

AD-A274 193



1

DTIC
ELECTE
DEC 23 1993
S A

**Synthetic BattleBridge:
Information Visualization and User Interface
Design Applications in a Large Virtual
Reality Environment**

THESIS

**Kirk G. Wilson, Capt, USAF
AFIT/GCS/ENG-93D-26**

This document has been approved
for public release and sale; its
distribution is unlimited.

93 12 22 1 17

93-31004


95P95

AFIT/GCS/ENG-93D-26

**Synthetic BattleBridge:
Information Visualization and User Interface
Design Applications in a Large Virtual Reality
Environment**

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 Systems**

**Kirk G. Wilson
Captain, USAF
December 1993**

Preface

I would like to thank Capt Brian Soltz for teaching me the incremental software development method and for making up half of the best team at AFIT (at least in the ability to work late hours). Thanks also to the other members of the simulation cadre: Major Mike Gardner, Capt Matt Erichsen, Capt Bill Gerhard, Capt Andrea Kunz, Capt Mark Snyder, and Mr Steve Sheasby. And thanks to Capt Alain Jones for the photography assistance.

I would also like to thank Lt Col Martin Stytz for his patience and guidance during a sometimes tumultuous seven months of development and coding. Thanks to Lt Col Phil Amburn and Lt Col Patricia Lawlis, the two other members of my thesis committee.

To my wife Karen and my four sons Daniel, Thomas, Matthew, and Ryan:
I missed you greatly and can't wait to see you again.

Kirk Wilson
27 Nov 93

Accession For	
NTIS	CRA&I ☒
DTIC	TAB ☐
Unannounced ☐	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

DTIC QUALITY INSPECTED 3

Table of Contents

Preface	ii
Table of Contents.....	iii
List of Figures	vii
Abstract.....	viii
1. Introduction	1
1.1 Background.....	1
1.2 Problem	1
1.3 Assumptions.....	2
1.4 Scope.....	2
1.5 Research Conduct.....	3
1.5.1 Virtual Reality Background.....	4
1.5.2 Tomorrow's Reality Gallery (SIGGRAPH '93).....	4
1.5.3 Enhancements to Synthetic BattleBridge.....	5
1.5.3.1 Navigation.....	5
1.5.3.2 Information.....	6
1.5.3.3 Interface Development.....	6
1.5.4 Thesis Preparation & Defense	6
1.6 Materials and Equipment	7
1.6.1 Hardware.....	7
1.6.2 Software.....	7
1.7 Complementary Efforts	7
1.7.1 Fuzzy Logic Sentinel.....	7
1.7.2 ObjectSim	8
1.7.3 Object Manager	8
1.7.4 Virtual Cockpit	8
1.7.5 Red Flag.....	8
1.7.6 Satellite Modeler	9
1.8 Schedule.....	9
2. Background	10
2.1 Introduction	10
2.2 Future.....	11
2.2.1 A Vision for the Future.....	11

2.2.2	No Interface to Design.....	12
2.3	Status.....	13
2.3.1	A Tutorial.....	13
2.3.2	Object Data Manipulation.....	14
2.3.3	Terrain Data Manipulation.....	14
2.4	Conclusion.....	15
3.	Design.....	17
3.1	SBB User Interface Design and Functionality.....	17
3.1.1	Description.....	18
3.1.2	Information Grid Design.....	21
3.1.3	FakeSpace BOOM Interface.....	22
3.1.4	Head Mounted Display Interface.....	23
3.2	Synthetic BattleBridge Simulation Design.....	24
3.2.1	ObjectSim.....	25
3.2.1.1	Simulation.....	26
3.2.1.2	View.....	27
3.2.1.3	Attachable Player.....	27
3.2.1.4	Terrain.....	28
3.2.1.5	Renderer.....	28
3.2.2	Object Manager.....	29
3.3	SBB Top Level Design.....	30
3.3.1	SBB Simulation Objects.....	32
3.3.1.1	SBB Application.....	33
3.3.1.2	SBB View.....	33
3.3.1.3	Shadow.....	34
3.3.1.4	Stealth.....	35
3.3.1.4.1	Stealth View Types.....	36
3.3.1.4.2	Stealth View Controls.....	37
3.3.1.5	SBB Net Manager.....	37
3.3.1.6	SBB Net Player.....	38
3.3.1.7	Locator.....	39
3.3.1.8	Trail.....	40
3.3.1.9	Model Manager.....	41
3.3.1.10	Clock.....	41
3.4	Interface and Other Object Design.....	42

3.4.1	Player Button.....	42
3.4.2	SBB Button	43
3.4.2.1	Rect Button	45
3.4.2.2	Push Button.....	45
3.4.2.3	Light Button	45
3.4.2.4	Hideable Button	45
3.4.2.5	Display Button	46
3.4.2.6	Menu Button.....	46
3.4.3	Drawstring.....	46
3.4.4	Mac Sounds	47
3.5	Conclusion.....	47
4.	Implementation	48
4.1	Overview	48
4.2	Object Implementation	49
4.3	SBB Top Level Interaction.....	49
4.3.1	SBB Simulation Objects.....	50
4.3.1.1	SBB Application (sbb_app.cc).....	51
4.3.1.2	SBB View (color_view.cc).....	52
4.3.1.3	Shadow (shadow.cc).....	53
4.3.1.4	Stealth (stealth.cc)	53
4.3.1.4.1	Stealth draw.....	53
4.3.1.4.2	Stealth propagate	54
4.3.1.5	SBB Net Manager (sbb_net_mgr.cc)	55
4.3.1.6	SBB Net Player (sbb_net_player.h).....	57
4.3.1.7	Locator (sbb_net_player.h).....	57
4.3.1.8	Trail (sbb_trail.cc).....	58
4.3.1.9	Model Manager (model_mgr.cc).....	58
4.3.1.10	Clock (clock.cc)	61
4.4	Interface and Other Object Design.....	61
4.4.1	Player Button (play_button.cc).....	61
4.4.2	SBB Button (button.cc).....	62
4.4.2.1	Rect Button : SBB Button.....	65
4.4.2.2	Push Button : Rect Button.....	65
4.4.2.3	Light Button : Push Button.....	66
4.4.2.4	Hideable Button : Push Button.....	66

4.4.2.5	Display Button : Light Button.....	66
4.4.2.6	Menu Button : Push Button.....	67
4.4.3	Drawstring	67
4.5	Problems and Praise	68
4.6	Conclusions	68
5.	Results, Future Work, and Conclusions.....	69
5.1	Results.....	69
5.1.1	Performance.....	69
5.1.2	User Interface	69
5.1.2.1	Transparent Dynamic Display Buttons.....	70
5.1.2.2	Views and State.....	71
5.1.2.3	Players.....	73
5.1.2.4	Navigation (Position and Orientation).....	74
5.1.2.5	Interface Manipulation	75
5.2	Future Work.....	75
5.2.1	Voice Interaction.....	76
5.2.2	In Scene Buttons.....	76
5.2.3	3D Sound.....	77
5.2.4	Selection of Objects in the Scene.....	77
5.2.5	Efficient Algorithms for Object Processing.....	78
5.2.5.1	Too Many Objects/Time Slice Processing.....	78
5.2.5.2	Culling/Drawing Bottleneck/Level of Detail Management.....	79
5.2.5.3	Unseen Objects Require Processing Too/Range Processing	80
5.2.6	Abstract Visualization of Force Distributions.....	80
5.2.6.1	Grid Architecture.....	81
5.2.6.2	Terrain Partitioning.....	81
5.2.6.3	Forces Partitioning.....	81
5.2.6.4	3D Chessboard.....	82
5.3	Conclusions	82
References.....		83
Vita		85

List of Figures

Figure 2-1 Fully Immersive Virtual Reality Suit (3).....	12
Figure 3-1 The SBB User Interface.....	18
Figure 3-2 Active Translation Button.....	19
Figure 3-3 Active Heading Button.....	20
Figure 3-4 The Information Grid.....	22
Figure 3-5 FakeSpace BOOM.....	23
Figure 3-6 Polhemus LookingGlass Head-Mounted Display	24
Figure 3-7 The ObjectSim Object Model.....	26
Figure 3-8 The Synthetic BattleBridge Task Object Model/Design.....	31
Figure 3-9 The Shadow Object Offset	35
Figure 3-10 The Object Manager/SBB Net Manager Design	38
Figure 3-11 Air Vehicle Locator	39
Figure 3-12 Ground Vehicle Locator.....	40
Figure 4-1 A Model Manager Example.....	60
Figure 4-2 A Model Manager Data File	60
Figure 5-1 The Transparent User Interface.....	70
Figure 5-2 The View & Player Buttons	72
Figure 5-3 The Information Grid.....	73
Figure 5-4 The Navigation Controls.....	75

Abstract

With shrinking budgets and fewer personnel, future military training will rely heavily on simulated environments. The goal for this training is to reduce cost while maintaining readiness and unparalleled capability for all levels of military command.

This thesis effort, the Synthetic BattleBridge (SBB), provides a real-time simulated environment for military commanders to observe on-going computer simulations of varying participation levels and helps them wring the most from a simulation. The SBB, designed for higher ranking personnel with little time to spend learning how to run the system, must exhibit three capabilities: ease-of-use, long-term retention, and adaptability.

Based on the ObjectSim framework and the Object Manager network management software, the SBB includes a unique transparent interface with dynamic display elements that ensures minimal intrusion. The SBB provides multiple views and direct attachment to simulation players, along with an information grid showing the distribution of forces within the current environment.

Synthetic BattleBridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment

1. Introduction

1.1 Background

With shrinking budgets and fewer personnel, future military training will rely heavily on simulated environments. The goal for this training is to reduce cost while maintaining readiness and capability unparalleled in the world. To make this happen, researchers must solve several problems, some hardware-related, some software, often both.

My research furthers the software knowledge base by providing a user interface management system (UIMS) for managing objects and information in a synthetic environment. I developed this system as an enhancement to the Synthetic BattleBridge (SBB) system, a real-time simulated environment for military commanders to observe on-going computer simulations of varying participation levels.

1.2 Problem

Specifically, I address navigation and information displays in a large, distributed simulation environment. Using current hardware and software technology, I developed navigation controls and information displays that exist to help the user wring the most from the system. Since the SBB is designed for higher ranking personnel with little time to spend learning how to run the system it must exhibit three capabilities: ease-of-use, long-term retention, and adaptability.

The first two capabilities come from a simple and straightforward interface to a complex system. A clear, structured integration system, capable of adapting to a changing environment and the needs of the user, provides the third.

1.3 Assumptions

I used the previous SBB incarnation (9) as the basis for the capabilities resulting from my research and implementation, with a few exceptions listed later. The focus of this effort, enhancement of the SBB user interface and increased frame-rate performance, required working within a new design architecture called ObjectSim (also introduced later). This did not limit my research, except in the time required to understand and modify the ObjectSim framework to accommodate the desired SBB capabilities.

1.4 Scope

The SBB must be easy to use and able to "go anywhere" the user desires, within the confines of the simulation. For ease-of-use, I attempted to blend the interface into a natural, reality emulating, "non-interface" discussed extensively in (6), refining the interface along a path that eventually (in some future research) leads to its disappearance. I focused on navigation and information displays within the system to provide an easy-to-use situational awareness tool.

For navigation, the user selects the level of observation and then the location and orientation of the view. Attachment to individual players occurs using multiple scenarios. The user may also choose a mini-view inset within the larger view (this will depend heavily on the resolution of the chosen display device). Movement control enhancements include

integration of the translation device into the interface, controlling the six directions of motion (in/out, up/down, left/right).

I also enhanced the SBB to provide a variety of information displays, each selectable and configurable by the user:

- orientation gauges showing the current heading, pitch, and roll
- grid/maps showing location of objects and views within the environment (possibly extending to terrain density/roughness, if time permits)

In conjunction with other research activities, I provided the ability to "jump" the user to particular views (saving the state of the present view), based on:

- "fuzzy-logic" feedback on hot-spots within the environment (in conjunction with other research)
- feedback on the history of the simulation, including trends or patterns to monitor, or viewing older data compared to new
- the state of the view consists of the location and orientation, whether attached to a network player, and the viewing mode (plan or flying in scene).

1.5 Research Conduct

I conducted my thesis research in four phases, as follows:

- **Virtual Reality Background**
- **Tomorrow's Reality Gallery (SIGGRAPH '93)**
- **Enhancements to Synthetic BattleBridge**
- **Thesis Preparation & Defense**

The following sections describe each phase and the associated tasks; the last section contains a schedule for each phase and task.

1.5.1 Virtual Reality Background

In this phase I focus on the current status in interface design for 3D virtual environments, develop the background required to conduct new research, and investigate the capabilities of the current SBB implementation.

I began by looking into the current issues relating to user interface design. Then my research moved into the current SBB system where I reviewed the current capabilities, design, and implementation before I developed an ObjectSim framework for my enhancements. Conduct of additional background research, such as learning SGI Performer (a graphics library for the Silicon Graphics workstations used at AFIT for virtual reality investigations) or studying new graphics/user interface issues, occurred as needed during each succeeding phase.

1.5.2 Tomorrow's Reality Gallery (SIGGRAPH '93)

The objective for this phase involved building a subset of the main enhancements into a form suitable for presentation at the Tomorrow's Reality Gallery (TRG) at the 1993 Conference on Interactive Computer Graphics (SIGGRAPH '93) at Anaheim, California held from 1 to 8 Aug 93. The SIGGRAPH preparation schedule required that the implementation of the majority of enhancements be in place by 7 Jul 93.

The first enhancement for SIGGRAPH involved porting the previous SBB capabilities into the ObjectSim framework developed in the first phase with a four-fold performance (frame-rate) improvement as the objective. Next I developed the list of enhancements to be included in the TRG user interface subset, envisioned as a "simple" two button arrangement centered mostly on navigating within the simulation environment. I implemented this

interface along with a stereoscopic view using slightly different viewpoints—simulating eye separation distance—and red/blue channel rendering.

In preparation for the big show, all implementations and equipment for SIGGRAPH were transported on 27 Jul 93 to California. Preparation for SIGGRAPH entailed training other AFIT students in the use of the systems (in case of illness or absence) and generating instructions for the general public. SIGGRAPH '93 provided a forum for showing off AFITs hard work and was a major success.

1.5.3 Enhancements to Synthetic BattleBridge

Upon return from the SIGGRAPH conference, I had two months to implement a representative system and prepare my thesis. Much of the ground work was complete due to SIGGRAPH.

1.5.3.1 Navigation

I first developed the navigation aspects of the system. The navigation controls, where the user either directly (using a movement control of some kind) or indirectly (using either voice activated commands or icon selection) modifies the view, was followed by the selection of views and players.

Direct movement controls functionally resemble the "stick" in a helicopter, except movement in any direction is possible. For indirect view manipulation, the user selects and defines viewing areas in a number of different ways. The most useful way involves defining a list of views and then being able to jump among the views. Other views allow attachment to particular objects within the environment or generation of views automatically based on some condition, based on timing, on the force distributions, or other important events.

The user can attach to players visible in the scene, either by directly selecting the player or by selecting a player identifier from the active list. The user has the ability to narrow the list to specific types of vehicles or domains, such as selecting only a list of air domain players.

1.5.3.2 Information

I completed the information displays when the navigation controls implementation neared completion. This phase consisted of doing as much as possible in the time allotted with a cutoff of 1 Oct 93. I developed the grid to show the composition and distribution of forces within the simulation environment.

1.5.3.3 Interface Development

In addition to the navigation controls and information displays, I developed a new and unique interface system to enhance the look, feel, and functionality of the SBB, to include transparent and dynamic display buttons. I also developed an interface for a dynamic representative model tool known as the Model Manager. In addition to these capabilities, I developed an external application interface allowing information flow between the Fuzzy Logic Sentinel (11) and the SBB.

1.5.4 Thesis Preparation & Defense

In this phase I prepared and then defended my thesis. Following the defense, I reworked my thesis according to the inputs from my thesis committee members. I defended in late-November.

1.6 Materials and Equipment

I used a Silicon Graphics workstation for this research along with many of the current data input devices associated with virtual reality technology, as follows:

1.6.1 Hardware

- FakeSpace BOOM2M (multichannel monochrome graphics display)
- Polhemus Head Mounted Display (HMD) & head tracker
- Apple Macintosh Quadra 800 w/Stereo Speakers (for sound)

1.6.2 Software

- C++ compiler
- Performer software library
- MultiGen object generation facility
- Sound Generation Facility v2.0 (13)

1.7 Complementary Efforts

This thesis effort exists due to the hard work of many people besides myself; the entire graphics research team continuously fed new ideas and insights to the other members of the team. I benefited greatly from sharing in the exploits of my colleagues. An overview of these efforts follows.

1.7.1 Fuzzy Logic Sentinel

Captain Brian Soltz (11) developed the Fuzzy Logic Sentinel as an add-in to the SBB. He and I worked together on the interface between our respective projects to provide a smooth and seamless integration of capabilities that provides the user with enhanced situational awareness.

1.7.2 ObjectSim

The framework for all current AFIT synthetic environment research, Captain Mark Snyder's ObjectSim (12) allows a wide variety of capabilities in similarly engineered applications. His toolbox enabled the rest of the graphics team to concentrate on features instead of form.

1.7.3 Object Manager

Mr. Steve Sheasby developed the Object Manager as a network player monitor. The SBB and all other DIS-related applications draw heavily on the Object Manager.

1.7.4 Virtual Cockpit

Captains William Gerhard (15) and Matthew Erichsen (14) significantly enhanced the reality of the Virtual Cockpit (VC) simulation. Much of their leadership into the DIS environment helped the rest of the graphics team advance. I especially appreciate the round-earth capabilities engineered by Captain Erichsen.

1.7.5 Red Flag

Major Mike Gardner (16) developed a close cousin to the SBB for RedFlag exercises called the Remote Debriefing Tool (RDT). The similar missions promoted a healthy competition and comparison, and a familiarity for many problems encountered that helped occasionally to correct an errant course. "Been there, done that, didn't work" was most apparent between me and Mike than between any other.

1.7.6 Satellite Modeler

The Satellite Modeler also shares some of the mission of the SBB, though on a grand scale. Captain Andrea Kunz (17) developed an impressive situational awareness tool for space and satellites that must be seen on a very large screen to be fully appreciated.

1.8 Schedule

The following schedule lists the phases and tasks included in this research project:

Phase	Schedule	
	Start	Finish
• Task		
Virtual Reality Background	1 Apr 93	1 Jun 93
• Virtual Reality User Interface Issues	1 Apr 93	15 May 93
• Current SBB Capabilities	15 May 93	23 May 93
• Basic ObjectSim Application	20 May 93	1 Jun 93
Tomorrow's Reality Gallery (SIGGRAPH '93)	1 Jun 93	8 Aug 93
• TRG Development	1 Jun 93	18 Jul 93
• Final System Checkout & Preparation	19 Jul 93	28 Jul 93
• SIGGRAPH '93 - Anaheim, California	30 Jul 93	8 Aug 93
Enhancements to Synthetic BattleBridge	1 Jul 93	1 Oct 93
• Navigation	1 Jul 93	30 Aug 93
• Information	15 Aug 93	30 Sep 93
• Interface	15 Sep 93	1 Oct 93
Thesis Preparation & Defense	1 Oct 93	29 Nov 93
• Thesis Preparation - Draft	1 Oct 93	18 Nov 93
• Defense Presentation Preparation	16 Nov 93	22 Nov 93
• Thesis Defense		23 Nov 93
• Thesis Preparation - Rework	23 Nov 93	29 Nov 93

2. Background

2.1 Introduction

The promise of simulated environments to provide a relatively cheap and effective means of practicing the art of war require improvements due to limitations in the presentation to the user. The user interface to a simulated environment must become closer to the real world interface systems our eyes, ears, even nose provide to make us believe in the reality of our simulated environment. Most simulations attempt to fool these on-board human systems with an alternate-virtual-reality using visual and aural cues to evoke feelings of "telepresence," the current gauge of a true "virtual reality" system that attempts to quantify the perception of "being there" (1:75).

This review focuses on building and interfacing computer-generated worlds so that virtual reality may achieve its full potential and allow users to "be there." Uses abound for a system that provides the opportunity to launch a thousand air sorties against Iraq without losing a single real pilot, sending ten divisions to assault Omaha Beach without the huge cost in life and materiel, or visualizing data in new and context-sensitive ways to develop other non-physical senses. The repeatability and analysis value alone makes developing such systems highly attractive.

I begin by presenting a few visions for the future of this potentially revolutionary technology before coming back to reality (not the virtual kind) with a discussion of the current status, and then conclude by further restricting my focus to the key challenges.

2.2 Future

Several authors suggest that the today's "virtual reality" technology represents the tip of the iceberg in the art of human-computer cooperation and interaction. The current advances in interface technology, from the Apple Macintosh point-and-click what-you-see-is-what-you-get interface to the fully decked out "virtual reality suit" (Figure 2-1) using a positioning glove or other sensor-based input device all have one thing in common: an interface that allows for directly selecting, creating, and manipulating objects in the environment.

2.2.1 A Vision for the Future

In the future, virtual reality may become a grandiose and limitless new space or "cyberspace"—as Benedikt predicts—where interactions with incredibly large amounts of information (assisted by virtual vehicles designed to help the user find and process the desired information) is commonplace and available to everyone (5:199-201).

Anything, anywhere, anytime is the motto for virtual reality, and Benedikt is no exception. As many technological prognosticators have done in the past, Benedikt sees a future of limitless potential based on merely imagining the possibilities given today's early steps.

He asserts that what is needed is a "cyberspace program" akin to the space programs of the past that develops the ideas of today into the virtual realities of tomorrow (5:189). Benedikt's take on virtual reality, based in part on his architectural background (5:433), has a certain appealing quality of order. His insistence on rules and principles for developers appears stifling at first but is presented well and, as always with projected opinions, up to developers to accept or modify to suit their needs (5:132).

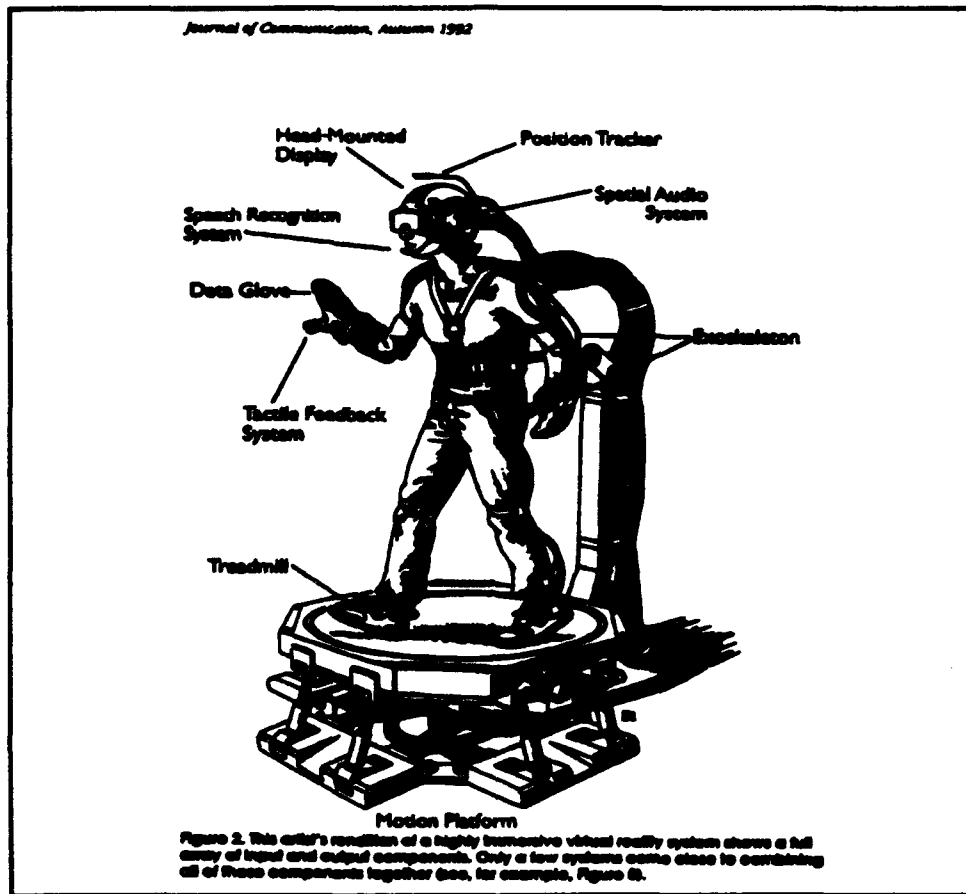


Figure 2-1 Fully Immersive Virtual Reality Suit (3)

2.2.2 No Interface to Design

Benedikt's vision of the future, regardless of validity, or more appropriately, viability, stretches the art of human-computer interface design. Others insist the challenge is to make the interface invisible or seemingly non-existent (6:364). Bricken contends that the term interface infers a boundary, and that elimination—not creation—of boundaries characterizes the user interface of tomorrow (6:365).

Bricken further contends that the main focus of a user is on what they can do within the simulated environment (6:371-373). Neat visuals interest users for only a very short time; soon they become bored and want to move on to the next adventure or experience. Bricken reports that users figure out these virtual worlds quickly and easily if they appear similar to the real world (6:377).

2.3 Status

2.3.1 A Tutorial

Generation of virtual worlds rivaling the quality of the human visual system requires a massive amount of computational power; some estimate it at eighty times the current processing power of our best graphics engine (3:61).

Biocca, in his tutorial on the current state of virtual reality technology, like Benedikt, believes in the limitless future of virtual reality. As quoted in Biocca, "...Warren Robinette, a key designer at NASA ... (believes) ' The electronic expansion of human perception has, as its manifest destiny, to cover the entire human sensorium' " (3:23-24). I don't know about manifest destiny, but I think he's trying to say that for virtual reality to succeed, users must believe that they are really within the simulated environment and that the environment must avoid placing artificial bounds on the user.

The future holds great promise of boundless new worlds we can create ourselves and modify at will. The ability to view and interact with objects from any perspective requires representation in a form that the computer understands. Objects within the simulated environment are manufactured using basic computer design techniques and "...marketing of virtual models

has already begun," though real-time generation of new objects may eventually replace these static models (3:60-61).

The problem is not just size and location, but how to determine and enter object information, such as color, texture, feel, hardness, etc., into the computer so that the object behaves as it should (3:60). All this information presents further problems like how to store and retrieve the information, how to present groups of objects, how to visualize non-physical information, and how to navigate in the virtual world.

Extensive research into database management and design, particularly object-oriented design may answer some of these very challenging questions. At the very least, research into modeling and the extension of computer-based modeling of objects will only help reduce the current problems.

2.3.2 Object Data Manipulation

Jacob has defined a method for standard manipulation of objects, once created (7). The main idea of his specification language for direct manipulation of objects concerns the ability to allow functions on the objects based on the selected object instead of "carrying on a dialog about" the object (7:283). Previous interfaces mostly allow the user to select the function first and then the object on which to operate.

Dynamic objects require constant update of position and heading. Direct manipulation due to the uncertainty of position of these objects presents an on-going problem.

2.3.3 Terrain Data Manipulation

Terrain data manipulation presents a different problem. The enormous size of a useful terrain database makes it nearly impossible to load into

computer memory all at once and even more difficult to determine which parts of the terrain to present. Searching the terrain database causes large delays in presenting the rendered image to the user by decreasing the frame rate significantly (8:66), leading to lower quality simulations (8:65).

Researchers from the Naval Postgraduate School (NPS) present methods for "culling" out unneeded information, using a hierarchical "tree" structure for the terrain data (8). The top level of the tree holds the lowest resolution, smallest size data, while lower levels hold increasingly higher resolution, larger size data. Location and distance from the viewer determine the level to use.

Combined with a moving grid of active terrain (called a bounding box), the NPS tree structure allows for effective management of most terrain data while decreasing the processing time spent weeding out unneeded information. Their studies have shown significant performance improvements over previous methods (8:68-69).

Though not really an emphasis of this research, terrain data manipulation must become more flexible and less of a performance handicap. Hopefully, advances in this area will relieve some of the draw bottleneck and reduced level of visual fidelity that occurs in most simulations.

2.4 Conclusion

I've attempted to bring the wide spectrum of virtual reality technology from the lofty visions of the future to the problems of representing largely static terrain information within the confines of the virtual new world. I've identified further areas of research along the way that define the paths for advancing the current state of the art to the next level where man and machine blend into one to work smarter, play harder, and train better.

To achieve the next level, a level I firmly believe will revolutionize the way we use computers to train and work in the future, the technical community must meet a wide variety of challenges. Lets get to work. Hand me that spaceball...

3. Design

The overall design philosophy for the SBB: provide powerful navigation and information presentation capabilities while maintaining an almost completely unobtrusive interface. A major objective during development of the SBB was to further the application of menu driven and object-oriented interface design to virtual reality systems that (someday) hope to detach from the standard keyboard input to voice command processing.

The previous iteration of the SBB (9) had a crude voice command interface that fell short of useful. Instead of integrating this system, I chose instead to develop an interface as suited to voice command input as mouse input. In a sense, the mouse substitutes for an effective voice command system, with the knowledge that a future iteration can incorporate useful voice processing (when effective voice translation and interpretation software becomes available) with little effort.

Before describing the design of the application and the objects that provide the design capabilities, I next describe the interface elements and the functionality present in the SBB. I then describe the overall simulation system design integrated with the interface, followed by the design of the individual objects enabling the simulation environment and the user interface.

3.1 SBB User Interface Design and Functionality

The Synthetic BattleBridge, compared to all other AFIT simulation applications, has a unique user interface due to the required transparency of the interface elements, and the dynamics of their operation; transparency allows the view, while in some ways restricted, to never be fully obscured. I designed these interface elements from scratch using graphics calls and a

Button class to be described later, building a "library" of common interface elements along the way.

The main objectives of my user interface design effort was to build an interface adaptable in the future to voice commands as that technology develops and to minimize the screen real estate used for the actual interface. I succeeded in both of these objectives using the transparent dynamic buttons described in the sections to follow. These buttons translate well to the chosen monochrome viewing device (the FakeSpace BOOM2M), and can easily adapt to the task of processing voice commands.

3.1.1 Description

The user interface consists of buttons and menus providing access to the variety of SBB functional elements, as shown in Figure 3-1.

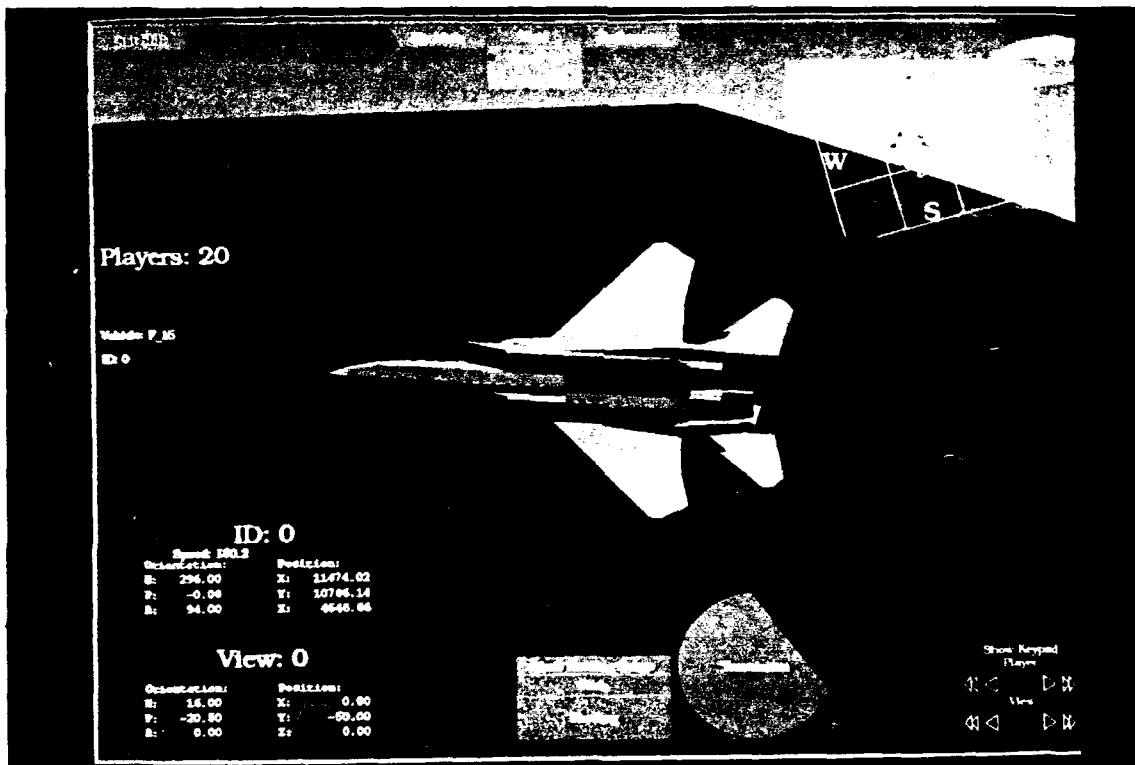


Figure 3-1 The SBB User Interface

At the bottom are the navigation controls; the lower left side holds the View and Player attachment controls; along the top are the preferences, grid, and mode controls. The grid, when enabled, has two modes: large and small. The small grid occupies the upper right corner of the screen, while the large grid occupies most of the central screen area; only one mode is active at a time.

Above the pitch control button, the Heading Gauge demonstrates another feature of the interface: certain elements have minimal representations that the user presses to activate the full-size button or area. Pressing the "Show Heading Gauge" button (shown in Figure 3-1) activates the full Heading Gauge (clasp in Figures 3-2 and 3-3) showing the current orientation of the view.

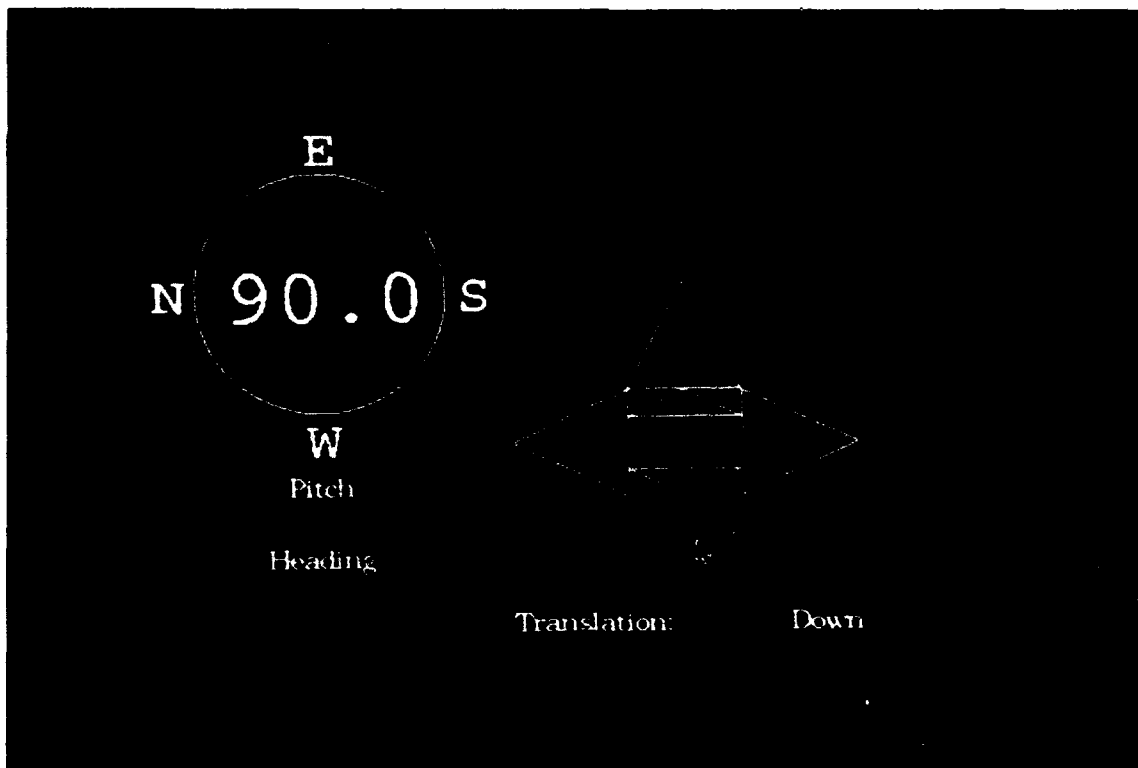


Figure 3-2 Active Translation Button

The Translation and Heading buttons show another unique aspect of the SBB user interface, what I call dynamic display. In Figure 3-2, notice that when the mouse is over the Translation button, the underlying structure and functionality shows through while the Heading button remains at a minimal configuration; when over the Heading button (Figure 3-3), the Translation button goes to the minimal configuration while the Heading button becomes more complex.

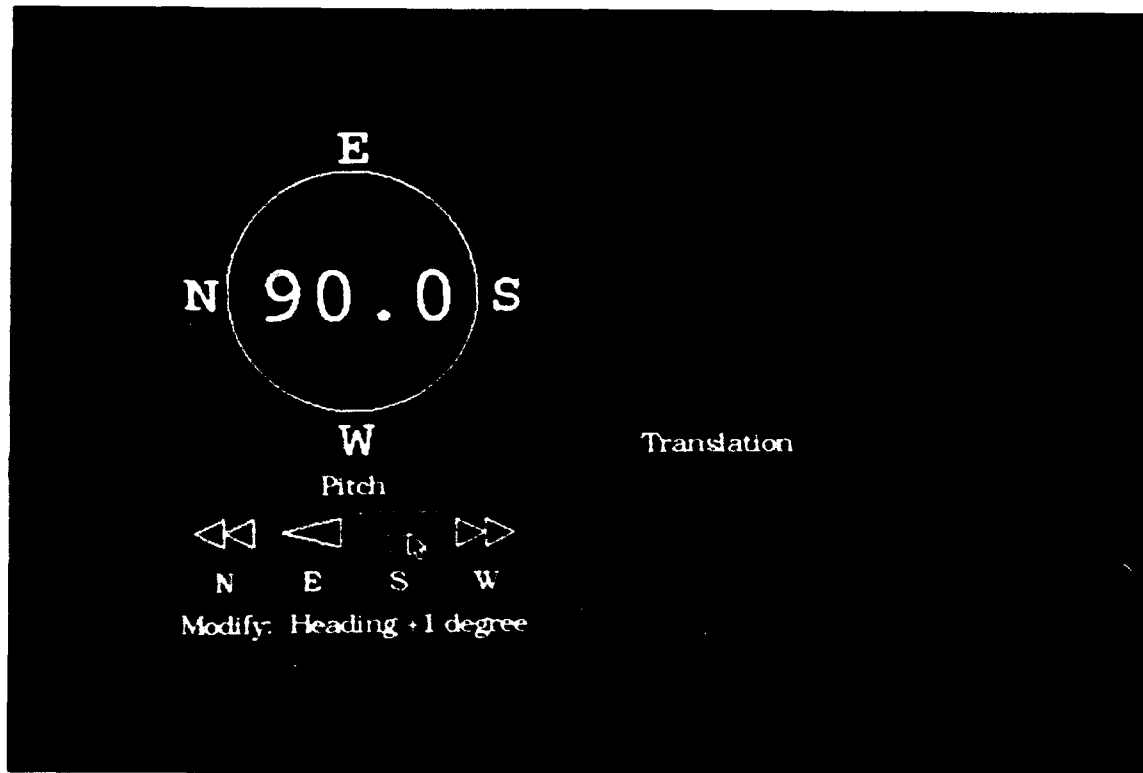


Figure 3-3 Active Heading Button

Notice the simplicity of the Translation and Heading buttons when not over them with the mouse. I designed the buttons this way so that they retreat to the background when the user focus is elsewhere.

Also, notice in both figures the message at the bottom of the control that tells the user the function of the button; this is my form of on-line, context sensitive help that also retreats to the background when not active. Different

button types have different messaging capabilities, for instance the Push Button type has the ability to display two different messages depending on the state of the button (pushed/not pushed).

All of the button types give direct feedback when pushed. Most indicate being pressed by filling the button with a designated color, while others turn on a "light" that indicates state.

3.1.2 Information Grid Design

The objectives for the information grid (Figure 3-4) included the ability to see the surrounding vehicles and force distributions within a user-specified distance of the current viewpoint location, and the ability to attach to players and views directly from the grid. My design incorporates these objectives.

The grid represents the immediate area surrounding the Stealth object, sort of like a radar, with the boundaries of the nearest grid sector and the orientation (NESW) clearly visible. In Plan Mode, the current viewable area is shown by the small rectangle at the center of the grid window. Objects on the grid/radar, in the colors and shapes denoting force and type (discussed later), can be seen within a 3700 square kilometer area (~6 sectors x (25km x 25km sectors)) centered at the current viewpoint coordinates.

Notice in Figure 3-4 that the ground objects on the grid appear in the same relative location with respect to the viewable area as the actual objects in the scene relative to the screen boundaries.

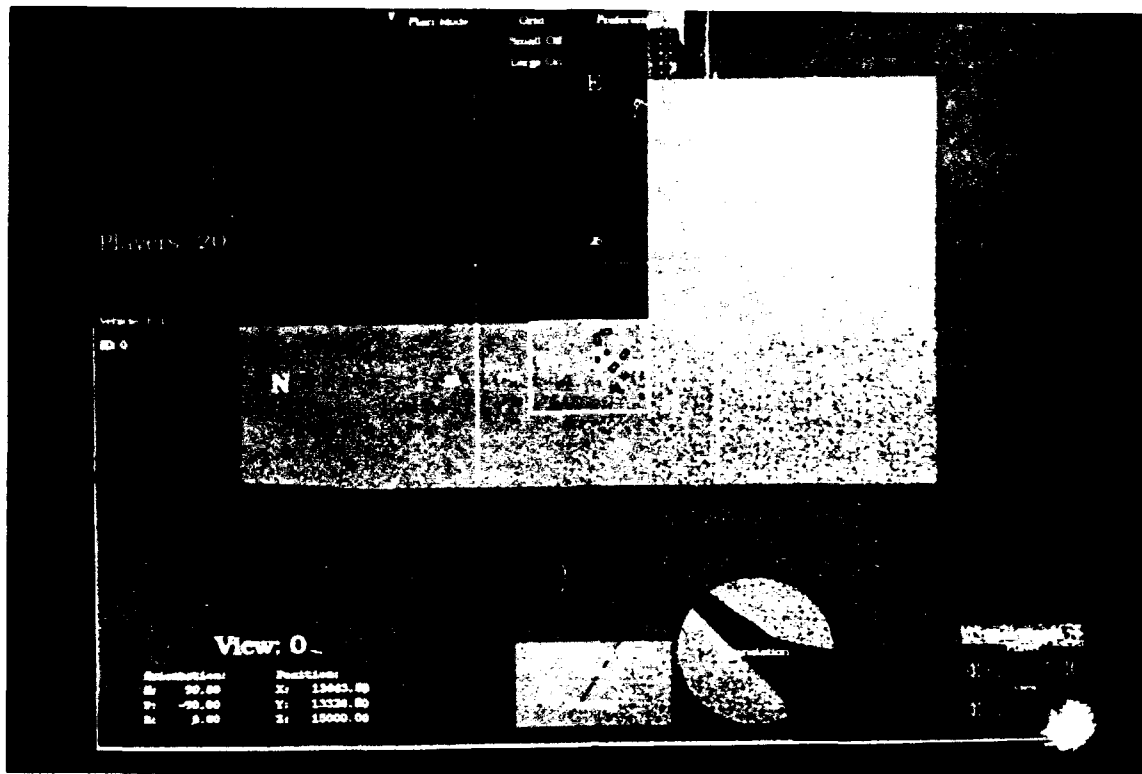


Figure 3-4 The Information Grid

3.1.3 FakeSpace BOOM Interface

The buttons and grid described above function well in a monochrome display device such as the FakeSpace BOOM (Figure 3-5), another reason I chose to design my own set of interface elements. The available tools such as the Forms library (19), provide lots of interface capability but at a performance cost that proved too expensive for the SBB. Plus, the Forms did not appear well on the BOOM, because of the colors and fonts, and small screen space available. Basically, the forms become unreadable.

My buttons have a gray scale equivalent that displays the fonts in white, and the lights as filled/unfilled white circles. With the transparency eliminating the screen space problem, the only real problem becomes how to select them. I consider the answer to this problem as future work that involves a "flying mouse" or voice commands, both of which the current

interface could handle without modification (well, maybe a little, but not much).

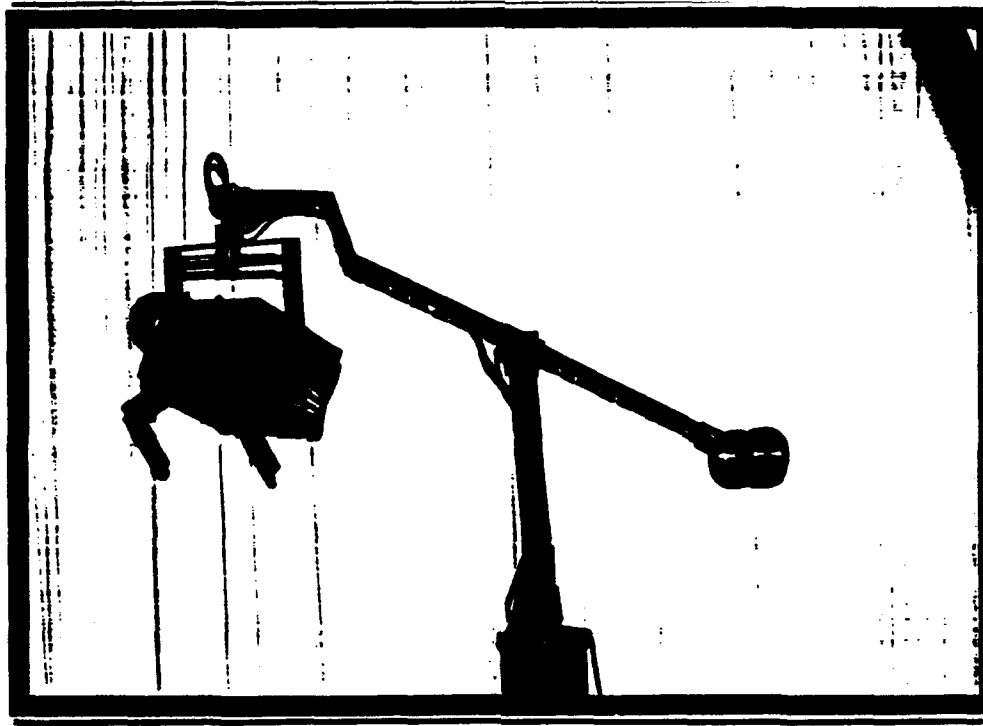


Figure 3-5 FakeSpace BOOM

The SBB operates in either gray-scale or 3D on the BOOM, with 3D having about half the frame-rate as the gray-scale due to the second viewpoint processing. The 3D viewing is best within a few meters of the focus object.

3.1.4 Head Mounted Display Interface

A second non-terminal display, the Polhemus LookingGlass Head-Mounted Display (Figure 3-6) permits color and gray-scale viewing. Currently, the HMD has been untested in the SBB, though the potential for its use exists within the ObjectSim framework.

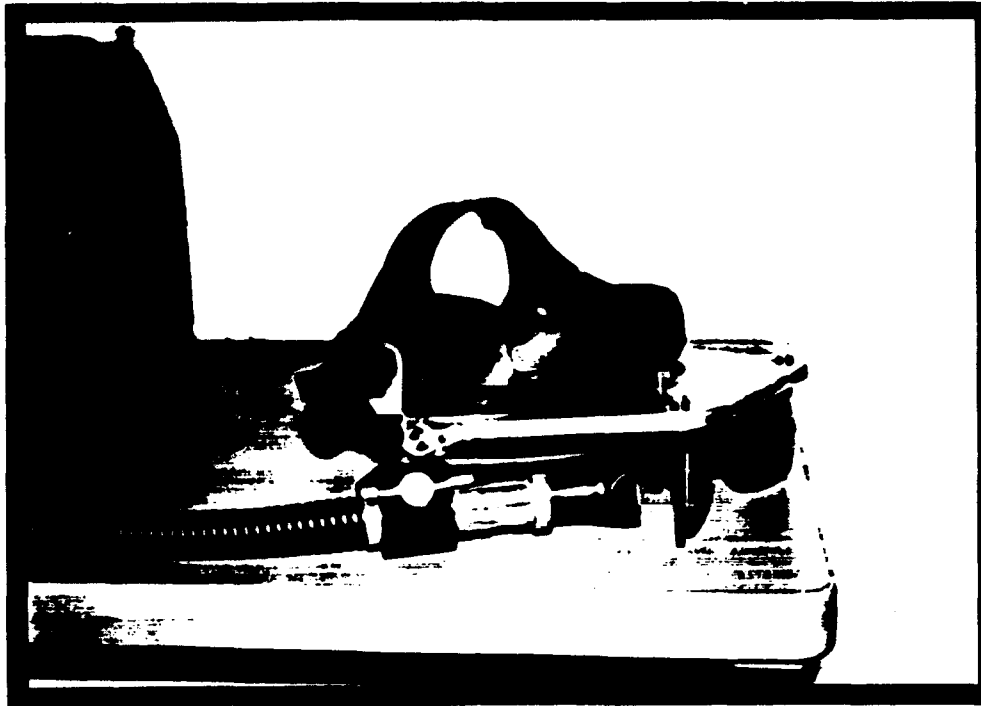


Figure 3-6 Polhemus LookingGlass Head-Mounted Display

3.2 Synthetic BattleBridge Simulation Design

In the following sections, I discuss the basis and framework for the Synthetic BattleBridge (SBB) application, a giant step forward in AFIT simulation work called ObjectSim, developed by Captain Mark Snyder (12). Then I discuss the SBB Object Manager developed by Steve Sheasby (10), the network interface solution that manages the Distributed Interactive Simulation (DIS) protocol data units (PDUs) as they pertain to specific SBB needs. Finally, I put it all together by describing the SBB-specific objects that are derived from ObjectSim, interface to the Object Manager, and provide user interface and information presentation capabilities unique to the SBB.

I designed the SBB to provide a view into the DIS simulation environment using the ObjectSim framework and the Object Manager net player management software. I developed the SBB with the focus of views

and tools to navigate and interrogate the environment to enhance the user's situational awareness. The SBB maintains this focus by abstracting the environment into less complicated visual information or enhancing the environment with other information on the disposition and composition of forces, the location of forces, on-going conflict, and patterns of conflict.

The shell of the SBB comprises a User Interface Management System (UIMS) with the tools and information displays integrated into the overall SBB architecture. I have also provided for other tools to be added to the SBB on an as needed basis; currently, the FLS is the only information tool integrated into the SBB, but others can easily follow using an approach described in (11).

Various objects are integrated into this shell to provide navigation and information presentation capabilities, 3D rendering, trails, locators, various modes, and a wide variety of views and view options. Before I get to these objects, I will explore ObjectSim as it pertains to the SBB.

Note that SBB objects are capitalized for recognition purposes.

3.2.1 ObjectSim

As shown in Figure 3-6, the object-oriented design of ObjectSim (12) allows subsequent designers to inherit and modify nearly all aspects of the simulation to suit their needs. I have used this framework to provide Stealth players (with no model to represent them), Locator players (with an abstract locator model representative of the type of vehicle) and Shadow players (to offset the view slightly for a 3D view). Each of these players derives some attributes and methods from an ObjectSim player in order to integrate into the ObjectSim framework.

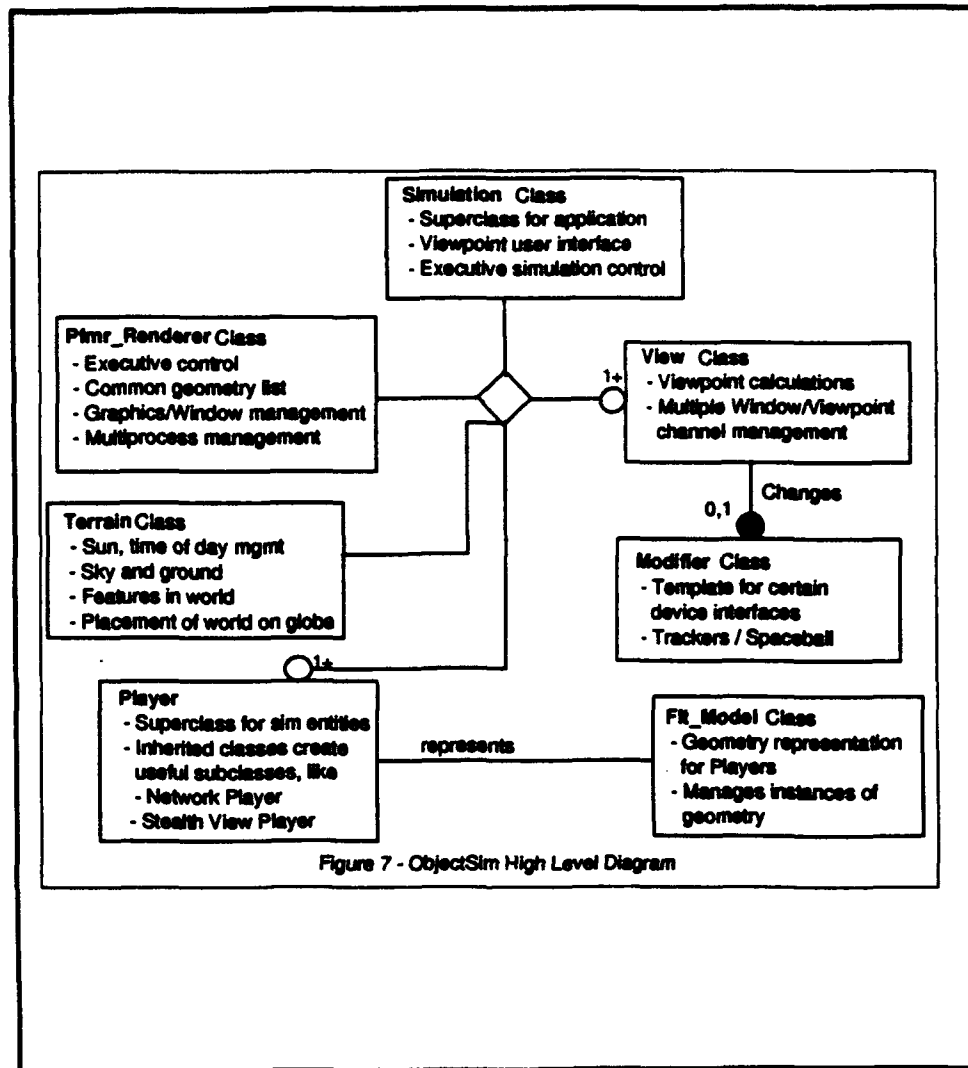


Figure 3-7 The ObjectSim Object Model

3.2.1.1 Simulation

The Simulation object is the main object of any ObjectSim-based application. Simulation contains the initialization parameters, the various other objects in the simulation, and the overall structure for the application. For example, the terrain, views, players, managers and other objects required to run are all initialized by the simulation.

3.2.1.2 View

During execution, the Simulation allows the user to control the View and the Player attached to the View. The View controls the relative location of the objects in the scene for the channel to which it is assigned by accessing the view player (a special type of player that builds on either a "real" object in the scene, or a Stealth object) attached to the View. The View object contains all the methods and data necessary to render the scene defined by the current attached Player.

3.2.1.3 Attachable Player

Players derived from the Attachable Player class have widely varying behavior depending on type, each usable within ObjectSim provided they meet certain minimum requirements for design. Due to the object oriented structure, any Attachable Player derivative can attach to a View allowing context/mode sensitive controls and great flexibility. *I use this flexibility to provide navigation and information displays that change based on user inputs, the type of view, and the current situation.*

Each player may or may not have a model that represents them in the display depending on the player type; net players are "mapped" to the nearest appropriate—or available—model using a Model Manager (described below), while Stealth players get no model at all. The network contains data for only the "live" players, that is the players currently active and present in the simulation environment. Stealth players are considered local, non-network players that affect only the SBB operation, and are handled differently.

3.2.1.4 Terrain

Each Simulation can have one or more Terrain player (really, player may not be the best description, but it will have to do) that provides context for the movement of net players within the scene. Terrain remains relatively constant in location and may have associated other features, such as buildings, trees, hedges, bridges, rivers, etc. Various methods for terrain resolution management exist (as discussed in Chapter 2), though the abstraction within ObjectSim hides the actual implementation from the application designer.

The Terrain database also determines whether the SBB uses a round-earth conversion algorithm or the straight flat-earth coordinates for net players. The wide variety of uses for the SBB, and the objective for as general a design as possible, made design decisions always lean toward flexibility and generality.

3.2.1.5 Renderer

Like the Terrain, the details for the Renderer are also hidden from the designer; except for allocating, initializing, and then starting the renderer, the application knows nothing more about it. To understand the renderer operation, one must understand the Performer tree structure. This structure allows the Renderer to bring all the pieces together and then render the resulting image. Each reachable branch of the tree is rendered each frame unless specifically told not to.

Following initialization, the Renderer repeatedly loops throughout the structure built by the Simulation to cull then draw the scene represented by the current Performer tree. The user actually interacts with the scene by manipulating elements of the tree and then repeatedly drawing the "new"

reality of the tree. ObjectSim handles this tree manipulation implicitly as "toolbox" commands within the ObjectSim framework, while only exceptional cases require direct access to the tree or knowledge of the Performer tree (such as for 3D rendering override, discussed below).

ObjectSim, a structured Performer toolbox expressly for simulations, handles most of the interface to Performer and control of the graphics pipeline(s). The structure of Performer necessitates thinking in three streams at all times: 1) Application, 2) Cull, and 3) Draw. Unless handled correctly, processes on one stream cannot "see" processes on another. Each of the three streams operate independently within a frame, but each must wait for the others to complete before starting a new frame, a process of synchronization that requires balance for efficient operation and high frame-rate. The operation of the Object Manager, my next topic, greatly affects the frame-rate.

3.2.2 Object Manager

The Object Manager (10) provides the net management facilities for client applications (such as reading and interpreting DIS packets from the network). The SBB Net Manager (providing specific utilities, as described in the next section) controls the SBB version of the Object Manager.

Since the SBB only receives and does not broadcast, the only task of the SBB Object Manager becomes one of monitoring the network for updates to net player² and providing access to the list of these players, a focused approach allowing for economies and optimizations within the update utilities (though there is plenty of room for improvement). Figure 3-10 contains a diagram of the SBB Object Manager/SBB Net Manager high level design and relationship.

A description of the information maintained for the net players using the SBB Object Manager follows (a combination of the information in the base net player and in the SBB net player):

<u>Information</u>	<u>Description</u>
Entity ID	A unique identifier used to distinguish between net players of the same type.
Entity Designation	Basically, the type of player; this information is used to map the entity to a model (e.g., F_15). Based on the DIS protocol.
Entity Domain	Air, Land, Surface, Subsurface, Space, etc.
Entity Category	Specifics for each of the domains.
Entity Model	The particular model for an entity, such as M1A1 or M1A2.
Forces Type	Friendly, opposing, neutral, and unknown.
Country	One of the DIS supported countries.
Position	XYZ coordinates for the player.
Orientation	Heading, pitch, and roll for the player.
Velocity	The velocity vector: <vx, vy, vz>.

Updating (player processing) consists mainly of updating player position and orientation—including round-earth conversions—and managing the player list to remove destroyed or inactive players.

3.3 SBB Top Level Design

I designed the SBB to build on ObjectSim and the Object Manager; I derived and modified the behavior of most of the objects available to make the SBB-specific objects (shown in Figure 3-8 and described below). These objects, tied together by the SBB Application object, include all the capabilities necessary for a DIS simulation stealth observer system.

Note:
Only steady state for arrows,
init assumed complete.

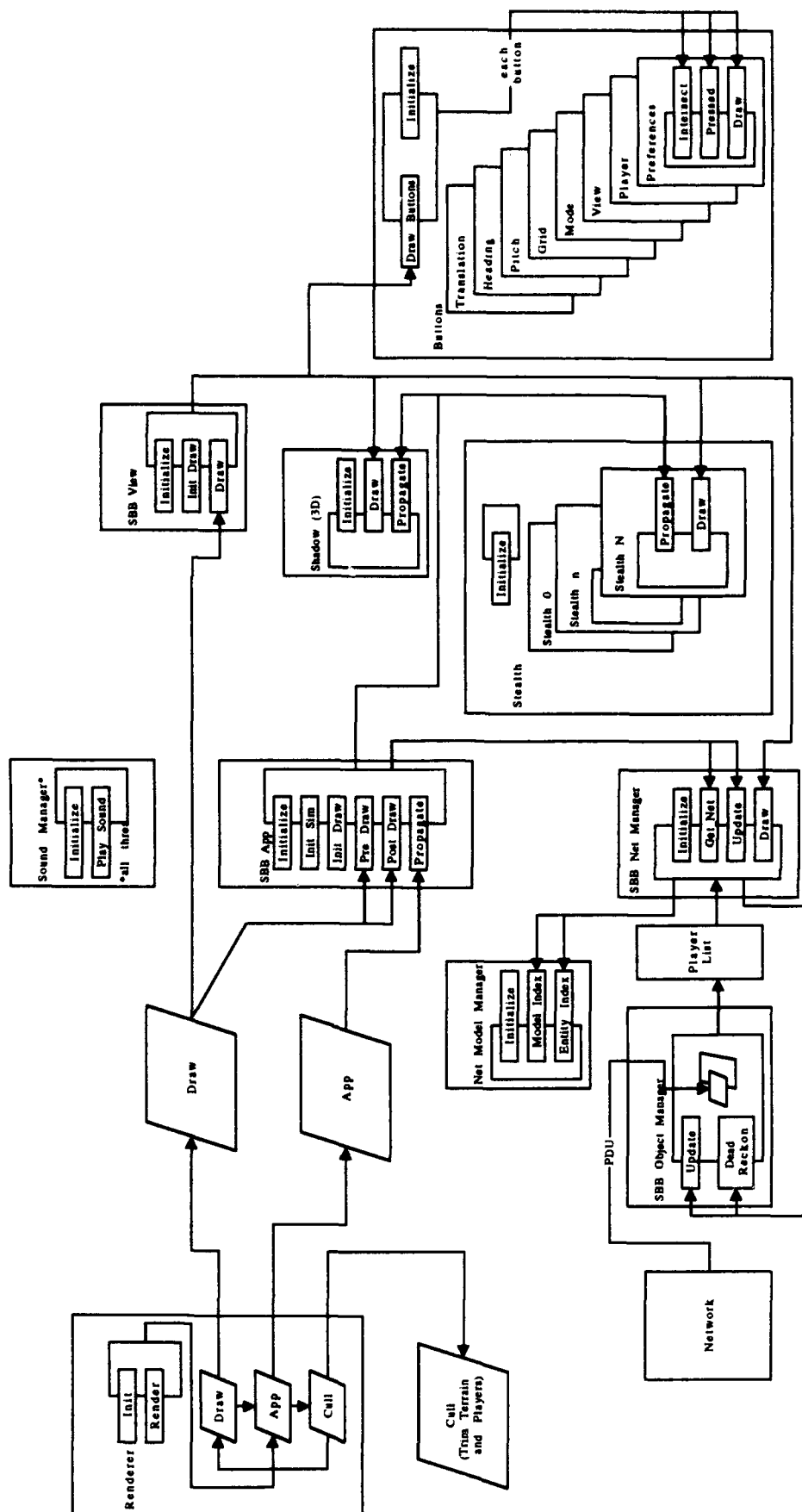


Figure 3-8 Synthetic Battle Bridge Task Object Model/Diagram

3.3.1 SBB Simulation Objects

The SBB specific objects include many derived from ObjectSim. The following table lists the SBB objects that provide the simulation capabilities and their relationship to ObjectSim objects. Figure 3-8 shows the steady state (following initialization) relationship of these objects.

<u>Object</u>	<u>Parent</u>	<u>Description</u>
•SBB Application	Simulation	The structure and timing object. Contains the entire SBB structure and initialization parameters.
•SBB View	View	SBB-specific view object, tailored for drawing buttons, trails, and other SBB interface elements.
•Stealth	Attachable Player	The workhorse of the SBB, the Stealth object contains the viewing and navigation capabilities of the SBB and most of the information processing utilities as well.
•Shadow	Attachable Player	Used to present 3D images by offsetting the stealth view. Provides means to tune the view to achieve better performance.
•SBB Net Manager	<none>	Described above, manages the network.
•SBB Net Player	Base Net Player	Contains the information the Object Manager compiles for each PDU from the network.
•Model Manager	<none>	Maps the player entities to the representative model.
•Clock	<none>	Keeps the simulation time. Contains functions for displaying and storing the time.

The following sections provide the details for each of these objects. An explanation of the other objects and utilities necessary for the interface and information follows the simulation objects description.

3.3.1.1 SBB Application

The major simulation object, the SBB Application contains the instantiations for the other objects within the simulation. The SBB Application also controls the Renderer operation and includes interfaces into the various rendering methods, such as for Cull or Draw. Following initialization, these interfaces provide for communication with the other objects; the SBB Application controls all aspects of the simulation and the management of the simulation and interface objects.

3.3.1.2 SBB View

The main simulation object, the SBB View class controls setting up and managing the draw operations for the View and the attached player for normal screen rendering and for the more complicated 3D rendering (in gray-scale only). The screen rendering mode is determined by the type of mask assigned to the instantiated view, as follows:

<u>Mask</u>	<u>Mode</u>	<u>Description</u>
•All	Normal	Does not alter the normal behavior of the view.
•Red/Blue	3D	Alters the behavior to enable only the bitplanes corresponding to the color of the mask.

In normal screen rendering mode, the SBB View performs the attached player's draw function and then draws the trails for the net players. In 3D

rendering mode, the SBB View does the operations from the normal mode with the additional task of overriding the renderer to perform specialized drawing into the individual color bitplanes.

Since the Silicon Graphics workstation produces a three-channel display output (corresponding to the red/green/blue bitplanes), instantiating two separate SBB Views, one for the red bitplanes and one for the blue, and offsetting the views using the procedure described below in the Shadow object, the resulting display contains a red/blue color separated image suitable for piping to a monochrome display device.

For this I use the FakeSpace BOOM2M, which allows separate control of the image sent to each eye viewer, each a monochrome equivalent of the intensities and hues present in the respective channel. By piping the red channel to the left eye and the slightly offset blue channel to the right eye (and discarding the green channel), the resulting monochrome image provides the visual cues present in a 3D image. These cues mostly include seeing a separate but consistent image with each eye.

3.3.1.3 Shadow

A view player, the Shadow object complements the Stealth object (discussed next) by providing the offset viewpoint for 3D rendering (as shown in Figure 3-9). The Shadow object simply adds a second view player that is offset from the Stealth view player by the eye separation distance (esd) and rotated to look at the same focal point as the Stealth.

The Shadow and Stealth objects each require a View, since they are both rendering a scene, though from slightly different vantage points. The separate views for Shadow and Stealth handle the view for the red and blue bitplanes.

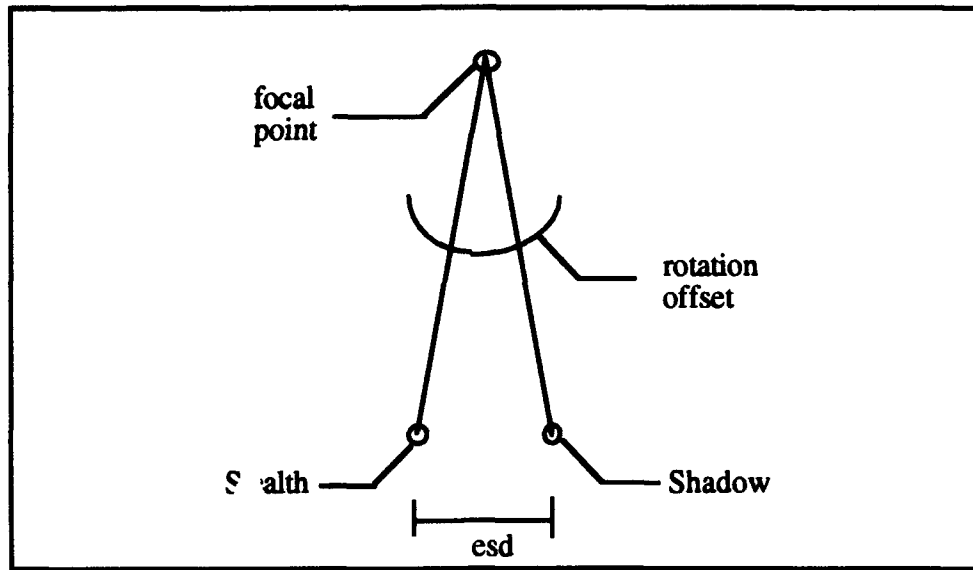


Figure 3-9 The Shadow Object Offset

The Shadow object provides built-in capabilities for modifying the eye separation distance (esd) and rotation offset that only become active upon instantiation, another demonstration of the overall design philosophy of context-sensitive controls and capabilities. This works for close up situations, but the problem with this set-up is that the manual modification of the rotation offset precludes the observer from focusing like the eye would. Until we know what the user is looking at (by scanning the eye maybe) to automatically alter the focus, the effects of 3D will be limited.

3.3.1.4 Stealth

The SBB contains a group of Stealth objects that provide the various types of views and the controls for navigating within the simulation environment. The scene rendered and the capabilities available to the user are determined by the particular Stealth attached to the View.

The design of the Stealth object is basically as an invisible vehicle (without a representative model) that transports the user within the

environment without affecting or being seen by the players within the environment (hence the term stealth). The Stealth object provides the principal interface capabilities to the user; it contains the methods and functions other objects (like the button/menu interface) call to affect the viewpoint or information presented to the user. I discuss the types of views before I discuss the controls.

3.3.1.4.1 Stealth View Types

Each view contains state variables that preserve the location and status of the view as of last attachment or initialization. The user controls which view they are attached to by selecting a Stealth object to attach to the SBB View. This allows for special purpose views, as many pre-defined views as desired (within reason), and state preservation.

The special purpose views include Fuzzy Logic Sentinel areas (11), and interrupt areas with special events programmable by the user, such as missile launches or timed viewing of areas where the user can set an alarm time that allows the SBB to take the user to a particular location, keyed to some timetable.

Unlike the special purpose areas, pre-defined areas are based on a setup file associated with a particular terrain, allowing preparation prior to simulation that eliminates setup time during the simulation. These pre-defined areas remain throughout the simulation, even if the user has moved the actual viewpoint to another location.

Again, regardless of which view is currently active, the user's previously visited views, the pre-defined views, and the interrupt and special purpose views hold their state. This preservation of state means that the user need

only define the views desired and then selectively cycle through them or define new ones.

3.3.1.4.2 Stealth View Controls

The navigation controls contained within the Stealth object allow the user to modify the viewpoint location and orientation. Basically event-driven, the Stealth controls allow the user to move along the three independent axes, orient heading and pitch (not roll), increase/decrease the rates of movement, attach to net players, and invoke pre-defined local viewpoints within a view (such as viewing from above or behind).

The event processor design of the Stealth means that processing occurs only when the user modifies the viewpoint, and then only to change the minimum variables required. When the user is simply observing and not changing the viewpoint, minimal processing occurs, as efficient a design as I could think of. The event processor operates by waterfalling through the state variables for the stealth object attached to the SBB view.

3.3.1.5 SBB Net Manager

As stated above, the SBB only receives information (due to the stealth nature of the application), so the SBB Net Manager includes the ability to get and update a list of the currently active net players. The SBB Net Manager, responsible for allocating and initializing the group of net players and their corresponding trails and locators, also controls the mapping of net players to the representation models and the abstract locator models, based on the entity type, designation, and forces type (friend or foe) using the net player Model Manager. During operation, the SBB Net Manager enables the trails and locators.

The SBB Net Manager also controls the instantiation and management of the SBB Object Manager. Figure 3-10 shows the relationship between the SBB Object Manager and the SBB Net Manager. Notice that the two objects share the list of net players. This allows for much faster processing and little wasted effort copying information.

3.3.1.6 SBB Net Player

The SBB Net Player trims the enormous volume of information available within the DIS standard to the minimum required for the SBB, and the place to modify to increase the information received from the net in the case of a complementary application (such as the FLS). The sole purpose of the SBB Net Player is to define the information requirements for the SBB.

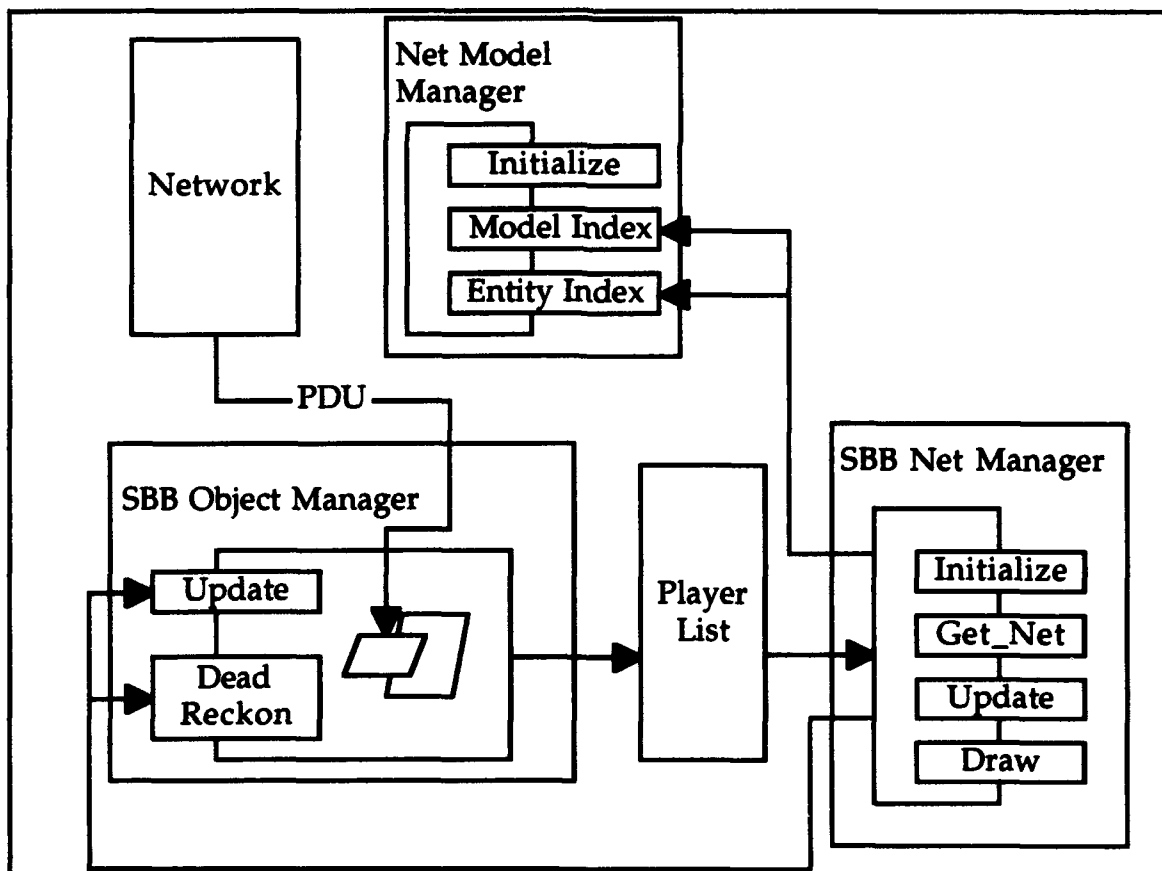


Figure 3-10 The Object Manager/SBB Net Manager Design

3.3.1.7 Locator

With the large amount of data coming into the SBB, and the wide variety of possible vehicle and entity types, I attempted to simplify and enhance the scene using the Locator player. Instead of seeing F-15s and F-4s, MiG-29s and Su-27s, M1s, M35s, and HUMMWVs, the skies would fill with locators that look like pills (some say footballs, see Figure 3-11), and the ground would cover with pyramid shaped locators (Figure 3-12) denoting the location and orientation of the vehicle, both of which are visible for (virtual) miles.

Notice in Figure 3-11 the other air vehicle about 10 kilometers in front of the closer air vehicle; without a locator, the vehicle would be invisible. To simply enlarge the vehicles would present an incorrect image with the vehicle appearing only a few hundred meters away.

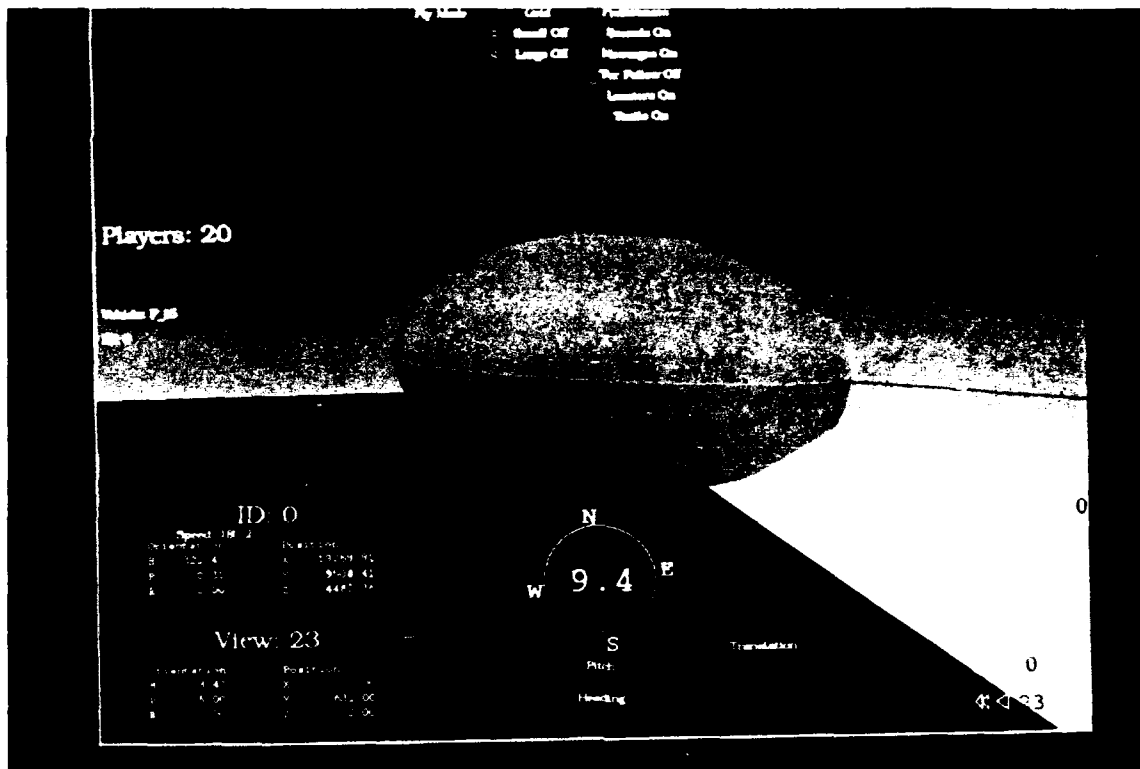


Figure 3-11 Air Vehicle Locator

In essence, the locator view is an abstraction of the battle space into types, simplifying the view while providing more information because the locators indicate friend or foe (by color), orientation (light on top, darker on bottom), and direction of motion (the colored ring at the front of the air locator, and the others simply point in the direction of motion).

The locators preserve the context of the image while simplifying; the user knows that another vehicle is out there, and the user also knows the orientation and forces type. The user gets this information from the ring on the front of the locator (or on the back polygon) or from the trail left by the vehicle, my next topic.

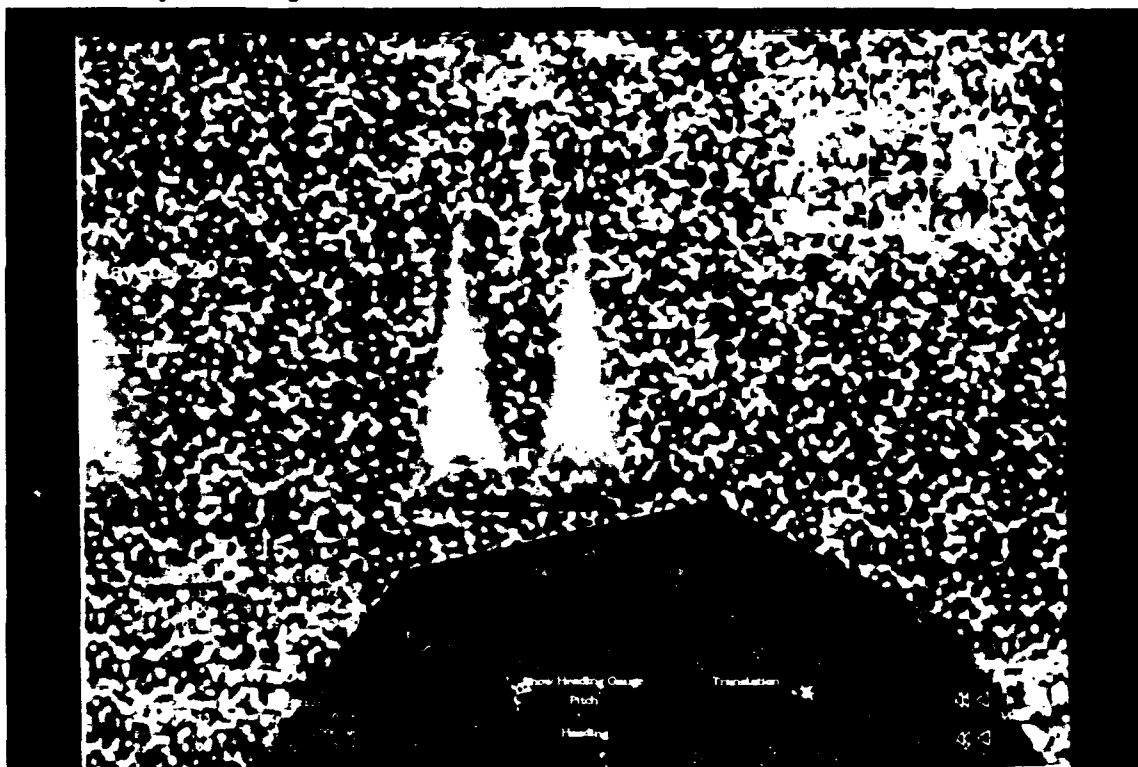


Figure 3-12 Ground Vehicle Locator

3.3.1.8 Trail

A close cousin to the Locator, the Trail indicates friend/foe and location history of the vehicle over some preset time period. Provisions exist for

setting each segment of the trail to a color based on the vehicle energy or velocity over the interval, while the algorithm for determining which color to use remains as future work.

3.3.1.9 Model Manager

Mapping of the SBB Net Players and any other players to actual representative models has been a problem for a long time. I solved the majority of problems by developing the Model Manager class, a data file interpreter designed to map the entity designations contained within the Object Manager/DIS standard to available models.

For instance, the user may decide to map all MiG-29s to the F/A-18 model, since no MiG-29 model is available. This mapping affects only the rendering of the entity within the scene and none of the data, the user will see an F/A-18, but the actual data and performance (given a correct driving application) will reflect a MiG-29.

I designed the Model Manager to allow methods for querying the mappings of entities to the representative model type. Other uses for the Model Manager include mapping of the terrain, animation effects (explosions, etc.), and locator models. I also use the comment field within the Model Manager to hold other information on the models such as the scale factors for the various locators.

3.3.1.10 Clock

The last simulation object, the clock, provides capabilities for setting and retrieving the system time based on either the simulation start time or the real time, based on user input. The clock produces a time format of "HH:MM:SS" usable by any object in the simulation.

3.4 Interface and Other Object Design

Along with the objects for simulation described above, the SBB also contains objects and utilities for information presentation and user interface, as described in the next table:

<u>Object</u>	<u>Parent</u>	<u>Description</u>
•Player Button	<none>	One for each player when the Grid is open in Large mode. Allows attachment to players.
•SBB Button	Button	User interface buttons. Transparent and modifiable on-screen, SBB Buttons allow great flexibility in user interface design.
•Drawstring	<utility>	Utilities that assist information presentation, user interface, and screen drawing objects.
Sounds	<utility>	Provides a sounds interface to play sounds based on the situation.

These objects provide the connection between the simulation environment and the user. The following sections describe the design of these objects.

3.4.1 Player Button

On screen only in the Large Grid display mode, Player Buttons allow direct attachment to Players and Views. Each button presents player information to the user, including domain, forces type, and ID number using icons, colors, and numbers, along with the present heading. The button becomes visible only when the user places the mouse directly over the player symbol.

The player/view symbol:

Aircraft	Ground	Munitions
		

The player/view color:

Friendly	Opposing	Neutral	Unknown
Blue	Red	Yellow	White

3.4.2 SBB Button

The SBB Buttons are the main user interface. They allow the user to manipulate the environment variables as well as the interface elements. There are many different varieties of SBB Buttons, as listed in the following table and described in the sections below.

<u>Object</u>	<u>Parent</u>	<u>Description</u>
Button	<none>	The base class. Contains the data and methods for the description of a basic interface button.
Rect Button	Button	A rectangular derivation of the base class. Intersection simplicity allows for very fast operation.
Push Button	Rect Button	A specific type of button that operates only once per change from pushed to not pushed.
Light Button	Push Button	Same as a push Button except includes built-in light for indicating pushed/not pushed status.
Hideable Button	Rect Button	Operation of this button includes built-in support for hiding and enabling other buttons when pushed.
Display Button	Light Button	A modifiable button used to display text or values. Includes memory.
Menu Button	Push Button	Selection of this button enables a list/menu of other buttons.

The Button base class includes data and methods supporting titles, messages (for on-line help or context), and enabling/disabling selection. All of the buttons operate on the same basic structure:

- get the mouse coordinates and whether pressed
- check for intersection with a button
- if over a button and mouse pressed,
then perform the button function
- draw the button

This structure allows for nesting of buttons into hierarchies so that only the desired buttons (active on-screen) are interrogated and processed. For instance, subordinate buttons are not processed if the parent button is disabled, trimming processing time to a minimum.

When pressed, Buttons have two potential operation types: continuous or leading edge. For the continuous button type, the button's function continuously fires, providing extremely quick response times and fast accumulation of desired input. For the second type, leading edge, the button function occurs when first pressed; the button must be released before another push button can fire.

Along with how the button fires, the designer has complete control over button drawing, a powerful mechanism for customizing the interface. The best example of this power allows the designer to structure a button such that the underlying function, or multiple functions, enabled by the button become visible only when the mouse is over the encompassing button. In this way, complicated buttons become complicated only when intending to use them, simplifying the interface at all other times.

All button types inherit from the base class, though some of the derived classes that follow further expand on the base class capabilities. Figure 3-1 shows examples of most button type.

3.4.2.1 Rect Button

The Rect Button provides the intersection routines for a rectangular button. The fastest button type, Rect Buttons are general purpose buttons with no added capabilities from the base class, except the intersection routine.

3.4.2.2 Push Button

A Push Button operates based on the state of the pushed/not pushed value. Also, provides methods for an off message along with the base class message, depending on the state of the button. Uses for Push Buttons include selecting from a list, or as pre-defined action buttons that logically occur only one time per push.

3.4.2.3 Light Button

Light Buttons, derived from the Push Button, present the value of the pushed state as a built-in light on the button. The type of light remains for the designer to specify.

3.4.2.4 Hideable Button

Hideable Buttons are completely inaccessible unless enabled. All subordinate buttons become inaccessible as well, so Hideable Buttons either work in pairs or in groups that enable and disable each other. A good use for the Hideable Button: present a minimal button representing another larger button in its hidden form; pressing the minimal button enables the larger button before disabling itself. Then, when the user hides the larger button,

they are really enabling the minimal button and disabling the larger button, an efficient and simple process.

3.4.2.5 Display Button

Display Buttons are input receptacles in the form of a button. By clicking on them, they become enabled and ready for input from any appropriate source. For instance, clicking on a Display Button for a particular value allows the user to modify that value directly, with the old value stored in the built-in memory (for undo purposes). Display Buttons derive from Light Buttons because a little indicator light comes on when enabled for input. The display value can assume any type from numbers to text.

3.4.2.6 Menu Button

The last derived type of button, the Menu Button, is a special Push Button and Hideable Button mix that allows the designer to build button hierarchies into a menuing structure. The user activates the subordinate group of buttons only by pressing the main Menu Button; pressing again deactivates them. Any type of button may be included in the subordinate grouping, including another Menu Button, a built-in hierarchy generation facility.

3.4.3 Drawstring

The Button classes and other interface elements perform much of the formatting and printing of text and drawing of geometric and iconic figures to the screen using the Drawstring package of utilities. The package contains methods for drawing various gauges, information groups, and for formatting strings, as well as font management interfaces for setting or changing the current font.

I designed this package to handle all drawing of text and non-Performer graphics images to the screen, using calls within the client objects. Most of the methods for text/graphics management contained within Drawstring are not specific to the SBB, enabling usage by other ObjectSim applications. Unlike with stroke-vector window-dependent salable, the Drawstring fonts (using the Font Manager library) must be explicitly scaled and are not automatically scaled in relation to the window size.

3.4.4 Mac Sounds

The last object class, a simple and straightforward sound interface designed by Captains Wright and Soltz (13), provides programmable sounds generation keyed to the user interface, events within the simulation environment, or any other situation the designer desires. To use this class, a simple class instantiation is all that's required.

3.5 Conclusion

A unique and powerful user interface makes the Synthetic BattleBridge an easy-to-use and adaptable observer system in a simulation environment. The ObjectSim framework and the Object Manager network management utilities allow for fast performance and wide usage within the DIS simulation arena, without affecting anyone else or disturbing the environment. The trick remains on how to implement and integrate the features and design described above, the topic for the next section.

4. Implementation

I implemented the SBB using the C++ programming language. As the best choice for object-oriented designs, ObjectSim, the Object Manager, and all of the objects within the SBB were implemented using C++.

4.1 Overview

The overall implementation began with an example application created by Captain Mark Snyder. I then took over and modified the various objects to create the SBB, with capabilities and functions necessary for navigating in the DIS simulation environment.

The application begins by parsing the command line options to determine the modes, options, and operating environment for the SBB. Then, the application instantiates the SBB objects and functional components, and the initializes the same, before entering the repeated loop that queries the system and culls/draws the resulting image.

Unless specifically overridden, the ObjectSim renderer handles the culling and drawing of the Performer tree without explicit instructions from the application. Special cases, such as the 3D rendering mode, override the renderer to handle the drawing (not the culling).

The Performer framework for the renderer consists of three threads of control, all synchronized on the frame boundary, but otherwise independent: the Application thread for computation and control, the Cull thread for removing unseen polygons from the scene to be drawn, and the Draw thread for the actual drawing of the scene.

User input occurs for real-time response on the Draw thread, while movement and updates to the player positions and views occurs on the Application thread. Some applications benefit from unloading some of the

Application work onto the Cull thread, but the SBB does not; the Application thread for the SBB is the least used of the three threads because of the event-processing design of the *propagate* and *update* functions.

The whole Performer system is geared toward optimized graphics for simulations and rendering of images. Each of the above threads of control works best on its own processor, with the Object Manager on its own processor as well, so the SBB requires four processors to provide optimum performance.

4.2 Object Implementation

I begin with a description of the implementation of each of the objects within the SBB and how the objects interact. The line between simulation, interface, and other objects blurs in the implementation discussion, because they are all in the same system and the interface objects drive the simulation objects, or modify their behavior. I discuss the implementation of the simulation objects first and only the direct SBB simulation objects, leaving discussion of the implementation of ObjectSim and the Object Manager for others (12 and 10, respectively). Last, I discuss the implementation of the interface objects.

4.3 SBB Top Level Interaction

The user interface elements provide user input to the simulation objects. The main method for this is through message passing, with the function of the interface element to send the message to the appropriate simulation object to modify the state variables. The general form of a message follows:

- `void message( int value );`

The *message.h* header file contains the list of messages, their recipients and the various locations of the calls. This system works well for Performer/ObjectSim because of the shared memory requirements for the multiple threads discussed above. All calls not directly accessible through the object-oriented structure (due again to the shared memory requirements) are accessed in this manner.

For instance, to change the view to a particular view number (say 53), the calling code will execute the following:

- `SetViewIndex( 53 );`

The *sbb_app* code contains the *SetViewIndex* function that increments the *attached_index* to the desired view. Instead of directly just changing the *attached_index*, *sbb_app* can execute the required set of statements or functions required to cleanly change the view to the desired number, to save the state variables for the current view, or other bookkeeping duties. I followed this style of coding throughout the SBB application.

As for other types of user input (such as keyboard input, or data from the BOOM2M), I set up a structure within each potential input receiver for processing input as a message as well. So when the user presses a button on-screen or a button on the keyboard, the message is sent to the correct simulation object and treated the same, a clean and easy to follow coding style.

4.3.1 SBB Simulation Objects

The SBB begins operation in the SBB Application (*sbb_app*) by initializing the various objects and kicking them off. Most of the simulation object require initialization to allocate shared memory or do other initialization tasks.

The operation of the simulation objects (following initialization) occurs in a regular pattern, based on the three-thread control structure discussed above. The renderer, once kicked off to run, knows the views (set up during the initialization process) and calls each one in order of creation to *draw* once each frame. The renderer also calls the application that initialized it (in this case the SBB Application) once before and once after each call to the view draws. These calls (*pre_draw* and *post_draw*), allow graphics or input processing to occur in the application as well as the other simulation objects. The Cull thread operates as described above, with no explicit calls within the SBB.

On the App thread, the renderer calls the SBB Application *propagate* function. The SBB Application is responsible for moving the views and players within the scene during the *propagate* call. The majority of computation occurs in the Stealth *propagate* call for the Stealth view attached to the SBB View.

This is the general process for the simulation objects. The following sections describe the details of the implementation for each of the simulation objects.

4.3.1.1 SBB Application (sbb_app.cc)

The initialization process for the SBB Application includes the following tasks:

- parse the command line
- initialize Performer
- initialize the various Model Managers
- initialize the terrain and the animation effects
- initialize the head modifiers and 3D rendering view

- initialize the Stealth class and Shadow class
- initialize the Renderer
- initialize the SBB Net Manager
- initialize the Views

Once completed, the SBB Application becomes the focal point for all subsequent processing and the main link to the Renderer. During the Renderer initialization, the SBB Application's *init_sim* call allocates and initializes the various view players and terrain. Prior to the first drawing, the Renderer calls the *init_draw* and *init_draw_thread* functions which set the window for graphics processing and enable the various input queue entries.

Once in normal operations, on the Draw thread the *pre_draw* function checks for user input; on the App thread the propagate function updates the calls the SBB Net Manager to update the network player list and then processes user input for modifying the view number. Once modified, the appropriate Stealth player (corresponding to the attached view index) gets *propagated*.

4.3.1.2 SBB View (color_view.cc)

As shown in the design section, the SBB View (or views, depending on mode) gets the job of calling the *draw* function for the attached Stealth player (and Shadow player, if 3D) and the SBB Net Manager for the player trails. Using *wmpack*, a graphics library call, the final task of the SBB View is to decide on the type of write mask used to load only the appropriate bitplanes: red or blue for a red or blue mask, or all planes if an all mask.

4.3.1.3 Shadow (shadow.cc)

The shadow object provides the offset viewpoint for 3D rendering using the Player attributes *base_offset* (translation) and *base_rot* (rotation). By sharing the currently attached Stealth player position and orientation, the Shadow includes functions for on-line tuning of the red/blue bitplane visual separation and orientation. The *base_offset* and *base_rot* vectors (really *pfVec3s*) work in local coordinates so to get a separation distance between eyes, *esd*, merely add the *esd* to *base_offset*[PF_X]. Likewise, the rotations are relative, so to cross the separated eyes merely add a heading change to the *base_rot*[PF_H].

4.3.1.4 Stealth (stealth.cc)

The Stealth class, the most difficult and complex implementation, bears the majority of user input processing because the Stealth player holds the viewpoint position and orientation for the rendered scene. Again, once initialized, the Stealth players repeat the same set of functions, *draw* and *propagate*, continually refining their position and orientation in line with user input, and (if attached) the location of the player they are attached to.

Since they are on separate threads, the *draw* function can only talk to the *propagate* function across shared memory and the user inputs are processed using the message approach described above. These functions process user input and draw information to the screen, and move the viewpoint within the scene as further explained in the following sections.

4.3.1.4.1 Stealth *draw*

Each view contains state variables that preserve the location and status of the view as of last attachment or initialization. The user modifies the state of

the Stealth player by modifying these variables. The major state variables are, of course, the position and orientation of the viewpoint.

The *draw* function algorithm, chosen for speed and functionality:

- set up the screen for drawing
- determine if currently attached to player
- read the mouse position and whether pressed
- if BOOM is present, read the BOOM for position and orientation
- read the keyboard to see if keys pressed
- draw the gauges
- draw the interface buttons

The *draw* function uses the message approach described earlier to process the variety of input sources. Messaging allows the BOOM to trigger the same as the keyboard the same as an interface button on the screen, etc.

4.3.1.4.2 *Stealth propagate*

The *propagate* function processes the messages left from the *draw* function, including handling the attachment and detachment from the net players (due to the use of the *base_offset* and *base_rot* vectors) and the changing of mode from Plan Mode to Fly Mode (or vice versa).

Free of attachment, the Stealth player propagates along an arbitrary vector, the orientation, if the user requests movement. Movement along this vector is easy: simply add the xyz components of the vector in measured increments input by the user to the position vector. *Propagate* also recalculates the orientation vector when the orientation changes due to user input.

The *propagate* algorithm then becomes:

- get the current player list

- determine (if currently attached to player) if the player is still in the list, if not detach
- if changing modes, reset offsets
- if player change desired, process changing the attached player and attaching
- if detaching, process detach to reset attachment variables
- process position and orientation
 - case attached
 - > copy the attached player coordinates
 - > move based on the base vectors
 - case not attached
 - > move based on the position and orientation vectors
- process pre-defined view messages
- save the presently attached Stealth variables into shared memory (for use in the draw function, next frame)

The fastest performance occurs when no messages occur. Usually, only one or two of the functional elements need process and only when changing the state of the view, so the *propagate* function should never cause noticeable performance degradation.

4.3.1.5 SBB Net Manager (sbb_net_mgr.cc)

The initialization of the SBB Net Manager includes the following tasks:

- spawn the SBB Object Manager
- allocate and initialize the list of SBB Net Players

SBB Net Manager spawns the SBB Object Manager as its own process (and processor, if a processor is available). Allocation and initialization of the SBB Net Players includes insertion into the Performer tree, assignment of the

locator switch used to turn the locators on and off, and allocation of space for the representative model.

During normal execution, the SBB Net Manager's single-minded purpose is to get and update a list of the currently active net players. These functions, *get_net* and *update*, are called together by the SBB Application during its *propagate* function. *get_net* simply has the SBB Object Manager fill in the array of SBB Net Players with the appropriate information for the SBB. *update* processes each player in the active list for changes, as follows:

- if inserted:
 - map the player to the representative model
 - assign a locator type and color
 - initialize a new trail
- if deleted:
 - remove the representative model
 - turn off the locator and trail
- copy the position and orientation
- if round earth:
 - convert the position and orientation
- if time for another trail segment:
 - record the trail segment
- if locators state changed:
 - process the change (turn them on or off)

The SBB Net Manager includes a pointer to the list of players for other objects to use. The biggest user for the player list is the Stealth objects.

4.3.1.6 SBB Net Player (sbb_net_player.h)

The SBB Net Players (defining the information requirements for the SBB) are implemented as an array, giving a list of data to interface. The SBB Object Manager fills the list with current values, the SBB Net Manager copies the list when told to by the SBB Application (during the *propagate* function), and updates any players that have new or modified information. The Pause button suspends the update of the list; when un-paused, the players will jump to their new locations.

4.3.1.7 Locator (sbb_net_player.h)

Locators exist to simplify the view. Captain Soltz and I tried many varieties of air locators, including cylinders, spheres, pyramids, and cones, but each fell short of the elongated-sphere football shape used for the air locator. The football shape, with the circular ring at the front indicating the force type and direction of the vehicle, proved the most appealing and what I considered the most informative; the shape always shows the orientation and is visible from a much further distance than the other shape designs. At seventy-two (72) polygons, the air locator was by far the most expensive design, but worth the minor performance cost (about a 10% frame-rate hit, which could easily be made up by turning off the representative model rendering when locators are on).

For ground locators, the first design proved the best: the pyramid. Centered at the vehicle location but pointed out the front and flat on the back, the pyramid ground locator looked so good from the start (mirroring classic military operations maps with the arrows pointed toward the front), the only other design tried was a small box for non-offensive ground vehicles (orientation problems made this a poor choice).

The decision that all ground vehicles share the same type of locator (for simplicity), should probably be revisited in future versions of the SBB to determine if separate shapes for the offensive versus support vehicles provides a better ground image. Clearly, a jeep is not as important as a T-80 Main Battle Tank.

Other types of locators exist for missiles and bombs, both just really big transparent versions of the basic missile and bomb models. Force type indication comes from the color.

4.3.1.8 Trail (sbb_trail.cc)

For the trails, I implemented a simple static-sized linked-list, with the only task being to manage the head and tail pointers. The head points to the oldest point in the trail list, and the tail points at the newest point, so drawing the trail in the scene consists of the following steps:

- start drawing at the head position
- for each point *i* in the trail list between the head and the tail do:
 - draw from the *i*-1 position to the *i* position
- draw from the last position to the current position

The current position is the world coordinates of the player at each frame. In this way, even though the trails are only updated every few seconds, the trail connects all the way to the vehicle.

4.3.1.9 Model Manager (model_mgr.cc)

The Model Manager is a group of array management utilities specifically geared toward the mapping of an entity or group of entities to a common model index. These utilities consist of three major functions, available only following initialization:

<u>Function</u>	<u>Description</u>
get_entity_model	Fills the model_index field with the mapping of the entity (from the DIS standard or other enumerated list of entities) to a representative model.
get_entity_comment	Fills the entity_comment field with a string describing the entity. Useful for providing string names as well as numbers for types of entities. For example, the MIG_31 mapping of 72 has a comment of "MiG-31 Foxfire."
get_model_comment	Fills the model_comment field with a string describing the model. Useful for providing string names for models. For example, the F_15 model in "models/f-15/f15+a.flt" has the comment "F-15."

So, when describing the mapping of entities to models, the developer may output the easily understood message:

Mapping the MiG-31 Foxfire (entity: 72) to model F-15.

Because the Performer tree encourages multiple instantiations of a shared model (multiple leaves from the tree with the same root), the logical choice was to map the various entities to an array, and then manage the array such that each Model Manager managed only that portion of the array that concerns its models, as shown in the example set of Model Managers in Figure 4-1.

This simplified example shows the results of mapping ten separate models in four Model Managers. The separation of the models into the logical groupings was a by-product of the Model Manager design, originally there was only one Model Manager. The first column holds the local (internal) array index for the model, the second column is the global index, and the third is the model file name.

Terrain Model Mgr		
0	0	redflag.flt
1	1	neyland.flt

Build Model Mgr		
0	2	bridge.flt
1	3	building.flt

Net Model Mgr		
0	4	m-1.flt
1	5	dest-m-1.flt
2	6	f-16.flt
3	7	molniya.flt

Local Model Mgr		
0	8	loc_air_friend.flt
1	9	loc_air_foe.flt

Figure 4-1 A Model Manager Example

```
//Net Model Mgr

[models] 4

0 models /m-1.flt      M-1 Abrams Tank
1 models /dest-m-1.flt Destroyed M-1
2 models /f-16.flt     F-16 Falcon
3 models /molniya.flt  Molniya Satellite

[entities] 9

0      0      1      M_1
1      0      1      T_80
3      0      1      Jeep
4      2      2      F_15
5      2      2      F_16
7      2      2      MiG_29
9      2      2      Su_27
15     3      3      Molniya_Satellite
23     3      3      GE_Satellite
```

Figure 4-2 A Model Manager Data File

Internal to each Model Manager is another array that maps entity numbers (a unique identifier for each) to models (Figure 4-2). Models can have multiple mappings from entities.

4.3.1.10 Clock (clock.cc)

The implementation of the clock was very straightforward. Really, the clock only formats the time sent to it, returning a properly formatted string or the input time filled in the "HH:MM:SS" format.

4.4 Interface and Other Object Design

With the implementation of the simulation objects complete, I turn next to the implementation of the interface objects: the Buttons, Drawstring, and Sounds. The Buttons proved difficult at first, since very little has been done with transparency, but once the first few Button types were developed and the majority of design considerations complete, the number and complexity of buttons grew rapidly. For the Drawstring implementation, I slowly developed the utilities, one at a time as needed, to meet the various challenges of the interface design. The Sounds integration was a snap.

4.4.1 Player Button (play_button.cc)

The player button is a specialized case of the SBB Button, implemented separately, with only the absolute minimum of data, to provide optimal performance. The description of the implementation for the SBB Buttons will also describe the implementation of the player button, only with more data.

4.4.2 SBB Button (button.cc)

As stated in the design section, all of the buttons operate on the same basic structure:

- get the mouse coordinates and whether pressed
- check for intersection with a button
- if over a button and mouse pressed,
then perform the button function
- draw the button

This simple structure provides a power and elegance to the Button class that appears (at first glance) obvious and straightforward. While I admit that the result was both of these things, the development was neither. I spent much of my time during the development and implementation of the buttons working on a tighter and tighter algorithm for the base class. This devotion to the base class enabled later derivatives to inherit most of their behavior and attributes. This allowed rapid development and easy integration of subsequent additions to the Button class. The member data and functions for the Button class follows:

<u>Object</u>	<u>Data</u>	<u>Functions</u>
Button	position over & select parent & sub title & message colors press_type select_lock value & call_back	get/set for data members ----- draw_fn draw_over draw_not_over ----- intersect pressed draw

The member data and functions for the derived Buttons follows:

<u>Object</u>	<u>Data</u>	<u>Functions</u>
Rect Button	<Button data> left top right bottom	<Button functions> set_rectangle ----- intersect
Push Button	<Rect Button data> pushed off_title off_message	<Rect Button functions> get/set_pushed ----- pressed draw
Light Button	<Push Button data> light_rect light_type light_justification	<Push Button functions> set_light
Hideable Button	<Rect Button data> enable	<Rect Button functions> set_enable ----- intersect pressed draw
Display Button	<Light Button data> display_value display_prec display_position display memory	<Light Button functions> set/cat/uncat/clear_display set/swap/clear_memory
Menu Button	<Push Button data>	<Push Button functions> ----- intersect pressed draw

The overridden functions for *intersect*, *pressed*, and *draw* allow the derivative class to modify the behavior of the base class. The overall function to *draw_buttons* handles calling the *intersect*, *pressed*, and *draw* functions for each button in the array, in order.

The base class *intersect* algorithm:

- if mouse coords > xpos - delta_x and
 - if mouse coords < xpos + delta_x and
 - if mouse coords > ypos - delta_y and
 - if mouse coords < ypos + delta_y then
 - mouse is over button
- if mouse is over button and button not locked then
 - button is selected
- call intersect for sub buttons

The base class *pressed* algorithm:

- if button is selected then
 - case press_type
 - when continuous
 - > execute call_back with value
 - when leading_edge
 - > if mouse has changed state then
 - execute call_back with value
- call pressed for sub buttons

The base class *draw* algorithm:

- if over the button then
 - draw_over the draw_fn
- else
 - draw_not_over the draw_fn
- call draw for sub buttons

The *draw_fn*, *draw_over*, and *draw_not_over* functions shared the same basic structure, as well. An example follows that draws a titled rectangular button when not over and a circle on the rectangle when over (with no title).

draw_fn:

- if filled
 - draw filled fn
- else
 - draw unfilled fn

draw_over:

- draw blended transparent rectangle background
- draw_fn (select) => fn = circle

draw_not_over:

- draw blended transparent rectangle background
- draw_fn (false) => fn = title

The final implementation is as simple as the Buttons are powerful. The base class demonstrates the majority of the attributes of the derived classes, so I will only discuss the differences in each instead of rehashing the similarities.

4.4.2.1 Rect Button : SBB Button

The Rect Button provides the intersection routines for a rectangular button. The intersection tests the left, right, top, and bottom. All other base class data and functions remain the same.

4.4.2.2 Push Button : Rect Button

A Push Button operates based on the state of the *pushed* value, and has an *off message* and *title*. So the Push Button has the added task of toggling the

pushed value when *pressed*, and checking the *pushed* value during draw to see if the regular (on) *title/message* or the *off_title/message* is displayed.

4.4.2.3 Light Button : Push Button

Light Buttons present the value of *pushed* as a built-in light on the button. This light (setup using the *set_light* call), occupies part of the button automatically based on the *type* and *justification* of the light. Simply, the *light* and *title* are treated as separate buttons linked together to form one.

4.4.2.4 Hideable Button : Push Button

Hideable Buttons override all of the *intersect*, *pressed*, and *draw* functions to wrap the *enable* check around them to preclude the wasting time on the sub buttons or on the button itself if *enable* is false (hidden). The really tricky part of the Hideable button is the synchronization of enabling the other buttons.

Originally, I did not disable selection of other buttons when hiding buttons (setting *enable* = false). So if the Hideable button enables a button directly underneath and the button directly underneath enables the Hideable button, the result is a very fast cycling between the buttons (because the frame rate is usually much faster than the duration of the click of the button), with random results. I fixed this problem by disabling selection of another Hideable button until the mouse button is released.

4.4.2.5 Display Button : Light Button

This is one of my favorite buttons, because of the power, utility, and simplicity of the implementation. The Display Button allows the designer to combine an output with an input in such a way that to change a value the user just selects the value to change directly from the display, popping up a

keypad to enter the numbers. This would be perfect in a simulation environment where the user has a "Virtual Hand" able to reach out and touch the interface elements.

I use the Display Button to implement the keypad display (hence the name) so the user can directly enter numbers into various data fields in the interface, such as player or view number. Extension to direct entry of position coordinates or orientation values would be fairly straightforward, I simply ran out of time.

4.4.2.6 Menu Button : Push Button

The Menu Button groups other buttons into a menu. Selection shows the subordinate buttons causing the Menu Button to override all of the *intersect*, *pressed*, and *draw* functions. The change is actually quite small: simply test the *pushed* value prior to executing each function for the sub buttons. Elegant.

4.4.3 Drawstring

The Button classes and other interface elements perform much of the formatting and printing of text and drawing of geometric and iconic figures to the screen using the drawstring package of utilities. The package contains methods for drawing various gauges, information groups, and for formatting strings, as well as font management interfaces for setting or changing the current font. I designed this package to handle all drawing of text and non-Performer graphics images to the screen, using calls within the client objects. Most of the methods for text/graphics management contained within Drawstring are portable to other applications.

4.5 Problems and Praise

The majority of problems I encountered had little to do with the implementation of the design, most had to do with modifying other's code or understanding and using the Performer and GL libraries. The ObjectSim framework was a dream to work with; elegant in design and flexible in practice, ObjectSim is a tremendous achievement for Mark Snyder in particular, and AFIT in general.

I took a week (no kidding, a full week) in May to sit down and try to understand the ObjectSim design and code, and even though it may not show, the week was extremely well spent. Unlike others who decided to rely on Mark Snyder (who I admit came through for most as patiently and quickly as possible), I wanted to understand and really test the limits of ObjectSim. I found no major drawbacks or places I would do differently.

4.6 Conclusions

The SBB implementation may have flaws, as expected for the rapid-prototyping approach, and the significant amount of code involved, but I designed on paper and tried to work out the potential problems well before sitting down to write the code. I know the future holds a lot of change for the SBB and the version next year may little resemble this year's version, but I did my best and am very proud of the results.

5. Results, Future Work, and Conclusions

5.1 Results

An unintrusive system designed to observe and inform on the DIS environment, the SBB meets the need for an easy to use and consistent situational awareness enhancement system. The next section holds the performance results achieved as of early November 1993, followed by a few sections summarizing the overall look and feel of the SBB.

5.1.1 Performance

The system embodies a significant improvement in speed over the previous version of the SBB, up to five to ten times faster on comparable equipment. With low numbers of players (<100), the frame rate consistently exceeds 20 frames per second (fps) and usually pegs at 30 fps. At 500 players, the frame rate hovers around ten (10) fps; over 1000 players, and the frame rate drops to around six (6) fps. Unfortunately, we could never test much higher than around 1200 to 1500 players because the test software caused numerous network difficulties resulting in frequent drop-offs from the network and poor test results.

5.1.2 User Interface

The first thing to notice about the SBB User Interface: it's transparent (you can see through it). As discussed above, the Button class allows me to tailor the SBB interface to many different possible configurations. Where others have used the Forms library (18) to develop their interfaces, mine is a direct route using direct graphics calls and optimized code to allow the user of the SBB to see through the interface for a good view of the environment without

sacrificing screen real estate (see Figure 5-1) or incurring any significant performance cost.

5.1.2.1 Transparent Dynamic Display Buttons

In fact, the buttons cause an insignificant slowdown during normal operations. With all possible buttons active, and much activity, the buttons incur a slight cost of about one fps, well below a comparable interface implemented from the Forms library.

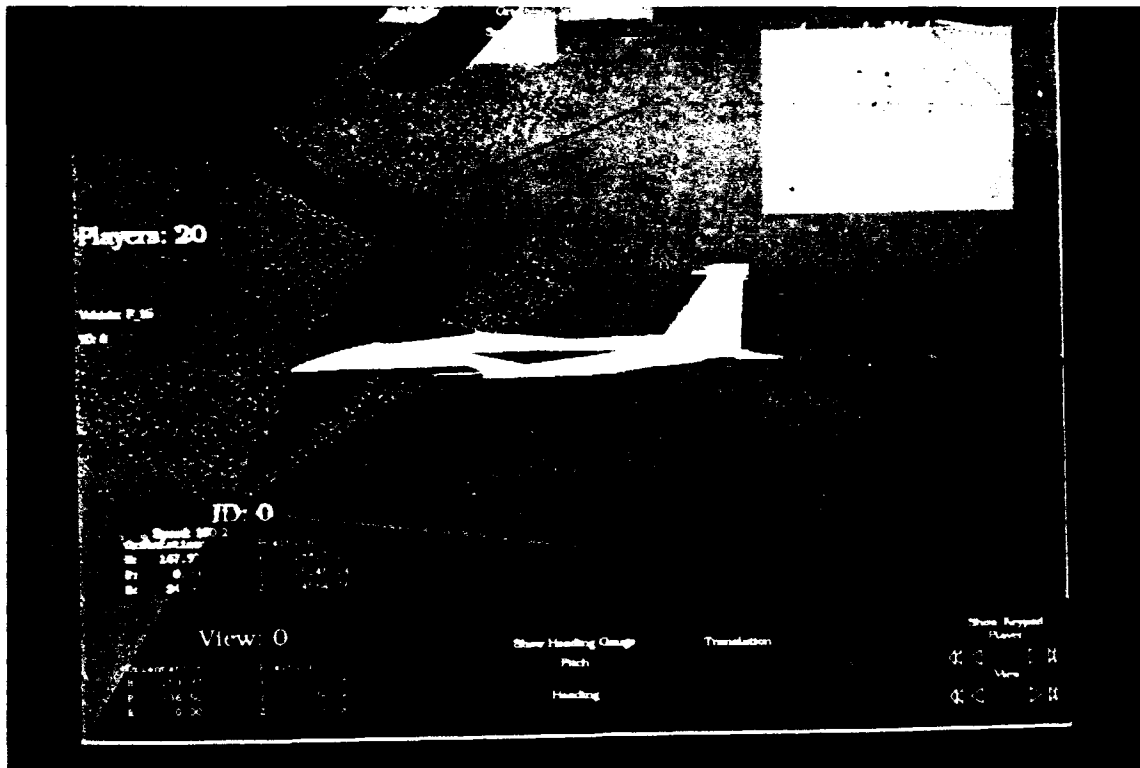


Figure 5-1 The Transparent User Interface

My intent throughout has been to maximize user control while maintaining a clear and uninterrupted view. The different interface areas become active only when the user places the mouse or other input device over the desired area. In this way, the display reverts to the simplest form when the mouse is not over the area, allowing the best view possible into the environment behind at all times.

The virtual buttons respond to user input very quickly. Each interface area also includes messages (like on-line help) to explain the operation of each button within the area.

Why did I choose to implement the interface as transparent dynamic display buttons instead of in the traditional opaque way? The answer goes back to the non-interface discussion of paragraph 2.2.2; the interface should intrude as little as possible on the user's view of the environment, a non-interface. Traditional static interface elements offer similar functionality but unquestionably intrude on the environment. Also, intrusion constrains the design to a smaller area than possible were the interface minimally obtrusive.

I also attempted something traditional static interface elements don't usually do: I made the buttons reconfigure when the user passes the mouse over them. The reconfiguration (pre-defined by the software developer) shows lower and lower levels of functionality and the active button that would fire if the user pressed the mouse button. Providing this feedback, along with the messages capability, helps the user focus quickly on the desired area.

The Button class and code care not if the resulting interface is the opaque traditional static interface or the transparent dynamic display interface, these features came from the structure and flexibility of the class and the manipulation of this structure to achieve the desired results. The Button class is more an Interface toolbox than I at first intended.

5.1.2.2 Views and State

The view button allows the user to enter or cycle to any desired view using mouse input (See Figure 5-2). Upon selection of the view display

button, the Virtual Keypad becomes active and ready for input; the user can select the view number (the total number of views is set during initialization) or enter data into any other display field using the keypad.

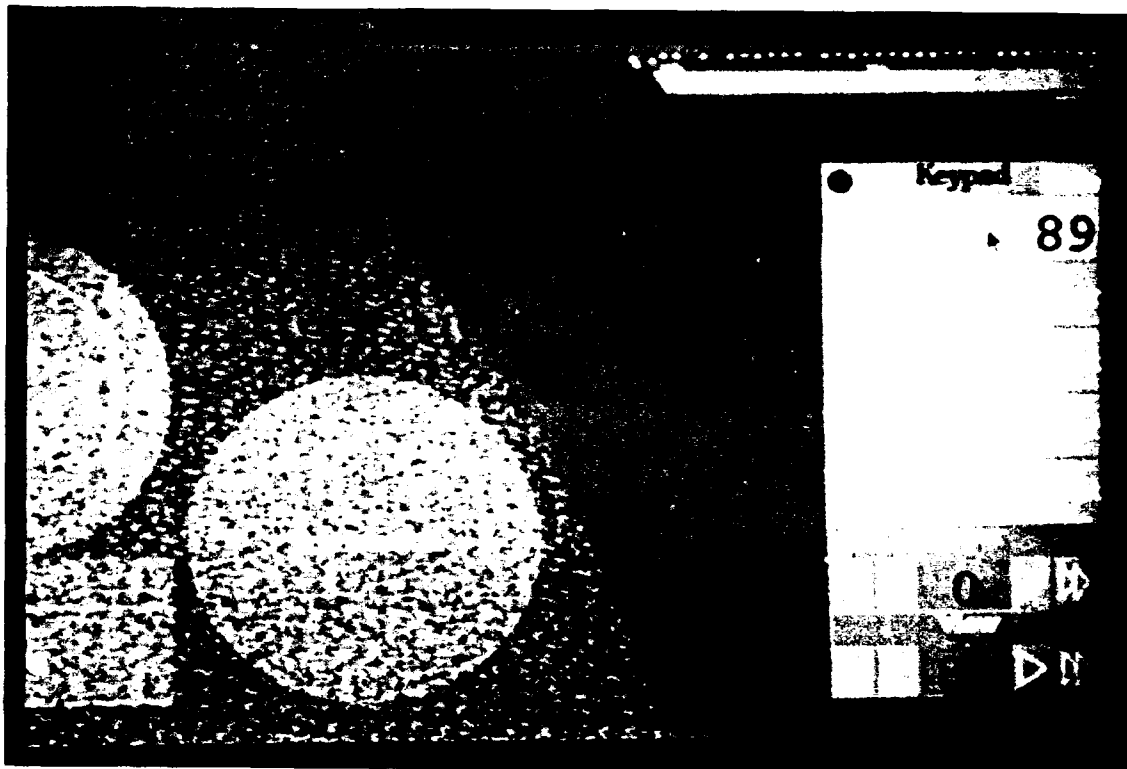


Figure 5-2 The View & Player Buttons

Each view holds the state information throughout the simulation. If the user leaves a particular view, all the information about that view is maintained, including location, orientation, attachment, and player information. The view becomes inactive, but ready to activate on the next frame.

Extension of the view button could include previews or multiple views (sub windows) within the main window. Other extensions could include management of display information using a view control panel.

5.1.2.3 Players

Like the view button, the player button (also in Figure 5-2) allows the user to select a particular player in the simulation environment using the player's position within the active array. Information presented upon selection includes the object manager ID and the type of entity of the player.

The user can also attach to players by selecting the player's icon from the Information Grid as shown in Figure 5-3. Each player within the area of the grid is represented using a symbol and a color, as described in the previous chapters.

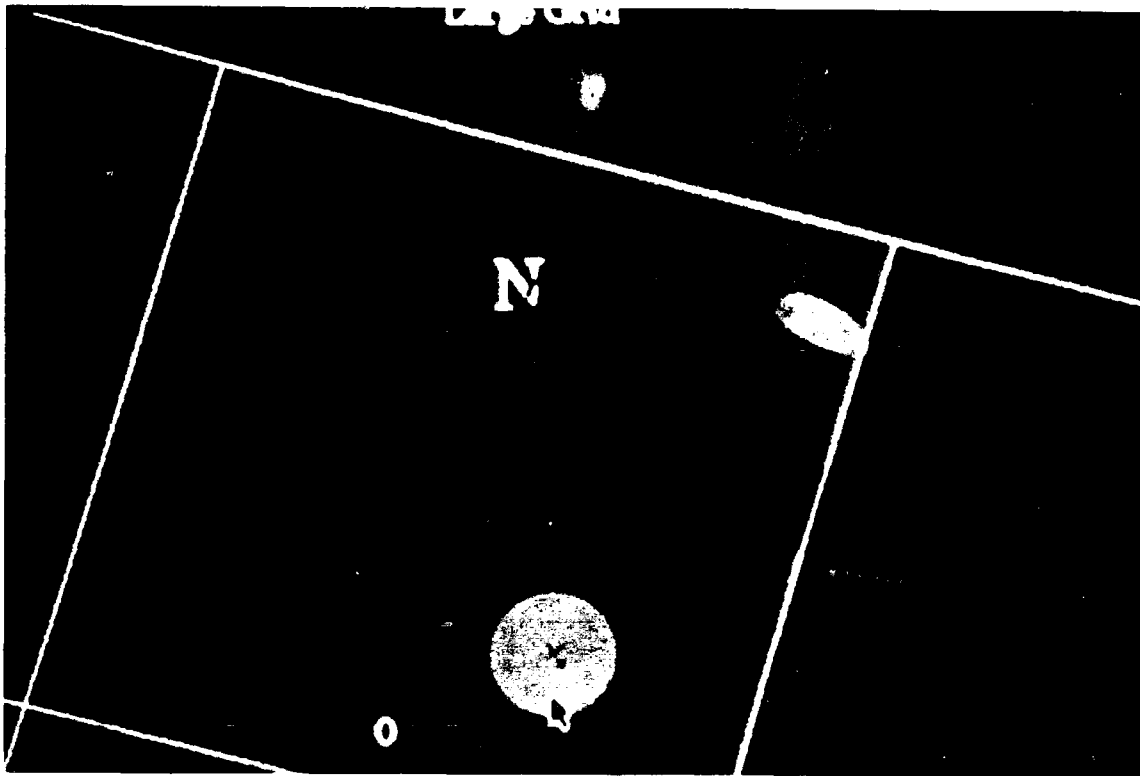


Figure 5-3 The Information Grid

Selection of players using the Information Grid should be extended to include selection of players and objects within the scene using the mouse or some other input device, such as a Virtual Hand (the specification as described above). Selection in this way requires transforming the two-

dimensional coordinates of the mouse/VH into the three-dimensional world coordinates, and then searching for the object that the mouse/VH intersects, not an easy problem based on the constantly changing environment and the required number of objects in the scene.

5.1.2.4 Navigation (Position and Orientation)

The user's view within the environment depends on position and orientation. Modifying either will modify the user's view. The on-screen controls that allow the user to modify the view are shown in Figure 5-4 (note that the user cannot modify the roll within the SBB, because of the possible disorientation this would cause). These controls are based on the standard aircraft control conventions, as follows:

Orientation:

Heading	Pitch	Roll
0 to 360 deg	-90 to +90 deg	0 to 180 deg

Position:

North	South	East	West
0/360 deg	180 deg	90 deg	270 deg

The Heading Gauge presents a look to the interface that the user will easily recognize and interpret. The Translation area (shaped like a throwing star) allows the user to manipulate the six direction of motion buttons and the stop button within a single area. The Heading and Pitch controls allow fast or slow operation, and like all of the dynamic display buttons, the minimal interface when not in focus.

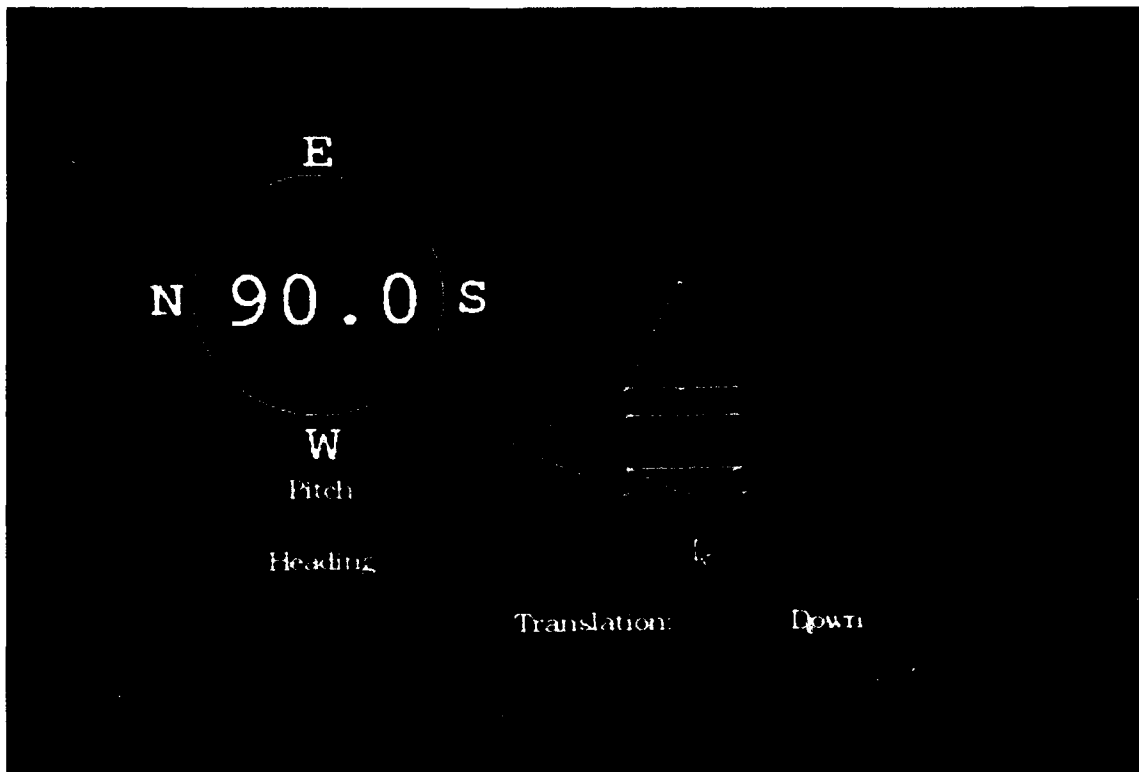


Figure 5-4 The Navigation Controls

5.1.2.5 Interface Manipulation

Knowing that each user may have a particular interest or preference, I specifically designed the interface to allow each user to modify and enhance the interface as desired. The user can hide or enlarge buttons or areas, selectively display the otherwise hidden button characteristics, or remove all interface elements from the display. The minimal interface includes only the menu to display the other elements, and in this minimal configuration, the user can still provide input using the keyboard equivalents for the interface areas.

5.2 Future Work

Many challenges lie ahead for future SBB and ObjectSim developers. Mainly, I see a great need for alternate ways of processing player information

in the environment, specifically the flat-earth/round-earth conversions and dead-reckoning; our current algorithms begin to see problems in the hundreds of vehicles, while with thousands the performance becomes unbearably slow. Before I get to this problem, and potential solutions, let me outline other problems not completely addressed in this thesis.

5.2.1 Voice Interaction

The previous iteration of the Synthetic BattleBridge used Voice Navigator software to enable the user to make one of several commands using voice only. I did not implement this. I chose instead to implement virtual buttons (as described in chapters three and four) with the future developer of the SBB extending these buttons into voice command processing (with visual feedback), using better voice recognition software. Ideally, the user would speak the name of a button (or part of the name) and the button would highlight and select itself as if selected by the mouse.

I think these visual cues combined with the power of a good voice recognition software may warrant research as a topic all to itself, if it hasn't already. The integration of such a system, using the present framework, would not be difficult and would apply to many of the Virtual Reality systems being developed at AFTT.

5.2.2 In Scene Buttons

Another useful attribute of the flexible Button class would be the ability to attach a Button to specific places or objects in the scene. I suggest further development of the Button class to allow the planting of elements in the scene, instead of just in the formal interface. The designer could associate a Button to a particular spot in the scene, say for a console switch or button.

Seeding the environment with selectable objects that look, behave, and act correctly is the essence of immersive technology.

The Virtual Cockpit (14 and 15) could greatly benefit from adding "real" cockpit control buttons (for the radio, weapons selection and arming, etc.) on the VC console that remain anchored to the correct spot, even should the viewpoint change. Maybe instead of transparency, the VC could use texture maps of the actual buttons and switches, and volumetric analysis of a Virtual Hand to determine intersection.

The volumetric (3D) intersection techniques are a natural extension of the current planar (2D) intersections and would require no other changes to employ; the draw function would describe a three-dimensional object, instead of a two-dimensional object, again a 3D extension of the current 2D design. Volumetric analysis, assuming accurate position data, could replace selection of objects using gestures such as swirling of the finger to denote selection.

5.2.3 3D Sound

A useful audio interface is noticeably lacking from the SBB. Due to time constraints, I made no effort to incorporate the spatially correct sound hardware received in September 1993. This hardware allows the user to hear sounds coming from (virtual) behind, or from the side, as in reality.

Used for messages or warnings, spatial audio would enhance the immersive feeling of the environment and provide means for additional information presentation.

5.2.4 Selection of Objects in the Scene

I suggest a menu as a means of navigating through the DIS standard to allow for selection of desired classes or even individual vehicles, countries, or

models. This set of menus could be developed using my Menu Button class described above. The context of a menu selection would depend on other preceding menu selections, each using a common set of sub-menus; a context-sensitive waterfall approach.

The repetitive structure for such a selection lends well to the idea of a menu, even to the point of an on-line DIS standard application that could drive the object manager.

5.2.5 Efficient Algorithms for Object Processing

As the number of objects in the simulation space continues to increase, I see a great need for alternate ways of processing these objects. Our current algorithms begin to see problems in the hundreds of vehicles, while with thousands the performance becomes unbearably slow.

My colleagues and I (during the lulls in our night shift) have discussed several ideas on how to combat these problems (though I am solely responsible for any errors or omissions in the presentation of these ideas). A summary of each problem with a suggested solution follows.

5.2.5.1 Too Many Objects/Time Slice Processing

The problem of too many objects *will not go away*. Today thousands, tomorrow tens of thousands, etc. The only solution that appears to me to have a chance comes from Mark Snyder, glorious developer of the ObjectSim framework and all-around Performer guru. He says that he has developed (recently) a technique to process these objects based on a frame modulus.

That is, use a round-robin (or priority) scheme to process the objects in the scene so that the frequent slow downs associated with object processing occur spread out over all frames. This would slow down the overall frame rate, but

the appearance of smooth operation and consistent frame rate performance may be more pleasing to the user.

Another (more expensive) solution would require many more processors. The idea involves spawning a new Object Manager process on another processor to scan a particular portion of the network whenever the count of objects reaches a certain level. The dynamics of this approach appear daunting to begin with, but think of the huge benefits and possible uses for dynamic managers of data and stress-tailoring of the simulation environment.

A good network scanning algorithm could allow for as many vehicles in the network as processors in the computer. Think of it as buying more object capacity; a four processor Onyx could handle X number of objects, an eight processor Onyx say 2X, etc. Of course, the resulting numbers would not be exactly linear due to the added management time, but the idea could definitely prove possible and the ticket to very high numbers of objects in the simulation (for a price).

5.2.5.2 Culling/Drawing Bottleneck/Level of Detail Management

Currently, our model management (within the models themselves) leaves a great deal to be desired. My development of the Model Manager class was the first step in the right direction, but more needs to be done. We currently don't really do any manipulation of the LOD Range capabilities supplied through Performer, we simply display the models as they exist within the Flight file. No one actively manages these files, though some have tried, and we still do not have the right library of models, at least not the lower and lower resolution of detail that LOD processing requires to work effectively.

At more than a few thousand meters, none of the smaller vehicles can be seen at the screen resolutions. We see only those models made arbitrarily large (or using locators), so take them out of the Performer tree (using pfSwitches of LOD Range values) at the long ranges so no processing time is required.

5.2.5.3 Unseen Objects Require Processing Too/Range Processing

The last problem I discuss concerns unseen objects, or objects too far away to see due to distance or orientation. Before processing, figure a way to determine if the player effects the scene. I know this sounds like culling, but I mean beyond that, or before that. If an initial test determines that the object has no affect on the scene, put it in the Process Later stack of objects, which trades frame rate for accuracy outside the current reasonable viewing horizon. The internal representation of the state of the whole virtual world may contain inconsistencies this way, but the apparent view to the user improves and the performance improves.

5.2.6 Abstract Visualization of Force Distributions

Another potential solution for the object processing problem may lie in the full abstraction of the battle space and the objects that represent this space. Since the abstract nature of information visualization allows for greater flexibility within the SBB for displaying a view into the simulated environment, the rendered image can take on unreal qualities that aid the digestion of important information.

The vehicle locators exhibit this abstract quality (see the discussion above). The extension of this idea to the entire space would greatly enhance the

information presentation capabilities of the SBB, as briefly described in the following paragraphs.

5.2.6.1 Grid Architecture

The main grid would consist of the terrain and related natural and artificial surfaces such as trees, buildings, bridges, etc. in one of many possible forms, i.e., terrain height. Other grid layers may consist of Friendly, Opposing, or Neutral force distributions.

Use of the grid for information processing precludes the rendering of the terrain and the other objects in the scene and would greatly reduce the rendering processing required. This grid could also be spawned as its own process and displayed on an as required basis much like the current location grid described above.

5.2.6.2 Terrain Partitioning

Plotting the height (above some arbitrary default height) over a grid, by color of course, would result in squares of potential rough terrain (steep changes in color) and smooth terrain (little change in color). By analyzing the grid, terrain influences on the force distributions may become clearer. Initialization of the grid during startup to determine the appropriate values may add to the start up time, but this time occurs only once per simulation.

Special color coding of the terrain to indicate buildings, towns, or population would again enhance the information capabilities.

5.2.6.3 Forces Partitioning

For vehicles/players in the simulation, an individual grid color may consist of the number of a certain type of aircraft within its confines. Or the color may indicate the relative strength of forces within the grid section. The

possible permutations using colors and forces/location of vehicles and terrain makes this an endless lesson in deriving information from the battle space.

5.2.6.4 3D Chessboard

By partitioning Friendly and Opposing vehicles into separate grids, the overlap, based on additive transparent colors, shows the overall force distributions; a 3D chessboard to manipulate and modify (possibly using Fuzzy Logic techniques). Or maybe the grid could take on an artificial intelligence aspect to predict future distributions. Viewed in an immersive environment, this could become a powerful information presentation tool.

5.3 Conclusions

The Synthetic BattleBridge has come a long way since I first started to explore the world of Virtual Reality applied to the presentation and visualization of information in a large battle space. I strived to maintain a consistent, easy to use, and minimally obtrusive interface while providing powerful navigation and information tools to the user. Ease of use (at least in the sense of self explanatory) with more power continues to drive Virtual Reality environment research.

The transparent dynamic display interface, though an intuitive improvement on traditional approaches requires several human-factors studies and extensive user testing to validate its effectiveness and ease of use gains. As with any human-computer interaction, if people can use it, they will use it.

References

1. Steuer, Jonathan. "Defining Virtual Reality: Dimensions Determining Telepresence," *Journal of Communication* 42(4): 73-93 (Autumn 1992).
2. Regian, J. Wesley, Wayne L. Shebilske, and John M. Monk. "Virtual Reality: An Instructional Medium for Visual-Spatial Tasks," *Journal of Communication* 42(4): 136-149 (Autumn 1992).
3. Biocca, Frank,. "Virtual Reality Technology: A Tutorial," *Journal of Communication* 42(4): 23-72 (Autumn 1992).
4. Lanier, Jaron, and Frank Biocca,. "An Insider's View of the Future of Virtual Reality" *Journal of Communication* 42(4): 150-172 (Autumn 1992).
5. Benedikt, Michael, editor. *Cyberspace*. The MIT Press: Cambridge MA, 1991.
6. Bricken, Meredith. "Virtual Worlds: No Interface to Design," Chapter 13 in *Cyberspace*. Ed. Michael Benedikt. The MIT Press: Cambridge MA, 1991.
7. Jacob, Robert J. K.. "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transactions on Graphics* 5(4): 283-317 (October 1986).
8. Falby, John S., Michael J. Zyda, David R. Pratt, and Randy L. Mackey. "NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation," *Computers & Graphics* 17(1): 65-69 (1993).
9. Haddix II, Rex G. *An Immersive Synthetic Environment for Observation and Interaction with a Large Volume of Interest*. MS thesis, AFIT/GCS/ENG/93M-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1993.
10. Sheasby, Steven M. *Management of SIMNET and DIS Entities in Synthetic Environments*. MS thesis, AFIT/GCS/ENG/92D-16, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.
11. Soltz, Brian. *Graphical Tools for Situational Awareness Assistance for Large Battle Spaces*. MS thesis, AFIT/GCS/ENG/93D-21, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.

12. Snyder, Mark. *ObjectSim - A Reusable Object-Oriented DIS Visual Simulation*. MS thesis, AFIT/GCS/ENG/93D-20, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.
13. Wright, Charles, and Brian Soltz. *Macintosh Sound Generation Facility 2.0*. Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 15 November 1992.
14. Erichsen, Matt. *Weapon System Sensor Integration for a DIS-Compatible Virtual Cockpit*. MS thesis, AFIT/GCS/ENG/93D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.
15. Gerhard, William. *Weapon System Integration for the AFIT Virtual Cockpit*. MS thesis, AFIT/GCS/ENG/93D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.
16. Gardner, Michael. *A Distributed Interactive Simulation Based Remote Debriefing Tool for Red Flag Missions*. MS thesis, AFIT/GCS/ENG/93D-09, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.
17. Kunz, Andrea. *A Virtual Environment for Satellite Modeling and Orbital Analysis in a Distributed Interactive Simulation*. MS thesis, AFIT/GCS/ENG/93D-14, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1993.
18. Overmars, Mark H. *Forms Library: A Graphical User Interface Toolkit for Silicon Graphics Workstations v2.1*. Department of Computer Science, Utrecht University, Utrecht, the Netherlands, November 10, 1992.
19. FakeSpace Labs. *BOOM2/BOOM2C Operations Manual*. April, 1992.
20. Silicon Graphics. *Performer Programming Guide*. September, 1992.

Vita

Captain Kirk Wilson entered the United States Air Force in 1983 as a Computer Operator. Selected to attend Arizona State University under the Airman's Education and Commissioning Program (AECP) while stationed at Sunnyvale AFS, California, Kirk earned a Bachelor of Science in Aerospace Engineering in May 1989.

Kirk joined the Joint Tactical Information Distribution System (JTIDS) Joint Program Office for his first assignment following graduation from Officer Training School. He gained acceptance to the Air Force Institute of Technology in 1992 and completed his Master of Science in Computer Systems in 1993.

Kirk hopes to never see Maui again.

REPORT DOCUMENTATION PAGE			Form Approved OMB No 0704-0188	
Public reporting burden for this report is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis		
4. TITLE AND SUBTITLE Synthetic BattleBridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment			5. FUNDING NUMBERS	
6. AUTHOR(S) Kirk G. Wilson, Captain, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93D-26	
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES) ARPA/ASTO 3701 North Fairfax Drive Arlington, Va 22203			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) With shrinking budgets and fewer personnel, future military training will rely heavily on simulated environments. The goal for this training is to reduce cost while maintaining readiness and unparalleled capability for all levels of military command. This thesis effort, the Synthetic BattleBridge (SBB), provides a real-time simulated environment for military commanders to observe on-going computer simulations of varying participation levels and helps them wring the most from a simulation. The SBB, designed for higher ranking personnel with little time to spend learning how to run the system, must exhibit three capabilities: ease-of-use, long-term retention, and adaptability. Based on the ObjectSim framework and the Object Manager network management software, the SBB includes a unique transparent interface with dynamic display elements that ensures minimal intrusion. The SBB provides multiple views and direct attachment to simulation players, along with an information grid showing the distribution of forces within the current environment.				
14. SUBJECT TERMS User Interface, Situational Awareness, Synthetic Environments, Object-Oriented, Computer Graphics			15. NUMBER OF PAGES 95	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	