
# COLLABORATIVE WORKSPACES WITHIN DISTRIBUTED VIRTUAL ENVIRONMENTS

THESIS

William David Wells, Capt, USAF
AFIT/GCS/ENG/96D-28







DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-28

# COLLABORATIVE WORKSPACES WITHIN DISTRIBUTED VIRTUAL ENVIRONMENTS

**THESIS**

**William David Wells, Capt, USAF**
**AFIT/GCS/ENG/96D-28**

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

AFIT/GCS/ENG-96D-28

# COLLABORATIVE WORKSPACES WITHIN DISTRIBUTED VIRTUAL ENVIRONMENTS

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

William David Wells, B.A.E.
Captain, USAF

December 1996

# Acknowledgments

Performing research and creating a thesis document is as much about inspiration as it is perspiration. Thus, I would like to thank a number of people for giving me the inspiration to keep going when all hope seemed lost. First, I would like to thank my mother and father for standing by me and listening to me complain about technical issues that you didn't understand. Hopefully, this document will show you what your son has been doing for the last eighteen months. Second, I would like to thank my best friend, my sister Marianne, for just being so cool and giving me some perspective about this whole ordeal. Your personal struggles and accomplishments make mine seem insignificant. I would also like to thank Gary Hicks for introducing me to computers and hacking code. You opened up a world for me that appears to have no bounds. Thank you for giving me an outlet for my creative energies. Closer to my survival at AFIT, I would like to thank Cheryl Colombi. You didn't know it, but your smile, good humor, and willingness to listen made the first half of my stay at AFIT a very enjoyable experience. I owe you a trip to Disneyland. Finally, a quackazillion thanks to Steven Sheasby. Your guidance, both technical and personal, went far beyond the expectations of a contractor. Thanks for being one of the "good guys."

# Table of Contents

# List of Figures

# List of Tables

# ABSTRACT

In warfare, be it a training simulation or actual combat, a commander's time is one of the most valuable and fleeting resources of a military unit. Thus, it is natural for a unit to have a plethora of personnel to analyze and filter information to the decision-maker. This dynamic exchange of ideas between analyst and commander is currently not available within the distributed interactive simulation (DIS) community. This lack of exchange limits the usefulness of the DIS experience to the commander and his troops. This thesis addresses the commander's isolation problem through the integration of a collaborative workspace within AFIT's Synthetic BattleBridge (SBB) as a technique to improve situational awareness. The SBB's Collaborative Workspace enhances battlespace awareness through CSCW (computer supported cooperative work) enabling communication technologies. The SBB's Collaborative Workspace allows the user to interact with other SBB users through the transmission and reception of public bulletins, private email, real-time chat sessions, shared viewpoints, shared video, and shared annotations to the virtual environment. Collaborative communication between SBB occurs through the use of standard and experimental DIS-compliant protocol data units. The SBB's Collaborative Workspace gives the battlespace commander the widest range of communication options available within a DIS virtual environment today.

# Collaborative Workspaces Within Distributed Virtual Environments

# 1 . Introduction

## 1.1 Overview

For many years, the military has taken advantage of computers to simulate battlefield scenarios. Computer simulations have been vital as training and planning tools because they require a relatively small amount of resources. Currently, the military is exploiting the virtual reality capabilities of computer simulations to create more realistic simulated battlefield conditions. Distributed virtual environment (DVE) simulators provide a means whereby many remote players can interact on the same battlefield. In 1989, the Distributed Interactive Simulation (DIS) workshop was created to develop standards by which multiple DVE simulations could play together. "The primary mission of DIS is to define an infrastructure for linking simulations of various types at multiple locations to create realistic, complex, virtual worlds for the simulation of highly interactive activities [DISV94]." These DIS simulations allow the military player to insert himself into a virtual environment and interact with other human-controlled entities.

The Synthetic BattleBridge (SBB) is a DIS application specifically designed for a decision-maker to monitor, assess, and analyze the activities within a virtual battlespace [WILS93]. One of the limitations of the SBB, and nearly all other DIS applications, is the inability to communicate directly with other DIS players in the exercise. This lack of communication prevents collaboration between users in the virtual battlespace. The capability to collaborate with other users is essential to managing the activity within a battlespace because commanders are dependent upon their staffs for analysis and summaries of activity in the battlespace. This research project will take the first steps toward ending the

isolation imposed by the current SBB by providing support for computer supported cooperative work (CSCW) activities. In a broader sense, this research effort will begin the process of identifying tools and supporting technologies that people need to operate and manage forces effectively within a virtual battlespace and with other virtual environments such as a virtual emergency room.

Placing a CSCW capability within the SBB will complement the individual and autonomous agent analysis with analysis from other SBB users. This will allow commanders to form a better mental model of the activity in the battlespace. Timely information about changes to the battlespace and how these changes affect the commander's plans are crucial to effective decision-making. Because they are no longer alone inside the virtual environment, the possibility that important information will be missed is reduced and the analysis of the state of the battlespace should be improved.

This thesis effort will determine what changes are needed to the SBB to support CSCW activities. These changes will involve devising a means of transmitting information to users of other SBBs, a means to display information sent by other users, and a means for storing this information for future use. Developing a means of transmitting CSCW-supporting information will require that changes be made to the existing DIS standard. This will involve defining new protocol data unit (PDU) types for communication between stealth players in a distributed virtual environment. Once these changes are identified, they will be proposed as changes to the DIS standards committee and be implemented for use in our lab. To help the user manage information, messages will be displayed on the user's console or posted to a location in the battlespace as an annotation. The user interface will be based on the SBB's Information Pod interface (hereafter referred to as the Pod), so changes to the interface will have to address the placement of panels, the placement of controls on these panels, and the placement of information display windows and annotations. The toolset that provides the capability of performing CSCW activities between stealth players will be collectively known as the SBB's Collaborative Workspace.

## 1.2 Thesis Statement

The goal of this thesis effort is to open up as many avenues of communication as possible between SBB users so that collaboration between remote players can occur. This capability will demonstrate that computer-supported collaborative work (CSCW) is possible within a DIS virtual environment. At a minimum, CSCW support will include the ability for users to send and receive the following: public bulletins, private email, chat messages, viewpoints, video clips and annotations. Performance of these tasks requires that some form of "virtual keyboard" or enhanced interface be created so that textual input can be easily generated while wearing a head-mounted display. All of these features will be incorporated into the Synthetic BattleBridge using the Pod as a framework for the user-interface. New and existing PDU types will be needed to pass stealth management and entity-state information between SBBs. Finally, a user-study will be conducted to evaluate the effectiveness of the collaborative workspace within a virtual environment.

## 1.3 Scope

The Synthetic BattleBridge's Collaborative Workspace will be limited to the sharing of text messages, viewpoints, annotations, and video communications due to bandwidth limitations and the lack of available voice-processing equipment within AFIT's Virtual Environments, 3D Medical Imaging, and Computer Graphics Laboratory. The bandwidth restriction stems from the need to transmit the CSCW data and not interfere with the transmittal of state data for entities within the distributed virtual battlespace. The input devices used in this effort will be limited to those that are currently available in

AFIT's Virtual Environments, 3D Medical Imaging, and Computer Graphics Laboratory: magnetic head tracker, mouse, and keyboard.

## 1.4 Assumptions

An assumption that impacts the support of video and audio transmissions is the availability of a completed ATM network and a distributed simulation interface that supports ATM networks. If a distributed simulation interface that supports ATM is available before November 1996, it will be incorporated into this research effort.

## 1.5 Standards

Communication between SBBs will be accomplished through the use of DIS protocol data units (PDUs). PDUs provide the means of passing simulation data over a network to other DIS participants. Existing PDUs will conform to IEEE-1278 standards and new experimental PDUs will be developed with the intention of proposing changes to the current DIS standard.

## 1.6 Approach/Methodology

The development of the Synthetic BattleBridge's Collaborative Workspace can be divided into five areas: software architecture, graphics performance, distributed simulation, user interface, and CSCW support. Each of these areas will have its own particular design and implementation issues. The focus of the entire development effort is the support for CSCW activities and most of the time spent on this effort will be placed in developing an assortment of communication techniques between SBBs.

### *1.6.1 Software Architecture*

The current SBB's software architecture based on ObjectSim is unsuitable for the research to be performed. A new architecture will be designed that takes the functionality of the current SBB into account while also considering the need to add new CSCW-supporting enhancements. The current SBB will be ported to this new architecture.

### *1.6.2 Graphics Performance*

The rendering performance of the current SBB implementation provides an acceptable 15 frames per second. Objects are correctly positioned using a user-centered viewpoint algorithm that overcomes Performer's positional resolution problem. This algorithm will be converted to the new architecture and incorporated into the new SBB. The graphics performance of the SBB will then be improved to meet acceptable frame rates on single processor platforms.

### *1.6.3 Distributed Simulation*

DIS Manager 3.0, developed for AFIT by Steven Sheasby, will be integrated into the new SBB. The DIS Manager provides a communication interface between distributed simulation applications and the distributed simulation environment, currently DIS. All communication between the application and network players will be accomplished using this interface. DIS Manager 3.0 will support all PDU types currently used by the SBB and new, experimental PDU types needed for collaboration between stealth players.

### *1.6.4 User Interface*

The current SBB user interface will be examined for its ability to support the CSCW features being added. An interface will be designed based on the SBB's current Pod interface that also considers the need to handle text input and manipulate annotations. This new interface will be integrated into the new SBB to provide the added capability to support CSCW activities without sacrificing the ease of use of the current Pod.

### *1.6.5 CSCW Support*

Taking into account the bandwidth and available equipment constraints, a collaborative workspace will be designed to support CSCW activities within the Synthetic BattleBridge. This CSCW support will include the ability for SBB users to send and receive the following: public bulletins, private email, chat messages, viewpoints, video clips and annotations. This design will be integrated into the new SBB by using the new software architecture, DIS interface and user interface enhancements developed previously.

## 1.7 Materials and Equipment

The primary piece of equipment needed for this thesis effort is a Silicon Graphics workstation with High Impact graphics or better. Software for the system must include a C++ compiler, Performer 2.0, XForms, and Coryphaeus' Designers Work Bench. The SBB can be viewed through a conventional computer monitor; however, to enhance the experience, a head-mounted display and a Polhemus Fastrak magnetic head tracker is required. All equipment needed for this effort is currently available in AFIT's Virtual Environments, 3D Medical Imaging, and Computer Graphics Lab.

## 1.8 Thesis Organization

The next chapter reviews the background information associated with creating a collaborative workspace within a distributed virtual environment. It begins with an introduction to virtual reality and the field of computer supported cooperate work (CSCW). A history of non-immersive, virtual communities is presented along with the current research of bringing cooperative work to immersive virtual environments. A discussion of distributed interactive simulations and two potentially useful PDU types is also presented. The chapter concludes with a synopsis of previous research efforts using the Synthetic BattleBridge, a examination of two competing distributed simulation software architectures, and a review of user interface design in a virtual world.

Chapter 3 begins with the general and specific requirements of the new SBB and Collaborative Workspace. The requirements and design areas are divided into five components: software architecture, graphics performance and design, distributed simulation interface, user interface, and CSCW capabilities. Each design component will be discussed in terms of its applicability to the overall system design and ability to be implemented. At the end of this chapter the overall design of the entire system is presented.

Chapter 4 discusses the implementation of each design component. After converting the SBB to the new architecture, the graphics performance is improved through the use of current industry standards for graphics programming,. The DIS interface is integrated into the SBB along with some enhancements to the SBB's user interface. Finally, the Collaborative Workspace is developed to bring CSCW capabilities to SBB users operating in a DIS exercise.

Chapter 5 presents the results of this research project. The significant accomplishments of the SBB's Collaborative Workspace are presented. Recommendations are also included for future research in this field.

# 2 . Background

## 2.1 Introduction

This chapter examines the technologies involved with developing collaborative workspace techniques within a distributed virtual environment. To understand the design of the Synthetic BattleBridge's Collaborative Workspace and the significance of its results, these topics must be familiar to and understood by the reader. The purpose of this chapter is to identify and introduce these subjects. The topics related to this project include virtual reality, computer supported cooperative work (CSCW), virtual communities, collaborative workspaces, and distributed interactive simulations (DIS). Relevant issues such as software architecture and user-interface design will also be presented. This chapter will provide descriptions of these areas along with research efforts that are pertinent to this thesis endeavor. These descriptions are not meant to be complete studies of the work in their respective fields, but should act as an introduction to the subject matter with appropriate references provided should further study in this area be pursued. This thesis effort will involve the integration of a collaborative workspace into the Synthetic BattleBridge (SBB); therefore, background material on the SBB is also presented.

## 2.2 Virtual Reality

"Virtual reality" (VR) is a term coined by Jaron Lanier, in 1989, to distinguish between the immersive digital worlds he was trying to create and traditional computer simulations.

Virtual reality involves the immersion of humans in virtual worlds that exist entirely within the computer. Immersion involves interaction with objects from the virtual world, their manipulation, and the feeling that the human user is a real participant in the virtual world. Current systems produce this immersion effect using displays with wide field of view (provided by specialized optics included in head-mounted displays), stereoscopic images (provided by LCD-shuttered glasses or twin displays), one or more six degree-of-freedom tracking devices used for head and hand tracking, and often a glove or 3D mouse input device. Most current VR projects aim to reproduce a perceptual experience such as giving users the sensation of walking along a hallway in a building, flying over a city, or riding in a vehicle [BOLT95]. One goal of this research project is to integrate a large amount of symbolic (i.e. textual) information with the existing perceptual experience.

There are three challenges facing the virtual environments research community. The first challenge concerns how users immersed in virtual worlds can communicate their intentions to the computer. The second challenge relates to the first and addresses the inherent user isolation caused by operating in a virtual environment. The last challenge is a user interface problem in creating appropriate three-dimensional symbols for abstract objects (i.e. annotations, telepointers, avatars, etc.) within a virtual environment. Each of these challenges is discussed below.

### *2.2.1 User Interaction Techniques*

User interaction within a virtual world is different from user interaction in a typical desktop application. In a virtual environment, there are many degrees of freedom, continuous response and feedback to user actions using high bandwidth I/O devices, and typically some amount of aggregation of the user's probabilistic input (e.g. tracking data) occurs [VDAM94].

The four basic forms of single-user interaction within virtual environments are user movement, object selection, object manipulation, and menu or control panel interaction [WEBE96]. User movement, or navigation within the virtual world, requires two attributes, direction and speed. This is typically accomplished by pointing to a particular location using a mouse, glove, or wand to indicate the direction of travel, and then using two or more buttons to control acceleration through the world. AFIT's Pod interface uses menus and control panels to navigate within its virtual environment. Selection allows the user to identify an object of interest. This is a three step process that starts with the user indicating the object to be selected. The computer then indicates to the user (usually through a highlighting technique) the object that was selected. The user confirms this selection or makes an attempt to select another object. Object manipulation can take the form of creating, deleting, positioning, scaling, or orienting a particular object. The techniques to perform these tasks vary greatly between virtual environments [PIME94]. One such manipulation technique will be presented in the section on user interface design. Menus and control panels do not exist within all virtual environments [GREE95]. One reason is that many virtual environments do not incorporate text and numerical information due to hardware technical limitations. The resolution of many head-mounted displays is barely adequate to produce readable text. As a result, VR applications limit themselves to the display of a minimal number of words or numbers, usually for the purposes of orientation or navigation. For this reason and the inability to effectively operate a keyboard while wearing a HMD, textual input is not considered in most virtual reality systems. The creation of menus and control panels will be discussed further in the user interface design section.

### *2.2.2 Isolation in Virtual Environments*

One of the drawbacks of working in a virtual environment is the inherent isolation of the user within the virtual world. Some of this isolation is caused by the hardware as a result of immersing the user within the virtual environment. For example, head-mounted displays (HMDs) typically block outside light from entering the visor so that the user is only able to see the images created by the computer. This focuses the user's attention on the displays, but prevents the use of the most common user input device, the keyboard. This limitation is removed through the use of see-through displays that are typically used with augmented reality applications [FEIN93]. Because of the inability to use a keyboard while wearing an immersive HMD, many research laboratories have given up on text as a communication medium in virtual environments and have instead turned their attention to gesture and voice recognition systems [BYHM92][WEBE96].

Isolation within a virtual world is not solely physical in nature. Users of VR systems usually find themselves alone in a world with their objects. Even when multiple users share a common virtual environment, communications are often limited to gestures and user position. This is especially true when bandwidth is a concern or the system does not support voice interaction. The performance requirements of a single-user VR system are difficult enough to maintain without the unpredictable nature of a multi-user environment [SHAW93]. Synchronization and control are also issues that arise when a multi-user VR system is developed. The representation of user actions must be synchronized between all participants to allow cooperative work to occur. Control is also an issue, especially in the manipulation of objects within the virtual world. If two users attempt to resize an object, one user needs to be locked out of the operation for the other to complete the resizing task. This is not a concern of a single-user VR environment.

### *2.2.3 Appropriate Metaphors*

[SHER95] believes that virtual reality is a new medium with its own specialized language of symbols, signs, gestures, and metaphors. Within each virtual environment, a series of metaphors are used to help the user interact with the virtual environment. These metaphors determine the way in which the user is supposed to relate to the virtual environment. Metaphors are common in the theory of human-computer interaction. [BRYS94] The idea is to allow the user to think in terms of interacting with objects that are directly related to the task at hand instead of thinking in terms of interacting with a computer. The most common example is that of a window on a desktop containing folders as a way of organizing, interacting with, and navigating through information stored inside the computer. The window may contain folders which themselves contain information. These windows, desktops, and folders do not literally refer to their real-world counterparts. A window on the computer screen is to be thought of as a view of an information space. Similarly the computer desktop is a metaphor for a place where folders are placed, and a folder is a metaphor for an object that contains categorized information.

Constructing a good metaphor for a virtual environment is a critical task, as this metaphor will determine the appearance and behavior of the environment, as well as how the user interacts with that environment. A good metaphor will allow the user to interact comfortably and effectively with the virtual environment to perform the application tasks. A bad metaphor will hinder the user's effectiveness, either by presenting a confusing environment or by causing the interactions in the environment to be difficult to perform.

[BRYS94] states that virtual environment metaphors can be separated into three categories:

- *the overall environment metaphor* determines the overall appearance of the environment, including the types of application objects that appear in the environment,
- *the information presentation metaphor* determines how information about the environment is presented to the user, and
- *the interaction metaphor* determines how the user interacts with the environment and objects in the environment.

Note that at each level of metaphor there may be several metaphors. For example the information presentation metaphor may include text that appears in the environment (perhaps in an information window) as well as information displayed by the color of objects (i.e. the owner of an annotation). The interaction metaphor may include any or all of the interaction methods described previously.

## 2.3 Computer Supported Cooperative Work (CSCW)

The term "computer supported cooperative work" (CSCW) was coined by Greif and Cashman in 1984 to describe the increasing research and development activities regarding the augmentation of group work by computers [GREI88]. This multidisciplinary field of research combines the understanding of the nature of group work with enabling technologies of computer networking, systems support and applications [RODD91b]. Before addressing some of the issues involved with the computer supported aspects of CSCW, it is necessary to examine the nature of cooperative work.

According to Bannon et. al. [BANN91], terms like cooperative work, collaborative work, collective work, coordinated work, and group work are not well established in the CSCW

community. They present a variety of viewpoints that are meant to explain the differences between these phrases. In this paper, I will limit discussion to two of these terms, collaborative work and cooperative work. To collaborate is to work together or with someone else, whereas to cooperate is to work or act together for a shared purpose [SORG87]. To cite Bannon et. al. again, "cooperative work is the general and neutral designation of multiple persons working together to produce a product or service. [BANN91]"

Both Bannon and Sørgaard state that in order for a work situation to be considered cooperative, it must fall within the following criteria:

- *Cooperative work is found where individuals work together due to the nature of their task.* This group task must be cooperative in nature, which means that it should fall within the upper half of McGrath's task circumplex model[1] [MCGR84] (See Figure 2-1).
- *People involved in cooperative work share the same goals, part of which is the fulfillment of the shared task.* Thus, cooperative work is clearly non-competitive since the participants share the responsibility of the outcome of the task.
- *Cooperative work is done in an informal, normally flat organization.* Commonly, small project groups are used for cooperative work. The formality of the organizational hierarchy is temporarily relaxed as the goal of the work group is to complete the task. This helps to facilitate the next criteria, horizontal communication.

[1] McGrath has combined the main ideas of a number of scientists into a conceptually related set of distinctions about tasks. McGrath concludes that his task circumplex model represents a reasonable attempt to classify group tasks: virtually all tasks used in group research can be accomodated. The horizontal dimension shows a contrast between behavioral or action tasks to the right and conceptual or intellectual tasks to the left. The vertical dimension reflects a contrast between cooperation or facilitative compliance at the top and conflict or contrient interdependence at the bottom.

- *Horizontal communication takes place in a cooperative setting.* This is a direct result of the informal, flat organization of cooperative work.
- *Cooperative work is relatively autonomous.* External influences on work tasks (e.g. external planning and control) reduce the cooperative nature of the work.

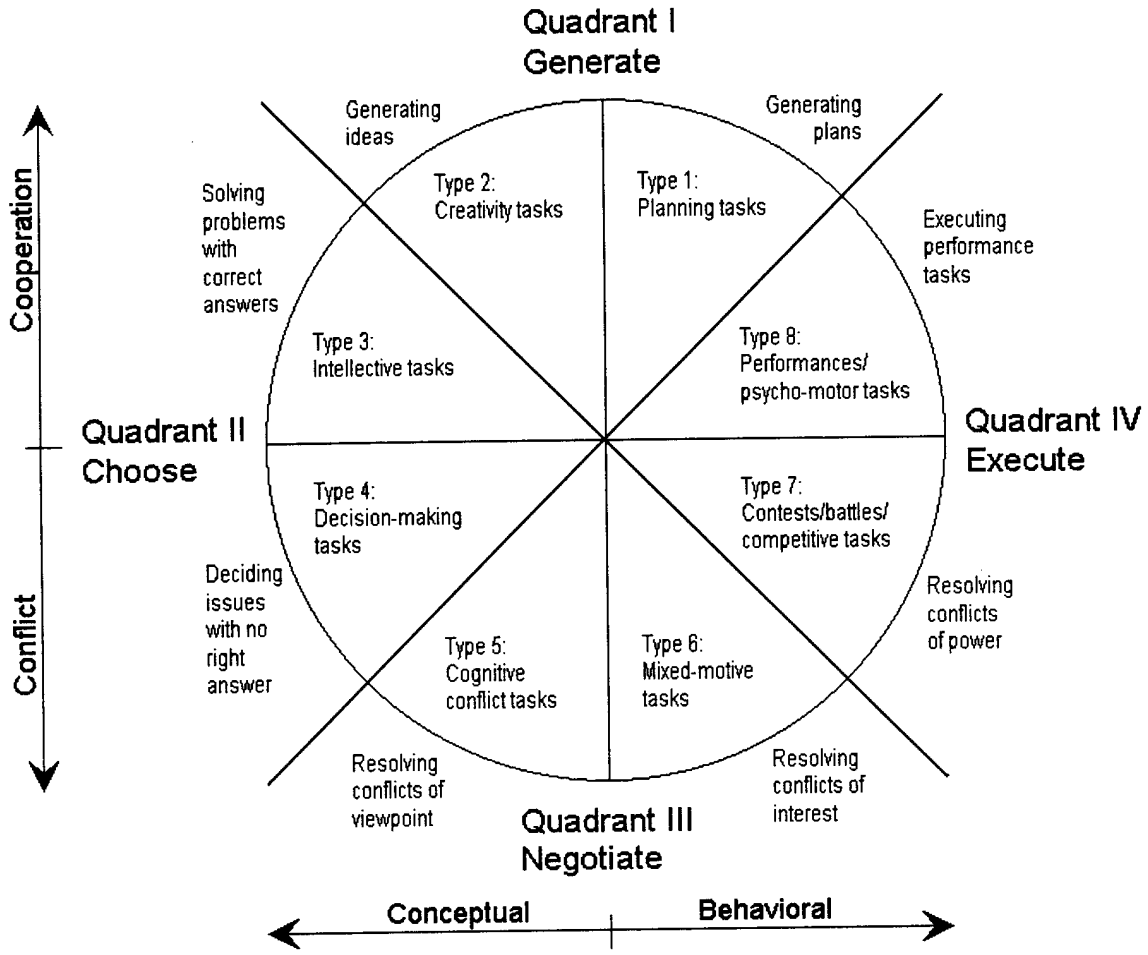



**Figure 2-1. McGrath's Task Circumplex Model**

Computer supported cooperative work systems are defined by Ellis as "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment [ELLI91]." This definition does not fully address some of the

characteristics of cooperative work stated above. For example, the communicative aspects required for cooperative work are not mentioned . Having access to a shared environment does not necessarily imply some form of interpersonal communication. An ideal example of this is a distributed interactive simulation where hundreds of users can participate in a shared world without direct communication with each other. While Ellis's definition is not perfect, it does allow a starting place to discuss some of the characteristics of CSCW systems.

CSCW systems can be characterized in many ways. Some of the research involved in this area include the work of Nunamaker, Roddin, and McGrath [NUNA91][RODD91a] [MCGR94]. CSCW systems can be classified based on three factors: the functional role of technology in the cooperative work, the environmental dimensions of group work, and the characteristics of the task that the group is trying to accomplish. The functional role of technology is typically that of the communication and control system for the participants. Levels of control and communication are discussed in detail in "CSCW and Distributed Systems : The Problem of Control" [RODD91b]. Nunamaker and Roddin distinguished various environmental facets of group work. Two common aspects are the form of the interaction (or time dimension) and the geographic dispersion of the group members (or spatial dimension). These two dimensions are further divided as seen in Figure 2.2. The last classification measure of CSCW systems concerns the nature of the group's task. Specifically, CSCW systems can be classified based on the type of support needed to accomplish the group's task. This support is typically classified as one of four systems: message systems, conferencing systems, coordination systems, and co-authoring and argumentation systems (see Figure 2-3).

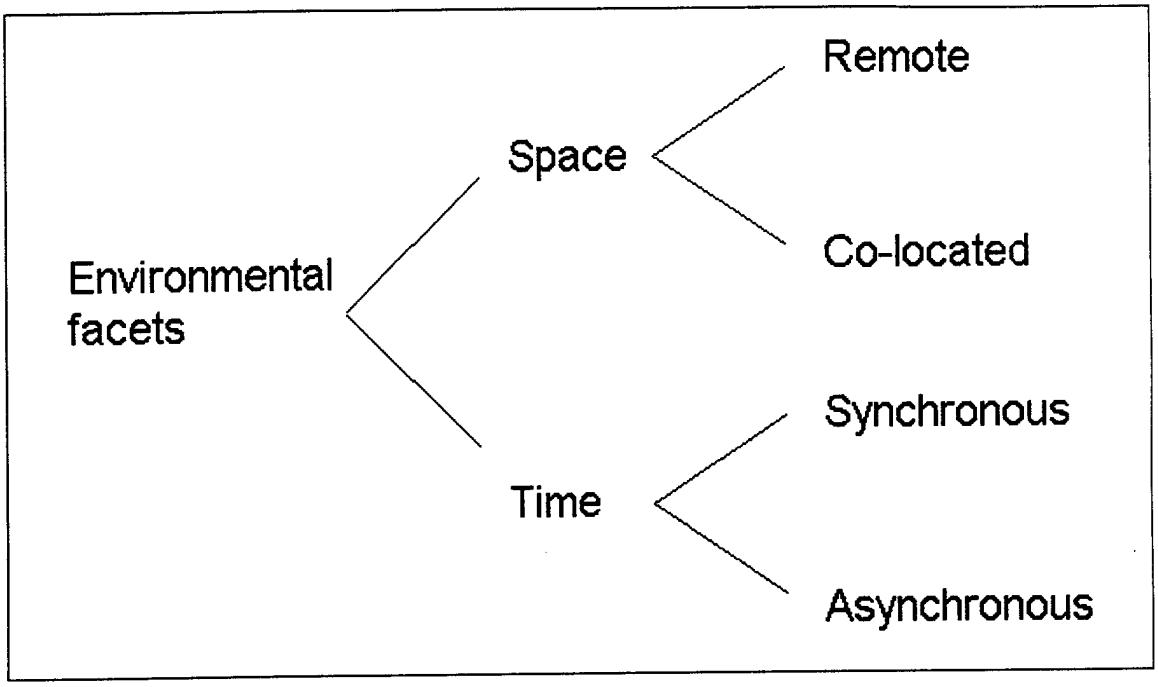



**Figure 2-2. Environment Facets of Group Work**

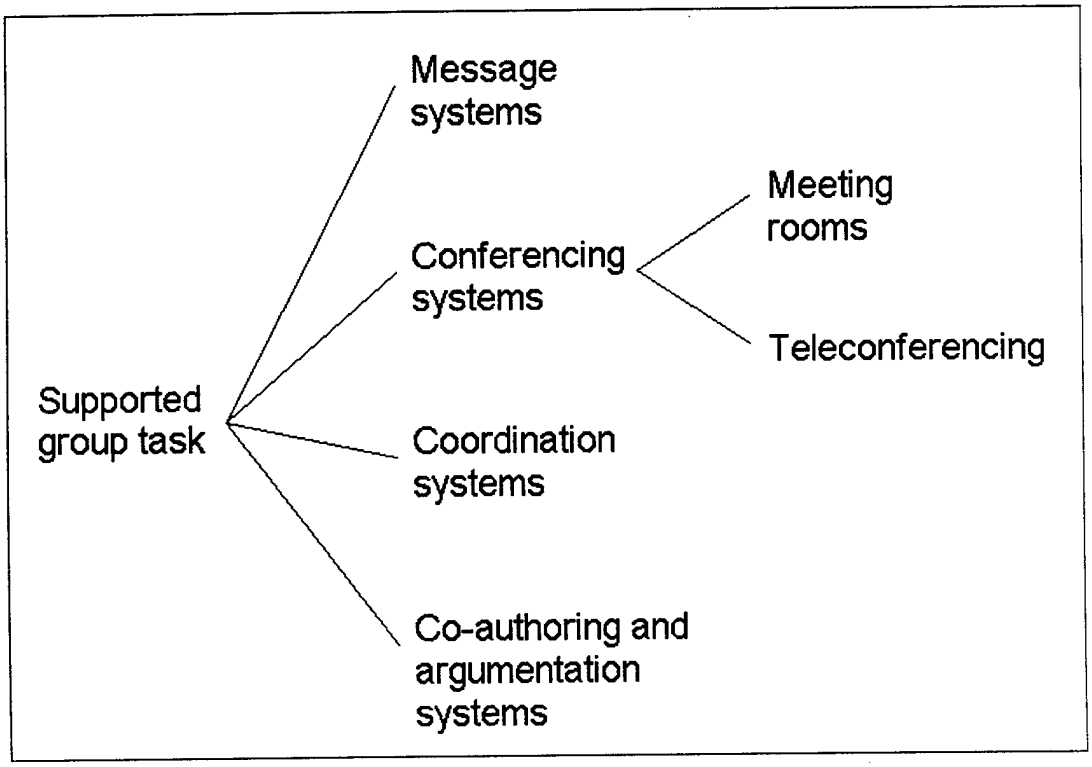



**Figure 2-3. Common Systems for Supporting Group Work**

Message systems are descendants of electronic mail (email) programs that allowed a user using a central machine to send textual messages to other users on the same machine. Since that time, the widespread use of wide area networks designed to support computer communications has increased the complexity and functionality of email. Message systems include the COSMOS [WILB88] and AMIGO [DANI86] projects and the Object Lens [MALO88], Studel [SHEP90], Coordinator [FLOR88], and ISM [RODD89] systems. Computer conferencing systems use the basic precepts of messaging systems, but a structure is imposed to allow the grouping of messages according to some criteria, typically a single topic. These topic groups, called conferences, must be joined by a user in order to access its contents. Examples of conferencing systems include Notepad [PULL86] and COM [PALM84]. Some conferencing systems also support the concepts of meeting rooms and teleconferencing. Examples of these systems are discussed by Ellis and Nunamaker [ELLI91][NUNA91]. Coordination systems address the problem of integrating an individual's work effort towards the achievement of the common goal. These systems are typically used by secretaries, managers, and peers to maintain a common calendar. Examples of coordination systems include electronic calendars and automated meeting scheduling software. The last computer supported task is the co-authoring and argumentation systems. This class of CSCW systems support the negotiation and argumentation involved with group working. Argumentation systems include gIBIS [CONK87] and SIBYL [LEE90] while co-authoring systems include Quilt [FISH88] and CoAuthor [HAHN89].

The computer support for tasks and the environmental dimensions in which they occur can be graphed together as was done by Roddin (see Figure 2-4). For the purposes of this thesis, only remote CSCW systems will be investigated due to the nature of DIS exercises.

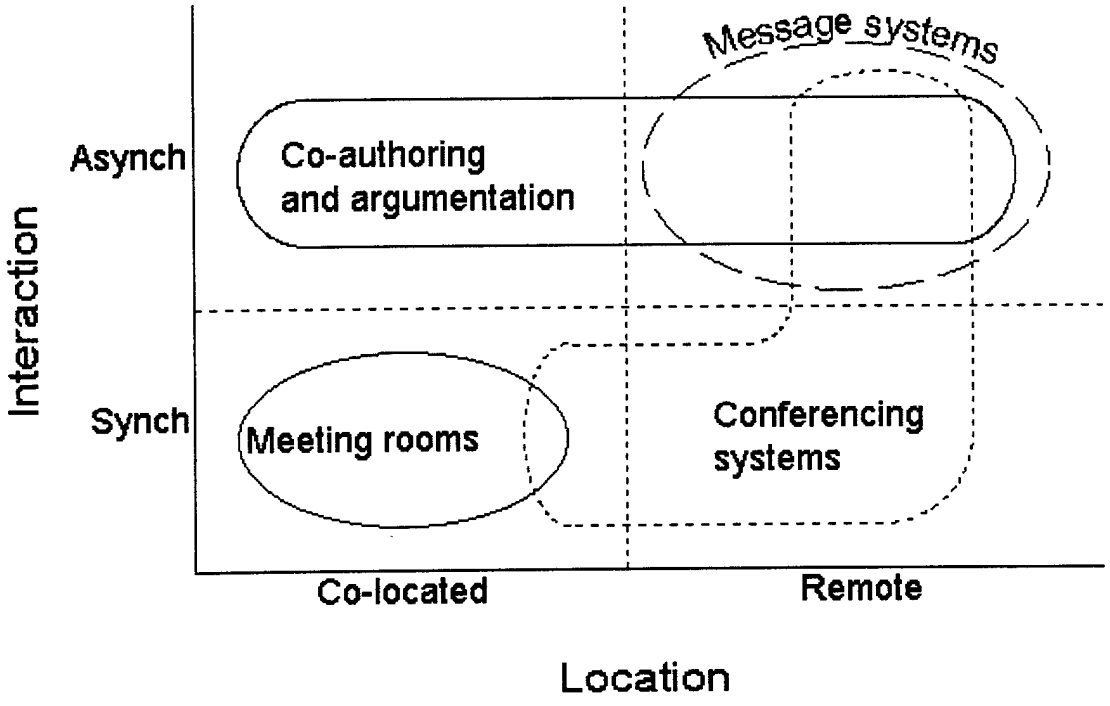



**Figure 2-4. Classification of CSCW Systems**

A justification for the study of CSCW system can be found in Alan Wexelblat's "The Reality of Cooperation: Virtual Reality and CSCW." In this paper, Wexelblat makes two important assumptions. The first is that the purpose of computers is to support human activity. He argues that while some esoteric computer applications exist (like calculating the value of pi), the point of using computers is to support what people do or what they want to do. Thus, computers extend the range of human activity, and support and augment existing activities. His second assertion is that human activity is inherently cooperative. He defends this statement by emphasizing that humans are social animals and that we cooperate in everything we do, from work to play to family. Most importantly, he contends, when humans want to get something done, we typically seek out people who can help us. If these two assertions are believed, then it is natural for people to investigate computer-supported collaborative work [WEXE93].

CSCW is still a relatively new field of study within computer science. Despite its newness, CSCW applications are used in most office environments, typically in the form of "groupware." Examples of groupware include Lotus Notes and Microsoft Office. Two important features of these groupware systems are email and documents that can be shared between users. Email has quickly become the standard form of interoffice communication and an Internet email address is now considered a minimal corporate presence on today's world-wide network of computers. Similarly, businesses have found that sharing documents allows workgroups to overcome some of the barriers of time and space that are typically imposed on people. The alternative is that people working on the same document must either be present in the same location or working at the same time.

Academic research papers concerning CSCW, such as those previously referenced, are typically aimed at providing a new enabling technology to the user to support cooperation. Usually, these CSCW systems are tested through a small series of user studies. These academic insights into CSCW trickle down to the corporate world where user studies on the order of a thousand people are not uncommon. From these studies, a CSCW product is created and refined until it meets the requirements of the greatest number of users. In the following sections, two commercially successful CSCW ventures, InPerson and NetMeeting, are presented. Features common to each system will be included in the design and implementation of the SBB's Collaborative Workspace.

### *2.3.1 InPerson & Iris Annotator*

Silicon Graphics' collaborative computing package is the InPerson desktop conferencing software. SGI's InPerson allows a user to place a conference call over a computer network to one or more users. Once connected, users can interact with each other via live video and audio

while working together on a shared file such as a text document, a 3D model, or a captured image using a collaborative workspace called the InPerson Whiteboard. The Whiteboard contains a number of annotation tools at the user's disposal. These tools include text, free-hand drawing, circles, squares, and arrowheads. A full palette of colors and line thicknesses are also available. Additionally, the InPerson Whiteboard provides a clipboard with the typical Cut, Copy, and Paste commands for editing objects drawn on the screen. Finally, when the call is completed, the Whiteboard can be saved to a file for future use.

IRIS Annotator is another collaborative product created by Silicon Graphics that allows a user to annotate a 3D model using the standard set of digital media tools found within InPerson's Whiteboard and then electronically mail the annotated models to others for review. This form of sequential collaboration is similar in nature to a in-turn letter, but with virtual objects. One interesting feature concerning IRIS Annotator is the ability to play back the series of actions performed on the model using simple VCR-like controls. This feature is particularly useful when the information being communicated is best presented in a linear fashion.

### *2.3.2 NetMeeting*

Recently, Microsoft has introduced its own collaborative computing software called NetMeeting. Microsoft's NetMeeting conferencing software is a standards-based, multipoint data conferencing product that does not currently support video, but does allow users a wide variety of communication techniques:

- Audio or text communication
- Chat rooms
- File transfers during collaboration
- Shared clipboard
- Shared whiteboard

- Shared applications

InPerson and NetMeeting share many CSCW-enabling technologies such as real-time chat (text and voice), file transfers that do not interrupt communication, the ability to save the resulting work of the group to a file for later use, and a shared whiteboard that includes a number of multimedia annotation tools. Both systems are also mindful of bandwidth limitations and can temporarily disable or scale-down functionality to meet the level of interaction required. This common feature set should be seen as the core functionality of any collaborative computing effort, including this thesis.

## 2.4 Virtual Communities

This section discusses the history of social virtual communities and is considered by many researchers in the field of collaborative virtual environments to be a significant part of their history as well [BENE91][BENF93a][BENF93b][BENF95b][GREE95] [SHER95]. It is included for completeness, but can be skipped without loss of clarity to the remaining sections.

For many people, the first time they heard about entering a computer world and flying through data was in William Gibson's science-fiction novel, *Neuromancer*. Published in 1984, it became an inspiration to would-be VR builders. *Neuromancer* describes life in the next century when the world's telephone system has been superseded by the Matrix, the interconnected sum total of all the world's computer networks [GIBS84]. Many parallels exist between the Matrix and today's Internet. Gibson used the term "cyberspace" to describe this alternative computer universe in which people use a special virtual reality console to connect to the Matrix.

The Gibsonian-view of cyberspace does not currently exist, but some founding steps have been made about how large groups of users would interact in a virtual world. Two of the

most noted examples of these large-scale virtual communities are TinyMud and Lucasfilm's Habitat project [MORN91]. A third example is the recently developed AlphaWorld. Although these virtual communities were essentially designed for entertainment, there is no reason why large amounts of useful information could not be located, accessed, and manipulated within the structure of their virtual worlds. These projects were used as reference points in the design of the Collaborative Workspace.

### *2.4.1 MUDs*

Multi-User Dungeons (MUDs) are text-based virtual communities based on the symbolic space of literature [BENE91]. The original MUD was written in 1979 by Roy Trubshaw and Richard Bartle on a DECsystem-10 at Essex University [BART90]. The first large-scale, Unix-based MUD was created in 1989 at Carnegie Mellon University and called TinyMud. [BURK93] TinyMud was the first easily portable, interactive, text-based, networked, user-extensible, city-metaphored venue for hundreds of participants. TinyMud supported the interaction of a large numbers of users, but limited the user to typed interaction. The benefit of this system is that a MUD can be accessed from any computer terminal, regardless of its graphics capability.

TinyMud players were able to communication with each other through talking, whispers, and actions. When a person talked, all participants of that room were able to hear what was being said. Whispers allowed private conversations to occur within the same room, or to talk to someone who is not in the same room by using a paging command. One interesting aspect of MUDs is that they imposed a spatial metaphor on the participants. Users may talk and interact easily with others in the same virtual room, but specialized means of communication (pages and emails) were required to converse with people in other locations [EVAR93]. MUDs provided a crude sense of space and lively interaction with other participants. In this manner, MUDs

foreshadow some of the types of interactions that can be expected in fully immersive virtual environments such as those included in the SBB's Collaborative Workspace (i.e. private email, public bulletins, and chat) [NRC95].

### *2.4.2 Worlds Chat and AlphaWorld*

Worlds, Inc. has recently created two Internet virtual communities, Worlds Chat and AlphaWorld, that take the notion of cyberspaces to the next level: real-time 3D graphics. Worlds Chat was the first system to take the features of a chat room and combine them with a navigational interface of a first-person shooter game (e.g. Doom). Each participant selects an avatar, a graphical representation of their character, that will be used within the chat rooms. Since the number of avatars is limited, a name tag is placed above the avatar to identity each user specifically. Communication occurs through textual chat messages that appear in the scene next to the speaker's nametag.

AlphaWorld is currently in the experimental stage of development, but already hosts over 100,000 users. AlphaWorld is a 3D MUD-like virtual world, but unlike traditional MUDs, where textual descriptions of the localities are used, AlphaWorld has been created using the Virtual Reality Markup Language (VRML) 2.0 specifications. In this virtual world, everything is realistically portrayed in three dimensions. Every object in the world can support a particular action, such as emailing a user, teleporting to another location, loading up a webpage, or playing a movie or a sound file. Experiences using AlphaWorld formed the design of the Collaborative Workspace's annotations.

## 2.5 Collaborative Workspaces

A collaborative workspace is a form of computer-supported cooperative work (CSCW) that is built into a virtual reality (VR) application. The term "collaborative workspaces" refers to the real or virtual gathering of people at a specific location for the purpose of interacting with each other and some group of objects [NRC95]. Commonly called "collaborative virtual environments (CVEs)", this aspect of CSCW has only come into existence in the past few years and has never been attempted within a DIS application. Collaborative virtual environments involve the use of distributed virtual reality technology to support group work. A necessary condition for a CVE is the provision of simultaneous multi-user access to a common virtual world. A CVE user must be explicitly represented to all other CVE users within the shared environment. Furthermore, they should be free to move around within this space, interacting with each other and also objects and information of common interest. Common goals of CVE systems are the support for natural spatial social skills, scalability of the CVE to support a large numbers of users (e.g. minimal bandwidth requirements), and support for the cooperative accomplishment of spatial-oriented tasks.

A sample of some of the collaborative workspace techniques used within non-DIS applications is presented in this section. This includes Georgia Tech's work on the Design Virtual Environment and their Virtual Venues project in support of the 1996 Olympics, Nottingham University's MASSIVE project, Cornell's Collaborative Virtual Environment, and perhaps the most well-known CVE, the Swedish Institute of Computer Science's Dive system. Other CVE systems include the COMIC ESPRIT III Basic Research Action [BENF93b], the UK's Virtuosi project [BENF94b], ATR's Virtual Workspace [TAKE92], MR Toolkit and the MR Peers package[SHAW93], and Manchester University's Aviary [SNOW94]. The systems

presented in this section represent the closest approximations to the Collaborative Workspace's design and implementation.

### *2.5.1 DVE and Virtual Venues*

The Design Virtual Environment (DVE) from Georgia Tech's Graphics Visualization Usability Lab is a virtual reality visualization tool for communicating design ideas and a teaching tool for design education. DVE allows users to experiment with new instructional concepts that are not currently possible with either traditional or CAD-based computer graphics techniques. In particular, DVE allows a student to get inside computer-designed structures to visually walk through and inspect the designs.

DVE has the capability to be used as a multi-participant virtual environment. Additionally, annotation tools can be used within DVE. This feature allows an instructor to walk through a design with a student, critique the design, discuss modifications, and leave behind markers with voice annotations to record specific suggestions for later design modification. This system is an example of a small-scale collaborative workspace based around annotations and voice communications.

The Virtual Venues project was created to visualize two Olympic environments, a natatorium for water sports and a stadium for track-and-field events [BOLT95]. Users are able to experience multimedia (e.g. audio, video and animation) presentations of sporting events, travel around the venue and leave annotations for other users. This project uses both 3D interface techniques such as user navigation, orientation, and annotations with 2D interface techniques like pull-down menus and plan-view maps. Menu items for navigation, orientation, annotation and help functions are selected by using a 3D pointer in an immersive environment or with a mouse in a screen-based environment. Video annotations projected onto a screen are

played back by selecting the VCR-like menu buttons in 3D space. This combined interface permits increased functionality of a single 3D input device with minimal loss of the display area caused by the top-level menu items.

### *2.5.2 MASSIVE*

MASSIVE (Model, Architecture and System for Spatial Interaction in Virtual Environments) is a VR conferencing system created at the University of Nottingham [BENF95] [GREE95]. The project is an expansion of the cooperative "Reality Built for Two" system created by Blanchard in 1990 [BLAN90]. The main goals of MASSIVE are scale (e.g. supporting as many simultaneous users as possible), heterogeneity (e.g. supporting interaction between users whose equipment has different capabilities), multimedia (i.e. visual, aural, and textual communication) and spatial mediation (e.g. spatially mediated conversation management). These goals closely follow the objectives of distributed interactive simulations and collaborative workspaces.

Inside the MASSIVE architecture, participants can interact through a combination of graphics, audio and text interfaces. The graphics interface renders objects visible in a 3D space and allows users to navigate with a full six degrees of freedom. The audio interface allows users to hear objects and supports real-time conversation and playback of pre-programmed sounds. The text interface provides a MUD-like representation of the world based on a 2D scrolling window that permits the user to move across a 2D plane. This text interface allows users to communicate by typing messages to one another. The MASSIVE interfaces can be arbitrarily combined according to the capabilities of the user's terminal equipment. Despite the capabilities of the interface, communication is controlled via a spatial model of interaction so that one user's perception of another user is sensitive to their relative positions and orientations.

MASSIVE's immersive environment currently requires too much bandwidth to permit the scalability needed for large group interactions [BENF95b]. In order to reduce the amount of bandwidth required, MASSIVE relaxes the synchronization of some of its components (such as viewpoints and actionpoints). A similar relaxation (i.e. dead reckoning) is used within distributed interactive simulations to reduce the bandwidth required to maintain an entity's state. The same technique is not feasible when transmitting a shared viewpoint that requires accurate positional information.

### *2.5.3 Cornell's Collaborative Virtual Environment*

Cornell University has constructed a system that allows individuals to work together within a virtual environment across a network connection [AMID95]. The users can see and talk with each other in real time while they explore and manipulate a common three dimensional data set. To provide the needed level of interaction, the researchers used the Continuous Media Toolkit, developed at the University of California at Berkeley, to transmit audio, video, user tracking and synthetic geometry information across the network. The primary importance of the Cornell Collaborative Virtual Environment system was that a pre-existing virtual environment application was quickly modified to allow cooperative work. Additionally, the basic structure of the original application was left undisturbed. This capability is also required for the SBB's Collaborative Workspace.

While the prototype system did meet with limited success, it suffered from expensive hardware requirements, problems with audio transmissions and a low video frame rate. The hardware utilized in the system consisted of two multiprocessor SGI Onyxes with Reality Engine graphics. One machine was equipped with a Sirius video card that was used to capture video images from both machines. Only one machine was equipped with a soundcard, so audio

transmissions could only be tested in loopback mode. Audio quality was acceptable, but since audio packets were not prioritized, gaps occurred in the audio datastream. Video quality in loopback mode was acceptable, ranging from a high of 30 frames per second (fps) to a low of 10 fps. Video quality between machines over the network was 3 fps due to a poor video compression scheme.

### *2.5.4 Dive*

The Distributed Interactive Virtual Environment (Dive) is an experimental platform created by Swedish Institute of Computer Science for the development of virtual environments, user interfaces and applications based on shared 3D synthetic environments [CALR93] [BENF95b]. Dive is designed for multi-user applications, where several networked participants interact over an internet. Users navigate in 3D space and see, meet and collaborate with other users and applications in the environment. A participant in a Dive world is called an actor, and is either a human user or an automated application process. An actor is represented by a "body-icon" (or avatar), that may be used as a template on which the actor's input devices are graphically modeled in 3D space. For example, changing the position of the eye, or changing the "eye" to an another object, will change the viewpoint. This roving viewpoint can support a wide range of I/O devices such as an HMD, wands, datagloves, etc. Dive permits actors to leave and enter worlds dynamically. Additionally, any number of application processes exist within a Dive world. Such application processes typically build their user interfaces by creating and introducing necessary graphical objects. Thereafter, they "listen" to events in the world, so that when an event occurs, the application reacts according to some control logic. An example is a virtual alarm clock whose hands move in accordance with the computer's system time. Upon reaching the user-defined alarm setting, the clock buzzes until the snooze button is pressed by a

user. The Dive system is an example of an object-oriented, multi-user collaborative workspace for a distributed virtual environment and will be used in the design of the Collaborative Workspace's annotations and shared viewpoints.

## 2.6 DIS and Current DIS Communication Techniques

In 1989, the Distributed Interactive Simulation (DIS) Workshop was created to develop standards by which multiple distributed virtual environment simulations could play together. "The primary mission of DIS is to define an infrastructure for linking simulations of various types at multiple locations to create realistic, complex, virtual worlds for the simulation of highly interactive activities" [DISV94]. These DIS simulations allow the military player to insert himself into a virtual environment and interact with other human-controlled entities and semi-autonomous forces. As a stealth application, the SBB allows a user to watch the events occurring within a distributed simulation without being seen by other players. This section introduces DIS, an implementation of an interface to the DIS protocols developed at AFIT, and the DIS protocols needed to collaborate with other stealth applications.

### *2.6.1 Distributed Interactive Simulation*

Distributed simulations are conducted by linking computer-based combat simulators together over a common network. Each simulator controls one or more entities (tanks, planes, etc.) that move within a common virtual environment (including terrain, weather, lighting, etc.). As an entity moves, its host simulator keeps all the other networked simulators informed of its location and status. A representation of each entity, local and remote, is displayed to the simulator user. A player can interact with other entities within the simulation using a predetermined protocol suite. The DIS protocols are an example of such a predetermined suite that enables networked simulators to interact within a distributed virtual environment. The DIS protocol suite allows potentially thousands of geographically dispersed users to participate concurrently within a shared virtual environment using a standardized set of

**Table 2-1. DIS PDU Families and Types.**

| Entity Information Interaction | Simulation Management | Logistics |
|---|---|---|
| Entity State | Start/Resume | Service Request |
| Collision | Stop/Freeze | Resupply Offer |
| | Acknowledge | Resupply Received |
| Warfare | Action Request | Resupply Cancel |
| Fire | Action Response | Repair Complete |
| Detonation | Data Query | Repair Response |
| | Set Data | |
| Radio Communications | Data | Emission Regeneration |
| Transmitter | Event Report | Electromagnetic Emission |
| Signal | Comment | Designator |
| Receiver | Create Entity | |
| | Remove Entity | |

communication protocols that have been codified into the Distributed Interactive Simulation Standard, IEEE 1278 [IEEE93]. The DIS Standard requires transmission of data in a format referred to as a protocol data unit (PDU). Each PDU has a fixed size and format header that includes the following information: protocol version, exercise ID, type and family of PDU, time stamp, and PDU length. The

PDU's header allows DIS entities to determine if the received PDU is of interest to that particular entity.. For example, if a PDU's exercise ID is not the same as the receiver's exercise ID, then the receiving application can safely ignore this PDU. The DIS Standard is comprised 27 PDU types divided into 6 families (see Table 2-1). Distributed simulation research has been thoroughly documented and the interested reader is referred to other sources for detailed information on DIS. "An Introductory Tutorial for Developing Multi-user Virtual Environments" [GOSS94] and "The DIS Vision"[DISV94] are good introductory tutorials on distributed simulations. "The Standard for Distributed Interactive Simulation" published by the Institute for Simulation and Training [IST96] contains a glossary of terms used in DIS. Finally, several DIS summaries exist [BLAU94][STYT95b].

AFIT has conducted DIS research since 1992 and has developed several DIS-compatible simulators [SWIT92], [ERIC93], [GERH93], [DIAZ94], [KUNZ94], [MCCA94], [VAND94], viewing platforms [HADD93], [SOLT93], [WILS93], [STYT94] and a DIS activity recorder analogous to a VCR [FORT94]. In order to facilitate the development of these projects, AFIT developed its own DIS interface, called the DIS Manager [SHEA92]. The DIS Manager removes the responsibility of the DIS interface from the application developer and is an important part of the Synthetic BattleBridge. The DIS Manager is responsible for sending and receiving all DIS-compliant PDUs. Two existing collaboration-related PDU types are the Comment PDU, from the Simulation Management family, and the experimental White Board PDU.

### *2.6.2 Simulation Management PDUs*

The Simulation Management family of PDUs are currently defined in the DIS standard and allow a DIS entity to create new entities, remove entities, request an entity to perform an action, query information about a DIS entity, change an entity's parameters, control time within a DIS exercise through start/stop/pause/resume mechanism, and send comments [IST96]. This

last PDU type, the Comment PDU is especially useful in communicating textual information between entities. A Comment PDU can be broadcast to all entities or sent to a specific entity. This enables a player to deliver what amounts to an email message or a public bulletin. The Comment PDU was not supported by previous versions of the DIS Manager.

### *2.6.3 White Board PDUs*

In the 12th and 13th DIS Workshops, Paul Gustavson outlines a mechanism for exchanging communications in a distributed interactive simulation between two or more DIS participants [GUST95]. This mechanism is called the White Board PDU (WBPDU) and is designed to allow participants to visually share ideas and information prior, during, or after a DIS exercise. The White Board PDU type permits the interactive sharing of text and graphic images and is meant to complement existing voice communications. The WBPDU is a message format that contains the following information: a sender ID, a receiver ID, the actual message data, and a message data type identifier; defined as either text (ASCII or binary), graphical image (BMP, TIFF, GIF, or JPEG), or a document (MS Word or Postscript). It is up to the application's developer to determine how these messages will be used and how the user interface will operate.

While this PDU type shows the greatest promise in being used in real-time DIS applications to support collaboration between participants, it was rejected by the Simulation Management Subgroup at the 13th DIS Workshop because of lack of interest to support this type of interaction within the DIS community [SIMU95]. Ironically, it received favorable reviews by the Validation, Verification and Analysis (VV&A) Subgroup at the same conference [VV&A95].

The WBPDU does not make full use of the features of a virtual environment, namely positional information of objects and entities. Messages cannot be attached to a location or a

player as annotations to the virtual world. Thus, the place metaphor is completely missing from the WBPDU. In email conversations with Paul Gustavson, he suggested that the capability to annotate objects in a 3D world is key to the survival of the White Board PDU and other DIS collaboration efforts. The WBPDU format will be used in the design of the Collaborative Workspace's Annotation Update PDU to permit the use of multiple media types for annotations.

## 2.7 Synthetic BattleBridge

Monitoring distributed interactive simulations without actively participating within the scenario is the job of the stealth entity. Stealth applications are used by commanders to gain situational awareness of the virtual battlespace. The Air Force Institute of Technology's (AFIT) Virtual Environments, 3D Medical Imaging, and Computer Graphics Laboratory is investigating ways to improve this situational awareness within large-scale virtual environments [STYT93] [STYT95a]. The use of collaborative workspaces is the latest technique to be explored in support of this effort. The AFIT Computer Graphics Lab has sponsored four thesis efforts in support of improving situational awareness through the use of the Synthetic BattleBridge. The latest theses to explore potential situational awareness improvements were by Kesterman [KEST94] and Rohrer [ROHR94]. Kesterman and Rohrer's thesis efforts culminated in the highly successful Pod interface that is now the standard user interface used within all Computer Graphics Lab DIS VR applications. Kesterman and Rohrer's work was based upon the thesis efforts of Haddix [HADD93] and Wilson [WILS93]. The result of Haddix and Wilson's research was the creation of a simple VR user interface and the integration of the Sentinels, autonomous agents capable of monitoring the battlefield and determining areas of interest to the commander. These Sentinels were created by Soltz [SOLT93].

The Synthetic BattleBridge is a DIS application specifically designed to help a decision-maker to monitor, assess, and analyze the activities within a virtual battlespace [WILS93]. The SBB's immersive user interface, known as the Pod, provides a way to control the user's view of the world, display information about the entities within the virtual environment, and provide a framework by which the application can be extended [HADD93][KEST94][ROHR94]. This framework permitted the seamless integration of a bio-chemical warfare component into the SBB and allowed for the Pod interface to be used inside of similar virtual environment applications such as the Satellite Modeler [VAND94] and the Virtual Cockpit [DIAZ94]. Using the Pod, the user can be presented with both low-level, unanalyzed data and data that has been analyzed and summarized by autonomous agents known as Sentinels.

The Synthetic BattleBridge provides the user with a first-person, immersive, synthetic environment observation post that permits unobtrusive surveillance of the environment without interfering with the activity in the environment. However, for large, complex synthetic environments, this type of support is not sufficient because the mere portrayal of raw, unanalyzed data about the objects in the virtual space can easily overwhelm the user with information. Thus, Sentinels were created to improve situational awareness by allowing the user to place analysis modules throughout the virtual environment [SOLT93][STYT93][STYT95]. These analysis modules gather information about their assigned regions and consolidate this information into an SBB display panel. This information allows the commander to determine the relative importance of various battlespaces by examining the rating provided by the Sentinels.

## 2.8 Software Architectures for Distributed Virtual Environments

Software architectures are needed to control information concerning the distributed virtual environment within the components of an application. Two approaches for communication infrastructures are the peer-to-peer configuration and the centralized configuration. A peer-to-peer configuration is used to permit communication between all of the software components in the system, resulting in increased software complexity. A centralized configuration allows communication through a centralized database that stores information needed by the system's components [AMSE95]. Both approaches have been used within AFIT's Computer Graphics Lab to design and implement distributed virtual environment applications. The peer-to-peer approach is represented by ObjectSim and the centralized approach is used by the Common Object Database. This section will present both software architectures and how they support the rapid prototyping of distributed virtual environments. Their comparative strengths and weaknesses are also discussed below.

### *2.8.1 ObjectSim*

The AFIT DIS virtual environment applications such as the Synthetic BattleBridge are built upon a set of libraries known as ObjectSim [SNYD93]. ObjectSim provides a peer-to-peer, object-oriented interface into the Silicon Graphics (SGI) Performer rendering library [HART94]. The goal of ObjectSim is to reduce the complexity a developer needs to master before attempting to write a visual simulation. This goal is accomplished by hiding most of the intricate details necessary for simulation creation inside a hierarchy of objects.

There are several problems with the ObjectSim approach to creating simulations. The first is the static nature of the ObjectSim libraries. Since ObjectSim is built upon SGI's

Performer, every time the Performer libraries are updated, the ObjectSim libraries must also be updated, patched, and recompiled. The second problem is in the extensibility of the ObjectSim classes. In order to create new objects, it is often necessary to partially inherit ObjectSim classes, thus creating a set of crippled ObjectSim objects with limited functionality and reusability. An example is the Crippled Polhemus Modifier class used to control the user's viewpoint through the keyboard. The third problem of ObjectSim is in the tight coupling between ObjectSim classes that results in a low cohesion between ObjectSim objects. Tight coupling and low cohesion defeats the purpose of object-oriented programming [RUMB91]. The largest drawback with ObjectSim is the added complexity created by the previously stated problems. An intimate knowledge of the ObjectSim classes is required to develop even the most basic of visual simulations.

### *2.8.2 Common Object Database*

A new software architecture effort known as the Common Object Database (CODB) suffers from none of the previous problems. It can be used with or without ObjectSim classes and permits data structures to be shared across all classes within an application. This centralized, flexible, object-oriented approach to sharing data allows the developer to store common data structures in a single location that can be accessed from anywhere inside the program. Semaphores and double buffering permits a single writer or multiple readers to safely access the same data structure at any given time. The double-buffering mechanism combined with the system's shared memory allows processes running at different rates to share data without excessive blocking. Given these advantages over ObjectSim, all current and future AFIT Graphics Lab distributed virtual environment applications will use the CODB architecture [STYT97].

Using this architecture, a developer creates a simulation class with a data structure that can be updated by any other class in the application. Specialized methods to access a class's data structure are no longer required. Instead, a common access method is provided by the CODB architecture to read and modify the shared simulation data. The CODB architecture is used for communicating data within the various components of an application and does not provide simulation management functionality. Currently, several CODB simulation classes have been created including support for four input devices, a renderer, a cockpit interface, AFIT's Pod user interface, a DIS interface, and a DIS entity generator.

## 2.9 User Interface Design in a Virtual World

"We can move within the virtual world, flying, floating, looking at any part from almost any perspective, from our vehicle: a virtual "pod" of some design that, with us always, provides us with interface and motion controls, navigation aids, communications devices, and many of the common features of GUIs. Almost all of today's two-dimensional GUI facilities are available "inside" the vehicle from its control panels. For example, windows and menus can pop up from the lower, or slip down from the upper, consoles. These facilities are thus not so much superseded as included in a more evolved system [BENE91]." Benedikt's vision of the ideal VR interface is a close approximation of the SBB's Pod interface. His vision also reminds VR interface designers of the importance of incorporating familiar 2D GUI techniques within VR interfaces.

Graphical user interfaces (GUIs) are considered to be the state of the art in 2D interface design. Windows, icons, pull-down menus and pointers are their hallmark [HORT95]. The paradigm shift from text interfaces to GUIs occurred due to the increased processor speeds and

graphical capabilities of today's machines [MART96]. Another paradigm shift is occurring as processor speeds and graphics capabilities keep increasing and as virtual worlds become pervasive in society. A VRML web browser is an example of this next step in 2½D interface design.

Some VR systems utilize control panels as the user interface. The SBB's Pod is one such example. There are several ways to create these panels. There could be 2D menus that surround a Window on the World (WoW; also known as "desktop VR") display, or are overlaid onto the image. An alternative is to place these control devices inside the virtual world. The simulation system must then note user interaction with these devices as providing control over the world. This is how the Pod interface works [KEST94]. The VR user interface could be restricted to direct interaction in the 3D world. However, this is extremely limiting and requires many 3D calculations. Thus it is desirable to have some form of 2D graphical user interface to assist in controlling the virtual world. These 'control panels' of the world appear to occlude portions of the 3D world, or perhaps the 3D world would appear as a window or viewport set in a 2D screen interface. The 2D interactions could also be represented as a flat panel floating in 3D space, with a 3D effector controlling them [ISDA93]. These 2½D interfaces provide a user with a familiar desktop metaphor capable of extensive functionality while retaining the immersive experience of the virtual world. To create these multi-modal, multidimensional interfaces, an understand of two dimensional and three dimensional controls is required.

### *2.9.1 Two Dimensional Controls*

There are four basic types of 2D controls and displays: buttons, sliders, gauges and text. Buttons may be menu items with either icons or text identifiers. Sliders are used as an analog control over various attributes. Gauges are graphical depiction's of the value of some attribute

or attributes within the world. Text may be used for both control and display. The user might enter text commands to a command parser. The system could use text displays to show the various attributes of the virtual world. An additional type of 2D display might be a map or locator display. This would provide a point of reference for navigating the virtual world or act as a radar screen to locate objects and players within the virtual world. GUI-builders such as XForms provide all of this functionality to the developer for the rapid prototyping of user interfaces.

### *2.9.2 Three Dimensional Controls*

Some systems place the user controls inside of the virtual world. These are often implemented as either a 3D widget or a floating control panel object. 3D widgets are useful for the selection and manipulation of objects. Control panels provide user functionality beyond the scope of widgets. Control panels contain the usual 2D buttons, gauges, menu items, etc. with a 3D representation and interaction style. Both types of 3D user interfaces are discussed here.

There have been a number of published articles on 3D control widgets. Three of these articles are referenced here. 3D control widgets provide interaction methods for directly manipulating 3D objects. One method implemented at both Brown University attaches control handles to the objects so that it can be grasped, moved, or twisted [CONN92]. For example, twisting one handle might rotate the object, while a "rack" widget would provide a number of handles that can be used to deform the object by twisting its geometry [SNIB92]. An extension to this project maps an input device into the virtual world to perform these operations based on the physical location of the input device, thus removing the need for handles in most cases [WLOK95].

Three dimensional control panels can be combined with or without the above techniques to enhance the functionality of a user input device [RIBA94]. Control panels and menus are used for a wide variety of tasks from simple navigation to managing the activity within the virtual world. Unlike the 2D menus described above or used by the Virtual Venues project [BOLT95], a 3D menu that exists as an object floating in space can itself be selected and manipulated to allow the user to take advantage of a menu as needed. When the menu is no longer required, it can be selected and moved out of the user's line of sight. The ability to hide a menu or control panel when no longer needed helps to reduce visual clutter and will be used in the design of the Collaborative Workspace's interface. Selection and movement are common themes in both 3D menus and 3D widgets and will also be used in the design of an interface to control annotations.

## 2.10 Conclusion

This chapter introduced several subject areas related to the creation of a collaborative workspace within the Synthetic BattleBridge. These included virtual reality, computer supported cooperative work, virtual communities, collaborative virtual environments, and distributed interaction simulations. A discussion of software architectures, user-interface design, and an introduction to the Synthetic BattleBridge were also presented.

From this review of background material, the reader should now have an understanding of the solitary nature of VR environments; the techniques used within the CSCW community to eliminate communication boundaries; the social character of large-scale, low-bandwidth, virtual communities; the current state-of-the-art in small-scale collaborative virtual environments; and the communication protocols needed to implement collaboration between DIS applications. The

reader should also understand the basics of the Synthetic BattleBridge, the software architecture that it was built upon and why it needed to change, and the user interface challenges in integrating a collaborative workspace into the current SBB.

# 3 . Requirements and Design

## 3.1 Introduction

This chapter presents the requirements and design for a collaborative workspace within a distributed virtual environment. The primary requirement of this research is to open as many communication channels as possible for collaboration between multiple DIS players. This requirement includes both direct (i.e. private email, public bulletins, and real-time chat) and indirect (i.e. shared viewpoints and annotations) forms of communication implemented through a DIS communication interface. The secondary requirement is to devise a learnable and usable user interface to perform these communication and collaboration tasks. Creating a user interface is an inherently difficult task that is more akin to art than science [MYER94]. This chapter will detail the user interface choices made and why the alternative approaches were inappropriate for this study. Lastly, a host of tertiary requirements were levied on this project such as software architecture, graphics performance, and the distributed simulation interface. These demands played heavily in the design of the collaborative workspace and will be covered as well. Finally, the chapter concludes with a description of the final design.

Table 3-1 presents an overview of the collaborative workspace's requirements and goals. This table provides both an itemized list of what will constitute a successful collaborative workspace and a road map of how this chapter is organized.

**Table 3-1. Collaborative Workspace Requirements**

| Requirement | Detailed Goal |
|---|---|
| **1. Software Architecture** | |
| 1.1. Increase flexibility of simulation framework by eliminating ObjectSim's constraints | Allow all Performer functionality to be available to developers by removing ObjectSim classes from SBB |
| 1.2 Provide architecture that will support all needed simulation components and be extensible | Utilize container-based approach to storing simulation data (CODB) |
| 1.3 Convert existing SBB to CODB architecture | CODB SBB works identically to ObjectSim SBB. Minimal loss of existing functionality. |
| 1.3.1 Pod must utilize CODB-based input and output | All input and output from Pod is CODB based |
| 1.3.2 Utilize CODB containers for simulation management data | Simulation management information will be available to all components in CODB |
| | |
| **2. Graphics Performance and Design** | |
| 2.1 Replace ObjectSim's rendering functionality | Create CODB renderer class using ObjectSim algorithms |
| 2.1.1 Structure Performer tree to support annotation models and viewpoint algorithm | Create similar top-level tree structure as previous ObjectSim-based SBB |
| 2.1.2 Overcome Performer viewpoint resolution problem | Replace origin-centered viewpoint algorithm |
| 2.2 Support current h/w and support future SBB work with industry standards for graphics programming. | Convert SBB from IRIS GL to OpenGL and X-Windows. |
| | |
| **3. Distributed Simulation Interface** | |
| 3.1 Minimize interaction between DIS Manager and SBB | Utilize CODB architecture to support independently running DIS Manager process. |
| 3.2 Manage incoming DIS entity information | Retrieve and store Entity State PDU information from DIS Manager using CODB containers |
| 3.3 Support stealth management and CSCW activities through DIS interface | Use Stealth Entity State, Comment, and Annotation Update PDUs for communication with other stealth players. |
| 3.3.1 Minimize bandwidth caused by additional stealth-generated PDUs | Stealth Entity State operates like Entity State PDU (including heartbeat), Comment PDU sent out once per message, Annotation Update PDU sent only on state change to annotations (no heartbeat mechanism). |
| | |
| **4. User Interface** | |
| 4.1 User interface must be easy to use | Keep consistent interface between textual modes of communication (email, bulletins, chat, annotations) |
| 4.2 Do not clutter the existing Pod controls | Create multi-layered interface, utilize switchable panels, and utilize Selection Manager for interaction with world annotations. Determine appropriate location of controls with respect to previous SBB designs. |
| 4.3 Create a method of text input suitable for use with an HMD. | Design virtual keyboard for future implementation. Expand multi-layered interface to include an text input windows that cannot be misplaced by the HMD user. |
| 4.4 Expand navigation controls | Allow user to detach from Pod. Overload mouse to support Performer "Fly" mode. Allow detached user to return to Pod or move Pod to current viewpoint. |
| | |
| **5. CSCW Capabilities** | |
| 5.1 Support communication of private email and public bulletins | Use Comment PDU with single user or public audience specifier. |
| 5.1.1 Support common message functions. | Implement sending messages, saving a message to a file, deleting a message, displaying messages in received order, and replying to a message. |
| 5.2 Support real-time chat communication | Use Comment PDU with chat audience specifier. |

| | |
|---|---|
| 5.2.1 Support common chat functions. | Implement sending a chat message, saving a chat log file, deleting a chat log file, displaying multiple chat messages in received order on scrolling display with identifiers for each participant. |
| 5.3 Support shared viewpoints | Use stealth Entity State PDU to send stealth position and orientation information. Create HUD to display stealth players' viewpoints with these coordinates. |
| 5.3.1 Support user teleportation to shared viewpoint | Allow user to jump to a viewpoint (with same position and orientation of viewpoint) and return to original coordinates. |
| 5.4 Support animation/video | Allow user to capture, transmit and playback a sequence of HUD images. |
| 5.5 Support annotations with the virtual environment | Create Annotation Update PDU. Create appropriate metaphor and display techniques needed for annotations. |
| 5.5.1 Support annotation functions | Implement creating an annotation, deleting an annotation, appending text to an annotation, clearing messages from an annotation, selecting an annotation, moving an annotation, and attaching an annotation to a DIS entity. |

## 3.2 Software Architecture

The SBB's software architecture must be able to quickly move data between the classes of the application. An example of this situation is that the radar panel displayed by the `SBB_Renderer` must share data concerning the DIS entities (i.e. entity location, entity medium, entity affiliation) with the `SBB_WSM` (World State Manager) and the `DIS_Manager`. The previous Synthetic BattleBridge was constructed with ObjectSim, a peer-to-peer architecture. In ObjectSim, `externs` are commonly used throughout the application to pass needed data structures, and their accompanying class methods, between classes. An alternative approach, the Common Object Database (CODB), uses a centralized data repository to store and access information. Each of these architectures has its own advantages and disadvantages in regards to design issues and these will be covered in this section.

### *3.2.1 ObjectSim*

ObjectSim has a large advantage over the CODB in that the previous SBBs were implemented using this architecture. Keeping the ObjectSim architecture would result in few structural changes to the SBB. The ObjectSim SBB (without the collaborative workspace) operates at acceptable frame rates ( >15 frames per second). The ObjectSim libraries provide extensive functionality for distributed simulation applications, include support for remote viewpoints, attachable players, and model management. This functionality also leads to ObjectSim's greatest drawback. There is a considerable learning curve involved with understanding the large quantity of ObjectSim classes and methods. To make effective use of these classes, the developer must also be intimately familiar with the Performer libraries. This is due to ObjectSim's tight coupling with the Performer libraries. The problem with this situation is the inherent difficulty tight coupling causes with regards to changing the libraries or providing new functionality to the existing libraries. For example, ObjectSim handles input devices through its `Modifier` class. This class assumes that the input device will somehow manipulate the simulation's current view. Thus, if the developer wishes to use the keyboard for some other purpose (i.e. typing text messages), he must override or disable these viewpoint modifying methods, effectively crippling the `Modifier` class. Finally, while the target platform for the new SBB is a single processor system, it should be noted that ObjectSim makes no attempt at trying to handle shared data between multiple processors. This was beyond ObjectSim's scope of providing a generic, Performer-based, simulation toolkit.

### *3.2.2 Common Object Database*

The Common Object Database (CODB) architecture was created to support rapid prototyping of distributed virtual environments. The CODB allows classes to communicate with each other through a centralized database. The CODB permits a certain amount of class abstraction by removing the need for a class to provide specialized data access methods. The CODB developer uses generic CODB-provided methods to access the data stored in the common database. To retrieve or modify the desired data, the developer only needs to know the name of the container in which the data is stored (see Figure 3-1). To support multiprocessing environments and to reduce the delays in accessing data, semaphores and double-buffering are used. Semaphores are used to control access to the containers, preventing a reader process from retrieving information while a writer process is modifying it. Double-buffers are used to allow separate processes to read from and write to the container simultaneously, thus reducing the access delays often associated with sharing data between processes. The disadvantage of the CODB is that it does not include a large number of simulation-development classes like ObjectSim. For this reason, many ObjectSim classes are being rewritten to support the CODB architecture.

It was required that the Common Object Database architecture would be used for the new SBB. The reasons for this decision were compatibility with the DIS Manager 3.0 architecture, the data-handling support for multiprocessing applications, the ease of use in reducing class access methods by using CODB containers, and the difficulty in expanding the ObjectSim libraries. All of the new distributed simulation applications within AFIT use version 3.0 of the DIS Manager. The current DIS Manager was created using the CODB architecture to

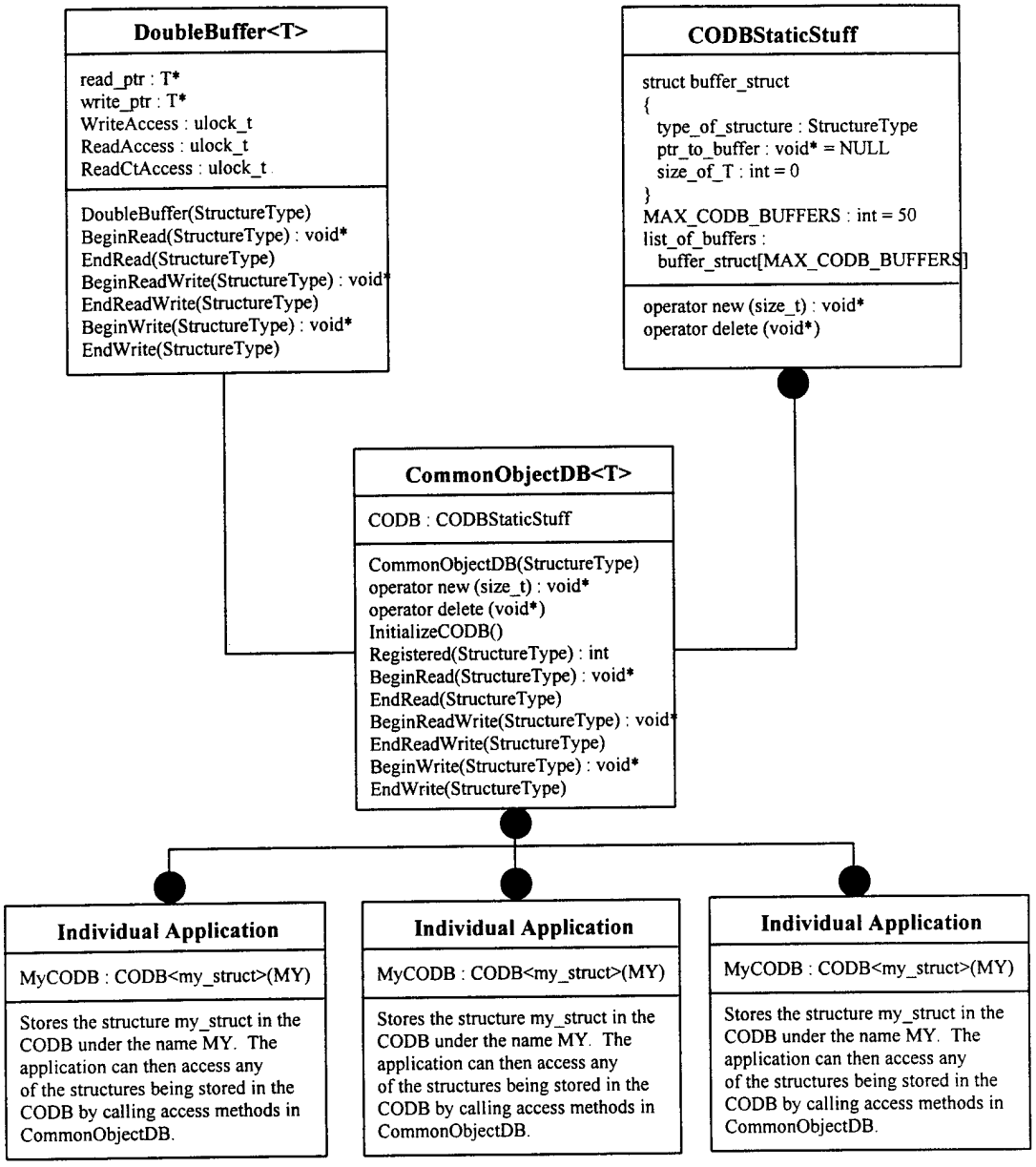



**Figure 3-1. Common Object Database Rumbaugh Diagram**

allow it to run in its own separate process, gathering information about the DIS environment as quickly as possible, with minimal interaction from the host application. The only contact between the DIS Manager and the host application is through shared data containers and calling

methods to dispatch data. The CODB's data-handling functions support the multiprocessing nature of the DIS Manager and the SBB through the use of the double-buffering mechanism. The previous ObjectSim-based version of the DIS Manager required a great deal of interactivity between itself and the host application. This amount of interaction lead to specialized versions of the DIS Manager being created exclusively for each distributed simulation. Using the CODB architecture, a single DIS Manager is now used for all of the current distributed simulation applications. More information about the interaction between the SBB and the DIS Manager 3.0 is included in section 3.3.

The ease of use in abstracting access to class data structures through the use of containerization cannot be overestimated. Only three methods are available to begin accessing a CODB container:

- `BeginRead`, where a filled container pointer is passed to the application for read access,
- `BeginWrite`, where an empty container pointer is passed to the app. for write access, and
- `BeginReadWrite`, that first performs a `BeginRead` to fill a write access container pointer that is passed back to the program for write access.

To finish accessing a CODB container, an `EndRead`, `EndWrite`, or `EndReadWrite` method is called as appropriate. A further advantage of the CODB is that since shared data is accessed through a container, that container's name must be present at the opening and closing of every CODB container. Thus, the "ownership" of particular data items within the container is clear to the reader. This is not always the case when `externs` are used to import the data from one class into the methods of another. Finally, the difficulty in expanding the ObjectSim library was another reason to choose the CODB architecture. For all of these reasons stated above, the CODB architecture was required for the development of the SBB's Collaborative Workspace.

## 3.3 Graphics Performance & Design

The ObjectSim SBB maintained acceptable levels of performance in graphics speed and image realism. The SBB's Collaborative Workspace must meet or exceed these levels of performance to be considered a success. Specifically, the SBB would need to maintain a rich graphical environment at 15 frames per second (fps) on a single processor machine. The 15 fps figure was the goal of the ObjectSim SBB (with Pod interface). In this section, I will discuss the decisions involved with designing the SBB's renderer class and why certain steps were taken to keep graphical performance levels high through the use of industry standards.

### *3.3.1 SBB's Renderer Class*

As stated by the software architecture requirements, the ObjectSim functionality found within the SBB must be replaced with CODB-based classes and methods. The most involved conversion occurs with the `SBB_Renderer` class. ObjectSim's rendering functionality involves the updating of simulation entities and the integration of terrain files into simulation. To promote as much reuse of ObjectSim code as possible, this design was adopted in the new `SBB_Renderer` class. The design approach taken was to utilize a Performer tree similar to the one maintained by the previous SBB. This Performer tree hierarchy places the Pod, the terrain, and the DIS entities all as separate Performer subtrees. See Figure 3-2 for a diagram depicting this arrangement. The `SBB_Renderer` class would then be used to manage all top-level Performer tree activity.

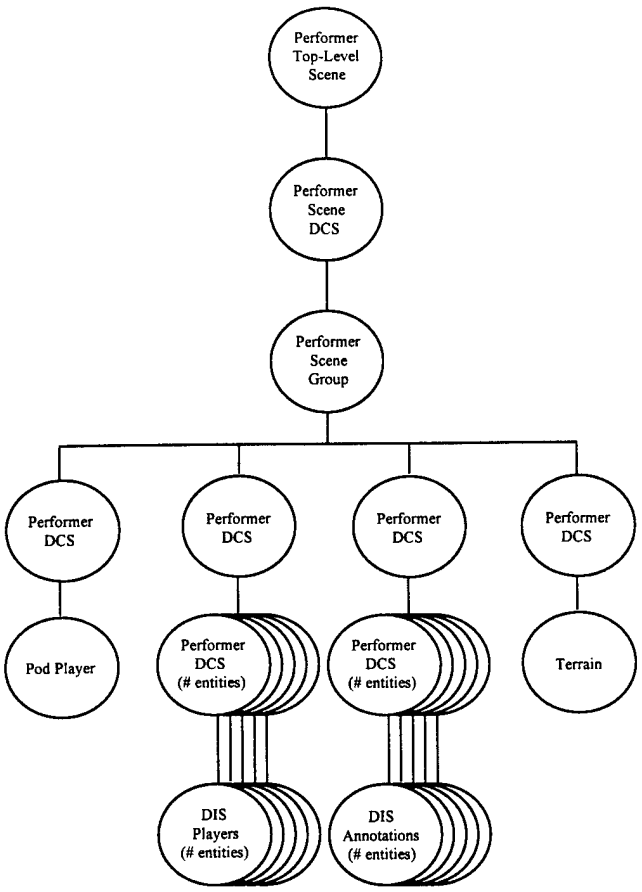



**Figure 3-2. SBB's Top-level Performer Tree.**

One of the most important ObjectSim renderer algorithms that needed to be converted to the CODB format involved the movement of the Pod in relation to the rest of the world (e.g. the terrain and the DIS entities). Performer uses an array of three floats (x, y, and z) to update the position of each object within the world. In large virtual environments, such as those used for DIS exercises, a problem in maintaining the resolution of these positional coordinates occurs as the floats increase in value. This problem results in the player's coordinates fluctuating enough that the user's viewpoint becomes unstable, resulting in an erratic display of the world. The solution to this problem is to place the user (e.g. the Pod) at the center of the coordinate system and have the rest of the world (e.g. terrain, DIS entities, annotations) move in relation to the

user's perceived position and orientation. The algorithm used to perform this task was taken from ObjectSim's `View` class and is shown in Figure 3-3.

```
// Algorithm to move the viewpoint of the Pod to the origin and to move other entities
// and terrain relative to the Pod. The algorithm also considers an offset.

// Set position of OriginCoord to origin (x = y = z = 0.0)
        PFSET_VEC3(OriginCoord.xyz, 0.0f, 0.0f, 0.0f);
// Set orientation of OriginCoord to Pod's orientation
        PFCOPY_VEC3(OriginCoord.hpr, Pod->Coords->hpr);
// Translate and rotate Pod's geometry by OriginCoord
        pfDCSCoord(My_Model->RotDCS, &OriginCoord);
// Set OriginCoord to Pod's orientation plus any offset
        PFADD_VEC3(OriginCoord.hpr, Pod->Coords->hpr, offset->base_rot);
// Set position of OriginCoord to origin (x = y = z = 0.0)
        PFSET_VEC3(OriginCoord.xyz, 0.0f, 0.0f, 0.0f);
// Make a coordinate transformation matrix from OriginCoord
        pfMakeCoordMat(viewmat, &OriginCoord);
// Transform the offset by coordinate transformation matrix
        pfXformPt3(Result, (*attached)->base_offst, viewmat);
// Set channel's View parameters
        pfChanView(chan,  Result, OriginCoord.hpr);
// Negate the resulting transform (opposite direction of Pod)
        PFNEGATE_VEC3(Result, (*attached)->Coords->xyz);
// Move the players in the opposite direction of the Pod
        pfDCSTrans(playertrans,  Result[PF_X], Result[PF_Y], Result[PF_Z]);
// Move the terrain in the opposite direction of the Pod
        pfDCSTrans(terraintrans, Result[PF_X], Result[PF_Y], Result[PF_Z]);
```

**Figure 3-3. Viewpoint Algorithm From ObjectSim's View Class**

Finally, the `SBB_Renderer` would also reuse ObjectSim-based code to display radar screens and perform model management. The radar screens project all DIS entities down onto a two dimensional grid to determine their position in relation to the Pod. If the entity falls within the user-selected range of the radar, the entity's information is passed to an icon generator for display onto the radar screen. Model management is an extensive process that registers each DIS

entity with a particular high-resolution model, a low-resolution locator model, an entity's sensor range, an entity's threat range, and a movement trail to display previous entity positions.

### *3.3.2 Graphics Standards and Performance Issues*

Upon talking to the developers of the Performer libraries at SIGGRAPH'96, it was decided to replace all of the SBB's IRIS GL (Graphics Library) code with the appropriate X-Windows and OpenGL counterparts. The reasons to undertake such a conversion were provided by the Performer developers. First, the current SGI graphics hardware is based on OpenGL and X-Windows instructions. IRIS GL-based programs must go through a software conversion layer known as "IGLOO" (IRIS GL On OpenGL) to perform their graphics instructions. This additional layer results in a significant graphical performance hit. Approximate figures are 10% for an SGI Impact machine and 50% for an SGI InfiniteReality machine [JONE96]. The second reason for this conversion is that the Performer 2.X libraries were written to support OpenGL and X-Windows functionality. IRIS GL compatibility libraries are available to maintain legacy code, but these libraries are slower due to the IGLOO layer and the nature of IRIS GL event handling. IRIS GL input devices are either polled or read from an event queue during each draw cycle to determine if any input has been received. OpenGL relies on X-Windows for event handling and window management. X-Windows allows input devices to be accessed at any time within the program, within any process. This means the SBB can increase its rendering performance by moving the input device queries outside of the time-critical draw process and into the application process. The final reason for switching to OpenGL and X-Windows was to facilitate future platform independence for the SBB's Collaborative Workspace. For these reasons of performance and future SBB expansion, it was decided to convert the SBB's legacy IRIS GL code into their OpenGL and X-Windows equivalents.

## 3.4 Distributed Simulation Interface

The SBB's distributed simulation interface will continue to use the Distributed Interactive Simulation (DIS) protocol. Originally, we intended to use the High Level Architecture (HLA) as the SBB's distributed simulation interface, but due to the lack of a completed HLA specification, we were unable to accomplish this task. The SBB will instead use the DIS Manager 3.0 libraries, created by Sheasby, for its distributed simulation interface. The current DIS Manager provides its host application with a CODB-based interface to the DIS environment. The DIS Manager uses a variety of CODB containers to pass information between itself and the SBB. These structures and their respective PDUs will be outlined in this section.

### *3.4.1 WSMEntityStruct*

The `WSMEntityStruct` is a CODB container that holds the most important information in the DIS exercise, the current state of each DIS entity as known through Entity-State PDUs and dead-reckoning algorithms. For each DIS entity, the following information is relayed to the SBB through the `WSMEntityStruct`: the entity's alliance, entity type (air vehicle, land vehicle, space vehicle, munition, etc.), the entity's position and orientation, an entity description, and the type of model (guise) used to represent the entity. Alliance and entity type are used to determine appropriate graphics for locators and radar images. The entity's coordinates are Earth-centered and must be converted to the flat-world coordinate system for use by the Performer application. The description of the model is used by the SBB to label each DIS player and the model number informs the `SBB_Renderer` class which graphical model to add to the Performer tree for that particular DIS player. This information is written to the

`WSMEntityStruct` container by the DIS Manager and is read by the SBB's World State Manager and renderer classes. No writing to this structure is allowed except through the DIS Manager. No calls are made to update this structure by the SBB. Updates are handled autonomously by the DIS Manager. Every cycle the structure is examined by the SBB for updated information. The SBB stores this information locally (and with Performer-based coordinates) within the `PerformerWSMStruct` container.

### *3.4.2 WSMMgtStruct and OwnMgtStruct*

Most DIS players broadcast their state to every player on the network. A DIS stealth player does not broadcast its state to the rest of the network, but silently listens to all of the DIS network traffic to gain a "big picture" perspective of the battlespace. The Collaborative Workspace forced this paradigm to change. In order for SBBs to communicate with each other, they must broadcast their own "Stealth Entity-State" PDU. This stealth state PDU includes the position and orientation of the stealth player and a description of the stealth player. Additionally, a Comment PDU was designed to broadcast text messages between stealth players. The Comment PDU contains a 256 character comment string (equivalent to a full screen of Pod text), an audience type specifier, the sender's id, and a receiver's id for sending private messages. The Annotation Update PDU was designed to be a combination of the Comment PDU and the Entity-State PDU. Each Annotation Update PDU includes the following information: a unique annotation identifier, the sender's id, the annotation's coordinates or the DIS entity's id that the annotation is currently attached to, the current text message associated with the annotation, a Boolean state to determine whether or not the annotation has been selected, and one of a series of annotation actions (create, delete, move, select, update, append, etc.). See Appendix B for the complete Annotation Update PDU specification. The information from

Stealth Entity-State PDUs, Comment PDUs and Annotation Update PDUs form the `WSMMgtStruct` CODB container. Within this container is stored the status of each stealth player, the messages currently being sent, and the status of each annotation. This information is available for read access by the SBB and write access by the DIS Manager. Similar to the `WSMEntityStruct`, it is examined each cycle to update a local copy of this information in the simulation manage section of the `PodStruct` container. In order to broadcast stealth information using the DIS Manager, a similar container was designed to be filled by the SBB application and passed to the DIS Manager for transmission. This CODB container is called the OwnMgtStruct. Broadcasting occurs as a result of any SBB movement, the need to send a message, or the desire to add or modify an annotation. Three DIS Manager calling functions were designed to handle each of these requirements. They are `broadcast_steath_state`, `broadcast_comment`, and `broadcast_annotation _update`.

## 3.5 User Interface

A guiding principle for the design of the workspace interface was to keep the interface of the Synthetic BattleBridge essentially the same. The rationale was to shorten the learning curve of the Collaborative Workspace by making maximum use of the existing interface structure. This meant adapting as many workspace controls to the Pod's interface as possible.

Unlike the complex user interface models of aircraft cockpits and medical equipment that rely upon a `Selection Manager` to successfully blend photo-realistic controls with user input [ADAM96] [GARC96], the Pod is well-equipped to handle most console-based user interface designs. This section will not detail the original Pod's user interface design. For an thorough coverage of the Pod interface, the reader should examine the work of Kestermann and

Rohrer [KEST94][ROHR94]. What will be discussed is how the Pod design was modified and how alternative interfaces were incorporated into the overall user interface.

### *3.5.1 The Pod*

The initial CODB demonstration effort resulted in a CODB Pod that the user could detach from and travel untethered throughout the virtual environment to quickly inspect an event that would be over before the standard Pod interface could travel to that location. Using a style similar to SGI's "Fly" mode, the user controlled movement solely through mouse input. The left mouse button accelerated the user, the right button decelerated the user, and the center button halted the user's momentum. Direction of travel and user viewpoint were controlled by the position of the mouse relative to the center of the screen. With the addition of a feature that allowed a detached player to jump back to the Pod's original position or teleport the Pod to the user's current viewpoint, this improved interface was too feature-laden to be replaced from scratch with the `Selection Manager`.

The Pod design is composed of five large control panels were available to the user in each of the forward-facing cardinal directions: center, left, right, top, bottom. The goal of the workspace's user interface design was to make the best use of these five screens without creating additional panels. The top and bottom panels were filled with radar information that could not be removed. The left panel contained detailed DIS entity information and a control for attaching the user to an entity or a predefined camera position. This meant that the center panel (used for navigation) and the right panel (primarily used for displaying HUDs) would be the most likely choices for adding collaborative workspace features.

The right panel could be easily adapted to include the shared viewpoint and animation capture/transmission/playback controls as described previously. The shared viewpoint would

only be available when other stealth entities were participating within the DIS exercise, although the animation controls could be used at any time to capture other HUD displays. Jumping to a viewpoint would be performed by storing the Pod's current coordinates for the return trip and then copying the viewpoint's coordinates into the Pod's CODB structure.

The center panel contained six subpanels, five used for navigational controls. The sixth, in the upper left corner of the center panel, can provide an access point to a unique feature available to the Pod interface, but not to the `Selection Manager`. Using object-oriented programming techniques, a panel can be temporarily hidden from the user and replaced with an entirely different set of controls. This functionality provides an unlimited amount of console space by simply "flipping through" various control panels until the desired controls are displayed [ROHR94]. Using this technique, it was decided that a single button would be placed on the center navigational panel that would clear the console and provide access to three full-sized panels "hidden underneath." These three panels would control email and bulletins, chat sessions, and annotations.

In addition to the access button, the workspace subpanel also displays statistics on the number of emails, bulletins, and annotations received. This status panel would let the user keep track of incoming messages in a manner similar to a telephone answering machine. An alternative approach would be to place a row of graphical tabs (like those used in spreadsheet programs) across the top of the center panel. Each tab would have a number and color representing the order that the message arrived and the type of message received. Clicking on the tab would display the message and its appropriate controls package. The problem with this approach is twofold: there is a finite amount of space along the top of the panel (e.g. a limited number of messages could be available for display) and placing graphical tabs large enough to be

functional would block too much of the user's view on the virtual world. Due to these problems, the "answering machine" design was chosen for implementation.

### *3.5.2 The Selection Manager*

The `Selection Manager` class is a tool for picking particular objects out of a scene and identifying them to the user (and the application). The Pod facilitates 2-D selection, but picking an object in 3-D space is the purview of the `Selection Manager`. The `Selection Manager` class makes use of an optimized picking function built into the Performer libraries. This Performer picking function computes an intersection with the visible Performer geometry based upon the mouse's position in the window. If any Performer geometry exists underneath the mouse cursor when the function is called, the path through the Performer tree leading to that geometry is returned. By naming "selectable" objects (such as annotation models) with a special prefix and integer, the `Selection Manager` can inform the developer when any of these special objects has been picked. This information can be used with the annotation to highlight selected annotations or pick annotations that need to be moved. Objects that have not been specially tagged (such as the Pod panels) can be ignored. This allows the developer to have a multi-layered interface (e.g. a 2½D Pod interface that is overlayed above the 3D `Selection Manager` interface).

### *3.5.3 Text Input - The Virtual Keyboard and the XForms Library*

The original design for text input used a virtual keyboard technique that would allow a user wearing an HMD to type messages using a standard keyboard. In brief, the idea was to place positional sensors on the user's index fingers. The user would start the program and be

prompted to calibrate the sensors by placing his fingers on the two home keys (F and J). Inside the virtual environment, the bottom panel would be replaced with a photo-realistic layout of the workstation keyboard. The tracked fingers would be represented by two green dots overlayed on the keyboard image. As the user typed on the physical keyboard, a screen above the virtual keyboard would display what was being typed. At the same time, every keystroke would be reflected on the virtual keyboard by a red dot at the corresponding finger's position. In this manner, a user in an HMD could perform simple hunt-and-peck typing with the physical feedback of the actual keyboard and the visual feedback of the virtual keyboard. Although this technique was not implemented in this thesis due to hardware failures in the positional trackers, it is believed that a thin glove attached with small, lightweight sensors will provide a viable means of using a keyboard with a head-mounted display.

Unable to implement a virtual keyboard, it was decided that a 2D window layer should reside above the Pod interface to provide text input to the collaborative workspace. This interface relies heavily on the keyboard and thus is only useable with see-through HMDs. The third layer of the multi-layered interface is a X-Windows dialogue box. Two X-Windows GUI-builders were immediately disqualified for this task: Motif [OSF91] because of its high learning curve and GL Utility Toolkit (GLUT) [KILG96] because of its lack of forms-based input. Forms-based input is provided by the XForms Library [ZHAO96], an API for quickly developing custom-built, Motif-styled dialogue boxes. The XForms Library is easy to use, freely available, and includes both sample code and a forms designer. XForms dialogue boxes are displayed at the top of the XWindows stack and thus the calling application can continue below the XForms window, completely unaffected. In visual simulations, this gives the illusion that the dialogue box is hovering in front of the eyes of the user. Wherever the user turns in the environment, the dialogue window stays on top and in view. This feature prevents a user from

losing the interface, an important concern in virtual reality applications. The multi-layered approach also gives a user access to each of the user interfaces without affecting the others.

## 3.6 CSCW Capabilities

The SBB's Collaborative Workspace must take advantage of as many methods of communication as possible to support CSCW activities between multiple SBB users. This interaction between players is not limited to the typical direct modes of communication found in most office environments (e.g. private email, public bulletins, and real-time chat), but also includes non-traditional indirect methods (e.g. shared viewpoints and annotations). Voice messages were not considered due to a lack of hardware support. This section describes the goals to be accomplished and the design paths chosen for implementation.

### *3.6.1 Private Email and Public Bulletins*

Email is the most common form of electronic communication used on computer networks today. According to Georgia Tech's Fifth Annual World Wide Web User Survey, 75% of all respondents said that they use the Internet for email, 36% accessed some form of public bulletin board or newsgroup and 25% visited chat rooms regularly [GEOR96]. These figures emphasize the need to support these forms of communication within the SBB's Collaborative Workspace.

Private email is simply a text message directed from a sender to a specific recipient. The recipient must be able to display this message and the identity of the sender so that a reply can be formed if necessary. Email messages are typically stored within the user's mailbox in the order that they arrived. Three other features found in most email systems are the ability to delete

messages, write messages to a file, and reply to a particular message. Messages deleted from the system cannot be recovered. Saved messages are commonly sent to a single file for simplicity. Replies to private emails are typically private themselves.

Public bulletins are announcements sent to all users on the network. Examples of this form of communication include messages about network outages, office events, paperwork suspenses, or items for sale. Public bulletins have the same requirements of a private email system with the exception that the targeted audience includes all members on the network. Replies to such messages are either public or private in nature.

Due to their commonalties, public bulletins and private emails are accessed through the same user interface. For this reason, it was decided that the only displayed difference between the two types of received messages would be an identifier of "Public" or "Private" next to the sender's name.

### *3.6.2 Real-time Chat*

Email and bulletins do not require the user to be aware of a message for the user to receive it. In a chat environment, the user must consciously decide to join the chat group to receive the messages being sent by its participants. Within a chat session, each player sees the messages of every active participant, typically with a prefix identifier added to the message giving the sender's name. Since the number of displayable messages is limited, old messages are scrolled off the top of the screen as new ones are appended to the bottom. This style is consistent with that used by MUDs and the MASSIVE text interface.

Unlike the asynchronous nature of email and bulletins, real-time chat requires some form of synchronous communication between the chat participants. This interaction can be immediately synchronized to display each keystroke as it is typed (i.e. the Unix *talk* utility) or

delayed so that a complete text string is sent when the user presses the `Enter` key (i.e. the Internet Relay Chat (IRC) environment). Due to the nature of DIS and the existing Comment PDU, the IRC-style interface for sending chat messages was used.

### *3.6.3 Shared Viewpoints*

With the SBB now communicating with other stealth players, it was necessary that a Stealth Entity-State PDU be transmitted to keep the stealth player "alive" in the exercise to the other stealth entities. An Entity-State PDU contains position and orientation information about a player that can be used to correctly display his representation (or avatar) within the virtual environment. This representation typically takes the form of a three-dimensional model or a radar image. This information can also be used to look through another stealth player's eyes and see the world from his perspective. The capability to view what another entity sees is called a "shared viewpoint."

Remote viewing through the use of a heads-up display (HUD) had already been implemented in the SBB by Kestermann and Rohrer [KEST94][ROHR94]. Remote cameras, video missiles, and radar displays proved to be valuable aids in situational awareness. The shared viewpoint design would mimic these previous implementations with two important additions: the ability to teleport to and from the shared viewpoint's position (with the correct orientation) and the ability to capture, transmit, and playback a sequence of HUD images.

The shared viewpoint teleportation mechanism would also be used within the annotation controls. Unlike the annotation scheme where the player is moved to an offset position of the annotation in question, shared viewpoint teleportation transports the user to the exact position and orientation of the remote stealth entity. This operation is similar to using a television with

picture-in-picture technology. Primary attention is focused on the main screen, but occasionally something of interest occurs within the inset screen. The user is able to toggle between the screens if the event warrants closer examination. This is the same thing that we propose to do with the shared viewpoint, only in 3-D space. Essentially, this allows a user to "step into the shoes" of another player with the ability to jump back to the original location.

A player may want to capture a sequence of images that occur within the HUD and then share these files with other players who could display the sequence as an single animation. These animation controls must be integrated together as a single unit within the HUD interface module to facilitate quick transfers.

### *3.6.4 Annotations*

The goal for the annotations was to create a three-dimensional whiteboard attached to a particular location in the virtual world. Players could communicate their ideas through this shared interface.

Several issues narrowed the scope of this aggressive plan. The first concern was graphical in nature. How should the computer display the contents of the whiteboard in three-dimensional space? Texture maps were too unwieldy, requiring each user to maintain a copy of the image currently being displayed. This severely hampered the interactive interface needed for collaboration and would likely create unacceptable amounts of network traffic to maintain synchronicity. Another idea was to take the `GraphText` and `GraphFont` classes to generate 3D text strings, similar to those used for the Pod's control labels, and attach the messages to 3D billboards. This design was tabled when a similar interface was seen within the AlphaWorld environment that was discussed in chapter two. AlphaWorld uses 3D billboards to display text messages, but one of the problems with this interface is that a user must correctly position

himself to read the messages displayed. Even when billboards are automatically turned towards the user, text could not be read from a reasonable distance due to the size of the fonts necessary to display lengthy messages (~256 characters). A solution to this problem would be to have the messages displayed on the Pod console, but the method of selecting or controlling the physical annotation models would still need to be resolved.

The final annotation interface design was to take the previously designed email and chat interfaces and Captain Garcia's `Selection Manager` class and combine them for use with controlling annotations. In this manner, annotations could be displayed on the Pod's console either by cycling through the annotations using the standard forward/back button controls or by clicking on the physical model of the annotation and requesting that its contents be displayed. The ability to select annotations also provides an intuitive method of repositioning annotations and thus, Captain Garcia's `Movement Manager` could be modified for this task. To easily locate annotations at a distance, screen-space labels were placed above them.

As well as moving annotations around the virtual environment, it was determined that they should have the ability to be attached to a DIS entity. While a moving annotation is not practical for interactive communication, it is useful for applying a label or some other pertinent information concerning the host entity. For example, workspace users could click on a DIS entity's annotation to discover the name of unit, its commanding officer, and its current mission.

The last annotation design issue concerned the icon used to represent an annotation. Ideal metaphors were not intuitively obvious and no standard for marking annotations in virtual environments existed. NPS had used large arrows to mark items of interest [MACE94], but arrows do not carry a message connotation and are primarily used as an attention-focusing device. Representations of actual whiteboards are used by Dive and MASSIVE [HAGS96] [BENF96], but as the annotation messages would be displayed on the Pod's console and not on

the annotation itself, this design was not used. It was decided that a flag or pennant be used to represent an annotation. Flags are a more generic icon than pushpins and thus provide an abstract symbol for attaching non-textual messages like pictures, sound files, and video clips.

## 3.7 Conclusion

Given the requirements in Table 3-1, the design for a collaborative workspace within the Synthetic BattleBridge was developed that utilizes the CODB architecture, supports open graphics standards for increased performance and future expandability, communicates with other stealth players through a DIS interface, provides a multi-layered user interface, and supports CSCW activities. This design is seen in Figure 3-4. The new SBB's design consists of a Common Object Database at the center of the application that is the primary mechanism for sharing data between simulation classes. The `SBB_Renderer`, located on the left side of the figure, is a super class of the `AFIT_CODB_Renderer`. The `SBB_Renderer` is responsible for setting up the Performer environment, managing the top-level Performer tree, managing entity models (including locators, trails, and range cones), and manipulation of the current view. The SBB_Renderer utilizes the `Selection_Manager` and `Motion_Manager` classes to select and move annotations within the virtual environment. Input devices, located on the right side of the figure, are responsible for user input into the simulation. The SBB's World State Manager is located at the bottom of the diagram and is responsible for communicating with the DIS Manager. The `Pod_Player` class takes up the right side of the figure and is responsible for maintaining all of the Pod's panels and their respective functions. Each panel is shown with an aggregated listing of its particular components. All of the simulation components Figure 3-4 use the CODB for both input and output.

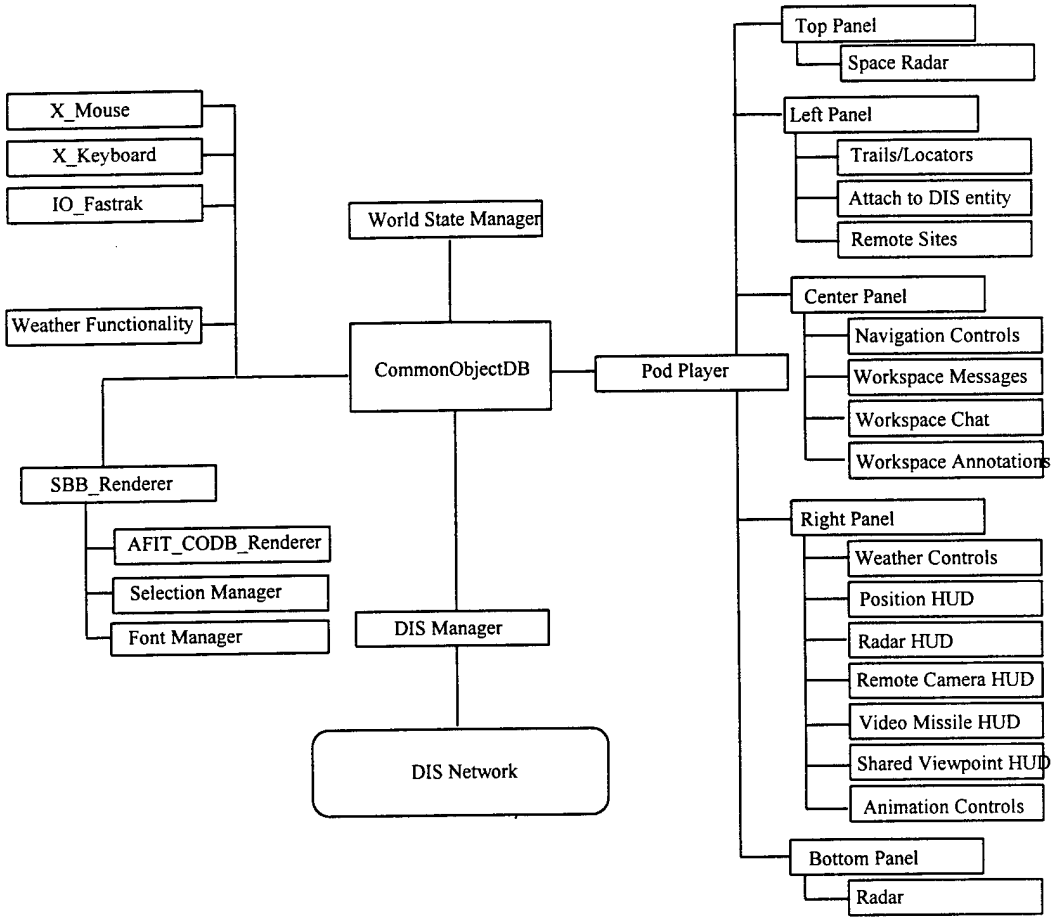



**Figure 3-4. Overall Design of the New SBB**

# 4. Implementation

## 4.1 Introduction

This chapter will cover the implementation of the requirements and design presented in Chapter 3. An incremental approach was taken in the implementation of the design, with each design component of the SBB's Collaborative Workspace representing a separate stage of development. The design components for this research project include the integration of the CODB software architecture, replacing ObjectSim functionality, improving performance through the use of graphics standards, interfacing with the DIS network, enhancing the user interface, and providing support for CSCW activities. Each design component was built upon an operational baseline, with the initial baseline represented by the CODB demonstration program. Subsequent baselines arose from each component's successful implementation. This approach allowed for progressive testing of the application after each stage of implementation. Each design component's testing included its interaction with previously implemented components. These component-wise implementations were not completely independent of each other as the DIS interface, the user interface, and the support for CSCW activities required tight integration. The design components were implemented in the order presented above.

## 4.2. CODB Software Architecture

The first task of this project was to port the previous Synthetic BattleBridge from the ObjectSim architecture to the CODB architecture. The challenge in performing this task was the removal of all ObjectSim components while insuring that minimal functionality was lost in the

conversion process. Instead of taking the previous Synthetic BattleBridge and removing all ObjectSim references and replacing them with CODB counterparts, it was decided to rebuild the SBB from scratch using the CODB architecture. One of the benefits of this approach was that it allowed the task of rebuilding the SBB to be broken down into manageable pieces. It also improved the understanding of each SBB component.

The first step in creating the new CODB-based SBB involved breaking apart the old SBB into its various components. Each class was taken out of the previous SBB and turned into its CODB counterpart. The design of the previous SBB involving many tightly-coupled components that communicated via shared memory structures. These individual structures could be grouped into one of three types: information concerning network entities, information concerning the Pod and its operation, and information concerning weather and the current view. Therefore, the shared memory structures were divided into CODB containers centered around three new CODB classes: the SBB's World State Manager, the Pod Player, and the SBB's Renderer.

The Pod Player component of the SBB includes classes for the each of the Pod's five panels (top, center, left, right, and bottom). Each panel is composed of separable control classes such as navigation, heads-up displays, weather, attaching to DIS entities, etc. These controls use `panel_types`, `sub_panel_types`, `button_types`, and a specialized `mouse` class to divide the functionality of each panel into manageable parts. This structure was maintained in the new SBB. The SBB's World State Manager class replaced the old `SBB_Net_Mgr` class due to the vast differences between the ObjectSim and CODB ways of communicating with the DIS Manager. Unlike the constant polling required in the ObjectSim SBB, the CODB architecture allows the DIS Manager to operate with minimal interaction from the SBB. Finally, the

rendering support functions found in the SBB_Net_Mgr were moved to the SBB_Renderer to isolate entity rendering into a single class. This conversion is discussed in section 4.2.

The first component of the new SBB to be converted to the CODB architecture was the Pod Player class and its center navigational panel. The Pod Player class is responsible for initializing the panels, attaching the panels' geometry to the Pod, updating the panels, and performing positional changes to the Pod as required by outside forces (i.e. resetting the Pod, attaching the Pod to an entity, etc.). The Pod Player does not use any ObjectSim classes and therefore was easy to integrate into the initial baseline. A single CODB container, PodStruct, was created to replace all shared data structures that referenced the Pod or one of its panels or subpanels. Each shared data structure reference was replaced with its CODB equivalent in PodStruct. This allowed the center panel and its six navigational subpanel classes to communicate with each other through the CODB. The Pod Player and its associated panel classes were well encapsulated from the rest of the SBB simulation components. As such, they provided a good first step at converting a complicated simulation component over to the CODB architecture.

This process continued for each of the remaining five panels. Given the time limitations imposed on this research project with over 20 distinct subpanel classes remaining to convert, the chemical weapons subpanel has not been implemented. Each subpanel usually adds another variable, typically a Boolean value, to the PodStruct. The final PodStruct, including variables for all Pod panels and the Collaborative Workspace, is shown in Figure 4-1. Variables names reflect the names used in the shared memory structure.

```
struct pod_struct
{
   pfCoord Pod_Position;

// Center Pod Panel
   boolean Hide_Panel;
   boolean Panic_Button_Pressed;
   boolean Pause_Button_Hook;
   boolean Toggle_Forward_Hook;

// Right Pod Panel
   boolean Radar_On;
   boolean Pod_Position_On;
   boolean Remote_Camera_On;
   boolean Video_Missile_On;
   boolean Shared_Viewpoint_On;
   boolean Nothing_On;
   boolean Capture_On;

// Left Pod Panel
   boolean Using_Site_To_Site;
   boolean Attached_To_Player_Last_Time_Flag;
   boolean Attach_To_A_Player_Flag_Hook;
   int     Attached_Player_Number_Hook;
   pfCoord Attached_Player_Coords_Hook;
   float   Attached_Player_Velocity;
   boolean Locators_On;
   boolean Trails_On;
   boolean Threats_On;
   boolean Ranges_On;
   boolean F_15_Threats_On;
   boolean F_15_Ranges_On;
   boolean Mig_31_Threats_On;
   boolean Mig_31_Ranges_On;
   boolean SA_6_Threats_On;
   boolean SA_6_Ranges_On;
   boolean Threat_Envelope_On;
   boolean Range_Envelope_On;

// Workspace Panels
   siman_struct siman;
   boolean Workspace_On;
   boolean Message_On;
   boolean Annotation_On;
   boolean Chat_On;
   int Annotation_Selected;
   int Current_Stealth_Player;
   message_struct* Current_Message;
   int    Current_Message_Num;
   message_struct* Current_Send_Message;
   int    Current_Send_Message_Num;
```

```
    message_struct* Current_Send_Chat;
    int     Current_Send_Chat_Num;
    annotation_struct* Current_Annotation;
    int     Current_Annotation_Num;
    annotation_struct* Current_MyAnnotation;
    int     Current_MyAnnotation_Num;
    annotation_struct* Current_Send_Annotation;
    int     Current_Send_Annotation_Num;
};
```

**Figure 4-1. PodStruct CODB Container.**

The SBB's World State Manager, My_WSM, was the second major CODB component to be added to the new SBB. The task of the SBB's WSM is to gather data about the DIS exercise and store this information within a CODB container, PerformerWSMStruct. The final PerformerWSMStruct container, including the temporary structures discussed in section 4.4, is shown in Figure 4-2. Variable names reflect those used in the original shared memory structure.

```
struct wsm_identity             //this identifies a single entity
{
   unsigned short player_id;
   player_state entity_state; //Empty, New, Playing, Removed
   char description[48];      //ex: T-72 Tank
   float x, y, z;             //entity position (local coord)
   float h, p, r;             //entity orientation
   entity_ally      team;          //UNK, Neutral, For, Against
   entity_domain    entity_medium; //Land, Space, Air, Munition
};

struct wsm_struct
{
  int Num_Net_Players_Hook;
  int Num_Stealth_Players_Hook;
  wsm_identity         wsm[MAX_NUMBER_OF_DIS_PLAYERS];
  wsm_identity     stealth[MAX_NUMBER_OF_STEALTH_VIEWERS];
  comment_identity comment[MAX_NUMBER_OF_COMMENT_STRINGS];
};
```

**Figure 4-2. PerformerWSMStruct CODB Container.**

## 4.3 Replacing ObjectSim Functionality

Replacing ObjectSim functionality in the SBB affected three areas of the design's implementation: changing the simulation viewpoint, rendering entities and annotations, and interacting with the DIS network. Each of these areas of functionality needed to be reproduced in the new SBB without using the ObjectSim architecture. Due to positional resolution limitations, the Pod's viewpoint is centered at the origin of the local coordinate system. The algorithm used to manipulate the scene this way was handled by an ObjectSim class. This algorithm must be incorporated into the SBB's Renderer. A structured Performer tree that includes entity subtrees had to be created to mimic entity management functions performed by ObjectSim. Finally, the DIS management routines that were performed by ObjectSim classes must be converted to the CODB architecture. These areas of functionality had to be implemented to preserve the functionality and performance associated with the previous SBB.

The user's viewpoint is centered at the origin of the local coordinate system because of Performer's positional resolution limitations. This means that the simulation must be Pod-centered (expected for an battlespace observatory), with the Pod positioned at the origin of the local coordinate system. The reason for this is because Performer has a limited resolution for viewpoint position. As the viewpoint moves further away from the origin, the viewpoint becomes unstable. Performer loses the precise resolution needed to maintain a steady viewpoint and begins to shake uncontrollably. This behavior is noticeable in the SBB's large virtual environment where viewpoint resolution is important. ObjectSim solved this problem by using an algorithm contained within its `View` class [SNYD93]. This algorithm translates the user's viewpoint and associated geometry (i.e. the Pod) to the origin and then moves the rest of the world (e.g. DIS entities, terrain, etc.) relative to the user's perceived position and orientation. In order to use the algorithm, the Performer tree must be structured in a manner that allows every

node to be moved relative to the Pod. This algorithm was moved to the SBB_Renderer's Make_Final_View method.

The Performer tree implemented in the SBB_Renderer has the same structure as the ObjectSim Performer tree structure with the addition of annotations. Figure 4-3 shows the Performer tree's structure for new SBB. The tree has four subtrees: the Pod, DIS entities, annotations, and the terrain. The pfDCS nodes above each of the subtrees allows the terrain, the annotations, and the DIS entities to be manipulated independently by the Make_Final_View method. The second level of pfDCS nodes in the annotations and the DIS Players' subtrees allow manipulation of the individual geometries. The pfSwitch node was added to the annotations subtree to allow switching between different annotation color schemes.

In the previous SBB, DIS management occurred through continual interaction between the SBB_Net_Mgr and the SBB's Object Manager. The DIS management routines performed by these two classes were converted into their CODB counterparts and divided between the SBB's World State Manager and the new DIS Manager. Interaction between the two classes will be reduced through the use of CODB containers. This conversion and the resulting CODB containers are discussed in section 4.4.

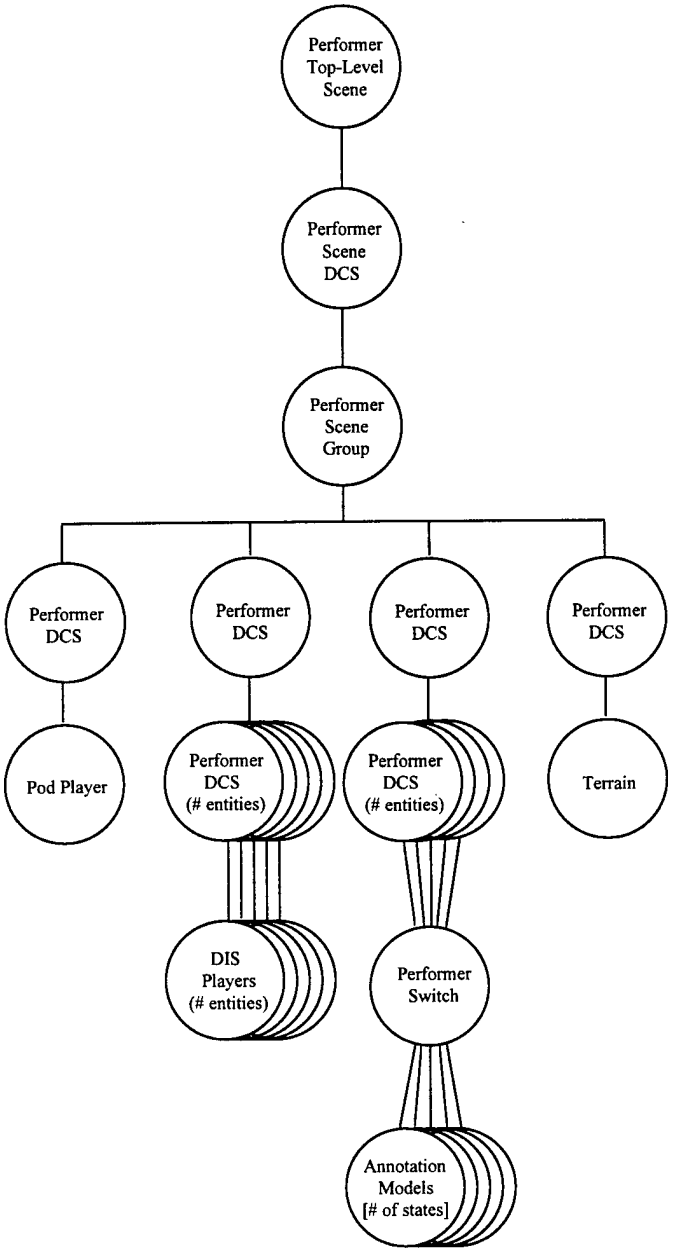



**Figure 4-3. SBB's Performer Tree With Annotation Switch Node.**

## 4.4 Improving Graphics Performance

After the Synthetic BattleBridge had been converted over to the CODB architecture and ObjectSim functionality had been replaced, it was time to start the third phase of the new SBB's development, improving graphics performance. The conversion consisted of three phases: converting IRIS GL window management and event-handling routines to X-Windows, converting all IRIS GL graphics routines to OpenGL, and finally, a comparison of the results. The conversions are discussed in Appendix A. The results appear in the next chapter.

This task was necessary in order for the SBB's Collaborative Workspace to reap the benefits of modern graphics hardware. The previous SBB required a multiprocessor platform in order to achieve acceptable frame rates. Performer divides the task of rendering a single frame into three separate processes: application, cull, and draw. Using a multiprocessor machine, each processor can be dedicated to a single Performer process. Thus, the relatively slow rendering capability of IRIS GL can be hidden from the user through parallel processing. Another way of achieving the same graphics performance is by using OpenGL. Current Silicon Graphics workstations have graphics subsystems that are highly optimized in performing OpenGL commands. This optimization allows OpenGL-based Performer applications to run under single processor machines at frame rates that were once only attainable by multiprocessor systems.

## 4.5 Distributed Simulation Interface

The distributed simulation interface used with the new SBB is the DIS Manager 3.0. The DIS Manager gives an application the ability to operate within a DIS exercise. Two separate interfaces are provided by the DIS Manager: the DIS network interface and the application interface. The DIS Manager communicates with other entities in the DIS exercise through the

transmission and reception of Protocol Data Units (PDUs). Outgoing PDUs are genrated by the DIS Manager in response to state changes in the application or in order to maintain the application as an active participant in the DIS exercise (i.e. the "heartbeat" function). The DIS Manager acts as a filtering device by weeding out PDUs that do not concern the application. PDUs that are of interest to the application are aggregated into CODB containers that are provided to the application for read access. In a similar manner, the application fills out CODB containers for use by the DIS manager when application state changes occur. For scalability performance concerns, namely interacting in an environment with a large number of DIS entities, our design used these CODB containers to minimize direct interaction between the SBB and the DIS Manager. Three CODB containers utilized by the new SBB to communicate with the DIS Manager: the `WSMEntityStruct`, the `WSMMgtStruct`, and the `OwnMgtStruct`. These containers are also known by their respective structures: the `entity_appearance_container`, the `stealth_state_container`, and the `own_stealth_state_container`. The `WSMEntityStruct` container can be seen in Figure 4-4 and the `WSMMgtStruct` and `OwnMgtStruct` containers can be seen in Figure 4-5.

### *4.5.1 WSMEntityStruct*

The DIS Manager uses the `WSMEntityStruct` CODB container to provide the SBB with accurate state information concerning DIS entities. Entity state information is used to update an entity's positional statistics, radar display icon, and graphical model in the virtual environment. Due to the graphical nature of maintaining information about non-stealth DIS entities, tight-coupling between the SBB's World State Manager class and the `SBB_Renderer`

class is required. Receiving entity state information is the most often used portion of the DIS interface and is a four step process:

1. Read `WSMEntityStruct` container from CODB,
2. Convert entity's DIS coordinates into Performer flat-earth coordinates using `RoundEarthUtils`,
3. Store entity's coordinates in `Performer-WSMStruct` container in CODB, and
4. Update entity's position and orientation in Performer geometry.

Note that if any entities are added or removed from the `WSMEntityStruct` container their corresponding entries in the `PerformerWSMStruct` and the DIS entities' Performer tree are added or deleted as well. In this manner, the SBB is able to use the `PerformerWSMStruct` as its own local copy of the `WSMEntityStruct` so that the DIS Manager is able to write to the `WSMEntity-Struct` without blocking. This is important because the `PerformerWSMStruct` also stores stealth entity state information. The four step process is divided between the SBB's World State Manager class and the `SBB_Renderer`. The SBB's World State Manager class uses its `Update` method to read the `WSMEntityStruct` from the

**Background Structure**

```
struct entity_appearance_record {
  // Entity Location in DIS coordinates
  double        x;          // meters
  double        y;          // meters
  double        z;          // meters

  // Entity Orientation in DIS coordinate system
  float         psi;        // radians
  float         phi;        // radians
  float         theta;      // radians

  // Entity Linear Velocity  Vector in DIS coordinates
  float_vector linear_velocity;    // m/sec

  // DIS ID
  unsigned int      site_id;
  unsigned int      application_id;
  unsigned int      entity_id;

  // Local Use Only
  unsigned short    model1;
  unsigned short    model2;

  // Entity Description
  char              description[40];

  // Entity Type Identifier
  entity_types      entity_type;

  // Force Identifier
  entity_alliance   team;

  // Enumerated type to determine type of entity
  container_state   entity_state;

  //Change indicator - used to tell whether change
  //has occurred in record from one cycle to the next.
  unsigned int      change_indicator
} entity_appearance_record;
```

**WSMEntityStruct**

```
typedef struct entity_appearance_container {
  entity_appearance_record
    DIS_entity[MAX_NUMBER_OF_ENTITIES];
  unsigned int num_of_active_entities;
} entity_appearance_container;
```

**Figure 4-4. Entity Appearance Container. (WSMEntityStruct)**

CODB, call the `RoundEarthUtils` to convert each entity's DIS coordinates into Performer flat-earth coordinates, and then write this info out to the `PerformerWSMStruct`. The `SBB_Renderer` class method `Update_Players` uses the `PerformerWSMStruct` container to update the position and orientation of the Performer geometry that represents each DIS entity.

### *4.5.2 WSMMgtStruct and OwnMgtStruct*

The `WSMMgtStruct` and `OwnMgtStruct` CODB containers receive stealth information and broadcasting one's own stealth information respectively. See Figure 4-5 to see each structure. The creation of these containers and their respective PDUs were implemented in three phases: stealth entity state, comments, and annotations.

For collaboration to occur between stealth entities, it was necessary to create a Stealth Entity State PDU so that basic stealth player information (name, ID, position, and orientation) could be maintained in a manner similar to non-stealth DIS entities. This information would be used to plot the position of stealth players on the radar screen and display their current viewpoint through the workspace's shared viewpoint HUD class. Broadcasting stealth entity state information consists of converting the Pod's local coordinates into DIS coordinates, filling the `own_stealth_record` structure with this information (including the stealth player's name) and then calling the DIS Manager's `broadcast_stealth_state` method. This process occurs entirely within the `Update` method of the SBB's World State Manager class and is performed every frame of the simulation. The DIS Manager uses dead-reckoning to determine when to broadcast a Stealth Entity State PDU to the network. The reception of stealth entity state information is identical to the non-stealth entity state process described previously.

The next component in the development of the WSMMgtStruct and OwnMgtStruct containers involved the transmission of text messages between stealth players. Sending and receiving text strings forms the heart of the Synthetic BattleBridge's Collaborative Workspace and is used in the Workspace_Message_Type, Workspace_Chat_Type, and Workspace_Annotation_Type classes. Text string transfers for email, bulletins, and chat are accomplished through the use of a Comment PDU. The Workspace_Annotation _Type grew too cumbersome to handle with the Comment PDU and eventually required the creation of the new Annotation Update PDU discussed later. Outgoing comments are sent by filling in the own_comment_record structure and calling the DIS Manager's broadcast_comment method. Filling the own_comment_record requires identification of the messages to be sent: private email, public bulletin, or chat message. This is accomplished through the comment_audience variable that accepts the following values: single_entity, all_entities, or curious_entities. A single_entity specifier requires a stealth entity ID to be given, sent_to_id. The actual message, limited to 256 characters, is copied to comment_string. The 256 character restriction is based upon the constraints of the Comment PDU type and the limits of the text display used within the Pod. Incoming messages are processed in a four step procedure:

1. Read WSMMgtStruct container from CODB,
2. Check for new comments from each stealth entity,
3. Temporarily store each stealth entity's comment in PerformerWSMStruct, and
4. Update simulation management structure (siman_struct) located within PodStruct as necessary.

In steps 1 and 2, the SBB's World State Manager opens the CODB, reads the contents of the WSMMgtStruct, and compares it with the contents of the PerformerWSMStruct. If the comment structures do not match, the incoming message replaces the old comment in the PerformerWSMStruct. Step 3 allows message updates to quickly occur within the SBB's World State Manager Update method. This minimizes the time that the WSMMgtStruct container is kept open. In step 4, each message's type, its contents, and sender information are determined within the Pod_Player's Update_Messages method. In this method, each stealth player's current message record is examined within the PerformerWSMStruct container to see if that message is old or new. Old messages are ignored and new messages are sorted by type (e.g. Email, Bulletin, or Chat) and immediately added to their respective structures with the exception of chat messages. Chat messages are dropped if the player does not have the workspace chat panel turned on.

The last component of the WSMMgtStruct and OwnMgtStruct containers concerns the creation, deletion, and manipulation of annotations. Originally, the annotations were a specialized class of comments, but the number of additional actions, state variables, coordinates, and messages grew beyond reasonable bounds, resulting in the crippling of the original comment_record. Thus, a new structure was formed, the annotation_record. This record encompassed all the annotation features intended for this research project and allowed future expansion of features and media types as well. The most important decision in the implementation of the annotation_record was to divide an annotation's state from its contents. This permits the annotation's state to be treated similar to a DIS entity's state and allows the annotation's contents to contain multiple messages and media types (i.e. sound, pictures, documents, movies). For this thesis, we limited annotations to the ability to hold multiple text messages. Annotations would perform unique actions: create and delete

annotations, update annotations, select and deselect annotations, attach and detach annotations from a DIS entity, move annotations, appending messages onto an annotation, and clearing messages from annotations. To request an annotation to perform an action, the `action` variable must contain the single requested action selected from an enumerated list of `annotation_actions`. Annotations are stored locally in the order in which they arrive to the stealth player. Thus, there is no reason that each stealth player would maintain the same listing of annotations in the same order. The problem of telling the DIS Manager which annotation needs to be acted upon was solved through the `annotation_index` variable. This index corresponds to the annotation's position in the local array of annotations and does not represent the annotation's unique ID. That information is handled internally by the DIS Manager. If an outgoing annotation needs to be attached to a DIS entity, the `attached_to_id` variable must be filled with that DIS entity's ID If a selection action is required, the `selected_state` variable must be set to TRUE. The opposite is true for deselecting an annotation. An annotation's `location` vector must be set when creating or moving an annotation. Note that annotations do not have an orientation associated with them for aesthetic reasons. Finally, a `comment_string` must be included when creating a new annotation or appending a message onto an existing annotation. When all the relevant fields have been filed out for the annotation action desired, the DIS Manager's `broadcast_annotation_update` method is called.

**Background Structures**

```
// Stealth Viewer "appearance" Record
struct stealth_record {
  // Entity Location in DIS coordinates
  double_vector    location;    //meters

  // Entity Orientation in DIS coordinate system
  tait_bryan_angles    orientation;      // radians

  // DIS ID
  unsigned int      site_id;
  unsigned int      application_id;
  unsigned int      entity_id;

  container_state    entity_state;    // Active / Inactive

  // Stealth Description
  char              description[40];    // ASCII characters
} stealth_record;

// Comment String (from a Comment PDU  record
struct comment_record {
  unsigned short    sender_id;
  audience_types    comment_audience;
  char              comment_string[256];
} comment_record;

// Individual Annotation Comment String record
struct annotation_comment_record {
   unsigned short    sender_id;
   char              comment_string[256];
} annotation_comment_record;

// Annotation record
struct annotation_record {
   container_state          annotation_state;
   unsigned int             change_indicator;
   unsigned short           sender_id;
   unsigned short           attached_to_id;
   boolean                  selected_state;

    // Annotation location is DIS coordinates
   double_vector            location;

   annotation_positions       positioning;
   unsigned short           num_of_active_comments;
   annotation_comment_record
       message[MAX_COMMENTS_PER_ANNOTATION];
} annotation_record;
```

**WSMMgtStruct**

```
// Stealth State Container
 struct stealth_state_container{
   unsigned short       num_of_active_stealth_viewers;
   stealth_record
     stealth_viewer[MAX_NUM_OF_STEALTH_VIEWERS];
   unsigned short       num_of_active_comments;
   comment_record
     comment[MAX_NUMBER_OF_COMMENT_STRINGS];
   unsigned short       num_of_active_annotations;
   annotation_record
     annotation[MAX_NUMBER_OF_ANNOTATIONS];
 } stealth_state_container;
```

**Background Structures**

```
// Stealth Viewer "appearance" Record
struct stealth_record {
   // Entity Location in DIS coordinates
   double_vector            location;        // meters

   // Entity Orientation in DIS coord. system (body coord)
   //  system
   tait_bryan_angles        orientation;     // radians

   // Entity Linear Velocity in World Coordinates
   float_vector       velocity;        // meters per second

   // Entity Linear Acceleration in World Coordinates
   float_vector       acceleration;    // meters per second

   // Entity Angular Velocities in Body Coordinates
   // Around the appropriate axis
   float_vector       around_axis;     // radians per second

   // Entity Description
   char               description[16]; // ASCII characters
} own_stealth_record;

// Comment String record
struct own_comment_record{
   unsigned short    send_to_id;
   audience_types    comment_audience;
   char              comment_string[256];
} own_comment_record;

// Possible Annotation Actions
enum annotation_actions {other_action,
   create_annotation, create_attached_annotation,
   delete_annotation, update_annotation,
   move_annotation, select_annotation,
   deselect_annotation, refresh_annotation,
   attach_to_entity,  detach_from_entity
};

// Own Annotation record
struct own_annotation_update_container{
   unsigned short           annotation_index;
   unsigned short           attached_to_id;
   boolean                  selected_state;
   annotation_actions       action;
   double_vector            location;

   char                     comment_string[256];
} own_annotation_update_record;
```

**OwnMgtStruct**

```
// Own Stealth State Container

 struct own_stealth_state_container{
   own_stealth_record          own_stealth;

   own_comment_record          own_comment;
   own_annotation_update_record
 own_annotation_update;
 } own_stealth_state_container;
```

**Figure 4-5. Send and Receive Stealth State Containers.**

**(WSMMgtStruct and OwnMgtStruct)**

Due to the graphical nature of annotations, incoming annotations are handled by the `SBB_Renderer` class in a similar manner as incoming entity state information. If the annotation does not exist within the simulation management portion of the `PodStruct` container, a graphical representation of the annotation is added to the Performer tree corresponding to the sender's ID. The annotation's information is added to the `PodStruct` container and is now considered an "active" entity. Active annotations that receive updates perform their respective actions as instructed. More information concerning annotations is found in the next section.

## 4.6 Enhancing the User Interface

In this section, the four methods of enhancing the user interface will be discussed: using hidden panels, the XForms user interface, the Selection Manager, and the Fly navigation interface. These interface improvements will be needed for Collaborative Workspace user to operate, communicate, and travel efficiently within the virtual world..

### *4.6.1 Hidden Panels*

The user interface for the Collaborative Workspace must be easy to use, meaning a consistent Pod interface between workspace components, and it should not clutter the existing Pod control panels. By design, the Pod supports the use of hidden panels; although a complete reconfiguration of a panel's controls has never been implemented. This feature is used to create and manage three new control panels (Workspace Messages, Workspace Chat, and Workspace Annotations) without increasing the total number of visible panels.

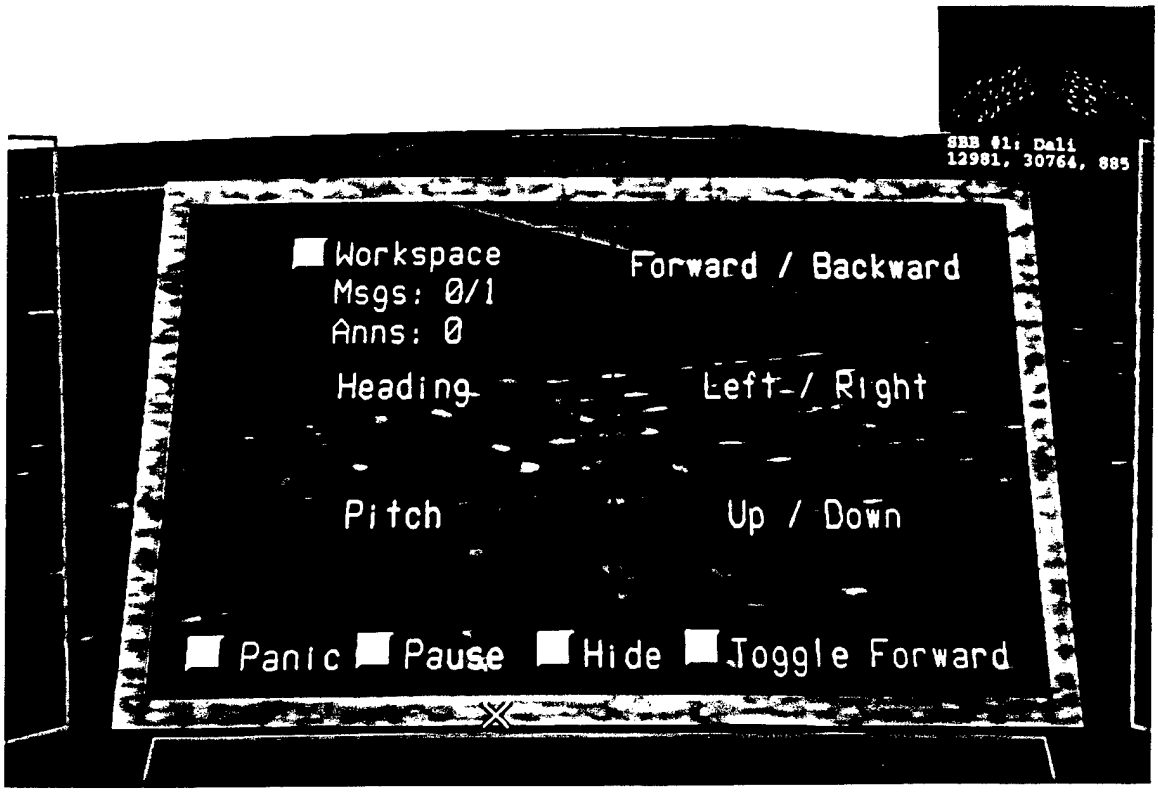



**Figure 4-6. Pod Center Panel With Workspace Subpanel**

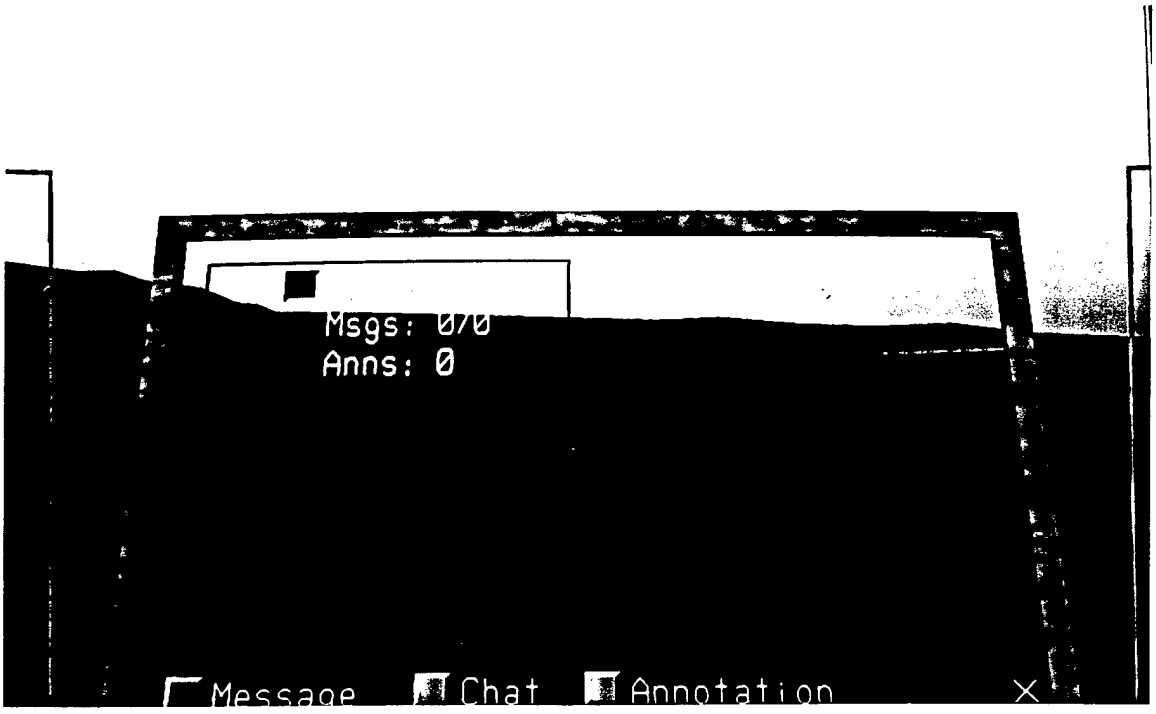



**Figure 4-7. Workspace Controls.**

The creation of hidden panels required that two intermediate subpanel classes be established to allow access between the main panel and the hidden panels, and to permit switching between the hidden panels. The first intermediate class is the `Workspace_Type` and it is the subpanel seen in the upper left-hand corner of the Pod's center panel( see Figure 4-6). The Workspace subpanel maintains statistics concerning the number of messages and annotations received and it also contains an access button to bring up the second intermediate subpanel class, the `Workspace_Controls_Type`. While the access button remains depressed, the Workspace Controls are available. The Workspace Controls subpanel is nothing more than a switching mechanism between the hidden workspace controls. The subpanel consists of a task bar at the bottom of the console with buttons that select the desired workspace panel (see Figure 4-7). The first time the Workspace access button is depressed, it will automatically bring up the Workspace Control's default panel, the Workspace Message panel. Subsequent visits to the Workspace Controls will bring up the last Workspace panel accessed.

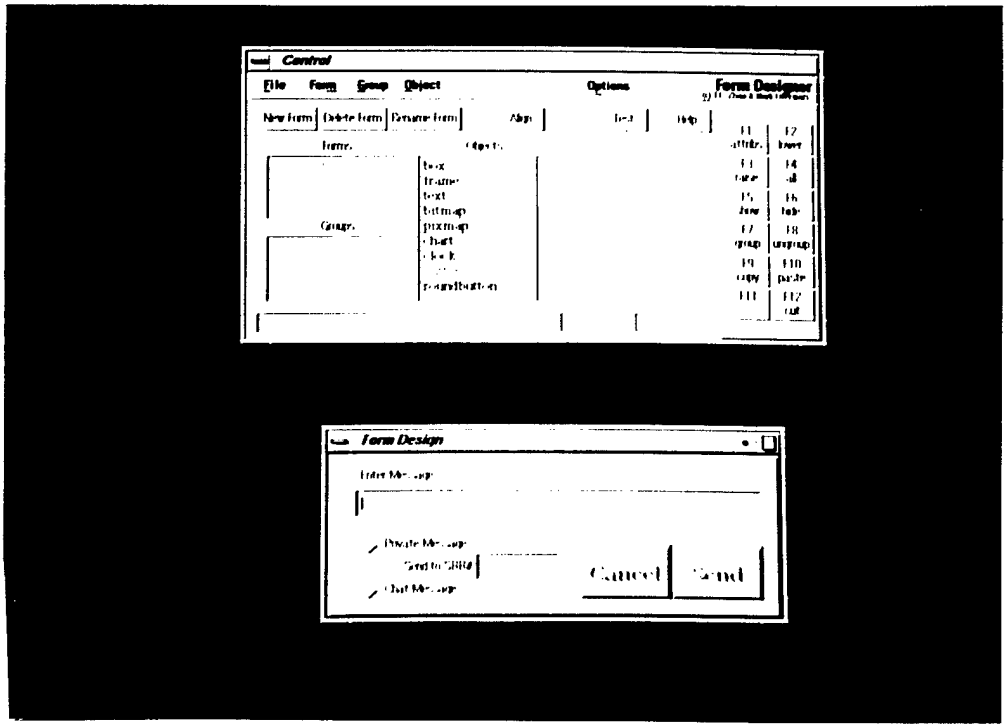
### *4.6.2 XForms User Interfaces*

Pod panels are not the only interface used by the Collaborative Workspace. The original the Pod had lacked two important features: first, it did not support text input and second, it could not interact with objects in the virtual world (beyond the Pod). These shortcoming were fixed with the creation of the multi-layered interface discussed in the design chapter. The XForms libraries were used to provide text input to the Workspace while the Selection and Movement Managers handle annotation interaction with the virtual world.

The XForms Library is an X-Windows implementation of the IRIS GL-based Forms Library that was used by Soltz in creating the Sentinel user interface for the SBB [SOLT94]. The XForms Library does not have the Forms Library limitation of producing slower frame rates

as experienced by Soltz. This is because the XForms-generated interface runs independent of the rendering process used by IRIS GL or OpenGL. Interaction with an XForms-generated interface is likewise independent of any interaction with the Performer environment. Thus, the Pod and the Selection Manager can be used at the same time as an XForms interface.

The XForms Library was chosen for its ability to quickly produce form-based dialogue boxes for user input. The Collaborative Workspace uses three interfaces that were designed using the Forms Designer: user name input, text message input, and annotation input. Once an XForms interface is complete, it can be interactively tested using the Forms Designer test console. Whenever the user changes the state of a particular forms object, the application is notified and can take action accordingly. Once the form has been tested, Forms Designer generates the corresponding C code to create the form within the application. Figures 4-8 and 4-9 show the text message interface and the C code used to generate the form.




**. Figure 4-8. Forms Designer with Text Message Interface**

Using a form is a four step process. The first step is including the Forms Designer-generated C code into the application. The second is initializing the XForms Library with the `fl_init` function. The third is creating a function to invoke the form, monitor state changes, remove the form when finished, perform desired actions as a result of the form input, and lastly terminate the function. The final step is to create a trigger for this function. This trigger can be a specific key-press (e.g. 'a' to invoke the annotation input form) that starts the above function in its own process. This allows the form to operate separately from the running application.

```
Typedef struct {
        FL_FORM *form;
        FL_OBJECT *box;
        FL_OBJECT *Message;
        FL_OBJECT *PrivateMessage;
        FL_OBJECT *SendTo;
        FL_OBJECT *ChatMessage;
        FL_OBJECT *Send;
        FL_OBJECT *Cancel;
        void *vdata;
        long ldata;
} FD_message_input;
```

```
FD_message_input *create_form_message_input(void)
{
  FL_OBJECT *obj;
  FD_message_input *fdui = (FD_message_input *)
     fl_calloc(1, sizeof(*fdui));

  fdui->form = fl_bgn_form(FL_NO_BOX, 490, 180);
  fdui->box = obj = fl_add_box(FL_UP_BOX,0,0,490,180,"");
  obj = fl_add_text(FL_NORMAL_TEXT,30,10,210,30,"Enter Message:");
     fl_set_object_lalign(obj,FL_ALIGN_LEFT|FL_ALIGN_INSIDE);
  fdui->Message = obj =   fl_add_input(FL_NORMAL_INPUT,30,40,440,30,"");
     fl_set_object_gravity(obj, FL_NorthWest, FL_SouthWest);
  fdui->PrivateMessage = obj =
     fl_add_checkbutton(FL_PUSH_BUTTON,30,80,200,40,"Private Message");
  fdui->SendTo = obj =
     fl_add_input(FL_INT_INPUT,160,110,90,30,"Send to SBB#");
  fdui->ChatMessage = obj =
     fl_add_checkbutton(FL_PUSH_BUTTON,30,120,200,40,"Chat Message");
  fdui->Send = obj =
     fl_add_button(FL_NORMAL_BUTTON,370,100,100,60,"Send");
        fl_set_button_shortcut(obj,"^M",1);
        fl_set_object_color(obj,FL_MCOL,FL_BLACK);
```

```
        fl_set_object_lcol(obj,FL_GREEN);
        fl_set_object_lsize(obj,FL_LARGE_SIZE);
        fl_set_object_lstyle(obj,13+FL_SHADOW_STYLE);
  fdui->Cancel = obj =
    fl_add_button(FL_NORMAL_BUTTON,270,100,100,60,"Cancel");
        fl_set_button_shortcut(obj,"^[",1);
        fl_set_object_color(obj,FL_MCOL,FL_BLACK);
        fl_set_object_lcol(obj,FL_RED);
        fl_set_object_lsize(obj,FL_LARGE_SIZE);
        fl_set_object_lstyle(obj,13+FL_SHADOW_STYLE);
  fl_end_form();

  return fdui;
}
```

**Figure 4-9. Text Message Interface Code**

### *4.6.3 The Selection Manager*

Interaction within the virtual world is needed by the Collaborative Workspace to manipulate annotations. The Pod supports interaction through the use of buttons set on a flat rectangular surface (i.e. the panel). The Selection Manager expands the user's interaction capability with a class that uses Performer's geometry-picking functions. The Selection Manager class provides a way of determining if a specially-tagged object has been selected by the user. By using Performer geometry, the class is able to accurately pick irregularly-shaped objects instead of picking against a flat surface area as in the Pod. The class requires the developer to name all potentially selectable Performer nodes. The `Selection_Mgr's` constructor is called and sent the root node of the Performer tree and the channel that renders the Performer tree. The Performer geometry is now ready for selection. The Selection Manager works by retrieving the path through the tree of the selected node. The Selection Manager's `poll` method will return an integer that identifies the selected node. For example, by naming an annotation node "ANNOTATION_005," the `poll` method will return a 5 to the application when that annotation

node is selected. In the SBB's Collaborative Workspace, selection occurs by placing the mouse cursor over a selectable object and pressing the middle mouse button.

### *4.6.4 The Fly Navigation Interface*

The final enhancement that was made to the Pod user interface involves navigation within the virtual world. Previous Pod implementations were limited to straight-line movement and single-axis rotations. One of the features imported from the CODB demonstration project was the ability to detach oneself from the Pod and travel quickly to any location using SGI's Fly interface. The Fly navigation interface uses only the mouse to control the velocity, acceleration, and direction of travel. Using the design described in chapter 3, the user can now detach himself from the pod and travel to any location in the virtual environment. In a technique similar to that used by the Workspace Shared Viewpoint (discussed in section 4.6.5), the user is able to transport the Pod to the exact coordinates and orientation of the detached player. The player is also able to jump back to the original location of the Pod.

## 4.7 Support for CSCW Activities

The heart of this thesis is the support of CSCW activities within a virtual environment. This section describes how the PDU types and CODB containers discussed in section 4.4 are used. The foundation has been laid for continued development using the CODB architecture, the ObjectSim functionality needed to support special workspace features like annotations and shared viewpoints is in place, the graphics performance has improved enough that new features will not create unacceptable frame rates, and a new multi-layered user interface has expanded the range of user input. It is time to open the channels of communication between SBBs. To give

some organization to the wide variety of communication tools implemented in the SBB's Collaborative Workspace, this section has been divided as follows: recognizing other stealth players, Workspace Messages, Workspace Chat, Workspace Annotations, Workspace Shared Viewpoints, and animation controls.

### *4.7.1 Recognizing Other Stealth Players*

One of the fundamental aspects of CSCW is that each participant has a unique identity [RODD91b]. Therefore, for the SBB's Collaborative Workspace to be considered a CSCW application, each stealth player must have a unique identity. Every DIS player, regular or stealth, is assigned a composite ID code by the DIS Manager consisting of site, application, and entity IDs. Entities are also labeled with a description of their particular vehicle. Stealth players need names instead of vehicle descriptions to identify the user behind the controls. Thus, upon starting the new SBB, the user is asked to input a name. This name is used to identify all transmissions from the user including positional information, messages, annotations, and the like. Incoming stealth players will receive stealth entity state PDUs from each stealth player within the first 12 seconds of the simulation, corresponding to each player's "heartbeat" function. As the stealth player moves within the virtual world, the player's coordinates are broadcast to all of the other stealth players. This information will be reflected in the player's radar image and shared viewpoint. If messages are sent out, Comment PDUs are received by the other stealth players. The same process occurs with annotations and Annotation Update PDUs. The more a stealth player interacts with the environment, the more information about the player's actions is passed to the other stealth players. Thus, the player is given embodiment through his actions.

### *4.7.2 Workspace Messages*

After a stealth player's state could be reliably tracked by another stealth player, the most basic form of direct communication was implemented, the text message. While it was relatively simple to send and receive messages using an IRIX console window, the challenge was how to manage sending and receiving messages using the Pod. The `Workspace_Message_Type` class was created for this purpose.

The implementation of the multi-layered Pod interface made the sending of text messages remarkably easy (see Figure 4-10), but unfortunately the virtual keyboard designed in chapter three was not constructed. This meant that the task of sending messages also needed to be accomplished without the benefit of a keyboard. Soltz experimented with a Forms-created virtual keyboard, but it was not implemented as a Pod-based device [SOLT94]. Thus, the easiest way to support sending messages using only the Pod interface was to have messages loaded from an input file upon SBB initialization. No limit is applied to the amount of preloaded messages, so long as each is under 256 characters. These messages can be cycled through until the appropriate message is seen in the display. The player can then either broadcast the message to all stealth players as a bulletin, or pick a particular stealth player and send the message out as email. Figure 4-11 shows the Workspace Message control panel with a preloaded message ready for broadcast.

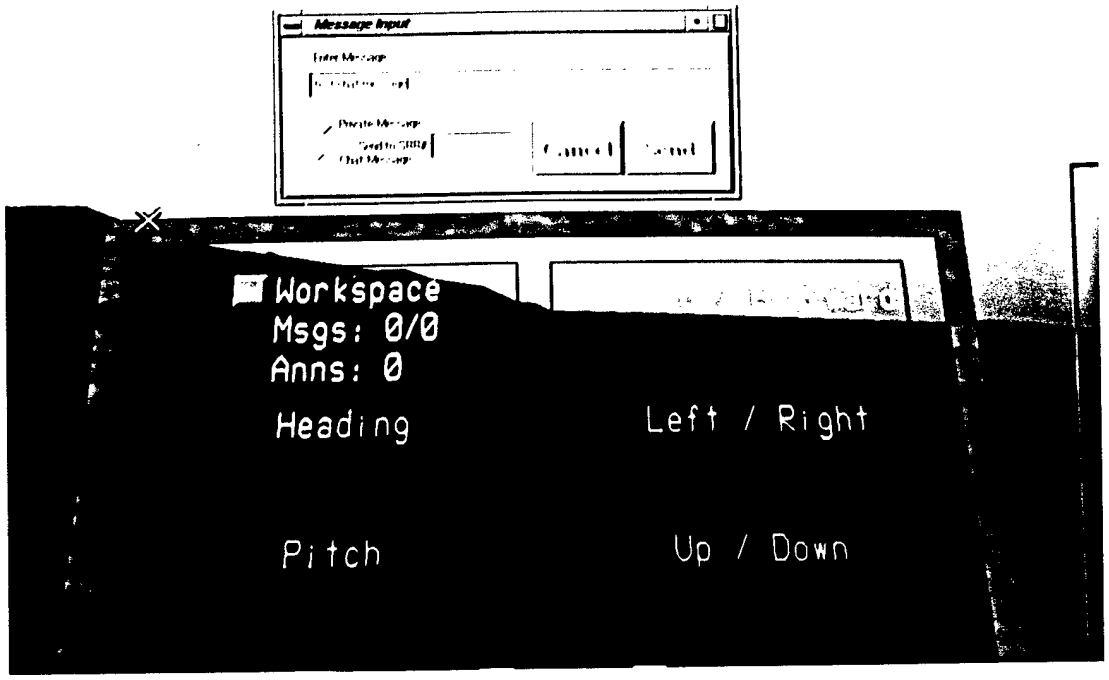



**Figure 4-10. Text Message Interface with Pod.**

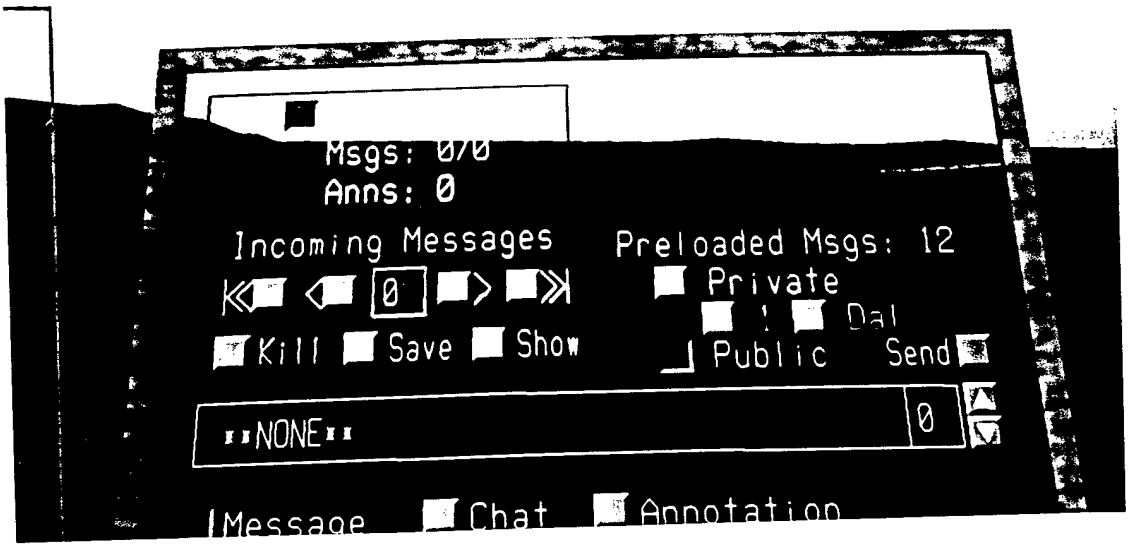



**Figure 4-11. Workspace Message Panel.**

Two issues must be resolved in implementing the receive side of the workspace messages: how would the message structure be organized and how would these messages be displayed? The nature of messages, sequential ordering, unlimited number of potential messages, and the ability to delete any message at will, presented only one logical choice: a doubly-linked list. A doubly-linked list allows the user to add recent messages to the end of the list, delete messages anywhere from within the list, and traverse the list forward or backwards as desired. Thus, the reception of new messages, the deletion of old messages, and the browsing of one's mailbox were performed by maintaining the message list and a pointer to the current message of interest. Buttons were created to allow the user to select a message, display it, save the message to a file, or delete it as necessary (see Figure 4-12).

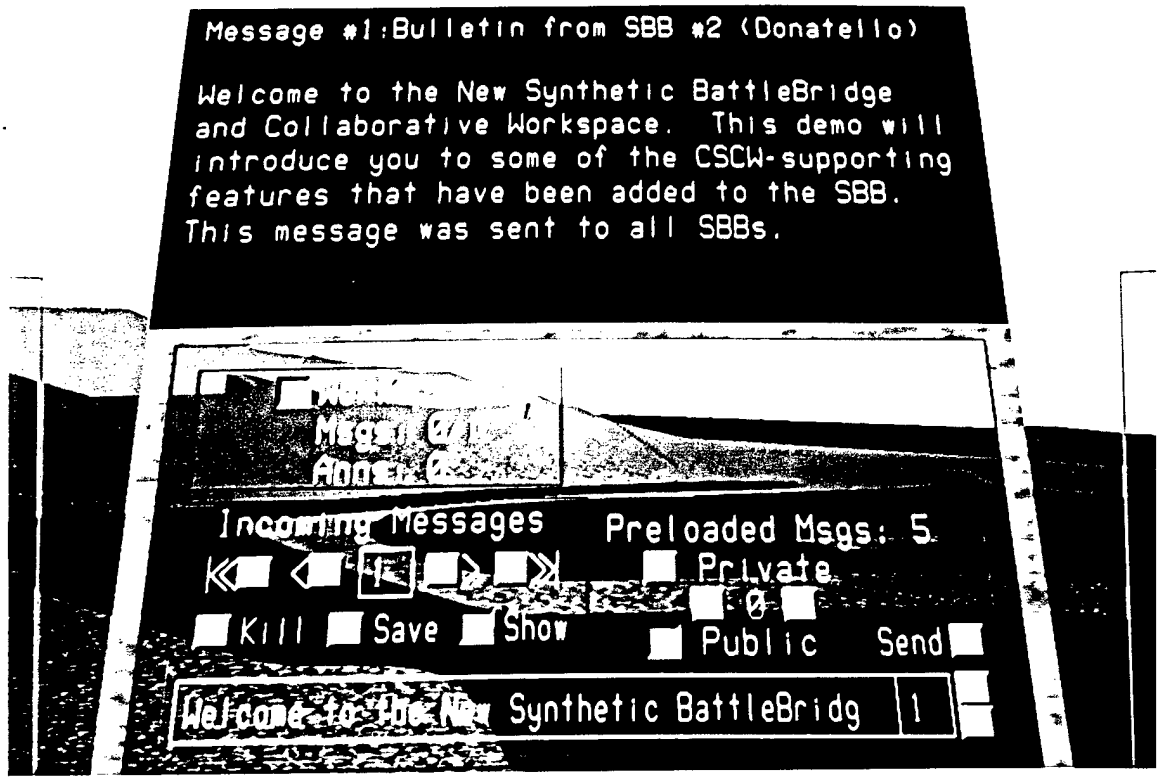



**Figure 4-12. Workspace Message Panel Displaying a Message.**

The displaying of messages was a more difficult problem than originally anticipated. The ability to display short 3D text strings was used throughout the construction of the Pod. The challenge was to find a way to display long 3D text strings in the smallest possible area. This was solved by creating a pseudo-panel that appeared between the center panel and the top panel. As a child of the center panel, the display panel inherits certain graphical properties such as the same tilt with respect to the user. The display panel is completely opaque with a black background and yellow text for maximum contrast. The type of message, the message number, and the sender's name form the message header with the message body following below. This is where the difficulties began. It is not a trivial matter to create a word-wrap function when each line of text and its position on the screen must be fully specified in three dimensions before being drawn. Unlike a standard console where not using word-wrap causes characters to either be cropped from the display or continued on the next line, the Pod has no concept of screen boundaries or lines of text. Such conventions must be maintained by the developer to meet user expectations. Therefore, a word-wrap algorithm was created to satisfy the unique requirements of displaying long text strings within the Pod. This algorithm is shown in Figure 4-13. There are no claims to its efficiency, but it does perform the task required. The result of word-wrapping a long text message is shown in Figure 4-12.

```
if (strlen(MessageContents) < 45)
  Workspace_Message_Text->Print(Message_label[PF_X],
                                Message_label[PF_Y] - 2.0f,
                                Message_label[PF_Z], MessageContents);
else
{
  TempString[0] = '\0';
  char *TS = TempString;
  char *MC = MessageContents;
  boolean done = FALSE;
  int i, // i is current character position on this line of text
      j, // j is current line number
      k; // k is current character position in incoming message
```

```
    int last_space;
    i = j = k = 1;
    while (!done)
    {
       TS[0] = '\0';
       i = 1;
       last_space = 0;

       while ((i<=45) && (k<=256) && (MC[k-1] != '\0'))
       {
          TS[i-1] = MC[k-1];
          if (MC[k-1] == ' ')
             last_space = i;
          i++; k++;
       }
       if ((last_space != 0) && (i > 45))  //word-wrap will occur
       {
          while (i != last_space+1)        //+1 to crop leading space
          {
             i--; k--;
          }
          //write one line of text to TempString
          strcpy(TempString,TS);
          strxfrm(TempString,TempString,last_space);
       }
       else                                //word wrap not needed
       {
          //write one line of text to TempString
          strcpy(TempString,TS);
          strxfrm(TempString,TempString, i);

          if ((strlen(TempString))<45)    //are we done with the message?
             done = TRUE;
       }
       if ((MC[k-1] == '\0')||(MC[k] == '\0'))
          done = TRUE;

       Workspace_Message_Text->Print(Message_label[PF_X],
                                     Message_label[PF_Y]-(2.0f*j),
                                     Message_label[PF_Z], TempString);

       j++;
    }//end while (!done)
}//end else (word-wrap needed)
```

**Figure 4-13. Workspace Message Type's Word-wrap Algorithm.**

### *4.7.3 Workspace Chat*

Using the Workspace Message control panel as a guide, the creation of Workspace Chat was a relatively simple task. The Workspace Chat control panel, seen in Figure 4-14, includes the ability to send preloaded chat messages, display the current chat session, kill the current chat log, or save the chat log to a file. These functions mirror those found in the Workspace Message control panel.

Due to the similar nature of chat messages and regular messages, a doubly-linked list was also used as the chat messages' structure. The difference between the two structures was that only the list as a whole could be saved or deleted, not a single entry. Also, only the last 10 chat messages could potentially be shown due to the size of the chat session display screen. Note that a larger screen was created, but it interfered with the top and center panels, resulting in a cluttered and confusing interface. The challenge of modifying the word wrap algorithm to display multiple messages while scrolling old messages off the screen needed to be resolved.

The word wrap algorithm itself was not touched, but instead was used in combination with two loops. An outside loop began the process with the most recent chat message and worked its way up to the $10^{th}$ most recent chat message if display space would allow it. Each string was processed by the existing word-wrap algorithm, but instead of immediately writing a line of text out to the display, it was saved within an array. After an entire chat message string was divided into screen-width lengths, the second loop was started to display the chat message. Unlike the Workspace Message display mechanism that prints out a line of text and moves down to the next line, the Workspace Chat display algorithm prints out the last line of text and works its way up as far as screen space will allow. At the first line of a chat message, the sender's name, cropped to three characters, is displayed as a prefix to the text string. Names are cropped to three letters as a tradeoff between recognizing the sender's name and presenting an organized

display. If the entire chat message can be printed, then the outer loop increments to the next older chat message and the process begins again. The Workspace Chat display can be seen in Figure 4-14.

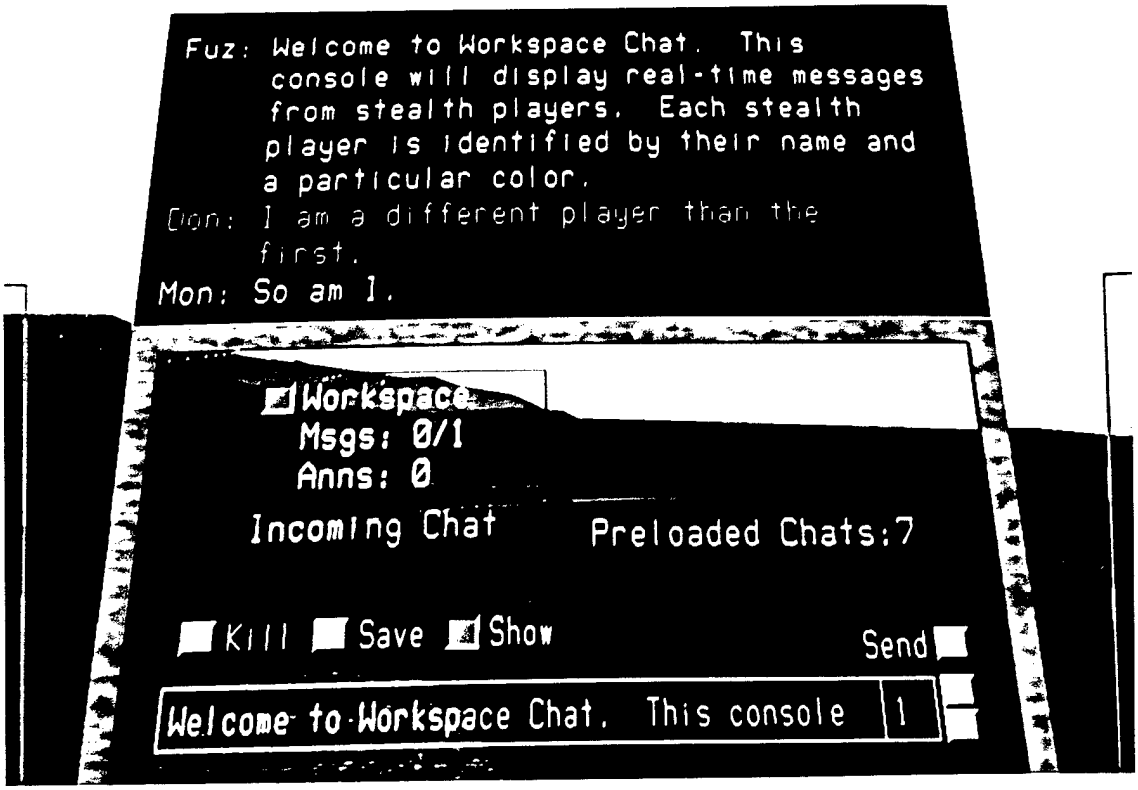



**Figure 4-14. Workspace Chat Panel Displaying a Chat Session.**

### *4.7.4 Workspace Annotations*

Workspace Annotations form a hybrid between Workspace Messages and DIS entities. For this reason, the Workspace Annotation control panel is a mixture of the Workspace Message control panel and the Attach To Entity control panel.

Annotations can be created in two ways: using the XForms annotation input window discussed in section 4.5 or through a more primitive approach using the annotation control panel. The first method is shown in Figure 4-15. The second method involves first preloading messages into the system in a similar manner as Workspace Message and Workspace Chat. The

user chooses the appropriate message and presses the Send button. The annotation is broadcast to the world at the terrain's origin. From this starting position, the annotation can be selected and moved to its correct location or attached to a DIS entity. To facilitate the proper graphical display of location-based annotations, the `SBB_Renderer` function `GroundTrack` is available to take the proposed annotation's coordinates, calculate the intersection point of a ray along the z-axis and the terrain, and return the corrected coordinates for use in creating the annotation. This process effectively clamps the annotation to the terrain. Newly created annotations are added to the Performer tree and their entries are updated as described in section 4.4.2.

Each annotation is represented in the virtual world with a pennant icon. Each pennant is color-coded to a single SBB player, so that an annotation's owner can be immediately determined visually. Pennant colors are also used to indicate newly created or recently changed annotations, selected annotations, and old (i.e. messages read) annotations. See Figures 4-16, 4-17, and 4-18 for examples of each type of annotation. Note that selected annotations are also highlighted with a bounding box. All annotations are labeled in 3D space by the new Font Manager described in section 4.3.2.

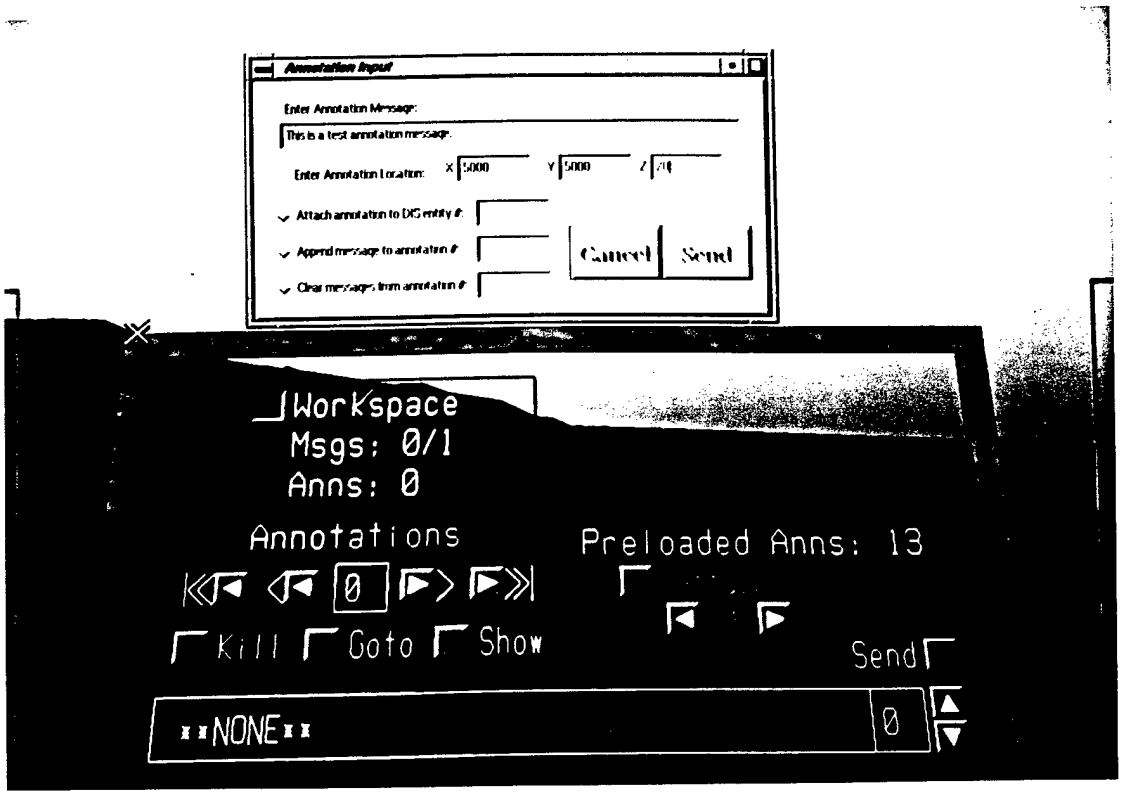



**Figure 4-15. Annotation Input Interface.**

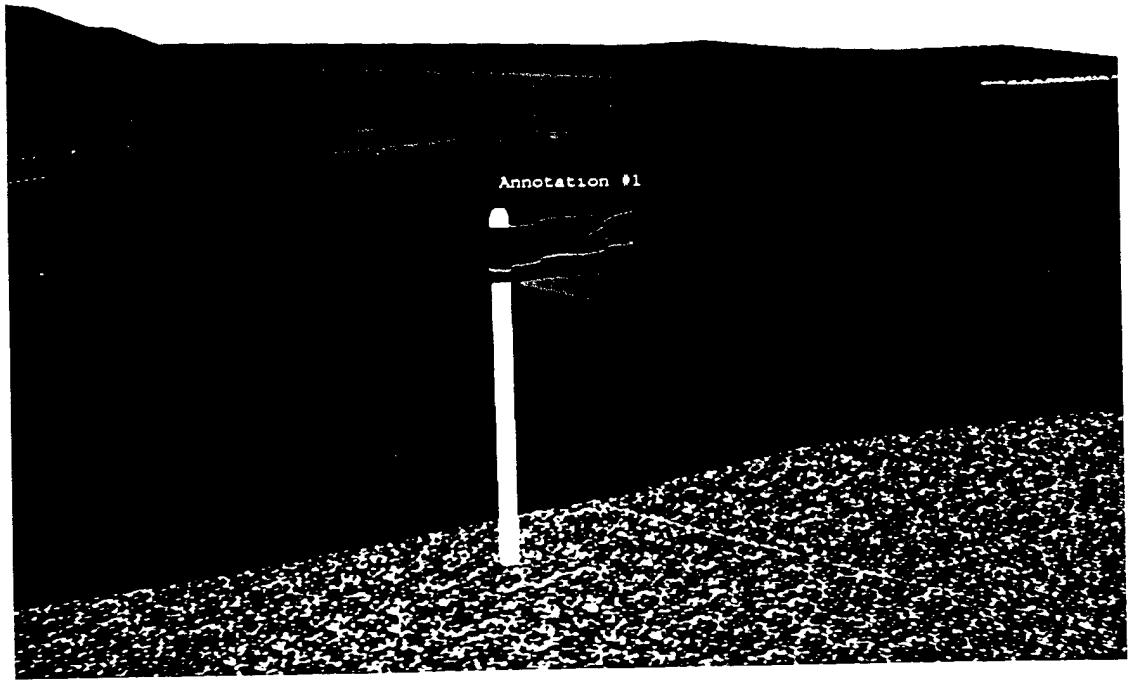



**Figure 4-16. New or Updated Annotation.**

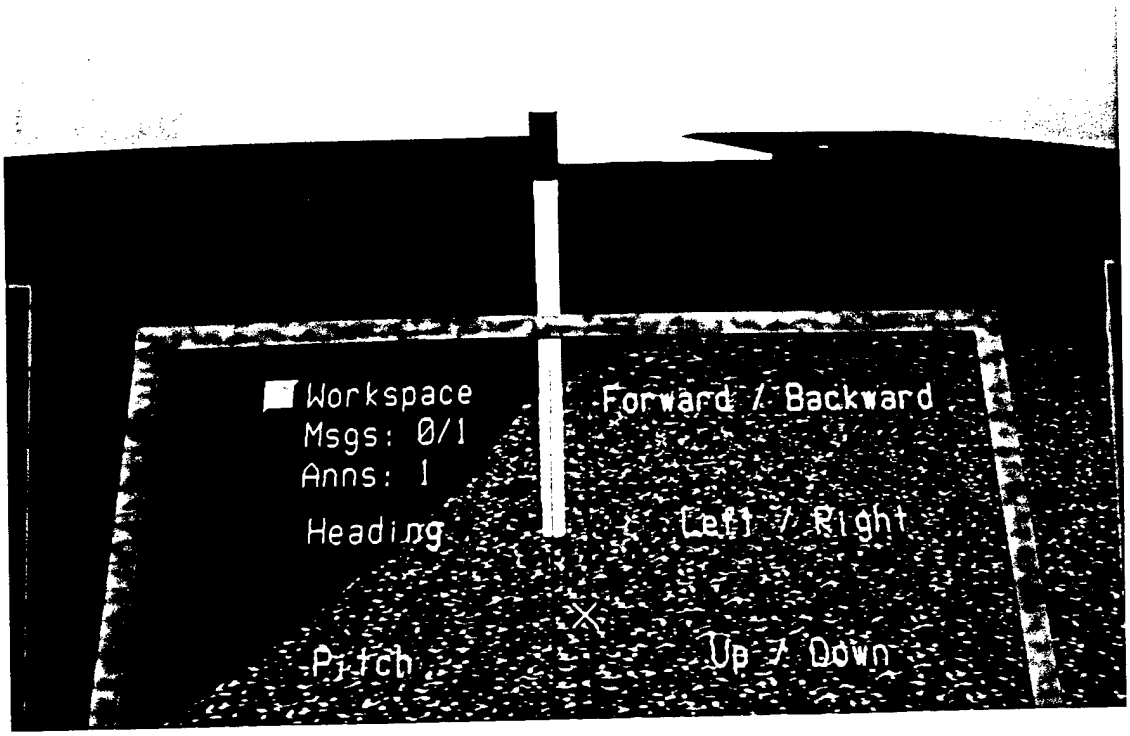



**Figure 4-17. Selected Annotation.**

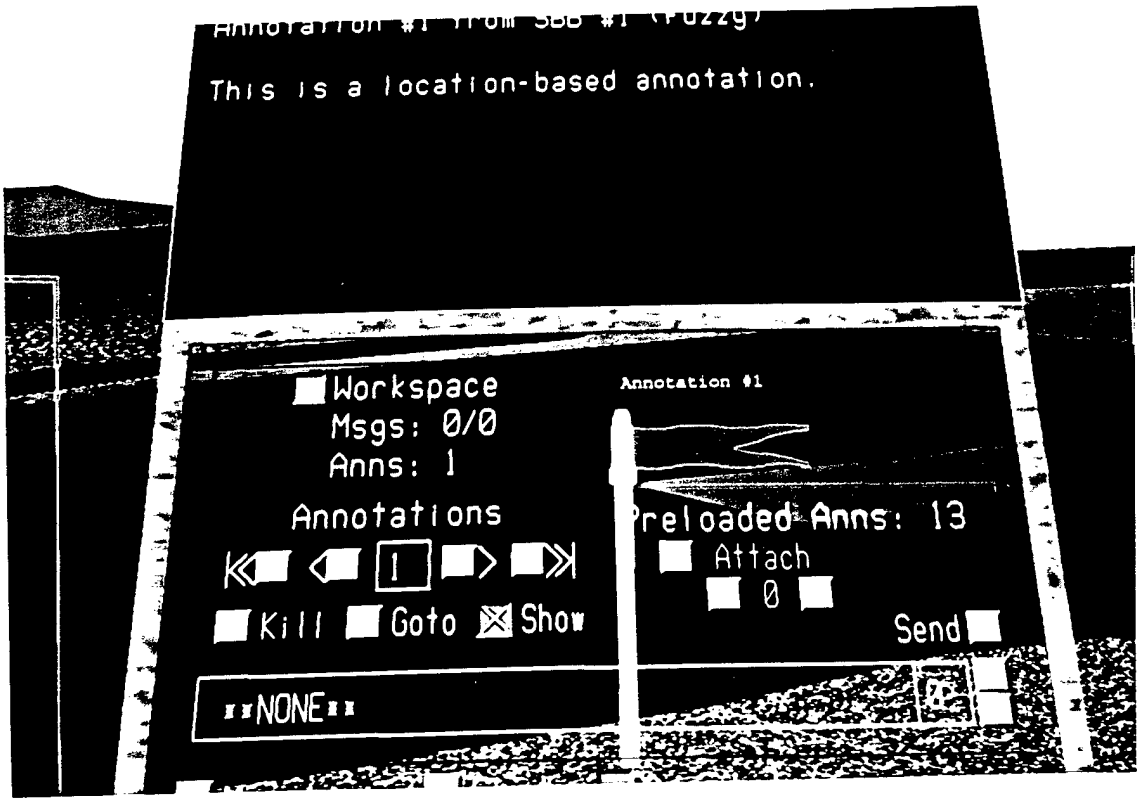



**Figure 4-18. Old Annotation.**

Viewing annotation messages involves first selecting the annotation and then displaying its contents. One means of selection has been discussed in section 4.5, but the annotation controls include the ability to select messages by cycling through all available annotations; similar to Workspace Messages. This is a local selection and will not be broadcast to other stealth players. The display of annotation messages is identical in nature to the Workspace Chat. Due to the limitations of screen space, only the most recent messages are shown on the text display screen with the sender's name, shortened to three letters, attached to the beginning of each message.

The remaining annotation management controls include deleting an annotation, clearing an annotation (i.e. removing the annotation's contents without deleting the annotation), appending a message to an annotation, moving an annotation, and attaching an annotation to a DIS entity. The process of deleting an annotation involves first selecting the desired annotation and then pressing the Delete button. The annotation's model is removed from the Performer tree and the annotation's information is cleared. Clearing an annotation requires selecting the annotation and pressing the Clear button. The annotation remains, but all of its messages are removed. Appending a message to an existing annotation can be accomplished in two ways. The first is to use the XForms annotation input interface described in section 4.5. The second involves selecting an annotation and then sending out a message. When an annotation is already selected, the Workspace Annotation controls assume the user's intention is to append the message onto the existing annotation and not to create a new annotation. Movement of annotations can only be performed by selecting an annotation and then dragging the annotation to the desired location while holding down the middle mouse button. Attaching an annotation to an DIS entity can be performed in two ways: using the XForms annotation interface or by choosing the desired annotation, increasing or decreasing the `Atch_ID` value to the match the

desired entity, and pressing the `Attach` button. Currently, there is no procedure for detaching an annotation as the annotation's location upon leaving the entity is undefined. The ability to perform such a feat is available using the Annotation Update PDU if desired in the future.

### *4.7.5 Workspace Shared Viewpoints*

Workspace shared viewpoints were implemented in the same fashion as the current Video Missile and Drop Camera HUDs. Using a style similar to the Drop Camera HUD controls, the SBB user is able to cycle through the viewpoints of stealth players based upon their current position and orientation. Note that the controls are only active when there is another stealth player in the DIS exercise. The viewpoints are labeled with the stealth viewer's name, number, and current position. To minimize screen clutter, this information is only available in full-screen mode (1280x1024)(see Figure 4-19). An additional feature described in the design

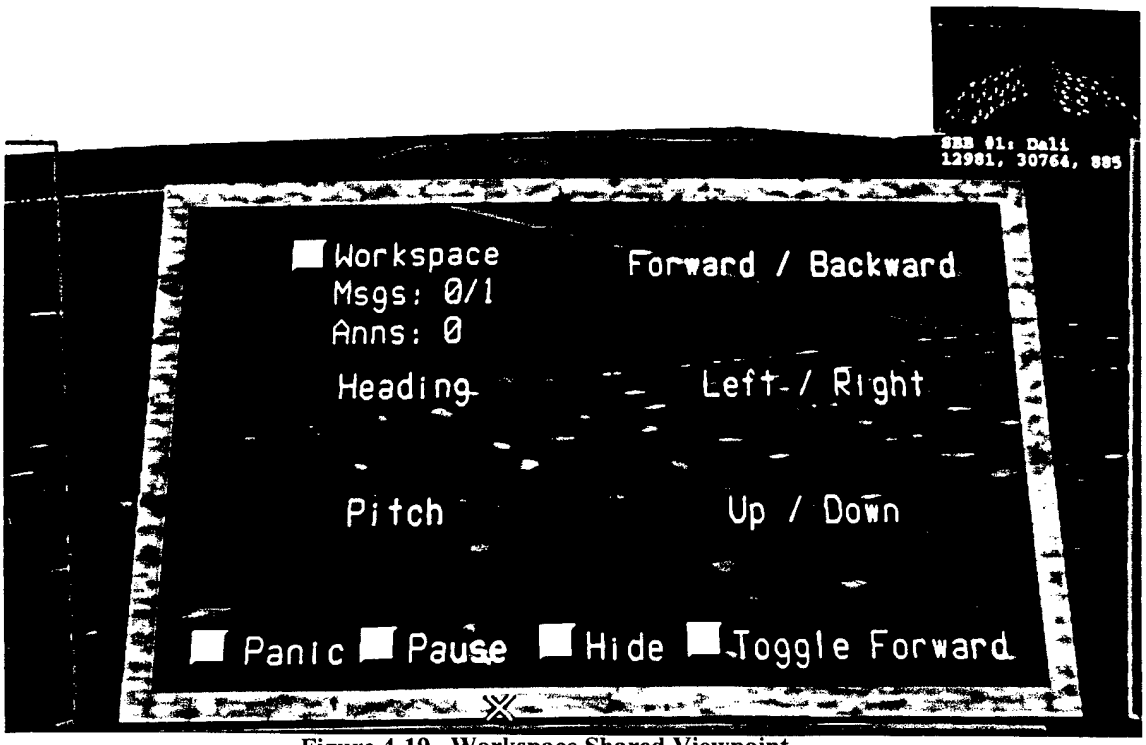



**Figure 4-19. Workspace Shared Viewpoint.**

allows the SBB user to jump to the currently watched viewpoint. After the jump, the Pod is located in the exact position and orientation of the shared viewpoint's owner. The SBB's pre-jump position and orientation parameters are stored to allow the user to return to the original location.

### *4.7.6 Animation Controls*

The final component of the SBB's Collaborative Workspace are the animation controls located within the HUD controls on the right panel. The controls consist of three buttons: Record, Playback, and Send. When the Record button is depressed, the top right-hand corner of the screen (where HUD images are displayed) is captured every other frame. The `pfuSaveImage` function used to capture the screen images is expensive in terms of performance, so the function is only called every alternating frame. This configuration maintains minimal application performance (i.e. 10 frames per second) while capturing enough

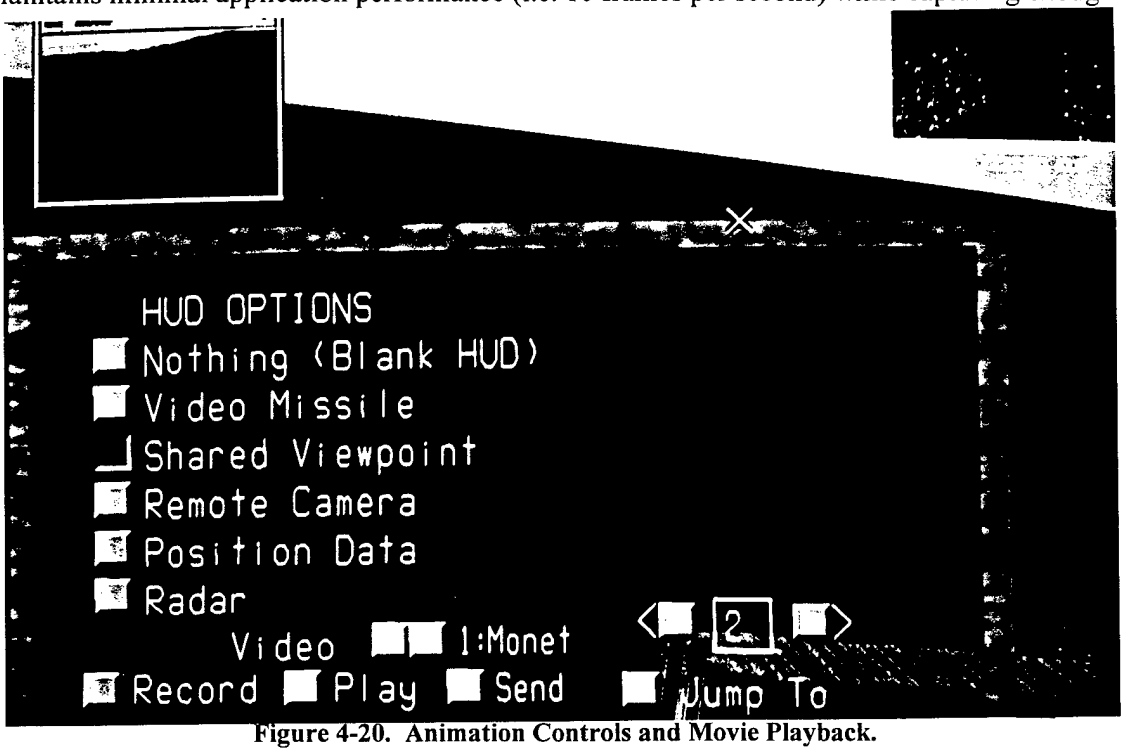



**Figure 4-20. Animation Controls and Movie Playback.**

images to produce a smooth animation. A captured image is saved to a file named "captureXXX.rgb," where XXX is the image's sequential number between 000 and 999. This gives the user the ability to capture 1000 images, every other frame, displayed at 15 frames per second, for a total of over two minutes of captured video. Playback of this video occurs through the use of SGI's movie player. The movie player reads in a series of images and displays them as ordered through their naming scheme. The movie is displayed at the same resolution of the captured images in a separately running window with its own controls (see Figure 4-20).

## 4.8. Conclusion

The development of the SBB's Collaborative Workspace has shown that computer supported cooperative work can be accomplished between multiple stealth entities. Each stage of the implementation provided a feature to the SBB that had not existed before. The CODB architecture allows simulation components to be created with easy access to shared memory. The SBB was rebuilt using the new CODB architecture and all ObjectSim code was removed. The graphics performance and potential for future expandability was improved by replacing all IRIS GL calls with OpenGL and X-Windows routines. The new DIS Manager and SBB World State Manager were integrated into the simulation to create a CODB-based interface to the DIS network. Several new interface components were developed and integrated into the application to enhance the existing Pod structure. These components include hidden panels, XForms-based dialogue boxes, the new Selection Manager, and the ability to quickly and fluidly navigate to any location with the virtual world. A collaborative workspace was developed and integrated into the new architecture providing basic CSCW capabilities between stealth entities; to include embodiment, message passing, chat sessions, annotations, shared viewpoints, and shared video.

The final result is a Collaborative Workspace running under the Synthetic BattleBridge that can support CSCW activities while operating in a DIS exercise. A Rumbaugh diagram of the Collaborative Workspace (as implemented), based on the SBB's overall design diagram in Figure 3-4 is shown in Figure 4-21.

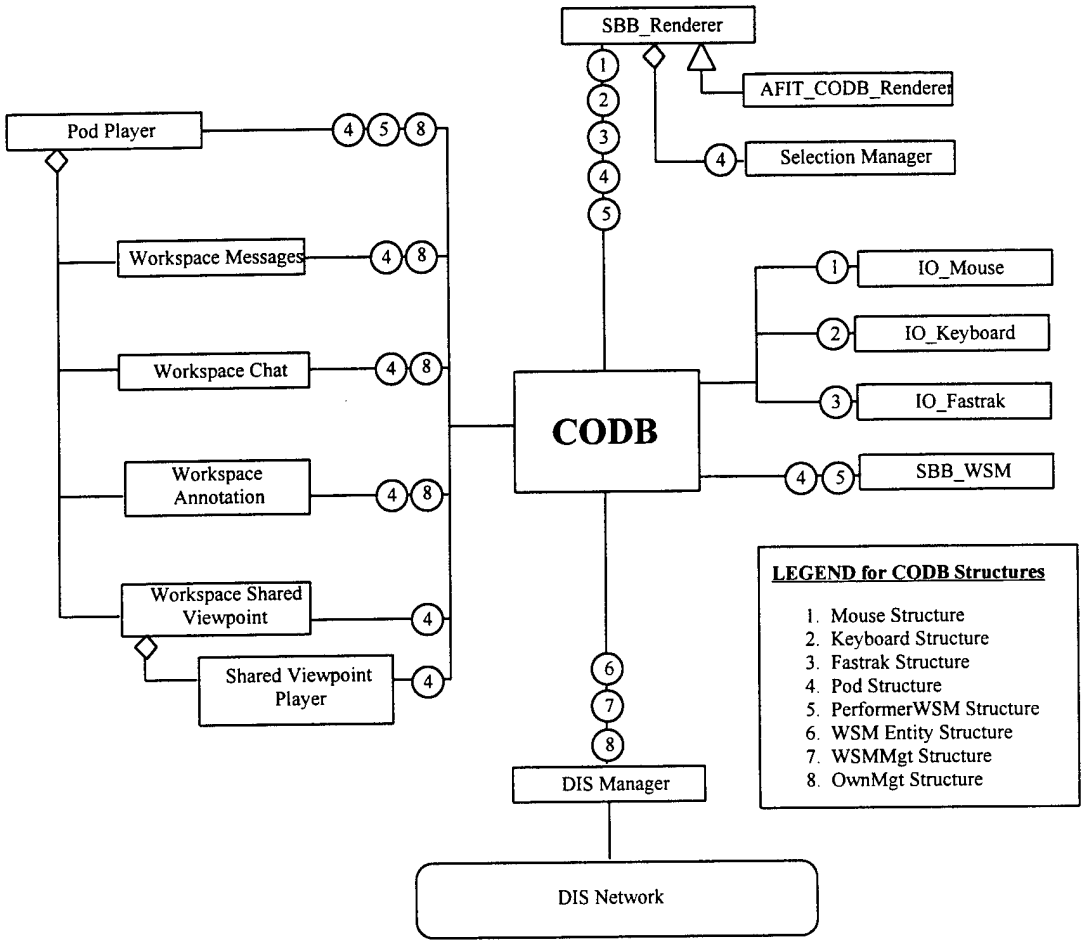



**Figure 4-21. Rumbaugh Diagram of the SBB's Collaborative Workspace.**

Note: Triangles represent inheritance, where the top-level component inherits the behavior of the lower level components. Diamonds represent aggregation, where the top-level component is made up of the lower level components.. Circled numbers beside a component represent the CODB data structures used by this component.

# 5. Results and Recommendations

## 5.1 Introduction

This chapter discusses the results of implementing each component of the SBB's Collaborative Workspace. Throughout this section, the requirements and goals found in chapter three (i.e. Table 3-1) will be referenced. The results in each design area were positive. The CODB architecture was successfully integrated into the SBB and all ObjectSim classes were removed. Graphics performance was increased through the use of industry standards in graphics programming. DIS support was provided by the CODB-based DIS Manager and allowed the reception of entity state PDUs, and the broadcast and reception of stealth management PDUs. A multi-layered interface was created to enhance the capabilities of the Pod. Support for CSCW activities was successfully implemented through the creation of the SBB's Collaborative Workspace. While the overall results were favorable, areas for improvement include the implementation of the virtual keyboard, support for audio, ATM support for audio and video transmission, and improved annotation manipulation techniques. The results in each of the five areas of the SBB's Collaborative Workspace design and implementation (software architecture, graphics performance and design, distributed simulation interface, user interface, and CSCW capabilities) are covered in detail in the following sections.

**Table 5-1. Software Architecture Requirements.**

| Requirement | Detailed Goal |
|---|---|
| **1. Software Architecture** | |
| 1.1. Increase flexibility of simulation framework by eliminating ObjectSim's constraints | Allow all Performer functionality to be available to developers by removing ObjectSim classes from SBB |
| 1.2 Provide architecture that will support all needed simulation components and be extensible | Utilize container-based approach to storing simulation data (CODB) |
| 1.3 Convert existing SBB to CODB architecture | CODB SBB works identically to ObjectSim SBB. Minimal loss of existing functionality. |
| 1.3.1 Pod must utilize CODB-based input and output | All input and output from Pod is CODB based |
| 1.3.2 Utilize CODB containers for simulation management data | Simulation management information will be available to all components in CODB |

## 5.2 Software Architecture

The results in the SBB's software architecture area were very good. The CODB was used to replace the ObjectSim classes contained in the previous SBB. Table 5-1 restates the software architecture requirements and goals found in Table 3-1. Each of these requirements and the results received are discussed below.

### *5.2.1. Requirement 1.1:*

*Increase flexibility of simulation framework by eliminating ObjectSim's constraints.* All ObjectSim classes were removed from the new SBB. This allowed input devices to be accessible at any place within the application. Input devices could also be overloaded with different functionality based on the current mode of the application. The removal of ObjectSim code also allowed the developer to access Performer functions directly. The replacement of ObjectSim's DIS interface is discussed in section 5.3 and the replacement of ObjectSim's renderer is discussed in section 5.2.

### 5.2.2. *Requirement 1.2:*

*Provide an architecture which will support all needed simulation components and be extensible.* The CODB architecture was used throughout the development of the new SBB and Collaborative Workspace. The CODB greatly eased the development process by allowing the programmer to concentrate on the data located in CODB containers and not the information stored within particular classes. An example is found in the development of the SBB's World State Manager. When the SBB's WSM was initially created, the DIS Manager was not able to pass entity state information to the SBB. Thus, to test the functionality of the WSM, a function was created to simulate DIS entities by filling the `WSMEntityStruct` CODB container with the same information that would be received from the DIS Manager. All knowledge of the non-stealth DIS environment occurs through this container. When the DIS Manager was ready for operation, the local entity-generating function was removed and the WSM worked exactly as tested.

### 5.2.3. *Requirement 1.3:*

*Convert existing SBB to CODB architecture.* From the baseline CODB demonstration program, the conversion of the SBB was relatively smooth.. The Pod Player and the center panel were implemented in a short period of time due to the lack of ObjectSim code within its hierarchy of subpanels. The left and right panels took longer to implement due to their extensive use of ObjectSim classes. The left panel retains the ability to attach to entities and locations, and supports threat ranges, trails, and locators. The right panel supports the full HUD and weather controls, but does not support bio-chemical functionality. The top and bottom panels are

identical in functionality to the previous SBB. All of the SBB's classes are able to access Pod and simulation management information through the CODB.

**Table 5-2. Graphics Performance and Design Requirements.**

| Requirement | Detailed Goal |
|---|---|
| **2. Graphics Performance and Design** | |
| 2.1 Replace ObjectSim's rendering functionality | Create CODB renderer class using ObjectSim algorithms |
| 2.1.1 Structure Performer tree to support annotation models and viewpoint algorithm | Create similar top-level tree structure as previous ObjectSim-based SBB |
| 2.1.2 Overcome Performer viewpoint resolution problem | Replace origin-centered viewpoint algorithm |
| 2.2 Support current h/w and support future SBB work with industry standards for graphics programming. | Convert SBB from IRIS GL to OpenGL and X-Windows. |

## 5.3 Graphics Performance and Design

The new SBB and Collaborative Workspace was able to meet all the graphics performance and design requirements presented in Chapter 3 (see Table 5-2). The two primary areas of concern were replacing the rendering functionality lost in the conversion to the CODB and improving graphics performance through the adherence to industry standards for graphics programming. The results in both of these areas are presented below.

### *5.3.1. Requirement 2.1:*

*Replace ObjectSim's rendering functionality.* The `SBB_Renderer` class was created to replace all ObjectSim rendering functionality needed within the new SBB and Collaborative Workspace. To be successful in fulfilling this requirement, two specific issues were addressed. First, the overall Performer tree was structured to hold separate subtrees for the terrain, DIS entities, and annotations. The resulting Performer tree is similar to the one created in the ObjectSim SBB. This was necessary to support the second goal, overcoming the Performer

resolution problem by using the viewpoint algorithm in ObjectSim's `View` class. Through the structure of the Performer tree and the integration of ObjectSim's viewpoint algorithm, the `SBB_Renderer` was able to produce smooth images, free of any jitter.

### *5.3.2. Requirement 2.2:*

*Support current hardware and support future SBB work with industry standards for graphics programming.* This goal was a proactive measure to improve graphics performance and aid in the future expansion of the SBB and its Collaborative Workspace. It met with surprising success that resulted in the entire SBB being converted from its IRIS GL-based legacy code to the new industry standard for creating advanced graphical applications, OpenGL and X-Windows. Three performance measures were used to determine the success of this conversion: porting time, overall application performance, and improved flexibility. The application's performance (to include graphics performance) received a negligible improvement under the multiple-processor, IRIS GL-optimized, Onyx RealityEngine$^2$ platform. This was to be expected as OpenGL tasks must be converted to IRIS GL calls, resulting in a overhead-generated performance hit. The SBB's performance under the Onyx platform was already high to begin with as a result of Performer's use of multiple processors and the graphics horsepower of the RealityEngine$^2$ architecture. Unexpected application performance results came from running the new SBB under the single-processor, OpenGL-optimized, Indigo$^2$ High Impact. The IRIS GL version of the new SBB ran at nearly 5 frames per second under the Indigo$^2$ platform. After the conversion, the OpenGL version ran at over 15 frames per second under the same platform. This allowed the remaining development of the Collaborative Workspace to be accomplished using the single-processor Indigo$^2$s. Improved flexibility was realized not only by the ability to use

more than one platform for development, but also by the capability to move input devices out of the time-critical rendering process and into the application process. This move was accomplished by converting the two IRIS GL input device classes, `IO_Mouse` and `IO_Keyboard` to their X-Windows-based counterparts, `X_Mouse` and `X_Keyboard`. These new classes are identical from the standpoint of the application programmer; producing the same data and called by the same methods. This similarity allowed the current IRIS GL-based research projects to convert to a mixed-model (IRIS GL and X-Windows) architecture with minimal changes in a manner of minutes.

**Table 5-3. Distributed Simulation Interface Requirements.**

| Requirement | Detailed Goal |
|---|---|
| **3. Distributed Simulation Interface** | |
| 3.1 Minimize interaction between DIS Manager and SBB | Utilize CODB architecture to support independently running DIS Manager process. |
| 3.2 Manage incoming DIS entity information | Retrieve and store Entity State PDU information from DIS Manager using CODB containers |
| 3.3 Support stealth management and CSCW activities through DIS interface | Use Stealth Entity State, Comment, and Annotation Update PDUs for communication with other stealth players. |
| 3.3.1 Minimize bandwidth caused by additional stealth-generated PDUs | Stealth Entity State operates like Entity State PDU (including heartbeat), Comment PDU sent out once per message, Annotation Update PDU sent only on state change to annotations (no heartbeat mechanism). |

## 5.4 Distributed Simulation Interface

The SBB's distributed simulation interface, DIS Manager 3.0, was used to satisfy all the requirements stated in Table 5-3. The DIS interface was able to receive entity state information, a feature common to both the previous and current versions of the SBB, and send and receive stealth management data, a feature unique to the current SBB. The specific results concerning each requirement are given below.

### *5.4.1 Requirement 3.1:*

*Minimize interaction between DIS Manager and SBB.* ObjectSim's distributed simulation interface required constant interaction between the DIS Manager and the host application. To increase speed, specialized versions of the DIS Manager were created to eliminate unused features. The new SBB runs a generic DIS Manager 3.0 interface under a separate process. Interaction has been reduced to three CODB containers (i.e. `WSMEntityStruct`, `WSMMgtStruct`, and `OwnMgtStruct`) and three calling functions (i.e. `broadcast_stealth_state`, `broadcast_comment`, and `broadcast_annotation_update`). No manipulation of `DIS_Manager` class data is performed by the SBB. Using the CODB containers allows the DIS Manager to process PDU information as quickly as possible without interruption from the SBB. For example, when the SBB needs to inform the DIS network of state changes, it fills the `OwnMgtStruct` container and makes the appropriate call to the DIS Manager. The converse is also true: the DIS Manager does not interrupt the SBB to inform it of changes to the DIS environment. Changes to the DIS environment are detected by examining change indicator variables within CODB containers.

### *5.4.2. Requirement 3.2:*

*Manage incoming DIS entity information*. This requirement was fulfilled by the creation of the `WSMEntityStruct` and the SBB's World State Manager class. The `WSMEntityStruct` holds all of the DIS entity state information used by the ObjectSim SBB. This CODB container was utilized by the SBB's World State Manager to retrieve entity state information from the DIS Manager and store the data in a local CODB container, `PerformerWSMStruct`. The

`PerformerWSMStruct` was used in turn by the `SBB_Renderer` to display entities and by various Pod panels (e.g. radar, attach to entity, entity statistics, etc.).

### *5.4.3. Requirement 3.3:*

*Support stealth management and CSCW activities through DIS interface*. The successful accomplishment of this requirement involved three areas: the use of existing PDUs and the creation of a new PDU type, the need to minimize network traffic between stealth players, and the creation of two new stealth management CODB containers.

The Entity State and Comment PDUs were used to broadcast stealth entity state changes and messages between stealth players. "Stealth Entity State PDUs" are not seen by non-stealth entities because the DIS Manager filters out Entity Types that correspond to stealth viewers. The Comment PDU is used for all text message traffic, including private email, public bulletins, and chat sessions. The Annotation Update PDU was created as the starting point for a new family of PDUs since none of the existing family of PDUs were appropriate for annotations created by stealth applications. The Action Request and Action Response PDUs are similar in nature to the action-based functionality required by the Annotation Update PDU, and were used as models, combined with features from the Comment and Entity State PDU types, to form this new experimental PDU. The format of the Annotation Update PDU is shown in Appendix B.

In keeping with the philosophy that stealth players are intended to be silent participant in the DIS exercise, I minimized the number of PDUs broadcast by stealth entities. Stealth players operate like normal entities, broadcasting movement changes that exceed dead-reckoning limits and "heartbeats" to keep the player "alive" in the exercise. Single messages are broadcast one time only. If a stealth player enters the DIS exercise late, he does not receive old messages.

Similarly with chat sessions, upon entering a chat session, the stealth player does not receive the current chat session log. Annotations operate under a different principle. In keeping with the philosophy of the DIS community, a central annotation management server was not created. Thus, annotations do not emit a "heartbeat" to keep them in the simulation. Instead, annotation information is received via annotation updates. Upon receiving an annotation update, the newly-arrived stealth player is able to retrieve a complete record of the annotation's state.

Two new CODB containers, `WSMMgtStruct` and `OwnMgtStruct`, were constructed for use in communicating stealth management information between the DIS Manager and the SBB. The `WSMMgtStruct` was used to receive information and the `OwnMgtStruct` was used for broadcasting information. Incoming stealth entity state information was passed to the `PerformerWSMStruct` and message/annotation information was transferred to the `PodStruct`.

**Table 5-4. User Interface Requirements.**

| Requirement | Detailed Goal |
|---|---|
| **4. User Interface** | |
| 4.1 User interface must be easy to use | Keep consistent interface between textual modes of communication (email, bulletins, chat, annotations) |
| 4.2 Do not clutter the existing Pod controls | Create multi-layered interface, utilize switchable panels, and utilize Selection Manager for interaction with world annotations. Determine appropriate location of controls with respect to previous SBB designs. |
| 4.3 Create a method of text input suitable for use with an HMD. | Design virtual keyboard for future implementation. Expand multi-layered interface to include an text input windows that cannot be misplaced by the HMD user. |
| 4.4 Expand navigation controls | Allow user to detach from Pod. Overload mouse to support Performer "Fly" mode. Allow detached user to return to Pod or move Pod to current viewpoint. |

## 5.5 User Interface

The user interface received a number of enhancements over the previous version of the SBB. Great care was taken in adding capability to the SBB without burdening the user with visual clutter. All the requirements in this area (see Table 5-4) were met and the specific results for each requirement are given below.

### *5.5.1. Requirement 4.1:*

*User interface must be easy to use.* The original plan of testing the user interface's ease of use was to perform a series of user studies comparing the previous SBB with the new SBB and Collaborative Workspace. One factor made such testing impractical. The current SBB user population consists of one person. Only the author is familiar with both interfaces and available for testing. While a handful of former students are knowledgeable enough in the SBB's Pod design to compare it against the new version, it was not practical to bring them to AFIT to test the new design.

Given the inability to conduct user tests, the ease of use goals were modified to state that interface between the old and new versions of the SBB must be consistent. The Pod's interface has not changed in appearance and the new workspace subpanels are either hidden or well-integrated into the existing structure. The workspace controls follow the style and conventions used in other Pod panels. The Workspace Message, Workspace Chat, and Workspace Annotation controls are all very similar in appearance and function, reducing the interface learning curve between CSCW activities. The Workspace Shared Viewpoint HUD operates in a fashion consistent with the existing Drop Camera HUD controls.

### *5.5.2. Requirement 4.2:*

*Do not clutter the existing Pod controls.* The new SBB uses a wide variety of interface improvements to keep the visual appearance of the Pod from looking too cluttered with new controls. A multi-layered interface was created that consists of XForms 2D windows for text input (discussed next), the existing Pod controls (a 2½D interface), and the Selection Manager for the selection and manipulation of 3D objects beyond the Pod's console. Hidden, switchable Pod panels were created to reduce the visual clutter of adding three new control panels (e.g. messages, chat, and annotations) to the existing Pod interface. Where possible, new workspace controls were integrated into the existing control panels. Examples of this include the Workspace Shared Viewpoint and the video/animation controls.

### *5.5.3. Requirement 4.3:*

*Create a method of text input suitable for use with an HMD.* Due to time constraints and hardware problems, the virtual keyboard designed in chapter three was not implemented. Instead, another method of providing text input to the HMD-wearing user was created. The result was a 2D Motif-style dialogue box created with XForms. The XForms window is overlayed on the current scene, creating an interface that is not dependent on the user's viewpoint. Therefore, a user can still change the viewpoint and move about the virtual world without losing the XForms interface. The dialogues boxes follow similar conventions: they are created under the mouse's current position, they can be moved to another location on the screen, each selection item can be reached by pressing the TAB key, the user can press the ESC key to cancel a form, and the form's input will not be sent until the ENTER key is pressed. Given these features, the XForms interface successfully meets its stated requirements.

### *5.5.4. Requirement 4.4:*

*Expand navigational controls.* The navigational controls were expanded by incorporating a mouse-based similar to SGI's Fly interface. This interface, created during the CODB demonstration project, allows the user to detach from the Pod and travel quickly to any location in the virtual environment. Two additional controls were added to allow the user to bring the Pod to the detached user's location and to return the detached user back to the Pod.

**Table 5-5. CSCW Capabilities Requirements.**

| Requirement | Detailed Goal |
|---|---|
| **5. CSCW Capabilities** | |
| 5.1 Support communication of private email and public bulletins | Use Comment PDU with single user or public audience specifier. |
| 5.1.1 Support common message functions. | Implement sending messages, saving a message to a file, deleting a message, displaying messages in received order, and replying to a message. |
| 5.2 Support real-time chat communication | Use Comment PDU with chat audience specifier. |
| 5.2.1 Support common chat functions. | Implement sending a chat message, saving a chat log file, deleting a chat log file, displaying multiple chat messages in received order on scrolling display with identifiers for each participant. |
| 5.3 Support shared viewpoints | Use stealth Entity State PDU to send stealth position and orientation information. Create HUD to display stealth players' viewpoints with these coordinates. |
| 5.3.1 Support user teleportation to shared viewpoint | Allow user to jump to a viewpoint (with same position and orientation of viewpoint) and return to original coordinates. |
| 5.4 Support animation/video | Allow user to capture, transmit and playback a sequence of HUD images. |
| 5.5 Support annotations with the virtual environment | Create Annotation Update PDU. Create appropriate metaphor and display techniques needed for annotations. |
| 5.5.1 Support annotation functions | Implement creating an annotation, deleting an annotation, appending text to an annotation, clearing messages from an annotation, selecting an annotation, moving an annotation, and attaching an annotation to a DIS entity. |

## 5.6 CSCW Capabilities

The primary goal of this research project was to implement CSCW-supporting capabilities within a DIS application to allow networked players to carry out some form of

collaborative work. Because of the importance of implementing CSCW support with the SBB's Collaborative Workspace, the requirements found in Table 5-5 were given top priority over all other requirement areas. Every requirement was satisfied and all goals were reached in this area.

### *5.6.1. Requirement 5.1:*

*Support communication of private email and public bulletins.* The goal of this requirement was to use the information transmitted by the Comment PDU to support a variety of common message-based functions. The DIS interface support for messages has been discussed in section 5.3. Audience specifiers in the Comment PDU allowed messages to be broadcast to a single entity, a group of entities, or all entities. This specifier identified each message as either a private email, a chat message, or a public bulletin. Received messages could then be sorted into their respective categories for use in the Workspace Message or Workspace Chat Pod panels.

The Workspace Message panel allowed private and public messages to be displayed, saved to a file, deleted, or sent via the touch of a button. The Pod interface did limit the user to sending only preloaded messages, but the XForms message interface expanded the Workspace Message panel's capability by allowing the freedom to input any message desired. Received messages were sorted sequentially and displayed the sender's ID and name in the header of each message. A word-wrap algorithm was created to correctly display long messages in 3D space.

### *5.6.2. Requirement 5.2:*

*Support real-time chat communication.* The Workspace Chat goals were similar to the Workspace Message goals. Using the information sent and received by a Comment PDU, the Workspace Chat console supported a variety of typical chat functions. The functions included

displaying a chat session, saving a chat session to a file, deleting the chat log, and sending chat messages using only the Pod interface. Once again the sender was limited to preloaded messages, but the capability was expanded through the use of the same XForms message interface used with the Workspace Message controls.

One difference between regular messages and chat messages was the dropping of chat messages if the user did not have the Workspace Chat controls turned on. Chat messages that were received used the same display panel as Workspace Messages. This panel displayed the conversation between stealth players in a style similar to MUDs and IRC. The word-wrap algorithm was used in combination with a scrolling technique that keep new messages at the bottom of the screen, while old messages scrolled off the top. Each stealth player in a chat session is referenced by a shortened user name and unique color code for his messages.

### *5.6.3. Requirement 5.3:*

*Support shared viewpoints.* Shared viewpoints operated in the same fashion as drop cameras. In each case, the user is able to watch a remote viewpoint based on provided Performer coordinates. In the case of shared viewpoints, these coordinates are provided by the stealth entity state information received from the DIS Manager. When more than one stealth player is operating in the DIS exercise, the user is able to cycle through the viewpoints of the other stealth players.

### *5.6.4. Requirement 5.4:*

*Support animation/video.* The proof-of-concept demonstration of sharing animation and video between stealth players was accomplished by capturing HUD images, transmitting these

images to a particular stealth player, and having that user playback the images in the form of a movie. Two areas of improvement are need for this capability to be used effectively. The first is that the current method of capturing images is very expensive in terms of computational power. A better technique is needed to maintain acceptable levels of application performance. The second is that the current method of transferring images is very slow. Due to time constraints, images are transferred through UNIX's remote copy command, rcp. A better approach would be to create an ATM client/server process that could transmit and receive multiple images on demand.

### *5.6.5. Requirement 5.5:*

*Support annotations within a virtual environment.* Annotations were successfully implemented in the SBB's Collaborative Workspace, meeting all of its requirements listed in Table 5-5. These annotations form the basis for a new class of DIS objects. While the current annotation implementation contains only text messages and performs basic annotation operations, the infrastructure is expandable to a wide variety of media types and actions. The Annotation Update PDU was designed to be extensible with the first proposed improvement being the replacement of annotation's text message with a White Board PDU-like format.

## 5.7 Conclusions and Recommendations

The SBB's Collaborative Workspace was a successful implementation of CSCW functionality within a distributed virtual environment using DIS protocols. The Collaborative Workspace also met all of the requirements of this research project. Many firsts were accomplished during this research effort. The new SBB was the first AFIT distributed

simulation application to be converted to the CODB architecture and Performer 2.0. The new SBB was the first AFIT distributed simulation application to use OpenGL and X-Windows instead of IRIS GL. The new SBB was the first AFIT distributed simulation application to use a multi-layered interface, DIS Manager 3.0, and an ATM interface. The SBB's Collaborative Workspace was the first AFIT DIS application to support real-time communication between players. The SBB's Collaborative Workspace was the first DIS application to support annotations, shared viewpoints, and shared video within a virtual environment.

Recommendations for continued research in the field of CSCW for virtual environments include implementation of the virtual keyboard designed in Chapter 3, support for audio communication, creation of an ATM client/server mechanism for transmission of audio and video messages, improvement of manipulation techniques for annotations, and use of expanded multimedia capabilities throughout the Collaborative Workspace (i.e. messages, chat, annotations). Recommendations for continued research in the field of providing CSCW support for DIS applications include the implementation of the White Board PDU and creation of the Annotation Data PDU. The White Board PDU should replace the Comment PDU for basic message transfers and be integrated into the Annotations Update PDU where future extensibility to the WBPDU is already supported. The Annotation family of PDUs should be expanded to permit late arrivals to the DIS environment to be informed of all annotations within the battlespace. The Annotation Update PDU already supports the broadcast of an `annotation_request` action, but the corresponding Annotation Data PDU type to permit SBBs to send their annotation information to the latecomer has not been designed.

The goal of this thesis effort was to open up as many avenues of communication as possible between SBB users so that collaboration between remote players could occur. The Collaborative Workspace was able to demonstrate that computer-supported collaborative work is

possible within a DIS virtual environment. This support was accomplished by providing CSCW-enabling technologies that included the ability for users to send and receive: public bulletins, private email, chat messages, shared viewpoints, video clips and annotations. All of these features were incorporated into the Synthetic BattleBridge using the Pod as a framework for the user-interface. The Comment PDU type was used to transfer text messages between users while a new experimental PDU type, the Annotation Update PDU was created to support the unique requirements of managing annotations. This research project successfully combined the fields of software architecture, graphics performance, user interface design, distributed simulations, and CSCW to create the world's first collaborative workspace for distributed virtual environments using DIS protocols.

# Appendix A

# Improving Graphics Performance Through the Use of Industry Standards for Graphics Programming

## A.1 Converting to X-Windows

As recommended in the "OpenGL Porting Guide" produced by Silicon Graphics, the first step in the overall conversion process is to create a mixed-model application where IRIS GL is still used for graphics and X-Windows is used for window management and event handling. Due to Performer's support for IRIS GL, OpenGL, and X-Windows, the window management changes were straightforward. The second part to creating a mixed-model application is the conversion of all IRIS GL-base input devices to their X-Windows counterparts.

The baseline CODB demonstration project resulted in four input classes being created: `IO_Mouse`, `IO_Keyboard`, `IO_Fastrak`, and `IO_Hotas`. Each of these classes were initialized in their respective constructors and their CODB structures were updated by calling the device's `poll` method. The goal of converting the two IRIS GL-based device classes, `IO_Mouse` and `IO_Keyboard`, was to keep the application interfaces identical to the current methods and CODB structures. The easiest approach to this problem was to make extensive use of the `pfutil` library. This Performer utility library is an extension to the Performer libraries and includes a wide variety of routines such as processor control, GLX mixed-model routine, input handling, cursor control, font control, basic user interface, smoke, texture loading and animation, and more. In order to make use of the `pfutil` library, it must first be initialized in

two places: directly after Performer's pfConfig call using the pfuInitUtil method and after the Performer window has been fully configured using the pfuInitInput method.

The IO_Mouse class was converted to the current X_Mouse class by first replacing the IRIS GL mouse device references with pfuMouse equivalents. Mouse output consists of a scaled X position [-1,1], a scaled Y position [-1,1], and a numerical representation of the number of buttons being depressed. In order to retrieve this information, the IRIS GL-based IO_Mouse called the getvaluator function for each axis, called functions to return the window's origin and size, determined whether or not the mouse position fell within the window boundaries, and then wrote the scaled values to the CODB MouseStruct container. Mouse buttons values were found with the getbutton function. Using X_Mouse, polling is now a three step process:

1. pfuGetMouse copies the current mouse values from the pfutil event collector (initially triggered with pfuInitInput) into the pfuMouse structure,
2. pfuMouseInChan maps the mouse screen coordinates (mouse->xpos, mouse->ypos) into coordinates in the range [-1, 1] (mouse->xchan, mouse->ychan) based on the Performer channel's viewport, and
3. processMouseInput copies the relevant pfuMouse information into the MouseStruct CODB container for use by the application.

The IO_Keyboard class was converted to the current X_Keyboard class by first removing all the IRIS GL device references and replacing them with pfuEventStream equivalents. Keyboard output consists of a keyboard value and a device number. In the IRIS GL-based IO_Keyboard, retrieval of this information consisted of queuing each device that the keyboard wanted to monitor (i.e. A-Z keys, keypad, function keys, etc.), and then testing the

queue to see if anything had been pressed. If nothing was in the queue, zeros were written to the `KeyboardStruct` CODB container; otherwise, the queue was read and the results were posted to `KeyboardStruct`. Only the first item in the queue was read, any remaining contents were flushed. "Polling" the `X_Keyboard` class involves reading the current event stream with `pfuGetEvents` and then processing the input. This processing first entails determining the number of events that occurred. X events are not limited to keypresses, but include such things as focus changes, creation or deletion of a window, and window redraws. If no events occurred, zeros are written to the `KeyboardStruct`. If an event has occurred, the device number is checked to see if it corresponds to a keyboard-related event. If this is the case, then that particular keystroke's value is recorded in the `KeyboardStruct` container.

The conversion to X-Windows took one day of research, two days of coding, and resulted in a minor change to the `AFIT_CODB_Renderer` and the creation of two new X-Windows-based input classes: `X_Mouse` and `X_Keyboard`. This process was portable enough that two other IRIS GL-based virtual environment projects, the Virtual ER and the Virtual Cockpit, were able to incorporate the changes within a manner of minutes. Speed improvements in the actual device classes were negligible, but graphics performance improvements were substantial due to moving input device polling out of the rendering process.

## A.2 Converting to OpenGL

The difficult task of converting over 20,000 lines of IRIS GL-based source code was greatly eased through the use of an SGI-supplied script called "toogl." Toogl (To OpenGL) is a script that takes IRIS GL code as input and produces commented, nearly equivalent OpenGL code as output. While toogl can't do everything, it is able to do the tedious job of changing

command names. When it comes upon a particularly difficult piece of code that may need to be ported by hand, it marks the potential problem with a comment starting with "OGLXXX". Toogl will also make an attempt at inserting the proper parameters into converted function calls. If there is any doubt, toogl will leave one of its attention-getting comments so that the developer can correct the parameters if needed. Note that toogl will not convert event handling or window management routines. This is why the mixed-model approach was performed first. Two items that toogl is unable to convert are color constants and font management.

In IRIS GL, there are a number of defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL does not provide these constants and toogl does not translate them, so they need to be ported by hand. Related to this issue is that the Pod uses its own defined palette of colors, found in `cp_colors.h`: CP_WHITE, CP_BLACK, CP_RED, CP_GREEN, CP_GRAY127, etc. The problem occurs in the way colors are set and the order that a color's components are defined. The following code fragments display the differences between IRIS GL and OpenGL colors:

**IRIS GL**

color defined with:

```
enum CP_COLOR {
//                    A B G R
CP_BLACK        = 0x00000000,
CP_WHITE        = 0xffffffff,
CP_RED          = 0x000000ff,
CP_GREEN        = 0x0000ff00,
CP_BLUE         = 0x00ff0000,
CP_CYAN         = 0x00ffff00,
```

color set with:

```
cpack (the_color);
```

**OpenGL**

color defined with:

```
typedef const GLubyte CP_COLOR[4];
//                          RED  GRN  BLUE ALPH
CP_COLOR CP_BLACK   = {0x00,0x00,0x00,0xff};
CP_COLOR CP_WHITE   = {0xff,0xff,0xff,0xff};
CP_COLOR CP_RED     = {0xff,0x00,0x00,0xff};
CP_COLOR CP_GREEN   = {0x00,0xff,0x00,0xff};
CP_COLOR CP_BLUE    = {0x00,0x00,0xff,0xff};
CP_COLOR CP_CYAN    = {0x00,0xff,0xff,0xff};
```

color set with:

```
glColor4ubv(the_color);
```

**Figure A-1. Comparison Between IRIS GL and OpenGL Color Definitions and Use.**

The most noticeable difference is the reversed order of the colors components. IRIS GL uses ABGR ordering and OpenGL uses the standard RGBA format. Color packing is not used in OpenGL, so an equivalent glColor command is called to read in the user-defined color's array of RGBA values.

The second problem concerned font management. IRIS GL relies on its Font Manager to render text strings to the IRIS GL window. Since this type of functionality would be needed for labels and the Pod's positional HUD, it was decided to create an OpenGL/X-Windows Font Manager class. The new Font Manager can draw text strings to the screen as an overlay or as text displayed in the 3D world. The routines needed to initialize the Font Manager and display text strings in 3D space (such as labels for annotations) are shown below.

```
static GLuint fontbase;

Font_Manager::Font_Manager ()
{
   initFont();
}

void Font_Manager::Draw_String3D(char* string, const pfVec3 Coords)
{
   static int do_once = 1;

   if (do_once)
   {
      makeRasterFont();
      do_once = 0;
   }
   pfPushState();
      pfBasicState();
      GLfloat Yellow_Color[3] = { 1.0f, 1.0f, 0.0f };
      glColor3fv(Yellow_Color);
      glRasterPos3f(Coords[PF_X], Coords[PF_Y], Coords[PF_Z]);
      printString (string);
      glFlush();
   pfPopState();
}

void Font_Manager::printString(char *s)
{
   glPushAttrib (GL_LIST_BIT);
      glListBase(fontbase);
      glCallLists(strlen(s), GL_UNSIGNED_BYTE, (unsigned char *)s);
   glPopAttrib ();
}
```

```
void Font_Manager::makeRasterFont(void)
{
   fontbase = glGenLists(last+1);
   glXUseXFont(fid, first, last-first+1, fontbase+first);
}

void Font_Manager::initFont(void)
{
   xdisplay = pfGetCurWSConnection();
   fontInfo = XLoadQueryFont(xdisplay,
              "-adobe-courier-medium-r-normal--20-400-0-0-m-100-iso8859-1");
   fid      = fontInfo->fid;
   first    = fontInfo->min_char_or_byte2;
   last     = fontInfo->max_char_or_byte2;
}
```

**Figure A-2. Font Manager Code Fragment.**

It is surprisingly difficult to obtain clear examples the Performer, OpenGL, and X-Windows libraries being used at the same time. The above code is presented in this thesis to show how to combine all three libraries' functionality to support a single operation. The new Font Manager class is currently being to label annotations in 3D space, to provide a positional HUD for the Pod, and to describe the owner of the current shared viewpoint HUD.

The conversion to OpenGL took three days of combined research and coding. Most of the coding took the form of fixing minor graphical bugs or tending to the comments left by toogl. A complete color palette was recreated for use by the Pod and a new OpenGL/X-Windows Font Manager class was created to replace the functionality lost by the IRIS GL Font Manager class. The SBB is now completely OpenGL and X-Windows-based and was recompiled using the OpenGL-specific Performer libraries. The outcome of the conversion is that acceptable graphics performance is now possible using a single processor Indigo$^2$ High Impact.

# Appendix B

# Annotation Update PDU Specifications

## B.1 Annotation Update PDU Format

| Field Size (bits) | Annotation Update PDU Fields | |
|---|---|---|
| 96 | PDU Header | Protocol Version — 8-bit enumeration |
| | | Exercise ID — 8-bit unsigned integer |
| | | PDU Type — 8-bit enumeration |
| | | Protocol Family — 8-bit enumeration |
| | | Time Stamp — 32-bit unsigned integer |
| | | Length — 16-bit unsigned integer |
| | | Padding — 16 bits unused |
| 48 | Originating Entity ID | Site — 16-bit unsigned integer |
| | | Application — 16-bit unsigned integer |
| | | Entity — 16-bit unsigned integer |
| 64 | Annotation ID | Site — 16-bit unsigned integer |
| | | Application — 16-bit unsigned integer |
| | | Entity — 16-bit unsigned integer |
| | | Annotation — 16-bit unsigned integer |
| 16 | Annotation Action | 16-bit enumeration |
| 32 | Padding | 32 bits unused |
| 32 | Number of Fixed Datum Records ($N$) | 32-bit unsigned integer |
| 32 | Number of Variable Datum Records ($M$) | 32-bit unsigned integer |
| 64 | Fixed Datum #1 | Fixed Datum ID — 32-bit enumeration |
| | | Fixed Datum Value — 32-bits |
| ... | | |
| 64 | Fixed Datum #$N$ | Fixed Datum ID — 32-bit enumeration |
| | | Fixed Datum Value — 32-bits |
| $64 + K_1 + P_1$ | Variable Datum #1 | Variable Datum ID — 32-bit enumeration |
| | | Variable Datum Length — 32-bit unsigned integer ($K_1$) |
| | | Variable Datum Value — $K_1$ bits |
| | | Padding — $P_1$ bits |
| ... | | |
| $64 + K_M + P_M$ | Variable Datum #$M$ | Variable Datum ID — 32-bit enumeration |
| | | Variable Datum Length — 32-bit unsigned integer ($K_M$) |
| | | Variable Datum Value — $K_M$ bits |
| | | Padding — $P_M$ bits |
| Total Annotation Update PDU size = 320 bits + sizeof all fixed datum records + sizeof all variable datum records | | |

## B.2 Annotation Update PDU Enumerations

### *B.2.1 Annotation Action Enumeration*

| Field Value | Annotation Action |
|---|---|
| 0 | Other |
| 1 | Create |
| 2 | Remove |
| 3 | Select |
| 4 | Deselect |
| 5 | Move |
| 6 | Append Text |
| 7 | Clear Text |
| 8 | Attach To Entity |
| 9 | Detach From Entity |
| 10 | Annotation Query |
| 11 | No Action |

### *B.2.2 Datum ID Enumerations*

| Field Value | Fixed/Variable Datum ID Name | Datum Value Size | Datum Value Type |
|---|---|---|---|
| 15000 | DIS Identity | | |
| 15100 | DIS Site ID | 32 | Unsigned Integer |
| 15200 | DIS Host ID | 32 | Unsigned Integer |
| 15300 | DIS Entity ID | 32 | Unsigned Integer |
| 31200 | Geocentric Coordinates | | |
| 31250 | X | 64 | 64-bit float |
| 31260 | Y | 64 | 64-bit float |
| 31270 | Z | 64 | 64-bit float |
| 102000 | Annotation Family | | |
| 102100 | Annotation Text | multiple of 64 | multiple 8-bit characters |

## B.3 DIS Basic Data Types and Records

From *IEEE Standard for Distributed Interactive Simulation -- Application Protocols (IEEE Standard 1278.1-1995)*

- PDU Header record defined in section 5.2.24 of *IEEE 1278.1-1995.*
- Datum Specification record defined in section 5.2.10 of *IEEE 1278.1-1995.*
- Fixed Datum record defined in section 5.2.20 of *IEEE 1278.1-1995.*
- Variable Datum record defined in section 5.2.32 of *IEEE 1278.1-1995.*

## B.4 DIS Enumerations

From *Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications* (Institute for Simulation and Training Document Number *IST-CR-95-14*):

- Enumerations 15000, 15100, 15200, and 15300 are from *IST-CR-95-14.*
- Enumeration 31200 is from *IST-CR-95-14.*
- Enumerations 31250, 31260, and 31270 are new.
- Enumerations in the 102000 family are new.

## B.5 Use of Annotation Update Datum Records

| **Datum Records** ⟶ **Annotation Action** | **Entity ID Datum Records (1)** | **Geocentric Location Datum Records (2)** | **Annotation Text Datum Record** |
|---|---|---|---|
| **Create** | Possible (3) | Possible (3) | Required |
| **Remove** | | | |
| **Select** | | | |
| **Deselect** | | | |
| **Move** | | Required | |
| **Append Text** | | | Required |
| **Clear Text** | | | |
| **Attach to Entity** | Required | | |
| **Detach From Entity** | | Required | |
| **Annotation Query** | | | |
| **No Action** | | | |

Notes:

1. The Entity ID Datum Records are actually the three DIS Identity Datum Records (DIS Site ID, DIS Host ID, and DIS Entity ID).
2. The Geocentric Location Datum Records are actually the three Geocentric Coordinates Datum Records (X, Y, and Z).

For the Create Annotation Action: if the annotation is to be immediately attached to an annotation, the Entity ID datum records are to be used and the location datum records not used. If the annotation is a location-based annotation, the location datum records are used and the Entity ID datum records are not used.

# Bibliography


[AMID95] Amidon, Bob and Brown, Dan, "Collaborative System for Virtual Environments," *Proceedings of the 1995 SIGCHI Conference*, p. 112-124, San Diego, CA, August 1995.

[AMSE95] Amselem, Denis, "A Window on Shared Virtual Environments," *Presence,* 4 (2), p. 130-145, Spring 1995.

[BANN91] Bannon, L. J. and Schmidt, K., "CSCW: Four Characters in Search of a Context, " *Studies in Computer Supported Cooperative Work*, Elsevier Science Publishers., North Holland, 1994.

[BARR95] Barrus, J.W., Waters, R.C., and Anderson, D.B., "Locales and Beacons: Precise and Efficient Support For Large Multi-User Virtual Environments," Mitsubishi Electric Research Laboratories Technical Report TR95-16, Cambridge, MA, 1995.

[BART90] R. Bartle, "Interactive Multi-User Computer Games," December 1990. ftp://ftp.ccs.neu.edu//pub/mud/docs/papers/mudreport.ps.gz

[BENE91]. Benedikt, Michael, "Cyberspace: Some Proposals," *Cyberspace: First Steps*, p. 119-224, Massachusetts Institute of Technology, 1991.

[BENF92] Benford, S., et. al., "MOCCA: A Distributed Environment for Collaboration," *Proceedings of Telepresence '93*, Lille, France, March 1993.

[BENF93a] Benford, S., et. al., "From Rooms to Cyberspace: Models of Interaction in Large Virtual Computer Spaces," *Interacting With Computers*, Butterworth-Heinmann, 1993.

[BENF93b] Benford, S., Fahlen, L., "A Spatial Mode of Interaction for Large Virtual Environments," *Proceedings of ECSCW'93*, Milan, September 1993.

[BENF94] Benford, S., et. al., "Managing Mutual Awareness in Collaborative Environments," *Proceedings of VRST'94*, Singapore, August 1994.

[BENF95a] Benford, Steve et al. "User Embodiment in Collaborative Virtual Environments," *Proceedings of the 1995 SIGCHI Conference*, p. 145-166, San Diego, CA, August 1995.

[BENF95b] Benford, S., Fahlen, L., "Viewpoints, Actionpoints and Spatial Frames for Collaborative User Interfaces," University of Nottingham Technical Report 95-XX-XXX.

[BLAN90] Blanchard, Chuck et al. "Reality Built for Two: A Virtual Reality Tool," *Proceedings of the 1990 Symposium on Interactive 3D Computer Graphics*, p. 35-36, Snowbird, UT, March 1990.

[BLAU94] Blau, B., et. al., "The DIS (Distributed Interactive Simulation) Protocols and Their Application to Virtual Environments," Institute for Simulation and Training, Orlando, FL, 1994.

[BLOC94] Block, Elizabeth and Stytz, Martin. "Tools for Commander and Staff Training in Large-Scale, Distributed Virtual Realities: Concepts and Implementation," *Proceedings of the 1994 Simulation Multiconference*, p. 43-48, San Diego, CA, April 1994.

[BOLT95] Bolter, J., Hodges, L., Meyer, T., Nichols, A., "Integrating Perceptual and Symbolic Information in VR," IEEE Computer graphics & Applications, July 1995.

[BRYS94] Bryson, Steve, "Approaches to the Successful Design and Implementation of VR Applications," *Proceedings of SIGGRAPH'94*, Orlando, FL, 1994.

[BURK93] Burka, Lauren P., *A Hypertext History of Multi-User Dimensions,* http://www.utopia.com/talent/lpb/muddex/

[BYHM92] Bvhm, K., et al., "GIVEN: Gesture Driven Interactions in Virtual Environments - A Toolkit Approach to 3D Interactions," *Proceedings of the Interfaces to Real and Virtual Worlds Conference*, Montpelier, France, March 1992.

[CARL93] Carlsson, C. and Hagsand, O., "DIVE: A Platform for Multi-User Virtual Environments," *Computers and Graphics*, Volume 17, Issue 6, p. 663-669, November/December 1993.

[CONK87] Conklin, J., "gIBIS: A Hypertext Tool for Team Design Deliberation," *Proceedings of Hypertext'87*, p 247-251, November 1987.

[CONN92] Conner, D.B., Snibbe, S.S., Herndon, K.P., Robbins, D.C., Zeleznik, R.C. and van Dam, A., "Three-dimensional Widgets," *Computer Graphics (Proceedings of the 1992 Symposium on Interactive 3D Graphics)*, ACM SIGGRAPH, p. 183-188, March 1992.

[DANI88] Danielson, T., Panoke-Babatz, U., "The AMIGO Activity Model," *Proceedings of EUTECO'88*, Vienna, Austria, April 20-22, 1988.

[DIAZ94] Diaz, Milton E. *The Photo Realistic AFIT Virtual Cockpit*, Masters Thesis, AFIT/GCS/ENG/94D-02, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1994.

[DISV91] *DIS Vision, The*. DIS Steering Committee, Institute for Simulation and Training, May 1991.

[ELLI91a] Ellis, C. A., Gibbs, S. J., and Rein, G. L., "Groupware: Some Issues and Experiences," *Communications of the ACM*, 34 (1), 1991.

[ELLI91b] Ellis, S. R., "Nature and Origins of Virtual Environments: A Bibliographic Essay," Computer Systems in Engineering, Pergamon Press, U.K., 1991.

[ERIC93] Erichsen, M. N., *Weapon System Sensor Integration for a DIS-Compatible Virtual Cockpit*, Masters Thesis, AFIT/GCS/ENG/93-07, Air Force Institute ofTechnology, Wright-Patterson AFB, OH, 1993.

[EVAR93] Evard, Rémy, "Collaborative Networked Communication: MUDs as Systems Tools," Proceedings of the Seventh Systems Administration Conference (LISA VII), p. 1-8, Monterey, CA, November 1993.

[FEIN93] Feiner, S., et. al., "Windows on the World: 2D Windows for 3D Augmented Reality," *Proceedings of UIST '93*, Atlanta GA, November 3-5, 1993.

[FIGU93] Figueiredo, M., Byhm, K., Teixeira, J., "Advanced Interaction Techniques in Virtual Environments," *Computers & Graphics*, 17 (6), p. 655-661 1993.

[FISH88] Fish, R. S., Kraut, R. E., Leland, M. D., Cohen, M., "Quilt: A Collaborative Tool for Cooperative Writing," *COIS'88 Proceedings*, Palo Alto, CA, March 23-25, 1988.

[FLOR88] Flores, F., Graves, M., Hartfield, B. and Winograd, T., "Computer Systems and the Design of Organizational Interaction," *Transactions on Office Information Systems*, 6 (2), pp. 153-172, April 1988.

[FORT94] Fortner, J., L., *"Distributed Interactive Simulation Virtual Cassette Recorder (DIS VCR): A Datalogger with Variable-Speed Relay*," Masters Thesis, AFIT/GCS/ENG/94-XX, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1994.

[GERH93] Gerhard, W., E., *Weapon System Integration for the AFIT Virtual Cockpit*, Masters Thesis, AFIT/GCS/ENG/93-XX, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1993.

[GIBS84] Gibson, William, *Neuromancer*, ACE Books, New York, NY, 1984.

[GOSS94] Gossweiler, Rich; Laferriere, Robert J.; Keller, Michael L.; Pausch, Randy, "An Introductory Tutorial for Developing Multi-User Virtual Environments," *PRESENCE: Teleoperators and Virtual Environments*, December 1994, 3 (4)., pages 255-264.

[GREE95] Greenhalgh, C. and Benford, S., "MASSIVE, A Collaborative Virtual Environment for Teleconferencing," *ACM Transactions on Computer, Human Interaction*, [Special Issue on Virtual Reality Software and Technology], Vol 2, Issue 3, p. 239- 261, 1995

[GREI88] Grief, I. (ed), *Computer Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann Publishers, San Mateo, CA, 1988.

[GUST95] Gustavson, Paul, "White Board PDU for Exercise Management and Feedback," *13th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, September 1995.

[HADD93] Haddix II, Rex G. *An Immersive Synthetic Environment for Observation and*

*Interaction with a Large Volume of Interest*, Masters Thesis, AFIT/GCS/ENG/93M-02, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1993.

[HAGS96] Hagsand, Olaf. "Interactive Multiuser VEs in the DIVE System," *IEEE MultiMedia,* 3 (1), p. 30-39, Spring 1996.

[HAHN89] Hahn, U., Jasrke, M., Kreplin, K., "CoAuthor: A Hypermedia Group Authoring Environment," *Proceedings of ECSCW'89*, Gatwick, UK, September 13-15, 1989.

[HART94] Hartman, Jed; and Creek, Patricia. *IRIS Performer Programming Guide*. Silicon Graphics, Inc. Mt. View, CA. 1994.

[HORT95] Horton, William, "Top Ten Blunders by Visual Designers," *Computer Graphics*, 29 (4), November 1995.

[IEEE93] Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 1278-1993, Standard for Information Technology, Protocols for Distributed Interactive Simulation, March 1993.

[ISDA93] Isdale, Jerry, "What Is Virtual Reality? A Homebrew Introduction," March 1993 ftp://sunee.uwaterloo.ca/pub/vr/documents/whatisvr.txt

[IST96] Standard for Distributed Interactive Simulation - Application Protocols, Version 2.1.3, IST-CR-96-11, Institute for Simulation and Training, September 1995.

[JACO93] Jacoby, R., Ellis, S., "Using virtual menus in a virtual environment," SIGGRAPH '93 Course Notes: Implementing Virtual Reality, course 43, 1993.

[JONE96] Interview with Michael Jones, Lead Developer, Silicon Graphics' Performer API, during SIGGRAPH'96, New orlean, LA, 8 August 1996.

[KEST94] Kesterman, Jim B. *Immersing the User in a Virtual Environment: The AFIT Information Pod Design and Implementation*, Masters Thesis, AFIT/GCS/ENG/94D-13, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1994.

[KUNZ94] Kunz, A., A., and Stytz, M., R., "A Virtual Environment for Satellite Modeling and Orbital Analysis in a Distributed Interactive Simulation," *Proceedings of the Military, Government and Aerospace Simulation Conference*, San Diego, CA., p. 55-60, 1994.

[LEE90] Lee, J., "SIBYL: A Tool for Managing Group Decision Rationale," *Proceedings of CSCW'90*, Los Angeles, CA, October 7-10, 1990.

[MACE94] Macedonia, Michael, Michael Zyda, David Pratt, Paul Barham, and Steven Zeswitz." NPSNET: A Network Software Architecture for Large-Scale Virtual Environments," Presence Volume 3, Number 4: 265-287 (Fall 1994).

[MALO88] Malone, T. W., Lai, K., "Object Lens : A Spreadsheet for Cooperative Work,"

*Proceedings of CSCW'88*, Portland, OR, September 1988.

[MAPE95] Mapes, D., Moshell, J., "A two-handed interface for object manipulation in virtual environments," *Presence*, 4 (4), Fall 1995.

[MART96] Martin, Alexander and Eastman, David, The User Interface Design Book, for the Applications Programmer, John Wiley & Sons, New York, NY, 1996.

[MCCA94] McCarty, W. D., et. al., "A Virtual Cockpit for a Distributed Interactive Simulation," *IEEE Computer Graphics and Applications,* p. 49-54, January 1994.

[MCGR84] McGrath, J. E., *Groups: Interaction and Performance*, Prentice Hall, Englewood Cliffs, New Jersey, 1984

[MCGR94] McGrath, J. E. and Hollingshead, A. B., *Groups Interacting with Technology*, Sage Publications Inc., 1994.

[MCLE92] McLendon, Patricia, *IRIS Performer Programming Guide*, Silicon Graphics, Inc., Mountain View, CA, 1992.

[MINE96] Mark Mine, "Virtual Environment Interaction Techniques," *Course Proceedings of SIGGRAPH'96,* Los Angeles, CA, 1996.

[MORN91]. Morningstar, Chip and Farmer, Randall F., "The Lessons of Lucasfilm's Habitat," *Cyberspace: First Steps*, Massachusetts Institute of Technology Press, p. 273-301, 1991.

[NUNA91] Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R., and George, J. F., "Electronic Meeting Systems to Support Group Work," *Communications of the ACM,* 34 (7), p. 40-61, July 1991.

[NRC95] National Research Council, *Virtual Reality: Scientific and Technological Challenges*, National Academy Press, Washington, D.C., 1995.

[PALM84] Palme, J., "COM/PortaCOM Conference System Design Goals and Principles," *Proceedings of INTERACT'84*, 1984 .

[PIME94] Pimentel, K., Blau, B., "Teaching Your System To Share," *IEEE Computer Graphics & Applications*, January 1994.

[PULL86] Pullinger, D. J., "Chit-Chat to Electronic Journals: Computer Concerencing Supports Scientific Communication," *IEEE Transactions on Professional Communications*, 29 (1), p23-29, March 1986.

[RIBA94] Ribarsky, W., et. al., "Visualization Analysis Using Virtual Reality," *IEEE Computer Graphics & Applications*, January 1994.

[ROBI92a] Robinett, W., Holloway, R., "Implementation of flying, scaling, and grabbing in virtual worlds," *ACM Computer Graphics: Proceedings of the 1992*

*Symposium on Interactive 3D Graphics*, Cambridge, MA, 1992.

[ROBI92b] Robinett, W., Holloway, R., " Implementation of flying, scaling, and grabbing in virtual worlds," *Proc. 1992 Symposium on Interactive 3D Graphics*, Cambridge MA, March, p.189-192, 1992.

[RODD89] Rodden, T., Sommerville, I., "Building Conversations Using Mailtrays," *Proceedings of ECSCW'89*, Gatwick, UK, September 13-15, 1989.

[RODD91a] Rodden, T., "A Survey of CSCW Systems," *Interacting with Computers*, 3 (3), p. 319-353, 1991.

[RODD91b] Rodden, T. and Blair, G., "CSCW and Distributed Systems : The Problem of Control," Proceedings of ECSCW'91, Amsterdam, 25-27 September, 1991.

[ROHR94] Rohrer, Jim J. *Design and Implementation of Tools to Increase User Control and Knowledge Elicitation in a Virtual Battlespace*, Masters Thesis, AFIT/GCS/ENG/94D-20, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1994.

[RUMB91] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall 1991.

[RYGO95] Rygol, M., et. al., "Tools and Metaphors for User Interaction in Virtual Environments", *Virtual Reality Applications* , Academic Press, pages 149-161, 1995.

[SHAW93] Shaw, C., Green, M., "The MR Peers Package," *VRAIS'93 Conference Proceedings*, p.463-469, 1993.

[SHEA92] Sheasby, S., M., *Management of SIMNET and DIS Entities in Synthetic Environments*, Masters Thesis, AFIT/GCS/ENG/92D-16, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1992.

[SHEP90] Sheperd, A., Mayer, N., Kuchinsky, A., "Strudel - An Extensible Electronic Conversation Toolkit," *Proceedings of CSCW'90*, Los Angeles, CA, October 7-10, 1990.

[SHER95] Sherman, W., Craig, A., "Literacy in Virtual Reality: a New Medium," *Computer Graphics*, 29 (4), November 1995.

[SILV94] Silverio, C.J., Fryer B., and Hartman, J., The OpenGL Porting Guide, Silicon Graphics, Inc., Document Number 007-1797-020, Mountain View, CA, 1994.

[SIMU95] Minutes of the Simulation Management Subgroup, *13th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, 20 September 1995.

[SNIB92] Snibbe, S.S., Herndon, K.P., Robbins, D.C., Conner, D.B. and van Dam, A., "Using

Deformations to Explore 3D Widget Design," *Computer Graphics (Proceedings of SIGGRAPH '92)*, ACM SIGGRAPH, pp. 351-352, July 1992.

[SNYD93] Snyder, Mark. *ObjectSim - A Reusable Object Oriented DIS Visual Simulation*, Masters Thesis, AFIT/GCS/ENG/93D-20, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.

[SOLT93] Soltz, Brian. *Graphical Tools for Situational Awareness Assistance for Large Battle Spaces*, Masters Thesis, AFIT/GCS/ENG/93D-21, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.

[SORG87] Sørgaard, P., "A Cooperative Work Perspective On Use and Development of Computer Artifacts," *Proceedings of the 10th Information Systems Research Seminar*, Scandinavia, 1987.

[STYT93] Stytz, Martin; Block, Elizabeth and Soltz, Brian. "Providing Situational Awareness Assistance to Users of Large-Scale, Dynamics, Complex Virtual Environments," *Presence*, 2 (4), p. 297-313, Fall 1993.

[STYT94] Stytz, M. R., et. al., "Providing Situational Awareness Assistance to Users of Large-Scale, Dynamic, Complex Virtual Environments," *Presence*, 2(4), Fall 93, 297-313.

[STYT95a] Stytz, Martin et al. "Portraying and Understanding Large-Scale Distributed Virtual Environments: Experience and Tentative Conclusions," *Presence*, Vol. 4, No. 2, p. 146-168, Spring 1995.

[STYT95b] Stytz, M., R., and E. Block, "Distributed Interactive Virtual Environments: Design, Implementation, and Experience," Air Force Institute of Technology, Wright-Patterson AFB, OH, 1995.

[STYT97] Stytz, Martin, Terry Adams, Brian Garcia, Steven Sheasby, and Brian Zurita, "Developments in Rapid Prototyping and software Architecture for Distributed Virtual Environments," *IEEE Software*, accepted for publication, not yet published.

[SWIT92] Switzer, J. C., *A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit*, Master's Thesis, AFIT/GCS/ENG/92D-17, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1992.

[TAKE92] Takemursa, H., Fumio, K., "Cooperative Work Environment Using Virtual Workspace," *Proceedings of CSCW'92*, Toronto, November 1992.

[VAND94] Vanderburgh, John C. *Space Modeler: An Expanded, Distributed, Virtual Environment for Space Visualization*, Masters Thesis, AFIT/GCS/ENG/94D-23, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1994.

[VDAM94] van Dam, A., "VR as a Forcing Function: Software Implications of a New Paradigm," *Proceedings of SIGGRAPH'94*, Orlando, FL, 1994.

[VV&A95] Minutes of the VV&A Subgroup, *13th DIS Workshop on Standards for the Interoperability of Distributed Simulations*, 18-22 September 1995.

[WEBE96] Weber, Hans, "How Do You Interact With A Virtual Environment?" *Course Proceedings of SIGGRAPH'96*, Los Angeles, 1996.

[WEXE93]. Wexelblat, Alan, "The Reality of Cooperation: Virtual Reality and CSCW," *Virtual Reality: Applications and Explorations*, p. 23-44, Academic Press, Inc, 1993.

[WILB88] Wilbur, S. B., Young, R. E., "The COSMOS Project : A Multi-Disciplinary Approach to Design for Computer Supported Group Working," *Proceedings of EUTECO'88*, Vienna, Austria, April 20-22, 1988.

[WILS93] Wilson, Kirk G. *Synthetic BattleBridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment*, Masters Thesis, AFIT/GCS/ENG/93D-26, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1993.

[WLOK95] Wloka, M. and Greenfield, E., "The Virtual Tricorder: A Uniform Interface for Virtual Reality," *Proceedings of UIST'95*, ACM Press, pp. 39-40, November 1995.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1996 | Master's Thesis |

**4. TITLE AND SUBTITLE**
COLLABORATIVE WORKSPACES WITHIN DISTRIBUTED VIRTUAL ENVIRONMENTS

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
William David Wells, Captain, USAF

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

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT/GCS/ENG/96D-28

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**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)*

In warfare, be it a training simulation or actual combat, a commander's time is one of the most valuable and fleeting resources of a military unit. Thus, it is natural for a unit to have a plethora of personnel to analyze and filter information to the decision-maker. This dynamic exchange of ideas between analyst and commander is currently not available within the distributed interactive simulation (DIS) community. This lack of exchange limits the usefulness of the DIS experience to the commander and his troops. This thesis addresses the commander's isolation problem through the integration of a collaborative workspace within AFIT's Synthetic BattleBridge (SBB) as a technique to improve situational awareness. The SBB's Collaborative Workspace enhances battlespace awareness through CSCW (computer supported cooperative work) enabling communication technologies. The SBB's Collaborative Workspace allows the user to interact with other SBB users through the transmission and reception of public bulletins, private email, real-time chat sessions, shared viewpoints, shared video, and shared annotations to the virtual environment. Collaborative communication between SBB occurs through the use of standard and experimental DIS-compliant protocol data units. The SBB's Collaborative Workspace gives the battlespace commander the widest range of communication options available within a DIS virtual environment today.

**14. SUBJECT TERMS**
Modeling, Simulation, Virtual Environments, Distributed Interactive Simulation (DIS), Collaborative Computing, Computer Supported Cooperative Work (CSCW), Collaborative Workspaces

**15. NUMBER OF PAGES**
152

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

**Block 1.** Agency Use Only *(Leave blank)*.

**Block 2.** Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

**Block 3.** Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

**Block 4.** Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

**Block 5.** Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | | | |
|---|---|---|---|
| C | - Contract | PR | - Project |
| G | - Grant | TA | - Task |
| PE | - Program Element | WU | - Work Unit Accession No. |

**Block 6.** Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

**Block 7.** Performing Organization Name(s) and Address(es). Self-explanatory.

**Block 8.** Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

**Block 9.** Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

**Block 10.** Sponsoring/Monitoring Agency Report Number. *(If known)*

**Block 11.** Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

**Block 12a.** Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- DOE - See authorities.
- NASA - See Handbook NHB 2200.2.
- NTIS - Leave blank.

**Block 12b.** Distribution Code.

- DOD - Leave blank.
- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- NASA - Leave blank.
- NTIS - Leave blank.

**Block 13.** Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

**Block 14.** Subject Terms. Keywords or phrases identifying major subjects in the report.

**Block 15.** Number of Pages. Enter the total number of pages.

**Block 16.** Price Code. Enter appropriate price code *(NTIS only)*.

**Blocks 17. - 19.** Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

**Block 20.** Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.