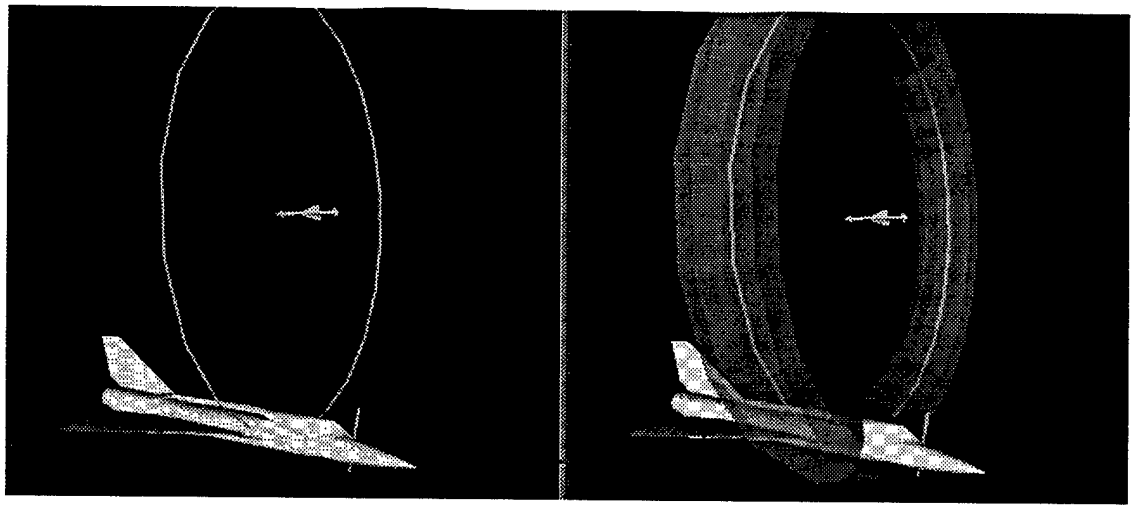


Figure 4-5A: Fly-out ring

Figure 4-5B: Fly-out Torus

4.2.5 Target Damage coloring

Component damage produced by OPEC is represented by a scalar value ranging from 0.0 to 1.0, with 1.0 representing complete damage. AMES visualized target component damage using a coloring index scheme in 1/10 increments with one color arbitrarily assigned to each of the 10 increments. No color index table was provided. Additionally, information is lost when the scalar value is converted to an integer ranging from one to ten. An improved technique is to implement a color gradient for visualizing target damage without data loss [Tufte90:91]. The color gradient applies the scalar component damage (*pk*) to the *rgb* (red, green blue) values of the geometric model as seen in equation [1]:

$$\begin{aligned} \text{red} &= pk \\ \text{green} &= 1.0 - pk \\ \text{blue} &= 0.0 \end{aligned} \qquad [1]$$

This produces a green color for components with the smallest damage, yellow for medium damage, and red for largest, or complete damage. Not only does this technique preserve the real nature of the *pk* value, but the green, yellow and red colors correspond to the familiar notation of

acceptable, warning, and danger respectively. A color index table is also available for display. The color index displays 20 spheres in .05 increments. Spheres are used to show how shading effects color on a three-dimensional object.

4.3 User Interface Design

4.3.1 Common Interface

Dialogs are designed for increased usability by creating a consistent user-interface. Consistency is important to an application [Nielson93:20] and is implemented easier through code re-use (explained in Section 4.6.5). Common features include similar action areas[4], file selection methods, and on-line help. Toggle buttons are used for all rendering object visibility selection while thumbwheels are used for transparency intensity settings. All text widgets allowing for entry of integer or float values have user input error checking.

4.3.2 View Control

AMVS includes control over the current viewpoint. In addition, a dialog is provided to display the current viewpoint in relation to the target. This includes position in target space as well as attack azimuth and elevation as calculated by ENCOUNT. Users can save and restore up to five views. The interface for saving and restoring the view is placed along the bottom of the main window, making it accessible at all times. All five views, along with the current viewpoint are written to disk when the simulation is saved.

4.3.3 User Interface Design for Animation Control

Although AMES' user interface for animation was easy to understand and use, it was over simplified and limiting to the user (see Figure 4-6). The user had no control over the

[4] Action areas are located at the bottom of a dialog and contain such buttons as Apply, Cancel, and Help.

animation's replay speed and no feedback as to the current simulation time. Because push buttons are used for starting, stopping and re-winding the animation, the user has to focus attention on the animation control dialog for each operation.

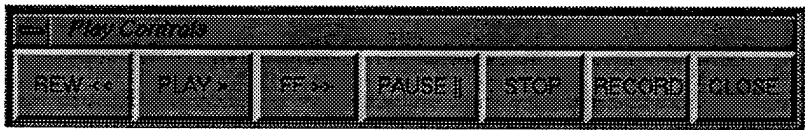


Figure 4-6: AMES Animation Control Dialog

To overcome these limitations when implementing AMVS, I have significantly modified the animation control user-interface by employing vertical prototyping and user testing. The animation dialog can be seen in Figure 4-7.

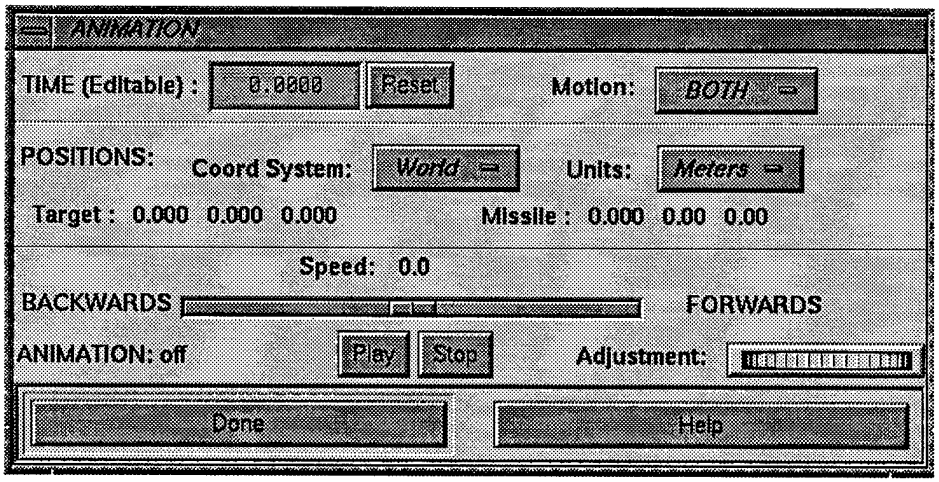


Figure 4-7: AMVS' Animation Control Dialog

This approach yielded the following improvements. The animation dialog provides current simulation time feedback along with greater control over the simulation time. It also allows the user three modes of modifying the time: 1) direct entry into the text widget used for displaying the current simulation time, 2) playing the animation (both forwards and backwards), including speed control, and 3) dial control of the simulation time using the time adjustment thumbwheel. Speed control is implemented as a percentage, from -100% to 100% of the maximum allowed. Using a slider bar for animation speed control and a thumbwheel for time adjustment increases

the animation controller usability, because once the user clicks down on either of these widgets, she can manipulate the simulation time while focusing her attention on the encounter.

As noted above, the thumbwheel allows adjustment of the simulation time. Dialing to the right moves the animation forward in time, while dialing to the left moves the animation backward. Varying the thumbwheel movement/simulation time change is needed depending upon the user's actions and intentions. When dialing the thumbwheel to move an object to a specific point in space, finer adjustments are required were as dialing to rewind the animation requires a larger animation movement. Using a factor of the focal distance is the preferred method of operation as revealed by user testing. Implementation of this technique is explained in Section 5.4.3.

The animation dialog also includes current target and missile position feedback in both meters and inches. This position can be displayed in World, Target, or Missile coordinate systems. Animation motion can likewise be done in a World, Target or Missile coordinate system by manipulating the "Motion" option button. A setting of "BOTH" causes both the missile and target to move relative to the world coordinate system. A setting of "MISSILE" causes the missile to move relative to the target, keeping the target stationary. A setting of "TARGET" leaves the missile stationary and moves the target relative to the missile.

4.3.4 Miscellaneous Usability Issues

A number of miscellaneous usability improvements have been made over AMES, a few of which are listed here:

- Menu items have been placed under appropriate headings.
- File selection dialogs have appropriate filters set to put the user in the correct directory, displaying the correct files for the given task.

- Files loaded automatically modify the rendering scene appropriately.

4.4 Library Selection

4.4.1 Graphical Rendering Library

AMES suffered from poor performance during animation. Using a different graphical rendering library was one possible solution to this problem. SGI's Performer was looked at as an alternative to Open Inventor. This section explains performance tests and results between AMES and a Performer prototype and talks about my final decision.

4.4.1.1 Performance Tests

Performance tests were done on a single processor SGI Indigo2, MIPS R4000, 250 MHz workstation. This machine is similar to the one used by our sponsors. Test data included full F106 and Falcon missile models containing 21124 and 284 polygons respectfully. Tests were also done with reduced models containing 746 and 140 polygons. I performed my tests under the same system load to reduce the effects of machine state inconstancy on the outcome. Test results varied with data size and machine state and are therefore represented as speed-up percentages.

I began by creating a Performer prototype, to compare against the existing version of AMES. This prototype yielded a 150% increase in frame rate over AMES. Knowing that some of the poor performance was due to AMES' implementation, I began looking into whether significant improvements could be made before abandoning Inventor altogether. My investigation determined that Moritz' use of Open Inventor-supplied animation engines inhibited performance. Another performance hit came from his means of providing missile and target position feedback during animation. This involved *SoSensor* nodes attached to the target and missile calling user defined callbacks for each position change. In the callback, the position is retrieved by implementing a *SoSearch* action on the entire rendering scene graph, thereby causing

transformation calculations to be made a second time for each animation frame. This method of performing animation with position feedback can be improved.

With 22 thousand lines of code to examine, I decided to build an Open Inventor prototype to do an unbiased comparison against Performer rather than continuing my search for areas of improvement within AMES. This prototype was almost identical to the Performer version. Tests between these two prototypes revealed almost identical results. When the minimum data set was used, both prototypes revealed frame rates up to 70 frames per second.

4.4.1.2 Conclusion/Decision

These tests revealed the primary cause of AMES' poor performance was due to its implementation and not attributable to the graphics libraries. For this application, Open Inventor fairs well against Performer in a single processor environment. Therefore, I decided to continue development with Open Inventor, because it has a better API for rapid development.

4.4.2 User Interface library and tool selection

Moritz used SGI's RapidApp for automatic generation of user interface classes. Although familiar with X-Windows and Motif, I wanted to see if this tool was useful to me. The code produced by RapidApp implements classes from the IRIS Viewkit library. This is a library of extended Motif widgets encapsulated into C++ classes. Although this is an efficient means of rapidly producing code for the user interface, I found the RapidApp classes to be too restrictive. Using RapidApp, I had no control over when the dialog was created, where they would appear on the screen, and what to do when the user killed the window, which is particularly important for the animation control dialog. If the user killed this dialog while animation is running, animation needs to be turned off. With all these in mind, I elected to write AMVS' user interface in Motif.

4.5 Evaluation of Existing Code

A significant effort went into the creation of AMES. The application, and its source code where available to me at the outset of my research. The extent to which I used this existing code needed to be decided upon early in the design phase. In order to make this significant decision, I evaluate AMES' usefulness to me in achieving my objectives given the list of requirements presented in Chapter II. To better make this decision, I evaluate AMES' code against my goals of stability, maintainability, extendibility and high rendering performance as outlined in Section 1.4.3. These attributes not only have effect on the final version of AMVS, but have significant impact on the development process.

4.5.1 Stability

The stability of AMES is questionable. Interviews with the author revealed that AMES was not extensively tested before delivery and that it had potential stability problems. Further tests revealed it was subject to faults and failures while use of automated debugging tools reveal many potential problems, some involving dynamic memory access.

4.5.2 Maintainability

Interviews with the author and code examination revealed that maintenance of AMES could be difficult. Furthermore, documentation is extremely lacking, and code is rather unstructured and difficult to follow making modifications and improvements difficult. In addition, global variables are extensively used and its software architecture yielded tight coupling. These attributes are known to lead to maintains problems [Holub95:47].

4.5.3 Extendibility

The existing version of AMES is not easily extended. Although written in C++, code examination revealed only one proper example of inheritance, and no examples of polymorphism

and encapsulation, other than code produced by RapidApp. Poor maintainability mentioned above also make extensions difficult.

4.5.4 Performance

Poor rendering performance appears to be an implementation problem that requires a significant amount of modification to improve. Inefficiencies in the rendering scene included extra light sources and superficial transformation nodes. Making necessary transformation calculations outside of the scene graph in order to reduce the number of transformation nodes could be performed, along with improving the lighting scheme; however, any change in the rendering scene could have unknown, most likely negative effects on AMES due to the tight cohesion between the rendering scene and code. AMES' code relies heavily upon the scene graph structure in any number of locations. Rather than keeping pointers to critical nodes in the scene tree, the tree is traversed each time node access is needed. Changing the structure of the scene graph will require an unknown number of modifications to the code, yet necessary to increase AMES' performance.

4.5.5 Final Decision

It was revealed that AMES poor performance was due to its underlying implementation and not its choice of graphical libraries (see Section 4.4.1.2). Improving performance requires significant modification to the scene graph and complete re-design of the animation process to avoid the use of the Inventor animation engine. In addition to hindering performance, using the Inventor animation engine hinders speed control and time feedback. Considering the tight coupling between scene graph and code, in conjunction with other maintenance problems listed above, making necessary changes proves to be a difficult task. Also extensions necessary to implement the requirements listed in Chapter II could prove to be difficult considering AMES' state of maintainability and extendibility. As a result, I avoid the use of AMES code for my

research.

Although AMES' source code is limiting, its implementation is of value. The result of Moritz's research yielded an effective prototype for displaying an endgame [Moritz96:6-5]. For this reason, I duplicate all of AMES features into AMVS with the exception of its method for displaying fragmentation fly-out, and technique for visualizing inter-object spatial relationships (see Sections 2.4.4.2 and 4.2.1 respectively).

4.6 Software Architecture Design

This section outlines my software architecture design starting with a presentation of AMVS' classes which will be referred to throughout the rest of this document. I then present the data flow of two key elements within AMVS: the scenario and the simulation time. Next, the sections that follow discuss this data flow while presenting the design of the *CEncount* and *CAnimControl* class. Afterwards, I present class designs for AMVS' dialogs, and saving and loading the simulation. Finally, I discuss how this software architecture design yields low coupling and high cohesion.

4.6.1 Class Hierarchy and Descriptions

The Class hierarchy for AMVS can be seen in Figure 4-8, followed by an aggregation and data flow diagram in Figure 4-9. However, Figure Figure 4-9 only presents the data flow relevant to the scenario and simulation time. Finally, Table 4-1 shows the class descriptions.

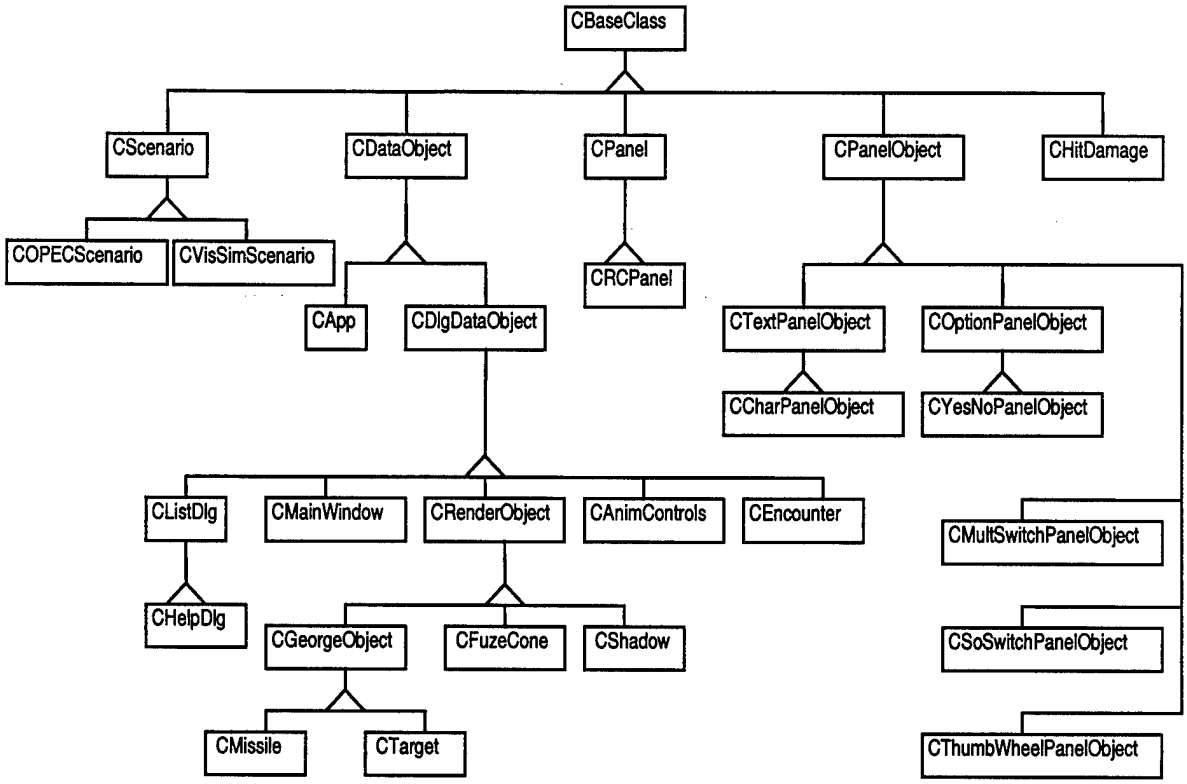


Figure 4-8: Class Hierarchy

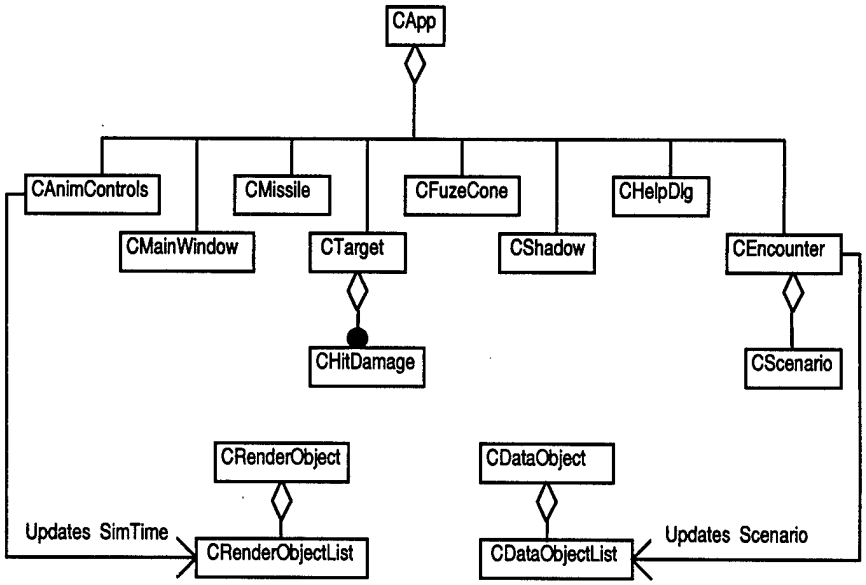


Figure 4-9: Aggregation and Data Flow

Table 4-1: Class Descriptions

CBaseClass	Base class for all other classes, provides run-time type identification (rtti).[5]
CScenario	Parent class for all scenarios. A scenario contains all information pertaining to the missile/target encounter
COPECScenario	Holds information about *endgame* encounters found in OPEC or ENCOUNT files.
CVisSimScenario	Holds information about *full fly-out* encounters produced by VisSim.
CDataObject	Base class for most AMES specific objects. *CDataObject*s accept a pointer to the current *CScenario* and are able to save and load their state to and from a Simulation file. Maintains a list of all instantiated *CDataObjects*.
CDataObjectList	List of *CDataObjects*. This list is a *static* member of *CDataObject*.
CApp	Main App class for AMES, instantiating all *CDlgDataObjects*. Initiates the saving and loading of a Simulation.
CDlgDataObject	Base class for all *CDataObject*'s requiring a user interface (Motif dialog). Registers help with the *CHelpDlg* class.
CListDlg	Implements a ScrolledList widget inside a *CDlgDataObject*.
CHelpDlg	Keeps a list of register help items, displays this list as a "help index" dialog. Instantiates a *CListDlg* and uses it to display the help for a topic read in from a file.
CMainWindow	Main window for AMES, containing menus and the Inventor *ExaminerViewer* (see Figure 5-1). Holds a pointer to the root of the scene graph. Implements saving/restoring of the rendering view.
CRenderObject	Abstract base class for all objects that will perform graphical rendering of information loaded in from a file. Contains a function *NewSimTime*, to accept the current time from the animation "clock". Maintains a list of all instantiated *CRenderObjects*.
CRenderObjectList	List of *CRenderObject*s. This is list is a *static* member of *CRenderObject*.
CGeorgeObject	Base class for the missile and target ("George" for the original file format used for missile and target models).

[5] The current compiler does not support rtti, and therefore needed to be implemented myself.

	Contains functionality similar to the missile and target.
CMissile	Missile class. Loads, displays and animates a missile model. Displays the fragmentation fly-out cone and animates fragmentation fly-out. Provides a user-interface for manipulating the missile (see Figure 5-5).
CTarget	Target class. Loads, displays and animates a target model. Displays OPEC target damage data. Provides a user-interface for manipulating the target (see Figure 5-4).
CFuzeCone	Fuze sensor pattern cones. Loads, displays and animates a fuze file. Provides a user-interface for editing fuze component settings (see Figure 5-6).
CShadow	Front, side, and top grids and 3-D shadows for missile and target. Provides a user-interface for manipulating shadows and grids (see Figure 5-7)
CAnimControl	Controls the animation "clock". Provides an interface for controlling the simulation time (see Figure 4-6). Passes the current simulation time to all *CRenderObjects*.
CEncounter	Displays information contained in a *CScenario* (see Figure 5-2). Allows loading, saving (ENCOUNT files) and editing of the current scenario, sends the current scenario to all *CDataObjects*.
CPanel	Used in the construction of Motif style user interfaces. Creates a form for the placement of *CPanelObjects*, keeps a list of all *CPanelObjects* added to this panel. Iterates through this list calling their *GetInfo* and *SetInfo* functions. If an error occurs in an object's *GetInfo* function, iteration stops and an error message is displayed.
CRCPanel	Inserts a RowColumn widget into the form created by a *CPanel*. Places *CPanelObjects* in this widget.
CPanelObject	Abstract base class for objects creating widgets for displaying/editing a single data variable. *SetInfo* updates widgets according to value held by the data variable. *GetInfo* performs error checking on user input and updates the data variable for valid input.
CTextPanelObject	Creates a TextField for displaying strings, integers or floats. Provides value range error checking and textual/numerical conversion for fields intended for integer or float input.
CCharPanelObject	TextField with width of one. Error checking can be performed against a valid string of characters.
COptionPanelObject	Implements an Motif Option Menu widget.
CYesNoPanelObject	*COptionPanelObject* with two menu items: Yes, No.

CSoSwitchPanelObject	Implements a toggle button to control an Inventor *SoSwitch* node. The *SoSwitch* node controls the visibility of objects in the rendering scene.
CMultSwitchPanelObject	Implements a toggle button to control a list of *SoSwitch* nodes. Used exclusively for level-of-detail models.
CThumbWheelPanelObject	Implements a ThumbWheel widget.
CHitDamage	Used by a *CTarget* for holding hit damage loaded in from an OPEC .out file.

4.6.2 Data Flow using an Observer Behavioral Pattern

The current simulation time and current scenario are controlled by the *CAnimControl* and *CEncounter* classes respectively. The simulation time is represented as a float value while the scenario is encapsulated in the *CScenario* class. When the simulation time changes during animation, all *CRenderObjects* must reflect this change graphically; likewise all *CDataObjects* must reflect changes to the current scenario. This change is generally graphical but could include changes to the associated user-interface as well. To notify other classes of changes to these variables, I use an *Observer* behavioral pattern[6]. In *Designing patterns: Elements of Reusable Object-Oriented Systems*, Gamma, et al. describes several patterns useful in software development. He classifies an *Observer* pattern under the behavioral category and defines it as follows:

> "Observer - defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
>
> [Gamma95:9,293-303]

In order to implement this pattern, I use a Standard Template Library (STL) *Vector* to define a *CRenderObjectList* and *CDataObjectList* to keep a list of all *CRenderObjects* and *CDataObjects*. These lists are static members of each respective class. In the constructor function of each of

[6] Fowler loosely defines a pattern to be "an idea that has been useful in one practical context and will probably be useful in others." [Fowler97:xv]

these classes, newly instantiated objects are added to the respective lists to ensure that all child classes are automatically. The *CRenderObject* class then defines a pure virtual *NewSimTime* function which takes the simulation time as a single parameter. Likewise, the *CDataObject* class defines a virtual *SetScenario* function taking a single *const* pointer to a *CScenario* object. These functions are then overloaded by child classes to take appropriate action on state changes. The *CRenderObjectList* and *CDataObjectList* are then used to notify all classes of state changes. Code to do so can be seen in Figures 5-10 and 5-14. The next two sections introduce the *CEncounter* and *CAnimControl* classes which control the scenario and simulation time respectively.

4.6.3 Scenario

A scenario is defined as all the information pertinent to a missile/target encounter. The *CScenario* class was designed to encapsulate this information for distribution throughout the rest of AMVS' classes.

AMVS is implemented to visualize and animate of both endgame and full fly-out encounters. Endgame parameters are found in ENCOUNT and OPEC files and are stored in the *COPECScenario* class. Both ENCOUNT and OPEC's parameters are similar enough that a single scenario object is used. The *COPECScenario* class contains a flag specifying whether the information contained within it was derived from OPEC or ENCOUNT data. Likewise, the *CVisSimScenario* class contains information regarding a VisSim fly-out simulation.

The *CEncounter* class controls the contents of a scenario as well as which scenario is currently being viewed by all rendering objects (ENCOUNT, OPEC or VisSim). The *CEncounter* dialog allows users to load and edit scenarios. *CEncounter* uses the *Observer* pattern described above to notify all *CDataObjects* of changes to the current scenario. When the current scenario changes, the *CEncounter* class iterates through the list of *CDataObject*'s, passing each a

pointer to the current scenario by use of the *SetScenario* function. Each keeps a *const* pointer to the *CScenario* controlled by the *CEncounter* class and uses this information to modify its user interface and/or scene graph accordingly. For instance, when a VisSim scenario is selected, the missile modifies its scene graph to include the missile's fly-out route and changes it dialog by removing the war-head user interface.

4.6.4 Rendering and Animation

All three-dimensional rendering and animation is performed by classes inheriting from the abstract base class *CRenderObject*. Each *CRenderObject* encapsulates data, functionality and user-interface unique to an object or set of objects for three-dimensional rendering. The *CRenderObject* class is designed for performance and extendibility.

Many design decisions went into the creation of the *CRenderObject* class to enhance AMVS' extendibility, most of which involve inheritance and polymorphism. Indirectly inheriting from *CDataObject* provides each render object with automatic notification of the current scenario. Each render object overloads the *SetScenario* function, performing any action necessary to render its scene graph according to the current scenario. Inheriting from *CDlgDataObject* provides functionality for the creation of the render object's user-interface (discussed in the next section). As mentioned above, the *CRenderObject* uses an STL *vector* to maintain a list of all instantiated objects inheriting from *CRenderObject*. The *CAnimControl* class calls each *CRenderObject*'s pure virtual *NewSimTime* function for each frame in the animation, or whenever the current simulation time changes. Any class inheriting from *CRenderObject* has automatic notification of the current simulation time. With this inheritance structure and the virtual functions we have defined, adding new rendering objects is simplified. To add a new rendering object to AMVS, only the following steps are required:

1. Create the Inventor scene graph in the constructor, keeping pointers to key nodes.
2. Overload *CDlgDataObject*'s *Create* function to create the dialog components.
3. Add a menu item to the *CMainWindow* to call this object's *Show* function.
4. Overload the *SetScenario* function, if this object is effected by the current scenario.
5. Overload the *NewSimTime* function, if this object is to be animated.

The *CRenderObject* class increases performance by having each child class create and manipulate its own rendering scene graph, as well as localizing all information needed for a single object to perform rendering during a single "frame" of animation. This information is derived from user input, the current simulation time, and data found in the current scenario. Localizing all rendering information avoids the overhead invoked when retrieving data from other classes through member access functions. When performance is not an issue, such encapsulation techniques are followed.

4.6.5 Dialogs

Design of the user interface classes proceeded with code re-use and ease of implementation in mind. The *CDlgDataObject* class encapsulates the creation and manipulation of Motif dialogs, and is used by all *CDataObjects* requiring a user interface. This class creates a dialog shell, a tailorable action area, messaging dialogs, and file selection dialogs. The action area includes buttons such as OK, APPLY, DONE, CANCEL and HELP, as well as the option to create user defined labels. Virtual callback functions are created for each button and can be overloaded as necessary. The HELP button brings up on-line help registered through the *RegisterHelp* function of any instantiated dialog. Pop-up message dialogs including a simple message dialog, and a question dialog with callbacks connected to the virtual *Yes* and *No* functions are implemented and easily invoked. The 'Open File' and 'Save As file' selection dialogs each having appropriate virtual callbacks which are overloaded by child classes. These

callbacks are passed the filename and path selected by the user.

The creation of the user interface takes place in the *CDlgDataObject*'s virtual *Create* function. This can be done with straight Motif code, or the use of panels. The *CPanel* and *CPanelObject* classes provide a convenient method of creating the user interface within the dialog. The abstract base class *CPanelObject* encapsulates the user interface for a single variable. A *CPanelObject*'s constructor function is passed the pointer to the variable and a description to be displayed with the associated widget. Classes inherited from *CPanelObject* create Motif widgets for displaying and editing the variable and take advantage of the *SetInfo* and *GetInfo* functions for updating the widget state or variable value respectively. When the variable changes value, the virtual *SetInfo* function can be used to set the widget's state based upon the new value. When the user changes the widget's state, the *GetInfo* function retrieves state information from the widget and sets the variable accordingly, if no error on input occurred. The classes also provided error checking based upon criteria for valid user input. For example, a *CTextPanelObject* can be set to display and edit an integer variable. The valid range of integer values is identified through the *IntRange* function error checking performed automatically in the *GetInfo* function.

The *CPanel* class is a container for *CPanelObjects*. This places panel objects in a Motif form or row-column widget. The *CPanel* class keeps a list of all *CPanelObjects* added to it. It provides iteration functions for calling the *GetInfo* and *SetInfo* functions of all *CPanelObjects* added to this panel. Once a panel has been constructed, a call to the *CPanel*'s *SetData* will automatically cause all *CPanelObjects* to read their associated variables and set their widgets appropriately. A call to *GetData* causes each object to read its widget values, perform valid input testing, and update the associated variables. If an error occurred, iteration stops and an error message is displayed in a pop-up dialog.

4.6.6 Saving/Loading of the Simulation

One recommendation for future work outlined by Moritz was saving the current encounter [Moritz96:6-4]. This originally involved saving the current ENCOUNT or OPEC files, along with the target and missile models selected by the user. I have extended this list to include potentially any of AMVS' variables or states. In order to do so, the *CDataObject* class includes virtual functions *SaveData* and *LoadData* for saving and loaded variables. During a save operation, the *CApp* class will iterate through the list of *CDataObjects* to save the current encounter by passing each data object a C++ iostream. Each *CDataObject* can write out any member variables in the *SaveData* function and later retrieve these values back into its member variables in the *LoadData* function. Placing the saving and loading functionality in the *CDataObject* provides me access to all state information within AMVS. Although the inheritance graph in Figure 4-8 shows some classes not inheriting from *CDataObject*, each of these are only instantiated from within a *CDataObject*, thereby ensuring the set of all *CDataObject*s has access to all state information.

4.6.7 High Cohesion, Low Coupling

This software architecture design yields high cohesion and low coupling. High cohesion results from having each rendering object class contain all information and functionality necessary for file I/O, graphical rendering, and user interfacing, as pertaining to the real world objects they model. This in turn results in low coupling by minimizing data flow between rendering objects to that information which is common to all: the simulation time and scenario.

4.7 Rendering Scene Graph Design

AMVS' *CMainWindow* class maintains a pointer to the root of the scene graph. All sub-scene graphs are encapsulated in a *CRenderObject* (defined in Section 4.6.4). Each

CRenderObject is responsible for the creation and maintenance of its sub graph. This section outlines design decisions involved in the creation of AMVS' rendering scene graph. Topics include transparency, performance, and use of the Open Inventor *SoSwitch* node for component visibility and levels of detail.

4.7.1 Transparency

Object ordering in the scene graph effects how Open Inventor renders transparency. Z-buffer based transparency algorithms require transparent objects to be drawn last during rendering [Foley92:755]. Therefore, the scene graph is designed with this in mind. Since each *CRenderObject* creates its own sub scene graph and attaches it to the root node, instantiation order is important. AMVS instantiates the *CRenderObjects* in the following order: *CShadow, CTarget, CMissile, CFuzeCone.* This order insures that the potentially transparent fuze sensor pattern cones and fragmentation fly-out pattern cone are rendered last. Additionally, since the target may have a transparent skin, I construct the target models with the skin components placed last in its scene graph.

4.7.2 Performance

This section addresses new improvements to the scene graph to increase rendering speed. First, AMVS uses fewer transformation nodes by performing many of the three dimensional transformation calculations outside the scene graph. For example, the target and missile each use only a single transformation node to replace the three which were used for the target and four for the missile. The calculations needed for the target and missile transformation are shown in Section 5.3. Second, when applicable, translation nodes were used instead of transformation nodes when no rotation calculations were necessary. Likewise rotation nodes were used when only a rotation was needed. Using specific node types where possible further reduced the overhead of three-dimensional transformation calculations during rendering by avoiding full

transformation calculations.

Another inefficiency in AMES' scene graph was its use of six point light sources to illuminate the cones interiors. Without these light sources, the insides of the cones received no illumination. An alternative solution to this problem is to place a single directional light in the fuze cone's scene graph pointing towards the cones interiors. Placing this directional light after the fuze cone's transformation node insures that the light will always point towards them. This method significantly increased performance for three reasons: 1) directional light sources require less computation [Wernecke94:92], 2) fewer light sources are required and 3) only a subset of the scene graph requires illumination calculations. Tests revealed a 30% increase in performance using this method.

4.7.3 Use of the SoSwitch Node

The Open Inventor *SoSwitch* node is a grouping node allowing for the selection of one, all or none of its children [OIAG94:616]; it is used to select sub-scene graph visibility. The *SoSwitch* node is also used to control target and missile levels of detail (LOD). Providing the user with multiple levels of detail allows them to select between improved performance or improved model appearance. Level-of-detail target and missile models are constructed with a *SoSwitch* node as the root. An option menu widget allows the user to select between each level of detail.

4.8 Conclusion

In this chapter, I cover design decisions regarding three-dimensional visualization, one of which involved the use of shadow projections to enhance inter-object spatial perception. I also discussed user interface design, including an improved animation control dialog. Critical decisions made early in development were presented including choice of rendering software

library and the user interface library, and avoiding use of existing code. Finally, I explained software and rendering scene graph design. In the next chapter, highlights of the final product are presented along with some specifics about the underlying implementation.

5. Implementation

5.1 Introduction

This chapter outlines the AFIT Missile Visualization System's (AMVS) implementation. The first section examines AMVS from a user's perspective, revealing its major features and explains how they relate to the requirements outlined in Chapter II. Subsequent sections provide insight into AMVS' underlying implementation. The final section discusses portable and platform independent animation of VisSim fly-out simulations.

5.2 AMVS' Features

5.2.1 Main Window

AMVS' main window can be seen in Figure 5-1. The main window provides the menuing interface, three-dimensional rendering window, and viewpoint control. The user controls the rendering scene viewpoint with the mouse. Viewpoints can be saved and later restored in one of five view-holders located at the bottom of the window. These five viewpoints, along with the current one are saved when writing the encounter to disk.

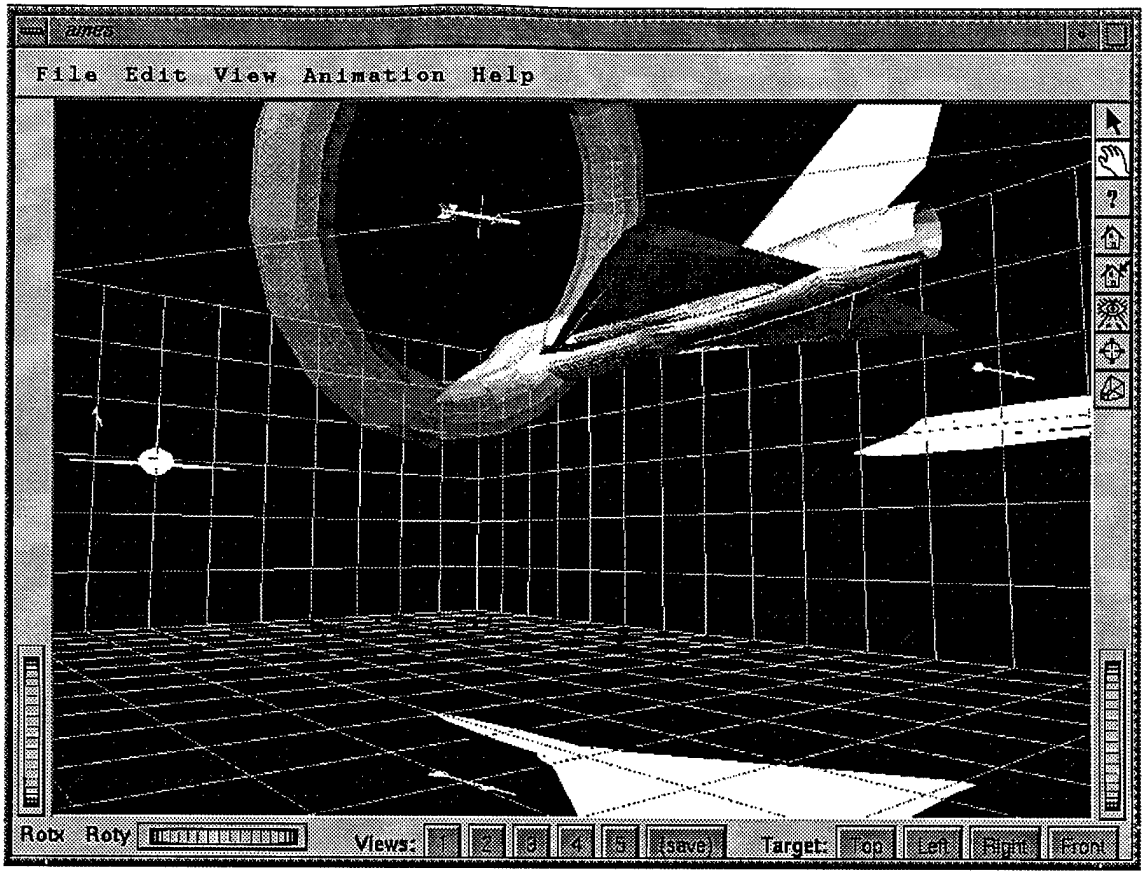


Figure 5-1: AMVS' Main Window

5.2.2 Encounter Dialog

The encounter dialog, shown in Figure 5-2, provides the user control over the current scenario. Scenarios are loaded, edited, and saved[7] using the graphical user interface. Currently AMVS' scenarios include ENCOUNT and OPEC endgame files along with VisSim full fly-out simulations. Changes made to fields in this dialog are immediately reflected in the graphical display.

[7] OPEC produced files may be loaded and edited but will not be saved. These files are output produced by OPEC and the sponsor does not need the ability to change them.

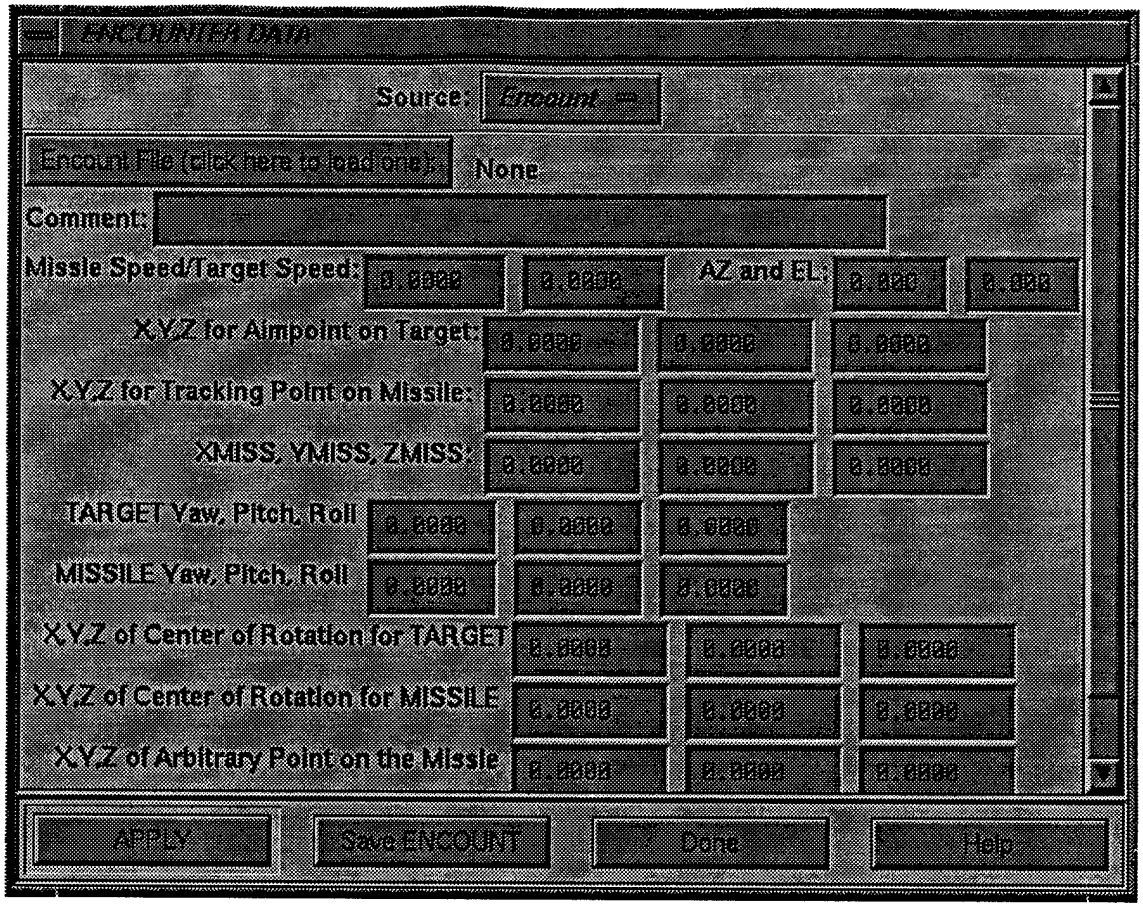


Figure 5-2: Encounter Dialog

Section 2.3.4 discusses how engineers use ENCOUNT in conjunction with IVAVIEW to set up and visualize an endgame encounter. AMVS provides an improved process: a single application to load, edit, and save an encounter file while displaying it both textually and graphically.

AMVS' ability to load OPEC generated files for textual and graphical display increases OPEC's usefulness as a tool for simulating endgames. As mentioned in Section 2.3.5, OPEC is a valuable tool for predicting missile component performance against selected targets; however, OPEC suffers from poor graphical rendering. AMVS is written to meet the need for graphical display of OPEC simulation results, improving the engineer's ability to visualize and understand the results.

Section 2.3.6 introduced the requirement for visualizing VisSim fly-out simulations. AMVS allows users to load VisSim files through the encounter dialog. Once loaded, the engineer can animate the fly-out using the same animation control dialog for endgame simulations.

Figure 5-3 shows the rendering of an example VisSim fly-out simulation involving a ground target.

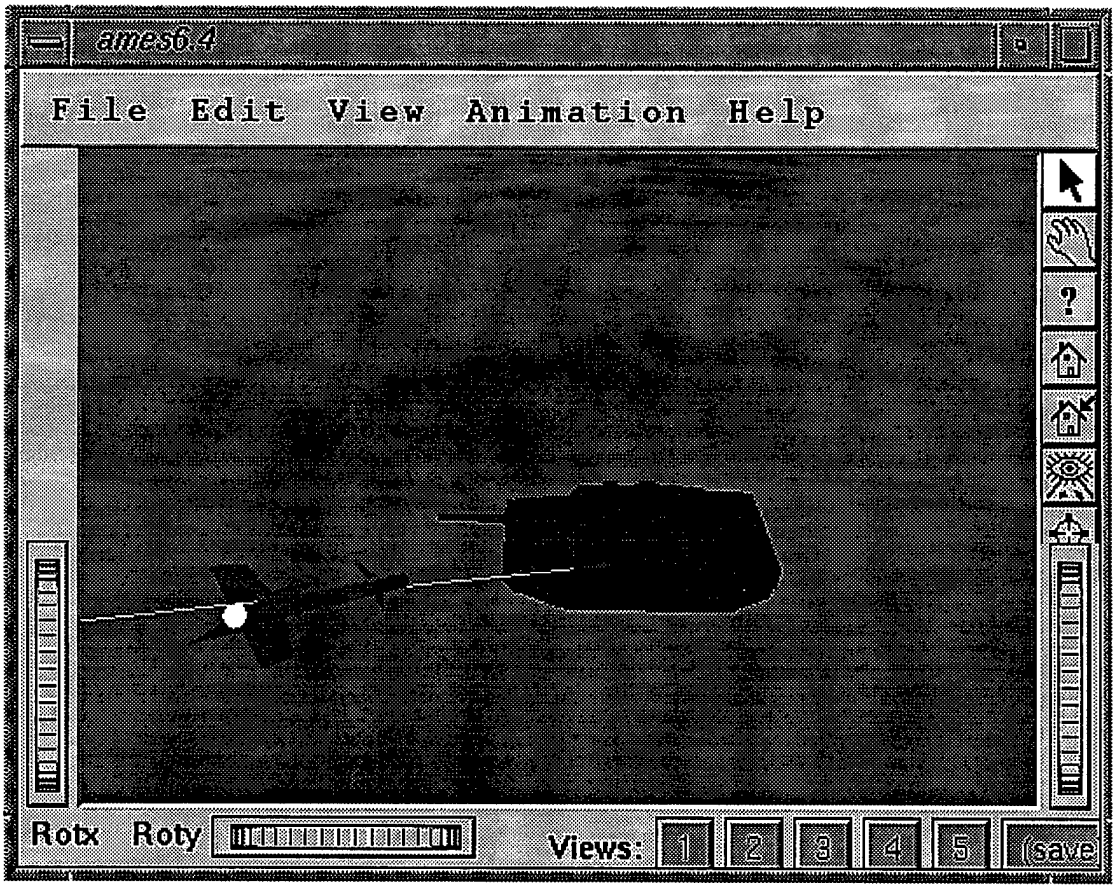


Figure 5-3: VisSim Full Fly-out Animation

5.2.3 Animation Controller

The animation controller is explained in detail in Section 4.3.3. Animating an encounter improves an analyst's understanding of the missile/target interaction during an endgame, particularly for encounters in which the missile intersects the target body. Knowing where

missile body components intersect the target during an encounter is important for calculating probability of kill (pk). Pk calculations are primarily based on warhead fragmentation intersecting target components, but include intersection of the missile body with the target [McCardle97]. ENCOUNT provides a means of visualizing this interaction by displaying a relative velocity vector line through a specified arbitrary point on the missile. This line shows the path of this point relative to the target, projecting where an intersection occurs. AMVS displays this line as well; however animation of the encounter improves upon this technique significantly by providing the engineer immediate understanding of missile/target intersection for the entire missile and not just a single user specified point.

Animating the encounter also enhances AMVS as a training and briefing tool for conveying concepts unique to an endgame. Sections 2.2.2 and 2.2.3 identified the fundamental roles of the missile's fuze and warhead components. AMVS animates these components to assist individuals in understanding the timing relationship between the fuze and warhead components. The fuze sensor cones transparency or visibility values are set to change when a target is passing within the fuze sensor's range. Warhead fragmentation is displayed as both an expanding ring and a torus. The user witnesses the correlation of fuze sensor target detection and warhead detonation. Simulation time during animation is displayed in the animation control dialog and assists in understanding timing issues.

5.2.4 Target Dialog

Figure 5-4 shows the target dialog. The user loads target models and SHAZAM-produced output files with this dialog as well as set the level of transparency for the aircraft skin components. Visibility of all aircraft component groups is toggled on or off by the user through an interface that is patterned off of the IVAVIEW program. If the loaded target file contains varying levels of detail, the user is allowed select between these using this dialog. This provides

the user the ability to increase AMVS' performance by reducing the scene graph complexity.

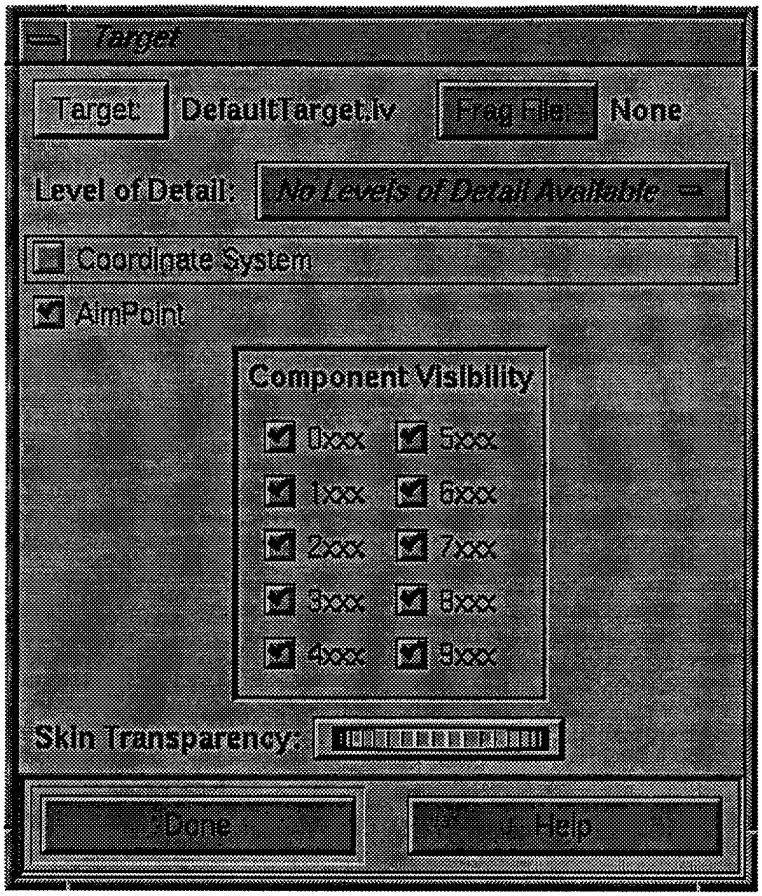


Figure 5-4: Target Dialog

The user can load SHAZAM files using the target dialog. The SHAZAM program produces simulation results showing where simulated fragmentation pieces are at specified times during the simulation. The fragmentation file flags components that have intersected a target with AMVS graphically displaying these results. During animation, the time of impact is used to determine when to turn on impact visibility.

OPEC *.out* files loaded in the encounter dialog contain target component damage data. This dialog provides the engineer the ability to view the colored component damage (component coloring is explained in Section 4.2.5) by setting aircraft skin transparency. The user can isolate specific target components groups, such as flight control system and hydraulics, by setting component group visibility.

5.2.5 Missile Dialog

The missile dialog, shown in Figure 5-5, allows the user to control which missile is loaded, level of detail for the missile, which velocity vector lines are displayed (both relative and actual), and information about the warhead. The user sets the warhead position on the missile through the missile

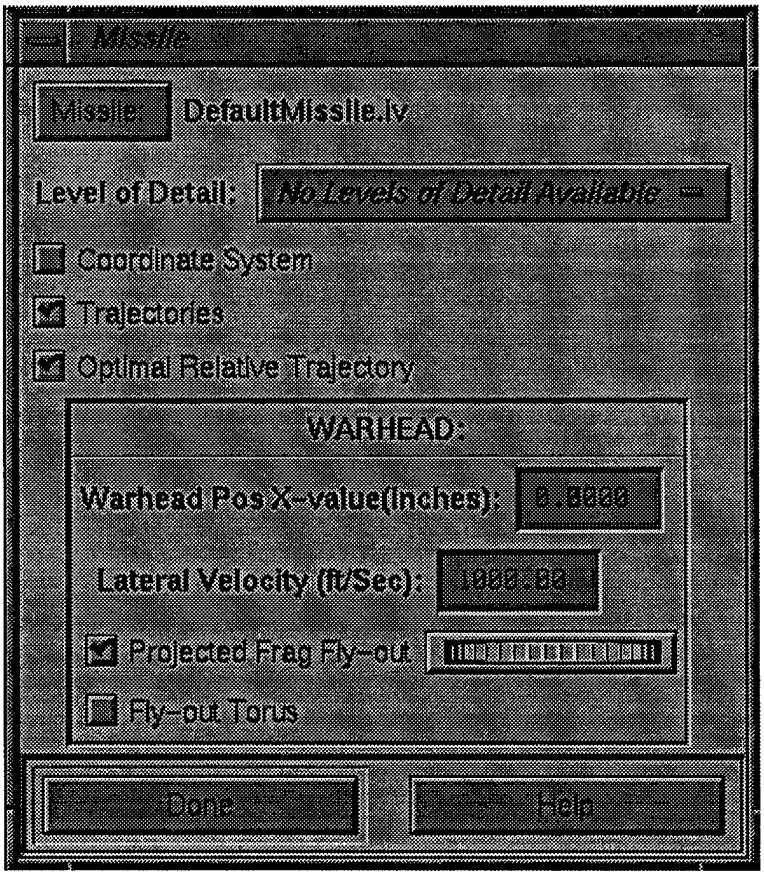


Figure 5-5: Missile Dialog

dialog. The warhead position is represented as a vector in the missile coordinate system with Y and Z values set to 0. Fragmentation lateral fly-out velocity is in inches per second and is set by the user. The user has control over visualizing the fragmentation fly-out. Fragmentation fly-out is visualized in two ways: first, as a cone representing the path of warhead fragmentation from the missile given the fragmentation fly-out velocity along with speed and orientation of the missile. Second, an animated ring or torus shows the fragmentation dispersal at specifics point in time.

AMVS includes the visualization of fragmentation fly-out only to convey concepts in fuzing and warhead detonation. It does not attempt to model the realistic physical phenomenon of fragmentation. Therefore, effects of air-resistance and gravity have been left out to simplify the model [Starfield90:8]. Never the less, the simplified model is sufficient to demonstrate the important effect missile velocity and orientation has on the fragmentation fly-out cone. This problem of fragmentation pattern skewing due to missile yaw or pitch and its effects on targeting is discussed in detail in Section 2.2.4. AMVS pictorial demonstration of this principle assists individuals in understanding this phenomenon and how it effects the performance of air intercept missiles. Derivation of the fragmentation fly-out skewed cone calculations appears in Section 5.3.2.

5.2.6 Fixed Fuze Cone Dialog

The missile fuze cone dialog is shown in Figure 5-6. This dialog provides the ability to load a fuze file, modify the fuze attributes, set fuze cone visibility and transparency, and specify information for animating the fuze cone. Fuze sensor patterns are displayed graphically as transparent cones.

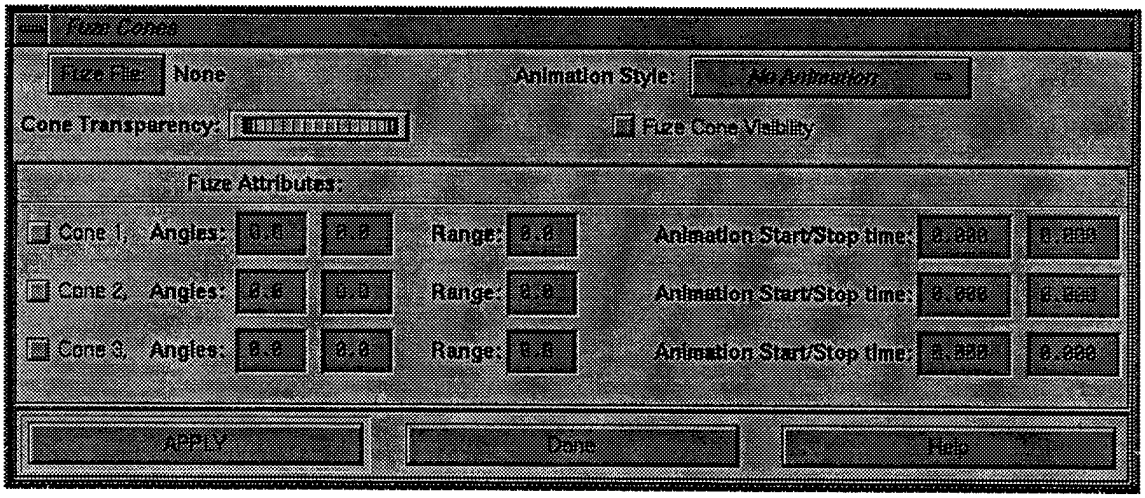


Figure 5-6: Fixed Fuze Cone Dialog

Engineers examine existing fuze components by loading fuze files containing the fuze sensor pattern specifications. They may also edit the fuze specifications with this dialog and examine the results graphically. As mentioned in Section 2.2.3, the fuze setting effects the type of missile/target encounter and ultimately the time-to-burst value. For example, larger fuze angles allow the missile to calculate a valid time-to-burst against a target approaching at a larger azimuth, while smaller fuze angles allow the missile to calculate a valid time-to-bust for high-speed head-on encounters. AMVS provides the engineer a tool for experimenting with different fuze settings for specific encounters. Because fuze cone animation is primarily for training and briefing purposes, and not sensor analysis, AMVS does not attempt to simulate real fuze sensor target detection. As a result, the target detection time must be entered manually by the engineer.

Currently, air-intercept missiles are designed and developed with only a single fuze sensor. Adding additional fuze components to a missile can potentially increase the range of encounters a missile can properly operate in to effectively kill a target. AMVS provides the engineer an environment for experimenting with up to three fuze sensors.

5.2.7 Shadow and Grid Dialog

Shadow projections, as introduced in Section 4.2.1, improve the perception and mental modeling of missile/target inter-object spatial relationships. This is most valuable in analyzing the endgame parameters of a scenario set by the encounter dialog (see Section 5.2.2). Figure 5-7 shows the dialog for controlling grids and shadows. An example of this visualization technique appears earlier in Figure 5-1. Using the shadow and grid dialog, the user has the option to turn the visibility of top, side and back shadows on or off. They may also change grid visibility, units dimensions and placement.

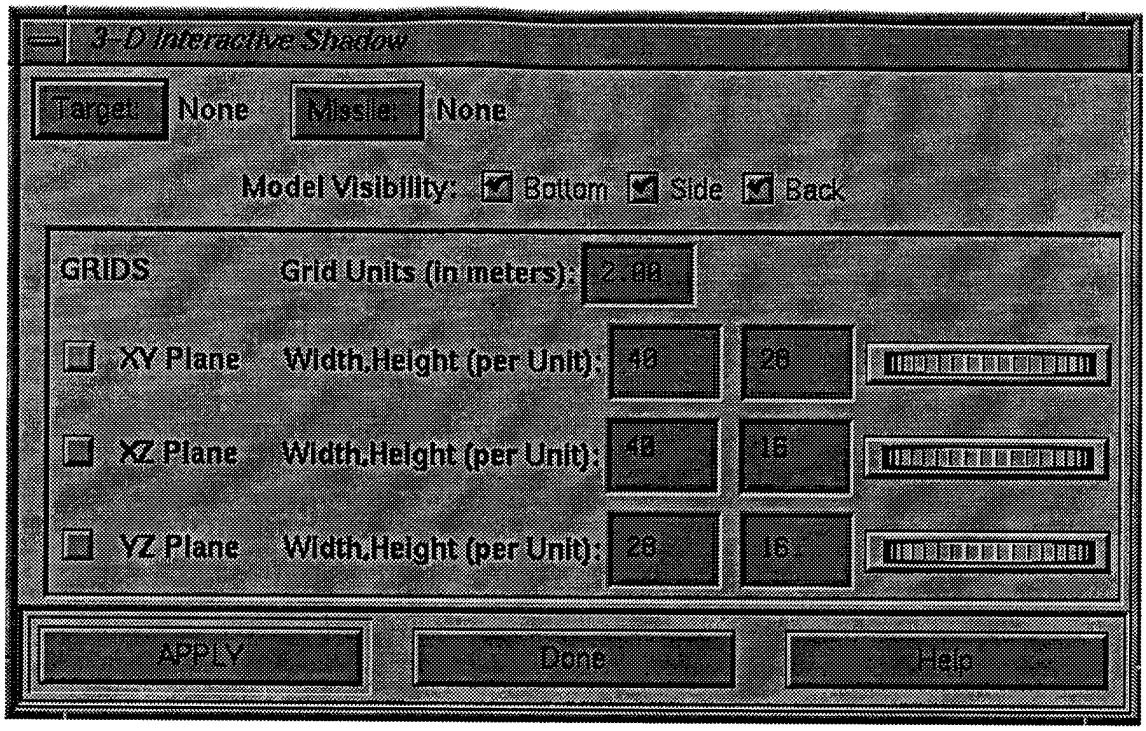


Figure 5-7: Shadow and Grid Dialog

5.2.8 Visual Cone Cross Sections

The dialog for setting fuze and fragmentation fly-out cones to 2D cross sections is shown in Figure 5-8. Figure 5-9 shows an example display of cone cross sections. Once the user selects the cross section modes, she dials the cone rotation to a desired position.

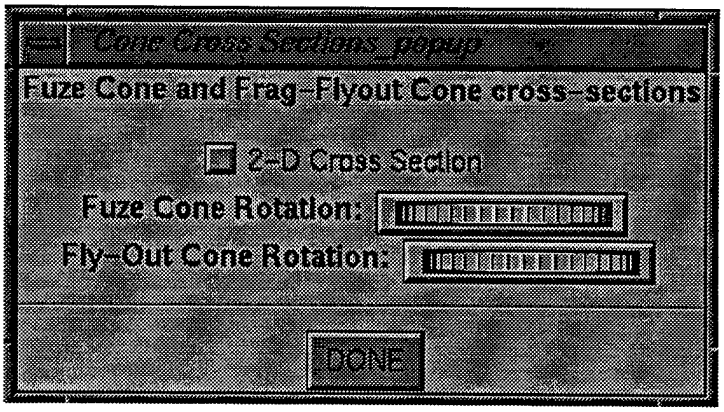


Figure 5-8: Cone Cross Section Dialog

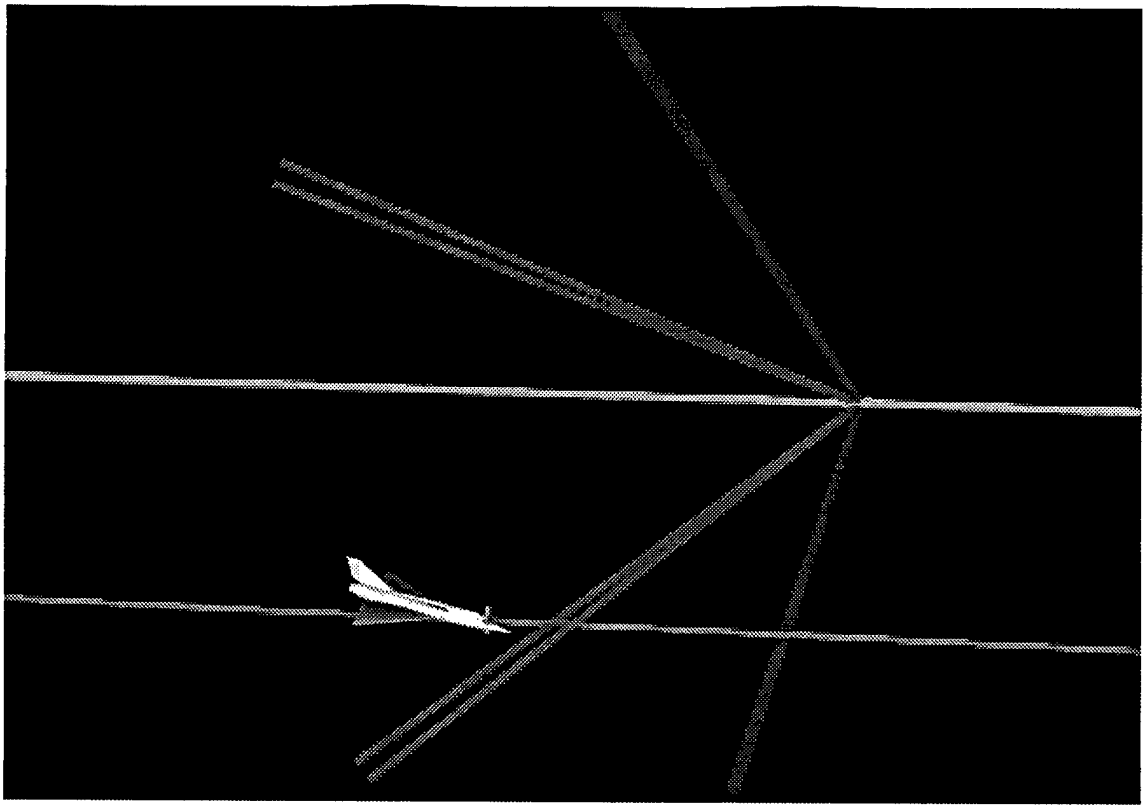


Figure 5-9: Example Cone Cross Section View

As mentioned in Section 4.2.3, this visualization technique, patterned after two-dimensional drawings in missile proximity sensor publications and manuals, is both familiar to the missile engineer and useful for modeling the interaction of the target and cones.. This technique is especially useful when the target is not directly intersecting the cone.

5.3 Setting up a Scenario

Now that we have examined AMVS from the user's point of view, we turn our attention to some of its underlying implementation, beginning with calculations for setting up an endgame scenario. The current scenario is controlled by the *CEncounter* class. When the scenario changes, the *CEncounter* class notifies all *CDataObject*s of the new scenario (see Figure 5-10). As data object in the list is visited the object's virtual *SetScenario* is called with the *CScenario* object holding the current scenario.

(This line of code is located in *CDataObject* constructor)

```
...
// Get a pointer to the DataObject lists
m_dataList = CDataObject::List();
...

...
// Notify all the CDataObjects of the new CScenario
for(m_listIndex=m_dataList->begin(); m_listIndex!=m_dataList->end();    \
                                        m_listIndex++ )
    (*m_listIndex)->SetScenario(m_scenario);
....
```

NOTE:
The following are defined in the *CDataObject* header file:

```
typedef vector<CDataObject*> DataObjectList;
typedef DataObjectList::iterator DataObjectListIterator;
```

The following are member variables of the *CEncounter* class:

```
DataObjectList*  m_dataList;
DataObjectListIterator  m_listIndex;
```

Figure 5-10: Notifying Data Objects of the New Scenario

Since *CDataObject's* constructor adds each newly instantiated data object to the data object list, all classes inheriting from *CDataObject* will be automatically notified of changes to the current scenario. Each *CDataObject* is then responsible to modify its state according to the new scenario. The following sections outline some of the state changes that occur with a new simulation.

5.3.1 Target, Missile and Relative Velocity Vector Transformations

Scenarios contain data derived from ENCOUNT or OPEC files. This data primarily effects the orientation of the target and missile. ENCOUNT and OPEC calculate missile and target orientation in the same way. This section outlines the transformations required to position the target, missile, and relative velocity vectors based upon the current scenario. Variables from the scenario, along with additional variables for calculation, can be seen in Table 5-1. All vectors are normalized unless otherwise specified.

Table 5-1: Endgame Scenario Variables

Variable Name	Type	Description
AZ	Degree	Azimuth of missile attach
EL	Degree	Elevation of missile attach
M_{AA}	Matrix	Attack angle (azimuth and elevation)
V_{MV}	Vector	Missile's unit velocity vector
Mvel	Float	Missile's velocity
V_{TV}	Vector	Target's unit velocity vector
Tvel	Float	Target's velocity
V_{RV}	Vector	Relative velocity vector
Rvel	Float	Relative velocity
M_{RVCS}	Matrix	Relative velocity vector coordinate system (rotational transformation only).
V_{AP}	Vector	Aimpoint in target space
V_{MP}	Vector	Miss point in relative velocity vector coordinate system.
$V_{XYZMiss}$	Vector	Aimpoint miss value in relative velocity vector coordinate system
V_{TP}	Vector	Tracking point on the missile
M_{TPT}	Matrix	Tracking point translation matrix, holds values in vector V_{TP}
M_{MYaw}, M_{MPitch}, M_{MRoll}	Matrix	Missile yaw, pitch and roll
M_{TYaw}, M_{TPitch}, M_{TRoll}	Matrix	Target yaw, pitch and roll
M_{MR}	Matrix	Missile rotation
M_{TT}	Matrix	Target's total transformation
M_{MT}	Matrix	Missile's total transformation
M_{RVAPT}	Matrix	Relative velocity vector and aimpoint transformation
M_{TPRVT}	Matrix	Tracking point relative velocity vector transformation
$V_{TPosAtZero}$	Vector	Target translation value for an animation time of 0.0. (used in Section 5.4.1) This value is extracted from M_{TT}.
$V_{MPosAtZero}$	Vector	Missile translation value for an animation time of 0.0. (used in Section 5.4.1) This value is extracted from M_{MT}.

5.3.1.1 Calculating the velocity vectors

The target velocity vector is always along the world coordinate x-axis and is not effected by target yaw, pitch or roll. The missile velocity vector is based upon the attack azimuth and elevation and is also not effected by missile yaw, pitch or roll. An azimuth value of 0 degrees is defined to be a head-on attack with the target, while an azimuth of 180 degrees is an attack from

behind. A positive elevation represents an attack from above. Equation 1 shows the angle of attack calculation. Equation 2 shows how to calculate the missile velocity vector (V_{MV}) by multiplying a unit vector in the X direction through the angle of attack matrix. For a discussion on three-dimensional transformations, see [Hearn97],[Rogers90] or [Watt93].

$$M_{AA} = \begin{bmatrix} \cos\ 180+AZ & -\sin\ 180+AZ & 0 & 0 \\ \sin\ 180+AZ & \cos\ 180+AZ & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\ EL & 0 & \sin\ EL & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\ EL & 0 & \cos\ EL & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

$$V_{MV} = M_{AA} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (2)$$

5.3.1.2 Calculating the relative velocity and relative velocity vector

The relative velocity vector is found using vector subtraction on the missile and target velocity vectors as shown in Equation 3. This vector has a magnitude of relative velocity. Equations 4 and 5 show the calculation of the unit relative velocity and the relative velocity vector, respectively.

$$V'_{RV} = (V_{MV}\, MVel) - (V_{TV}\, TVel) \quad (3)$$

$$V_{RV} = \frac{V'_{RV}}{|V'_{RV}|} \quad (4)$$

$$Rvel = |V'_{RV}| \quad (5)$$

5.3.1.3 Calculating relative velocity vector coordinate system for XYZMiss translation

A miss occurs when the missile is unable to intercept the target at the aimpoint during an

endgame [Mack87:2]. The miss value is a point in a coordinate system centered at the aimpoint with x-axis being the relative velocity vector. This coordinate system must be calculated for proper placement of the missile as well as proper orientation of the aimpoint cross-hairs. More specifically, to calculate the transformation for the aimpoint cross-hairs, we need the inverse rotation of this coordinate system (see Section 5.3.1.6) and can ignore aimpoint translation.

The relative velocity vector rotation matrix is calculated using unit the local coordinate system's unit axis vectors (described below) as columns in the matrix [Hearn97:428-9]. The relative velocity vector is used as the x-axis unit vector for this matrix (positive x represents the direction of the missile's movement towards the target). The z-axis unit vector is found by taking the cross product of the relative velocity vector with the world-coordinate y-axis unit vector. The cross product of the x-axis unit vector and z-axis unit vector define the y-axis unit vector for the miss coordinate system. These calculations appear in Equations 6, 7 and 8. The rotation matrix for defining the relative velocity vector coordinate system centered at the origin is shown in Equation 9.

$$V'_X = V_{RV} \tag{6}$$

$$V'_Z = V'_X \times [0 \quad 1 \quad 0] \tag{7}$$

$$V'_Y = V'_X \times V'_Z \tag{8}$$

$$M_{RVCS} = \begin{bmatrix} V'_{X1} & V'_{Y1} & V'_{Z1} & 0 \\ V'_{X2} & V'_{Y2} & V'_{Z2} & 0 \\ V'_{X3} & V'_{Y3} & V'_{Z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

5.3.1.4 Calculating the target transformation

The target's initial orientation is located at the origin, facing in the positive x-axis with the y-axis out the left wing. The target's orientation is defined by yaw, pitch and roll values. The

calculation for the target's rotation is shown in Equation 10.

$$M_{TT} = M_{TYaw} M_{TPitch} M_{TRoll} \quad (10)$$

5.3.1.5 Calculating the missile transformation

The missile's initial orientation is the same as the target's. The transformations for orienting the missile according to the current scenario are summarized as follows:

The missile is aligned with the attack angle,

The yaw, pitch and roll rotation is applied to the missile,

The missile is translated by the miss vector in the relative velocity vector coordinate system.

The missile is translated to the aimpoint in the target coordinate system.

To make the final two transformations, we introduce two temporary vectors V_{APW} and V_{MPW} representing the aimpoint and miss point translation vectors respectively in the *world-coordinate system*, and two temporary matrices M_{APT} and M_{MPT} to represent these two translations. These calculations are shown in Equations 11 through 14. M_{APT} and M_{MPT} are used again later when calculating the transformations of the relative velocity vector lines.

$$V_{APW} = M_{TT} V_{AP} \quad (11)$$

$$V_{MPW} = M_{RVCS} V_{MP} \quad (12)$$

$$M_{APT} = \begin{bmatrix} 1 & 0 & 0 & V_{APW_x} \\ 0 & 1 & 0 & V_{APW_y} \\ 0 & 0 & 1 & V_{APW_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

$$M_{MPT} = \begin{bmatrix} 1 & 0 & 0 & V_{MPW_x} \\ 0 & 1 & 0 & V_{MPW_y} \\ 0 & 0 & 1 & V_{MPW_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (14)$$

The missile's total rotation matrix calculation is shown in Equation 15. This matrix is saved for later calculation of the displayed relative velocity vector through the missile's tracking point. Equation 16 shows the missile's total transformation.

$$M_{MR} = M_{MYaw} M_{MPitch} M_{MRoll} M_{AA} \quad (15)$$

$$M_{MT} = M_{APT} M_{MPT} M_{MR} \quad (16)$$

5.3.1.6 Calculating transformations for the relative velocity vector lines

At this point, the calculations of the missile and target transformations are complete. The next step is to calculate transformations for the relative velocity vectors. Figure 5-11 shows the visible aimpoint, missile tracking point and relative velocity vectors. In this figure, the

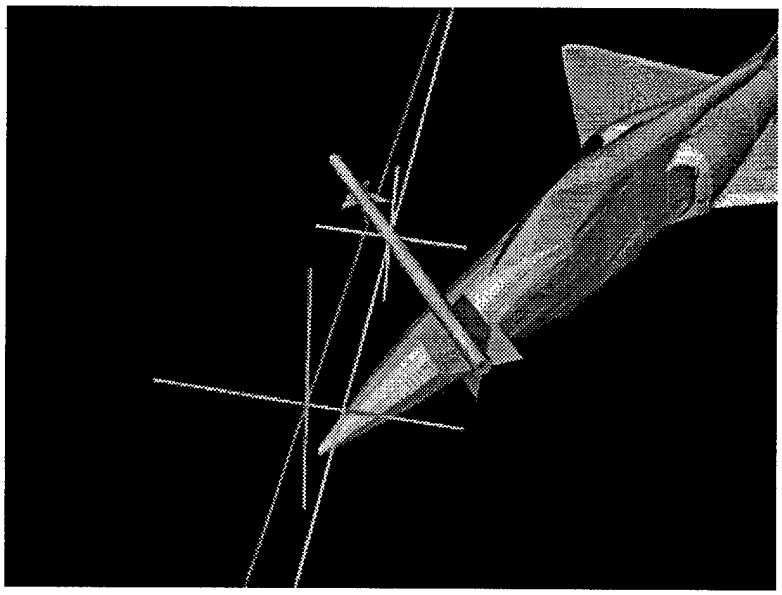


Figure 5-11: Aimpoint on the Target and Tracking Point on the Missile

aimpoint is placed directly above the nose of the target (larger cross-hair), the missile tracking

point is centered on the missile, the missile has a miss point of [0, -25, 75]. The ideal relative velocity represents the flight path of a missile with zero miss value and is therefore always displayed directly at the aimpoint. The missile's actual relative velocity vector is centered at the tracking point on the missile.

The ideal relative velocity vector and aimpoint on the target are placed in the target's scene graph and will follow the target during animation. The relative velocity vector displayed through the tracking point on the missile is placed in the missile's scene graph and moves with it during animation. Calculating the ideal relative trajectory and aimpoint transformation, shown in Equation 17, requires first an inverse of the target's rotation, a rotation according to the relative velocity vector coordinate system, then a translation to the aimpoint. The transformation of the relative velocity vector through the missile's tracking point likewise requires first an inverse of the missile's total rotation, a rotation according to the relative velocity vector coordinate system, then a translation to the tracking point on the missile. This can be seen in Equation 18.

$$M_{RVAPT} = M_{APT} M_{RVCS} M_{TR}^{-1} \tag{17}$$

$$M_{TPRVT} = M_{TPT} M_{RVCS} M_{MR}^{-1} \tag{18}$$

5.3.2 Calculating the fly-out pattern's skewed cone

Section 2.2.4 discusses the fragmentation fly-out skew pattern. Recall that the missile flies along the x-axis of a coordinate system defined by the attack azimuth and elevation. Warhead fragmentation is projected perpendicular to the missile's longitudinal axis. Elapsed over time, the warhead fragmentation pattern produces a cone. If the missile has a yaw or pitch at the point of detonation, a skewing of the cone results due to the fact that the missile's flight path and missile's orientation are not the same. This causes one side of the missile's fragments to travel away from the missile's original position faster than those on the opposite side. The

resulting pattern is a cone skewing towards the flight path of the missile.

To properly display the skewing effect on the fly-out cone, we use a shearing matrix. This section demonstrates how to set up the fragmentation fly-out cone's shear matrix. Table 5-2 shows the important variables.

Table 5-2: Skewed Cone Calculation Variables

Variable Name	Type	Description
M_{MR}	Matrix	Missile's yaw, pitch and roll rotation
Mvel	Float	Missile's velocity
Fvel	Float	Warhead fly-out velocity (perpendicular to the missile)
height	Float	Cone height
radius	Float	Cone base radius
V_{xunit}	Vector	X-axis unit vector: (1,0,0)
$V_{rotUnit}$	Vector	xunit rotated by MR^{-1}
XRot	Degree	Amount of rotation needed to rotate the sheared cone to the x-axis
SA	Degree	Shear angle: angle between the missile's axis and the missile's flight path.
M_{TO}	Matrix	Matrix to translate the cone in its default position to cone tip at the origin.
M_{AM}	Matrix	Matrix to align the cone with the missile.
M_{Shear}	Matrix	Matrix to shear the cone in the positive y direction.
M_{RX}	Matrix	Matrix to rotate the sheared cone around the missile's x-axis to the origin's x-axis.
M_{TW}	Matrix	Matrix to translate the cone to the position of the warhead on the missile.
Time	Float	Simulation Time in milliseconds.

First, the shearing angle must be calculated. This is the angle between the missile's longitudinal axis and its flight path (the x-axis), and it is found using a dot product between the missile's longitudinal axis and the flight path. The missile's longitudinal axis is simply the flight path vector multiplied by the inverse of the missile rotation matrix, M_{MR}. An inverse is used because our goal is to rotate the cone back to the x-axis after the shear. These calculations are shown in Equations 19 and 20.

$$V_{rotUnit} = M_{MR}^{-1} \bullet xunit \qquad (19)$$

$$SA = \arccos\ V_{rotUnit} \bullet xunit \qquad (20)$$

Next we calculate the height and base radius of the cone is calculated. The height of the cone is a user specified value ψ controlled via a thumbwheel widget. This gives the user control over the size of the cone. The height and base radius must have the same ratio as the missile velocity and fragmentation fly-out velocity. Shearing the cone will alter the height of the cone and must be taken into account. Equations 21 and 22 shows the calculation of the cone's height and base radius.

$$height = \psi / \cos SA \qquad (21)$$

$$radius = (height \times Fvel) / Mvel \qquad (22)$$

At this point, we are ready to set up our transformation matrices. The first transformation is a translation to move the default cone (centered at the origin of the missile coordinate system) along the y-axis to position the apex at the origin. Next, the cone is rotated 90 degrees to align with the missile. Figure 5-12:A shows the cone now oriented with its apex at the origin and aligned with the x-axis of the missile's coordinate system. Now a shearing in the positive-y direction can be applied based upon the shearing angle. The shearing matrix is shown in Equation 23 [Hearn97:203,423]. Figure 5-12:B shows the shearing effect for our example.

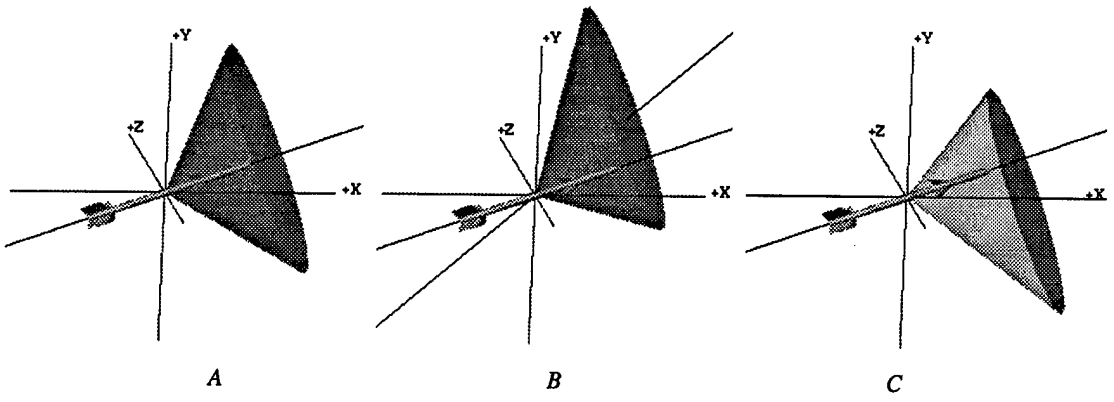


Figure 5-12: Sheering and Orienting the Fragmentation Cone

$$M_{Shear} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \tan SA & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (23)$$

The cone is then rotated around the missile's x-axis to align it with the origin's x-axis. To calculate this rotation, we use the *rotUnit* vector. This vector points along the x-axis of the origin (recall, we used the inverse of the missile's rotation to calculate this vector). We can therefore use this vector's y and z components to find the necessary rotation amount. This rotation amount is calculated in Equation 24, and placed in the rotation matrix in Equation 25. Figure 5-12:C shows the effects of this rotation.

$$M_{XRot} = \arctan\ V_{rotUnit_z} / V_{rotUnit_y} \quad (24)$$

$$M_{RX} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos XRot & -\sin XRot & 0 \\ 0 & \sin XRot & \cos XRot & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (25)$$

The final transformation is a translation to the missile's warhead and is calculated by the matrix *TW*. Equation 26 now shows the combination of these transformations to properly shear the cone and orient it along the missile's flight path.

$$M_{ShearConeTransform} = M_{TW} M_{RX} M_{Shear} M_{AM} M_{TO} \quad (26)$$

5.3.3 Shadows

Target and missile shadows are implemented in the *CShadow* class (see Figure 5-13 for an example). Shadowing is done by manipulating the polygonal coordinate values to match the orientation of the target or missile while setting one of the principle coordinate values to zero. The result is a "flattened" projection of the original model, having the same orientation.

Shadowing is implemented as follows: Four reduced models are loaded, one for the top, side and back shadows and one to control the original polygonal values. The three shadow models are then placed in the scene graph along with their respective grids. Flat shading is applied to the models to improved rendering performance. As the scenario changes, the shadows are set to match the missile and target orientations. This is done by iterating through all the model's coordinates values (found in *SoCoordinate3* nodes). Iteration is done simultaneously for all four models. Each polygon vector coordinate is passed through the respective transformation (missile or target). This vector is then placed in the each of the respective shadow models (top, side or back), zeroing out one of the axis values (e.g., models representing side shadows will have their Y values set to zero).

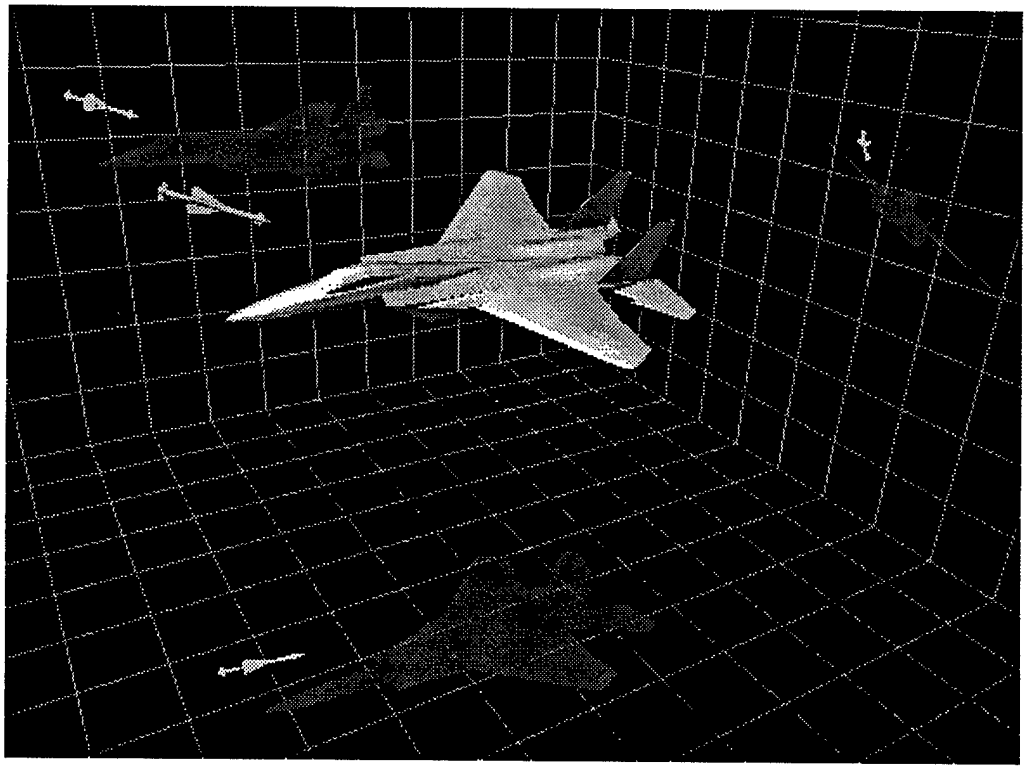


Figure 5-13: Example View using Shadow Projections

5.4 Animation

The current simulation time is controlled by the *CAnimControl* class. When the

simulation time changes, the *CAnimControl* class notifies all *CRenderObjects* of the new time with its *UpdatePosition* function (see Figure 5-14). As *UpdatePosition* iterates through its list of render objects, each render object's virtual *NewSimTime* is called with the current simulation time. Since *CRenderObject*'s constructor adds each newly instantiated render object to the render object list, all classes inheriting from *CRenderObject* will be automatically notified of changes to the current simulation time.

(This line of code is located in *CAnimControls* constructor)

```
...
// Get a pointer to the RenderObject lists
m_renderList = CRenderObject::List();
...

void CAnimControls::UpdatePosition()
  {
    // Notify all the CRenderObjects of the new simulation time
    for(m_listIndex=m_renderList->begin(); m_listIndex!=m_renderList->end();   \
                                        m_listIndex++ )
        (*m_listIndex)->NewSimTime(m_SIM_TIME);
  }
```

NOTE:
The following are defined in the *CRenderObject* header file:

```
typedef vector<CRenderObject*> RenderObjectList;
typedef RenderObjectList::iterator RenderObjectListIterator;
```

The following are member variables of the *CAnimControls* class:

```
RenderObjectList*  m_renderList;
RenderObjectListIterator  m_listIndex;
```

Figure 5-14: Updating the Simulation Time

5.4.1 Target and Missile Animation

The target and missile position during animation is dependent upon the animation mode the user has selected. If the user has selected "BOTH" for animation motion, then both positions are modified (see Equations 27 and 28). Equations 29 and 30 show missile and target position calculations for the "MISSILE" animation mode. In this mode, the missile moves along the relative trajectory while the target remains stationary. Equations 31 and 32 show position

calculations for the "TARGET" mode. Where not specified, variable descriptions can be found in Table 5-1 and Table 5-2.

$$V_{TPos} = V_{TPosAtZero} + (V_{TV} \times Tvel \times Time) \quad (27)$$

$$V_{MPos} = V_{MPosAtZero} + (V_{MV} \times Mvel \times Time) \quad (28)$$

$$V_{TPos} = V_{TPosAtZero} \quad (29)$$

$$V_{MPos} = V_{MPosAtZero} + (V_{RV} \times Rvel \times Time) \quad (30)$$

$$V_{TPos} = V_{TPosAtZero} + (-V_{RV} \times Rvel \times Time) \quad (31)$$

$$V_{MPos} = V_{MPosAtZero} \quad (32)$$

5.4.2 Fragmentation Fly-out Animation

Warhead fragmentation is displayed as an expanding ring. The ring represents the fragmentation's center of mass while the torus represents fragmentation dispersal. In general, fragmentation extends perpendicularly from the missile depending upon warhead type. For AMVS, the sponsor has chosen a dispersal pattern of about 80 to 100 degrees [McCown97]. As stated previously in Section 5.2.5, animation of warhead fragmentation fly-out is for briefing and training purposes and not for scientific analysis.

Fragmentation fly-out is animated by scaling the ring and torus. The fragmentation ring is centered at the origin with a radius of one meter. The warhead fly-out velocity, specified by the user, is first converted to meters per millisecond, then when the missile receives a new simulation time, the ring and torus are scaled as shown in Equation 33.

$$ScaleValue = FVel \times Time \quad (33)$$

5.4.3 Time Adjustment by Focal Distance

As mentioned in Sections 4.3.3 and 5.2.3, the animation control dialog's time adjustment

thumbwheel is configured to adjust the simulation time based upon the viewing distance. When the user is zoomed in on a component, movement of the thumbwheel results in small changes to the simulation time and vice-versa for a zoomed out configuration. My goal is to create a means for controlling the animation movement that positions the missile and target roughly from one screen edge to the other in a single movement of the mouse, regardless of the focal distance. Equation 34 shows the function of thumbwheel movement to simulation time change that achieves this goal.

$$SimTime = SimTime + TM \times (1 - e^{-0.005 \times FD}) \qquad (34)$$

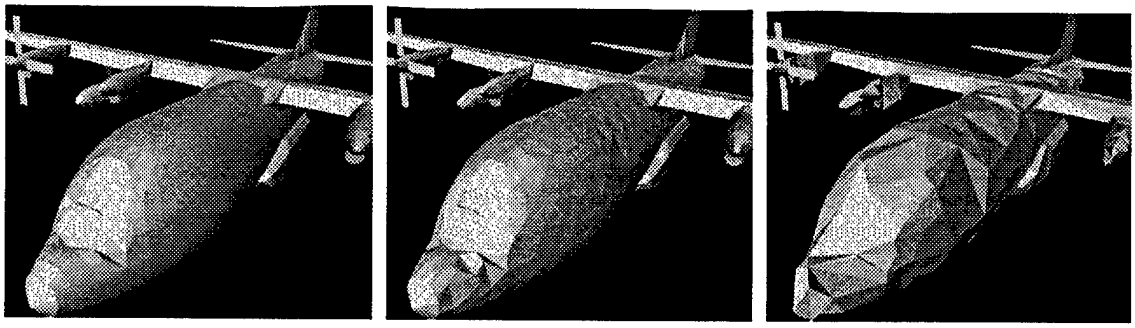
Where : TM = Thumbwheel movement
FD = Focal distance

5.5 Target and Missile Levels of Detail

AMVS' performance is increased by using reduced target and missile models. At times, however, the user may prefer more complete and accurate models. To provide the user control between rendering quality and performance, AMVS provides configurable levels of detail (LOD) for selected target and missile models.

5.5.1 Using Reduced Models Through Decimation and Web Retrieval.

Target and missile models are received from the sponsor in GEORGE format. Many of these models have a large number of polygons (over 20 thousand) which have an adverse effect on AMVS' performance. To alleviate this problem, I include polygon reduction in the conversion process from GEORGE to Inventor format using a decimation algorithm from the *Visualization Tool Kit* [Schroeder96]. To create LOD models, I apply a series of varying levels of decimation to the model. The resulting decimated models are later combined into a single Inventor file (see Section 5.5.2). Figure 5-15 shows a C-130 LOD model created using varying degrees of decimation.



Full Model *55% Reduced* *80% Reduced*

Figure 5-15: C-130 LOD Model

Since AMVS reads Inventor files, other target and missile models can be used to improve performance. Many aircraft models available on the World Wide Web are designed with reduced polygon counts. I use Coryphaeus' Software's Designers Workbench (DWB) to scale, oriented and convert those models downloaded from the Web for use within AMVS.

5.5.2 Creating LOD Target and missile models.

The reduced models described above can each be loaded directly into AMVS. However, these models are more easily managed when combined into a single Inventor file. The *MakeLOD* program takes *.iv* files containing target or missile models and makes a new model having an Inventor *SoSwitch* as the scene graph root. *MakeLOD* takes file description comments found in the *.iv* files and places them in the newly created file. These comments are used by AMVS to label the LOD option menu button found in the *CMissile* and *CTarget* dialogs (see Figure 5-4 and Figure 5-5). LOD models can be made from any combination of GEORGE-derived files, or other Inventor compatible formats mentioned above.

5.5.3 Using LOD Models in AMVS

When a new model is loaded, AMVS checks to determine if it is a GEORGE-derived model, non-GEORGE derived model, or an LOD model containing any combination of these. If AMVS determines that the newly loaded model contains levels of detail, the LOD option button

is updated with the LOD description of each sub-model.

AMVS handles GEORGE derived and non-GEORGE derived target models differently. If a GEORGE derived target model is loaded, the model is searched for key component visibility root nodes (*SoSwitch*'s) and the skin transparency node (*SoMaterial*). These nodes are manipulated when the user modifies skin transparency or component visibility. When non-GEORGE derived models are loaded, these features are disabled. When a user manipulates component visibility or transparency on an LOD target model, AMVS updates all GEORGE-derived models accordingly.

5.6 Multi-threading AMVS

Multi-threading a GUI-based application improves user interface responsiveness, particularly when the applications performs file I/O or computation that can be deferred [Kleiman96:4]. For this reason, I have multi-threaded two of AMVS' features: 1) loading a simulation and 2) setting target damage for LOD models. Threading simulation loading gives the benefit of allowing the user to view the scene as it loads. The user may perceive a reduced response time, but is not kept from viewing components that have already been loaded and has complete control over viewpoint manipulation during this process. Likewise, threading target damage coloring for LOD models improves user responsiveness. When the user requests to see component damage on an LOD model, the current level of detail is colored first, then other levels of detail are colored within the thread. The user is then allowed to view target damage on the immediately visible model while component coloring of other levels of detail is deferred.

5.6.1 Software Quality

In Section 1.4.3, I listed four development intentions. These include creating an application with increased stability, maintainability, extendibility and performance. This section

addresses each of these intentions in relation to AMES.

5.6.1.1 Stability

I evaluate AMVS' stability in the same manner as was done for AMES (as mentioned in Section 4.5.1): use of automated tools and stress testing. Parasoft's Insure++, as mentioned in Section 3.5.2.4, is an effective tool for ensuring the stability of a program. Insure's evaluation of AMES' code revealed only a few warnings, mostly regarding unused or un-initialized variables, which were later fixed.

5.6.1.2 Maintainability

AMVS is more maintainable than AMES due to improved design and documentation, along with cleaner, more efficient code. AMVS has been well documented, using both in-line documents and documentation found in AMVS' continuity directory. Also, AMVS has been implemented in fewer lines of code, resulting from good design and code re-use. Writing AMVS with the same capabilities of AMES was done in one-third the number of lines of code. It is generally accepted that an application written with fewer statements results in cleaner, more maintainable code.

5.6.1.3 Extendibility

Extending AMVS will most likely involve adding new rendering object, new dialogs, or perhaps, even new scenario types. The process to do so has been simple. The ease in which new rendering objects are added to AMVS was outlined in Section 4.6.4. The creation of new dialogs in simplified through code re-use. Classes inheriting from *CDlgDataObject* can take advantage of all methods provided by it. New interfaces can be quickly constructed using the panel and panel object classes AMVS provides.

AMVS has been designed to allow for the implementation of new missile/target

scenarios. One such example was the extension of AMVS to include animation of a VisSim fly-out simulation. All that was required was the creation of a *CVisSimScenario* class inheriting from *CScenario*. Each *CDataObject* was then modified to handle this new scenario type. The only modifications necessary for each object involved small changes to the user-interface, and actions performed in the *SetScenario* and *NewSimTime* functions.

5.6.1.4 Performance

Through-out this document, I have mentioned methods used to improve AMVS' performance. I will summarize techniques for increasing performance here:

1. Using fewer transformation nodes in the scene graph by performing many of the calculations prior to rendering. Also using translation or rotation nodes rather than full transformation matrices where applicable.

2. Providing each *CRenderObject* with all necessary information (through the *CScenario*) needed for calculating each frame. This low coupling reduces function call overhead. In addition, objects calculate and save as many variables as possible before animation begins (such as velocity vectors).

3. Only performing computation when necessary. If an object's visibility is turned off, the object will not perform calculations for position updates during animation.

4. Inlining functions [Meyers92:10].

5. Using a single directional light in the fuze-cone sub-scene graph rather then six point light sources for the entire scene-graph.

6. Flat shading shadows.

7. Using reduced models (described in Section 5.5.1).

Performance test results between AMES and AMVS can be seen in Table 5-3. All tests were performed with a missile containing 284 polygons. These test results show an average performance increase of almost 200%.

Table 5-3: AMES / AMVS Performance Tests

Target polygon count	*AMES frame rate*	*AMVS frame rate*	*Speed up*
21124	2.985	8.95	**200%**
688	26.785	72.25	**170%**
70434	0.830	2.60	**213%**
34433	1.71	5.15	**201%**
47879	1.10	3.15	**186%**

5.7 PC Based Animation of VisSim results

As mentioned in Chapter II, the sponsor is interested in seeing VisSim results animated on the PC. I have elected to explore this area by creating two prototypes to run on both the PC and SGI, one written in C++/OpenGL, the other written in Java/VRML and viewed through a web browser.

5.7.1 OpenGL

Figure 5-16 shows the C++/OpenGL VisSim animation prototype. This program was developed on an SGI workstation and then rehosted on the PC. To make this program completely portable, I did not use any existing libraries for the user interface. The user interface is written entirely in OpenGL. This prototype provides the basic capability to load a VisSim file from the command line, display the full fly-out path, and animate the missile along this path.

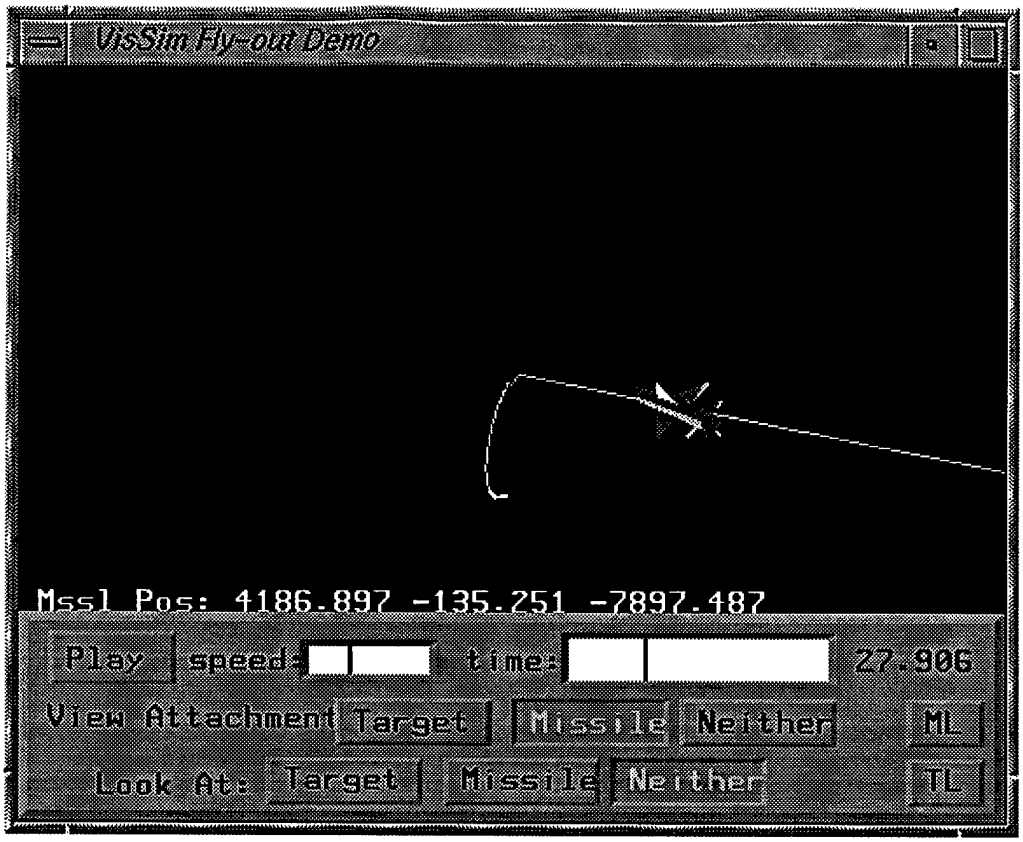


Figure 5-16: C++/OpenGL VisSim Animation Prototype

5.7.2 Java/VRML platform independent version.

Figure 5-17 shows the Java/VRML implementation as viewed in a Netscape browser. This version provides the same capabilities as stated above, with one addition: being able to load a new VisSim file with the Java applet. Also, the animation control more closely matches the one found in AMVS.

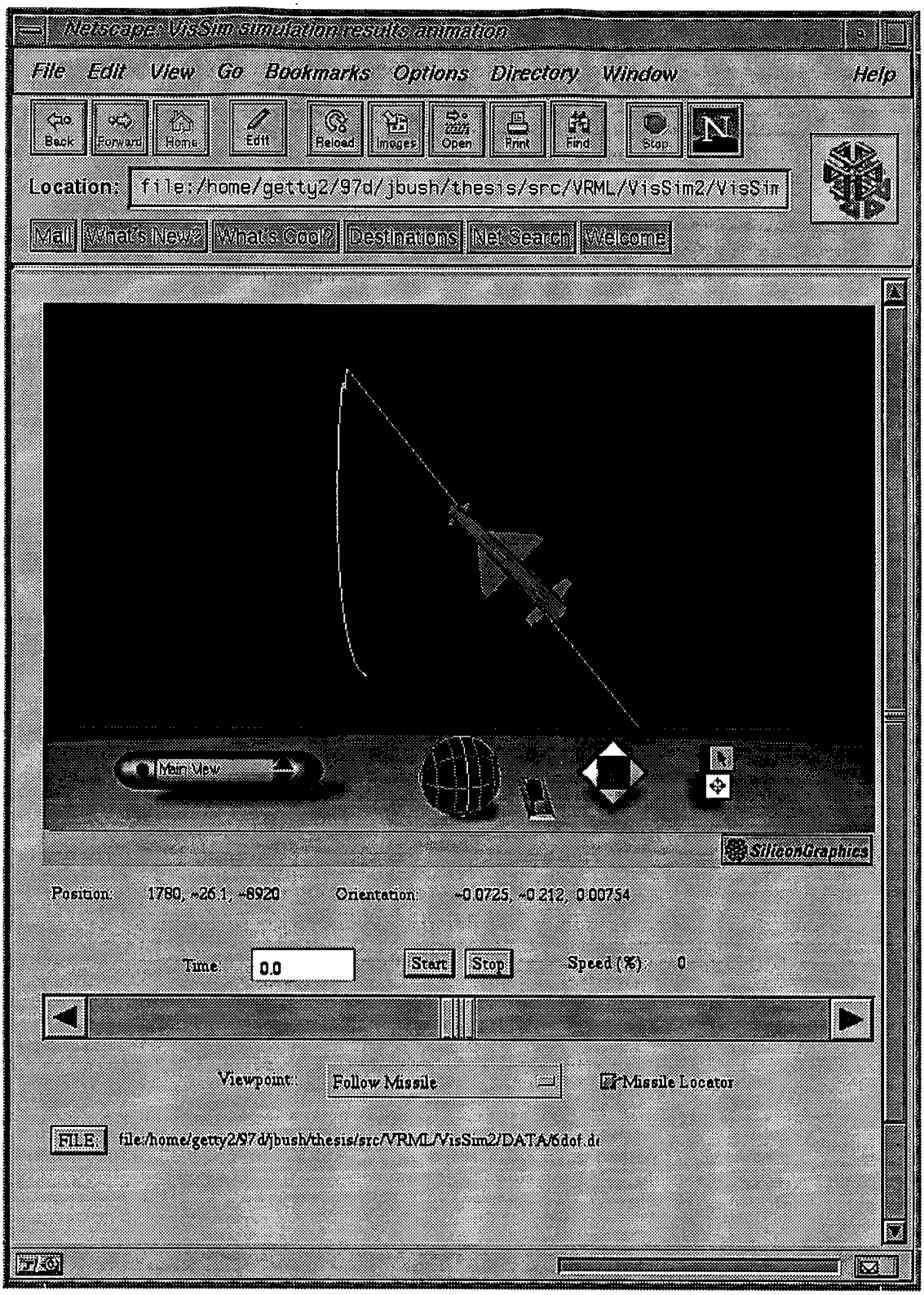


Figure 5-17: Java/VRML VisSim Animation Prototype

5.7.3 Comparison

Both software platforms allowed for object oriented programming. The biggest advantage OpenGL has over Java/VRML is in rendering performance. The Java/VRML version viewed through the web browser rendered at about half the former's frame rate. However, the Java/VRML software platforms has a few advantages over OpenGL. First, both Java and VRML are platform independent; no additional effort was needed to port it to the PC. As a result, software delivery has been greatly simplified. The Java/VRML application can also be posted at a web site, readily available to anyone with Internet access. Finally, since Java includes a user interface API, the GUI was easier to develop. Although the Java/VRML software platform (currently) has lower rendering performance, it has potential in the area of missile fly-out visualization. The next chapter summarizes my contributions to visualization and animation of missile/target encounters.

6. Contributions

6.1 Introduction

This chapter summarizes my research efforts to enhance visualization and animation of endgame and full fly-out missile/target encounters. I begin by presenting major and minor contributions to the field followed by major and minor enhancements to the previous research done by Lt. Joseph Moritz. Next I make suggestions for future work in this field and then conclude with a summary of this research.

6.2 Contributions

Previous work at AFIT in the field of visualizing endgame scenarios was accomplished by Lt. Joseph Moritz [Moritz96]. His research resulted in the creation of the AFIT Missile Endgame Simulator (AMES). My research extends Moritz's work. In particular, this research focuses upon visualizing the fragmentation fly-out skewing, improving endgame animation and extending the animation to include the complete endgame encounter from target detection to fragmentation impact. This research also investigates animation of a full fly-out simulation. Furthermore, it explores new techniques for visualizing the spatial relationships present in a missile/target encounter. These new capabilities and concepts in visualizing and animation missile/target encounters are implemented in the AFIT Missile Visualization System (AMVS). While AMVS development necessarily involved reusing ideas found in AMES, the sections to follow represent only those original contributions embodied in AMVS.

6.2.1.1 Visualizing the Fragmentation Fly-out Skewed Cone

The fragmentation fly-out cone skewing phenomenon, as discussed in Section 2.2.4, effects the performance of air-intercept missiles. Previously, engineers had no graphical tool for dynamically visualizing this phenomenon. AMVS displays the skewed cone both as a two-dimensional cross section and as full, transparent cone. Visualization of the skewed cone assists engineers in achieving a better understanding fly-out cone skewing and thereby discover solutions to combat its effects. It is also useful in briefing decision makers on this problem.

6.2.1.2 Full Animation of an Endgame Scenario

Previously, there did not exists a three-dimensional graphical tool for animating the full endgame encounter to include visualizing fuze-cone target detection, warhead fragmentation fly-out, and fragmentation/target impacts. AMVS animates the full endgame encounter. Fuze cones are animated by changing their transparency or visibility during target detection (see Section 5.2.6). Warhead fly-out is displayed as an expanding ring or torus (see Sections 5.2.5 and 5.4.2). Fragmentation/target impacts are displayed using SHAZAM output and correctly correlated with the impact time.

Such an animation tool not only helps the engineers understand encounter timing issues and the missile/target interaction during an endgame, it also provides an improved method for communicating endgame concepts to people not familiar with them.

6.2.1.3 Visualization and Animation of Full Fly-out Simulation Results

Engineers previously had no three-dimensional tool for visualizing and animating VisSim full fly-out simulation results. The lack of such a tool makes assimilation of these results difficult. Engineers needed to be able to understand the missiles flight path in relation to the target, as well as its orientation through out the flight. AMVS reads, displays, and animates the

missile's flight path using a VisSim output file.

6.2.1.4 Fixed Fuze-cone Attribute Experimentation

AMES was implemented to allow the engineer to visualize fixed fuze cone sensor pattern coverage based upon antenna azimuth and range attributes loaded in from a FUZE file. However, engineers needed an environment not only to visualize this sensor coverage, but to experiment with fuze cone antenna azimuth and range settings within a single graphical application. AMVS provides such an environment. Engineers can load a fuze file, modify the fuze attributes, examine the results interactively, and save their work.

6.2.1.5 Endgame Scenario Creation

As stated in Section 2.4.4, engineers need an efficient means of creating an endgame scenario. Although AMES was implemented to load an ENCOUNT file, no effort was made to allow for saving or creating new ones. As a result, engineers were still left editing these files via text editors. The task of creating an endgame scenario is much more efficient in AMVS due to its GUI interface for parameter entry, including error checking, and a graphical display to show the results of the settings. Furthermore, these modifications can then be written out to a user specified ENCOUNT file for archival or reuse.

6.2.2 Minor Contributions

6.2.2.1 Modifiable Levels of Detail

To improve performance while still providing high fidelity models, Moritz suggested the implementation of user modifiable levels of detail (LOD) with future versions of AMES [Mortiz96:6-3]. AMVS allows the user to select between varying levels of detail for the missile and target. Thus, the user may choose between efficiency and accuracy depending upon the task at hand. During some tasks, such as examining target/fuze-cone interaction, lower fidelity in

target and missile models may suffice. When using AMVS as a briefing tool, however, improved rendering may be more desirable.

6.2.2.2 Saving a Simulation

AMVS enables the user to work efficiently by allowing the engineer to save her work and return to it later. AMVS not only saves such information as currently selected missile and target models and as the currently loaded ENCOUNT or OPEC file, but also such states as the fuze cone attribute settings, the cone, grid and shadow visibility settings, the current rendering view, and the five "saved" rendering views.

6.2.2.3 Visualized Missile and Target Velocity Vectors

AMVS displays the missile and target's actual velocity vector lines in addition displaying relative velocities. Displaying the missile's actual velocity vector is critical when examining fragmentation skewing, since the fragmentation fly-out cone is centered along the missile's flight path and not the missile's longitudinal axis.

6.2.2.4 Target Component Group Visibility

Each target consists of several hundred components. To reduce scene complexity, IVAVIEW provides the user the ability to set component group visibility. AMVS likewise provides this capability.

6.2.2.5 View Control

To further enhance understanding of the scenario, engineers are given the current view position and orientation feedback relative to the target. Also, the engineer is able to save and restore key viewpoints.

6.2.2.6 Rendering Scene Image Capturing

To support training and briefing, AMVS captures its rendering scene and save it to a user specified image file. These image files can then be incorporated into briefing slides or training manuals.

6.2.3 Major Enhancements to AMES' Capabilities

6.2.3.1 Improved Rendering Performance

As mentioned in Section 2.4.3.2, AMES suffers from poor rendering performance, resulting in a degraded animation and lower fidelity. Section 5.6.1.4 summarizes the efforts to improve rendering performance in AMVS' implementation. Performance test results show almost a 200% increase in rendering performance.

6.2.3.2 Improved Animation Control User-Interface

AMES user interface for animation limited the user. AMVS' animation control dialog is significantly improved as a result of prototyping and user testing. It now provides valuable feedback during the animation and complete control of the simulation time. The feedback given to the engineer includes the current simulation time and the current target and missile positions displayed in target, missile, or world coordinate systems. Additionally, positions are displayed in inches or meters. The engineer also has three separate means of controlling the animation time including text field direct entry, dial control, and animation start/stop with speed control. Finally, the user has complete control over the simulation time without the need to focus on the animation control dialog.

6.2.4 Minor Enhancements over AMES' Implementation

6.2.4.1 Corrected Endgame Scenario Calculations

AMES makes a few calculation errors when displaying an endgame based upon parameters found in ENCOUNT and OPEC files. Incorrect calculations include miss-placed visible target coordinate system, arbitrary point on the missile, and visual display of velocity vectors as well as miss-calculated center of rotation of the missile and direction of flight for missile and target during animation. AMVS corrects these problems.

6.2.4.2 Ability to View OPEC Target Damage

Although AMES implemented color coding target internal components based upon the component damage information found in an OPEC *.out* file, component damage was not easily visible due to occlusion by the aircraft skin. AMVS fixes this problem by applying user configurable transparency to the aircraft skin, allowing the internal components to be visible.

6.2.4.3 Inter-object Visualization Techniques

In order to provide insight into inter-object spatial relationships, AMES used three additional windows showing the top, side and front views of the encounter in addition to the main rendering window. Section 4.2.1 discusses problems with this approach and presents an improved visualization technique using shadow projections. AMVS implements shadow projections of the missile and target in order to improve visualization of inter-object spatial relations between them.

AMVS also improves visualization of the target/fuze cone relationship. Although using transparent cones to visualize fixed fuze cone sensor coverage patterns effectively shows target/cone relationships when the target is penetrating the cone, it is ineffective in revealing the relationship when the target is positioned before or behind the cone. A better visualization

technique in this case is to use cone "cross sections." AMVS optionally displays the fuze cones as two-dimensional cross sections patterned after text book and manual drawings. Rendering the cone as a cross-section allows the user to better determine how far the target is from the cone at a specific point in time.

6.2.4.4 Alternative Warhead Fragmentation Visualization

AMES displays warhead fragmentation trajectories as lines emanating from the warhead origin point [Moritz96:4-17]. The sponsors expressed disinterest in this approach. Therefore, AMVS implements a preferred visualization technique involving an expanding ring or torus representing a mass of fragmentation emanating from the missile at a specific point in time. The size of the ring is modified accordingly during animation.

6.2.4.5 Improved Target Damage Coloring Scheme

Component damage produced by OPEC is represented by a scalar value ranging from 0.0 to 1.0, with 1.0 representing complete damage. AMES visualizes this target component damage using a coloring index scheme in 1/10 increments with one color arbitrarily assigned to each of the ten increments, thereby loosing some information. AMVS uses a coloring technique which does not loose information and more logically conveys damage amounts (see Section 4.2.5).

6.3 Recommendations for Future Work

AMVS significantly improves visualization of missile/target encounters. Graphically displaying the results of computer based simulation systems makes these systems more valuable to the engineers. However, there is one drawback. The engineer must run the endgame on one system, and view the results on another. For the case of viewing OPEC and VisSim results, this requires transferring simulation results from a PC to the SGI before viewing them in AMVS. A single application combining the modeling and simulation capability of OPEC, SHAZAM and

VisSim with the visualization and animation capability found in AMVS would be a valuable step in improving the engineer's efficiency and realizing the objective of providing a fully capable visual environment for missile fuze engineering.

6.4 Conclusion

My research results in the discovery and implementation of improved techniques for visualizing and animating missile/target encounters. For the first time, engineers are provided an interactive three-dimensional graphical display of the fragmentation fly-out skewing phenomenon. Visualizing this phenomenon allows engineers to understand its negative effects against air-intercept missile performance, leading them further towards the development of fuzing and warhead components that overcome this problem. Furthermore, engineers are provided a full animation of the endgame, from target detection to fragmentation/target interception. In addition, the engineer is given complete temporal and spatial control over the animation through an improved animation control interface and viewpoint feedback and control. Visualization techniques such as shadow projections and cone cross-sections enhance the animation by providing the engineer more information about complex spatial relationships during the endgame. This results in an environment which allows the engineer the ability to enter into, control, and freely witness an endgame in ways previously not possible. Furthermore, engineers are provided a simple to use graphical application for creating endgame scenarios and experimenting with fixed fuze-cone attributes. Finally, during my thesis efforts, I have extended my research to include three-dimensional visualization and animation of full fly-out simulations, where previously only static two-dimensional display of these results where available to the engineer. Displaying and animating the fly-out simulation results allows the engineer complete comprehension of simulation results, thereby increasing the value of the system producing them, and enhancing the engineers efforts in developing air-intercept missiles.

Bibliography

[Adams88] Adams, Lee. *High-Perforamance Graphics in C.* Summit PA: Windcrest Books. 1988.

[Ames97] Ames, Andrea L. et al. *VRML 2.0 Sourcebook.* New York: John Wiley & Sons, Inc. 1997.

[Arnold96] Arnold, Ken and Gosling, James. *The Java Programming Language.* Menlo Park, CA: Addison Welsey Publishing Company. 1996.

[Booch91] Booch, Grady. *Object Oriented Design with Application.* Benjamin/Cummings Publishing Company Inc. 1991.

[Card86] Card, Stuart K. and Moran, Thomas P. "User Technology: From Pointing to Pondering," *Association for Computer Machinery.* 1986.

[Coad93] Coad, Peter and Nicola, Jill. *Object Oriented Programming.* Yourdon Press. 1993.

[Coffield86] Coffield, Patrick C. et al. *User Manual for the Air-to-Air Effectiveness Program SHAZAM(U).* Eglin AFB, FL, Defense Technical Information Center, ADB-104959L. 1986.

[Cooper84] Cooper, C.N. and Shepard, R. N. "Turning something over in the mind". *Scientific America,* 251(6):106-114. 1984.

[Cramer85] Cramer, Russel E. and Hilbrand, Roy. *FASTGEN 3 Target Description Computer Program.* Washington D.C., Defense Technical Information Center, AD-B103-850. 1985.

[Cunard] Cunard, Donald A. *Programmable Integrated Ordnance Suite (PIOS).* WL/MNMF, Eglin AFB FL. Briefing slides. Unpublished.

[Cunard97] Cunard, Donald A. Proximity Fuze Team Leader, USAF Wright Laboratory. Armament Directorate. Fuzes Branch (WL/MNMF). Personal Interview. 19 June 1997.

[Ege92] Ege, Raimund K. *Programming in an Object Oriented Environment.* San Diego, CA: Academic Press Inc. 1992.

[Foley92] Folye, James D. et al. *Computer Graphics.* Menlo Park, CA: Addison-Wesley Publishing Company. 1992

[Fowler97] Fowler, Martin. *Analysis Patterns, Reusable Object Models.* Menlo Park, CA: Addison-Wesly. 1997.

[Gamma95] Gamma, Erich. et al. *Designing Patterns: Elements of Reusable Object-*

Oriented Systems. Menlo Park, CA: Addison-Wesly. 1995.

[Goldman91] Goldman, Ronald N. "More Matrices and Transformations: Shear and Pseudo-Perspective". *Graphics Gems II*. San Diego, CA: Academic Press Inc. 1991.

[Gottisdiner95] Gottisdiener, Ellen. "RAD Realities: Beyond the Hype To How RAD Really Works". *Application Development Trends*. Vol 2, No 8. August 1995. pp. 28-38.

[Grotch83] Grotch, Stanley L. "Three-dimensional and stereoscopic graphics for scientific data display and analysis". *IEEE Computer Graphics and Applications*. pp. 31-43. November 1983.

[Holub95] Holub, Allen I. *Enough Rope to Shoot Yourself in the Foot*. New York: McGraw-Hill. 1995.

[IRIS95] *IRIS Performer Programmer's Guide. Silicon Graphics, Inc. 1995.*

[Kleiman96] Kleiman, Steve. et al. *Programming with Threads*. New Jersy: Prentice Hall. 1996.

[Kobza74] Kobza, Norm. et al. *Air-to-Air Missile Fuze Sensor*. Eglin AFB, FL. Defense Technical Information Center, AFATL-TR-74-197. 1974.

[Heller92] Heller, Dan. *Motif Programming Manual*. O'Reilly & Associates, Inc. 1992.

[Hearn97] Hearn, Donald and Baker, M. Pauline. *Computer Graphics, C Version*. New Jersey: Prentice Hall. 1997.

[Herndon92] Herndon, Kenneth P. et al. "Interactive Shadows". Monterey, CA, *ACM UIST '92*. 1992.

[PMC] PMC Inc. *Ordnance Package Evaluation Code User's Manual*. Socorro, NM. Unpublished.

[Mack87] Mack, Rodney E. *A Time-To-Go Algorithm for Optimal Two-Dimensional Target Intercept*. Masters Thesis, University of Texas at Austin. December 1987.

[Marcus93] Marcus, Aaron; Letz, Grant; and Heidrich, Wolfgang. "Graphic Design for User Interfaces," *ACM SIGGRAPH Course Notes 24*, 1993.

[McCardle97] McCardle, Kevin. Branch Chief, Weapons Effects Engineer. USAF Wright Laboratory. Development Planning Directorate, Weapons Effects Division (ASC/XRWA). Personal Interview. 18 June 1997.

[McGown] McGown, Douglas. *Aerial Target Lethality Analysis Using SHAZAM*. ASC/XRWA, Eglin AFB FL. Briefing slides. Unpublished.

[McGown87] McGown, Douglas. *A Computer Program for the Graphical Illustration of a Missile and Target Encounter (GIMATE) User Manual.* Defense Technical Information Center, ADB-110-553L. 1987.

[McGown97] McGown, Douglas. Weapons Effects Engineer, USAF Wright Laboratory. Development Planning Directorate, Weapons Effects Division (ASC/XRWA). Personal Interview. 19 June 1997.

[Moritz96] Mortiz, Joseph E. *Grahpical Display of a Missile Endgame Scenario.* Masters Thesis, AFIT/GCS/ENG/96D-20, Air Force Institute of Technology, Wright-Patterson AFB, OH. 1996.

[Nielson93] Neilson, Jakob. *Usability Engineering.* London: Academic Press Inc. 1993.

[Nye92] Nye, Adrian and O'Reilly, Tim. *X Toolkit Intrinsics Programming Manual.* Sebastopol CA: O'Reilly & Associates, Inc. 1992.

[O'Brien96] O'Brien, Larry. "The RAD Stuff". *Software Development.* Vol 4. No 4. April 1996. pp. 27-33.

[OIAG94] Open Inventor Architecture Group, *Open Inventor C++ Reference Manual.* Menlo Park, CA: Addison Wesley Publishing Company. 1994.

[ParaSoft96] Unknown, *Insure++ User's Guide, Version 3.0.1*, ParaSoft Corporation. 1996.

[Pohl97] Pohl, Ira, *Object Oriented Programming using C++*, Menlo Park, CA: Addison-Wesley. 1997.

[Rogers90] Rogers, David F. and Adams, J. Alan. *Mathematical Elements for Computer Graphics, 2nd Edition.* New York: McGraw-Hill Publishing Company. 1990.

[Rumbaugh91] Rumbaugh, James. et al. *Object Oriented Modeling and Design.* New Jersey, Prentice-Hall Inc. 1991.

[Schroeder92] Schroeder, William J. et al. "Decimation of Triangle Meshes". SIGGRAPH '92 Conference Proceedings. pp. 65-68. New York: ACM Press. 1992.

[Schroeder96] Schroeder, William J. et al. *The Visualization Toolkit.* New Jersey: Prentice Hall PTR. 1996.

[Sessions92] Sessions, Roger, *Class Construction in C and C++: Object-Oriented Programming Fundamentals.*, Prentice Hall. 1992.

[Shirley93] Shirly, William K. *A Guide to FASTGEN Target Geometric Modeling.* Fort Walton Beach: Defense Technical Information Center, ADA-273-171, 1993.

[Starfield90] Starfield, Anthony M. et al. *How to Model It, Problem Solving for the Computer Age*. New York: McGraw-Hill, Inc. 1990.

[Stroustrup91] Stroustrup, Bjarne. *The C++ Programming Language, Second Edition.* Menlo Park, CA: Addison-Wesley. 1991.

[SURVICE92] The SURVICE Engineering Company. *FASTGEN Ivaview User's Manual.* Aberdeen, Maryland, Unpublished. 1992.

[Tufte87] Tufte, Edwared R. *The Visual Display of Quantitative Information.* Cheshire CT: Graphics Press. 1987.

[Tufte90] Tufte, Edward R. *Envisioning Information* Cheshire CT: Graphics Press. 1990.

[Tufte97] Tufte, Edward R. *Visual Explanations*, Cheshire CT: Graphics Press. 1997.

[Wanger92] Wanger, Leonard R. et al. "Perceiving spatial relationships in computer generated images". *IEEE Computer Graphics and Applications.* May 1992.

[Watt93] Watt, Alan. *3D Computer Graphics.* Menlo Park, CA: Addison Wesley. 1993.

[Wernecke94] Wernecke, Josie. *The Inventor Mentor.* Menlo Park, CA: Addison Wesley Publishing Company. 1994.

[Woo97] Woo, Mason. Neider, Jackie. Davis, Tom. *OpenGL Programming Guide, Second Edition.* Menlo Park, CA: Addison Wesley. 1997.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
Visualization and Animation of a Missile/Target Encounter

5. FUNDING NUMBERS

6. AUTHOR(S)
Jeffrey T. Bush, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Thechnology
2750 P Street
WPAFB, OH 45433-7126

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/97D-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
WL/MNMF
Don Cunard
101 West Eglin Blvd, Suite 219
Eglin AFB, FL 32542-6810

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Existing missile/target encounter modeling and simulation systems focus on improving probability of kill models. Little research has been done to visualize these encounters. These systems can be made more useful to the engineers by incorporating current computer graphics technology for visualizing and animating the encounter. Our research has been to develop a graphical simulation package for visualizing both endgame and full fly-out encounters. Endgame visualization includes showing the interaction of a missile, its fuze cone proximity sensors, and its target during the final fraction of a second of the missile/target encounter. Additionally, this system displays dynamic effects such as the warhead fragmentation pattern and the specific skewing of the fragment scattering due to missile yaw at the point of detonation. Fly-out visualization, on the other hand, involves full animation of a missile from launch to target. Animating the results of VisSim fly-out simulations provides the engineer a more efficient means of analyzing his data. This research also involves investigating fly-out animation via the World Wide Web.

14. SUBJECT TERMS
Computer graphics, modeling and simulation, missile endgame simulation, data visualization

15. NUMBER OF PAGES
114

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL