Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

12-1993

ObjectSim - A Reusable Object Oriented DIS Visual Simulation

Mark I. Snyder

Follow this and additional works at: https://scholar.afit.edu/etd

Part of the Graphics and Human Computer Interfaces Commons

Recommended Citation

Snyder, Mark I., "ObjectSim - A Reusable Object Oriented DIS Visual Simulation" (1993). *Theses and Dissertations*. 6680.
https://scholar.afit.edu/etd/6680

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.

AD-A274 030

AFIT/GCS/ENG/93D-20

1

DTIC
ELECTE
DEC 23 1993
S A

ObjectSim - A Reusable Object Oriented DIS Visual Simulation

THESIS

Mark I. Snyder, Captain, USAF

AFIT/GCS/ENG/93D-20

93-30989

Approved for public release, distribution unlimited

93 12 22 102

AFIT/GCS/ENG/93D-20

ObjectSim - A Reusable Object Oriented DIS Visual Simulation

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

Mark I. Snyder, B.S.
Captain, USAF

December 1993

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

DTIC QUALITY INSPECTED 3

Approved for public release, distribution unlimited

Preface

This research designed and implemented a reusable model for Distributed Interactive Simulation (DIS) visual simulations on Silicon Graphics machines using the Iris Performer library. The model was implemented in C++ using extension by inheritance to provide a customizable simulation architecture. The model was applied successfully to four simulation projects to reduce their development cycle time and increase their capability.

The visual simulations implemented using the architecture met many goals for capability, adaptability, and maintability. The student developers using the architecture all reported good productivity due to the object oriented nature of the architecture. Since this research has the potential for creating new simulation applications quickly, it should be a focus for future research efforts aimed at bettering the capability of DIS visual simulations.

I owe many thanks for the help and encouragement I received in doing this research. I would like to thank Lt Col Pat Lawlis for letting me have the freedom to explore my ideas and for helping me in gaining acceptance for them. Also, she helped keep my research focused and on track. I thank Lt Cols Phil Amburn and Martin Stytz for trusting in the promise of object oriented design and reuse, and trusting me with the core of their simulation projects. I thank each one of the student developers, Maj Mike Gardner, Capt Matt Erichsen, Capt Bill Gerhard, Capt Brian Soltz, Capt Kirk Wilson, and Capt Andrea Kunz for trusting their projects to an unproven idea and putting up with configuration woes. I also received much help from Mike Jones of Silicon Graphics, who always had the answer for my Performer questions. Finally, I thank my wife Roseanne and Jonathan and Timothy for their patiance and understanding when I had to work and couldn't be with them.

Mark I. Snyder

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/93D-20

Abstract

This research designed and implemented a reusable Distributed Interactive Simulation (DIS) visual simulation architecture for Silicon Graphics platforms. The goal was to research software architecture technologies and to create a design and implementation using these ideas. The architecture was designed using object oriented techniques to provide the ability to customize it via inheritance extension. The resulting design was implemented using C++ and applied to several DIS visual simulation projects in the Graphics Lab at AFIT. The architecture, named ObjectSim, was successful in its goal of providing a reusable core for the DIS visual simulation projects in the Graphics Lab at AFIT. It provides simulation developers reusable capabilities in the areas of rendering , data display, device interfacing, and DIS network interfacing. The projects designed and implemented with ObjectSim exceeded their research goals. Data on reuse effectiveness and several different performance areas was collected.

I Introduction

1.1 Overview

This research investigated methods of improving the process of creating visual simulations. I designed a reusable software analysis and design model on which to base current and future visual simulations, and implemented this model using the C++ language. This research used the latest high performance software and hardware tools from Silicon Graphics, Inc. to aid in the implementation. The results of this research, in the form of reusable analyses, designs, and software, are a basis for future work in visual simulation.

1.2 Research Motivation

The Air Force Institute of Technology (AFIT) is involved in research to help determine military uses for rapidly evolving visual simulation technology. A visual simulation is a computer generated environment that allows a human participant in a simulated reality to see objects in the simulation as if the participant were actually there. Examples include flight simulation, a simulated view from a tank or a ship, or a simulated automobile. In the Graphics Lab at AFIT, several simulations exist or are being developed which use this technology (Figure 1). Since the lab is working on so many similar projects, there are many benefits to be gained through software reuse.

Though reuse has been applied to the efforts in the Graphics Lab in the past, no methodology or process existed to organize or foster reuse of past work in a standard way. Since thesis students have limited time to become experts in graphics programming, the more past effort they can reuse, the more they accomplish in their research. These two factors led to the formulation of a small research group to work specifically in the Graphics Lab to apply software engineering methods. The goal was to improve the software designs, software maintainability, and student productivity in the simulation research efforts . This research was a part of that group's effort.

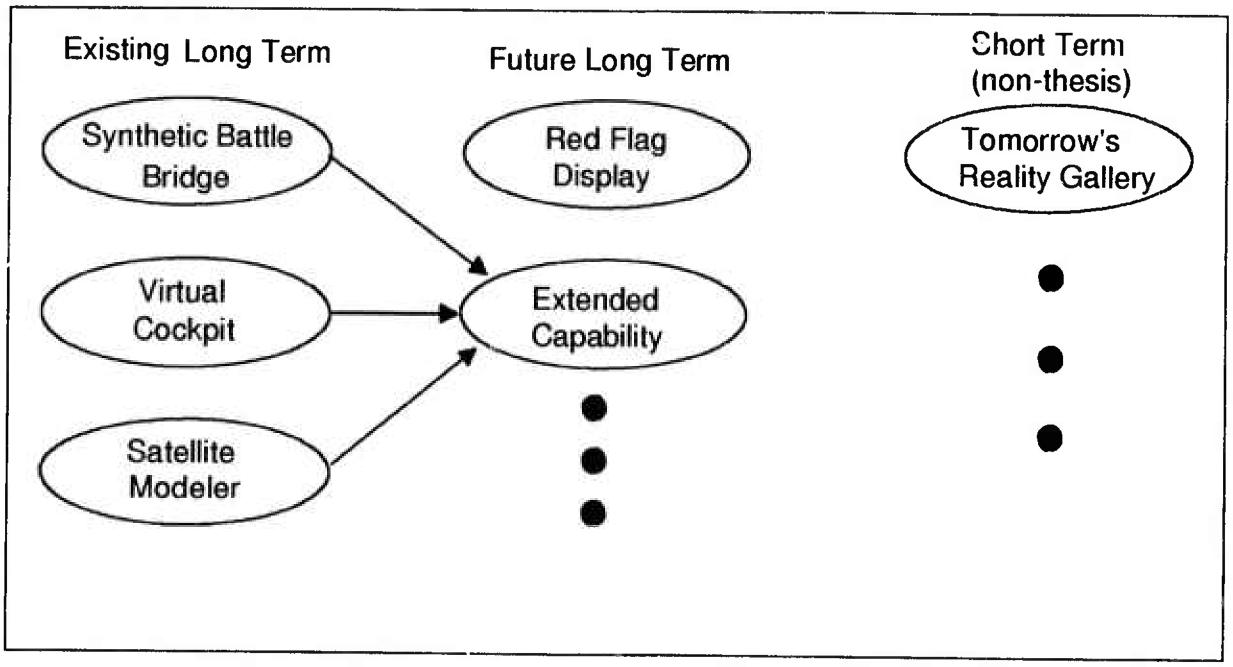


Figure 1 - AFIT Visual Simulation Projects

Creating a visual simulation has traditionally been a lot of work. These simulations require complex software to be written to manage graphics drawing (*rendering*), as well as to simulate vehicle dynamics, or moving objects in the scene. Also, software to communicate with various input and display devices is becoming increasingly complex with the introduction of head mounted displays, voice recognition, data gloves, and other user communication devices. As simulators begin to interoperate over networks, software to manage this communication must be written. There are many needs for software in the simulation realm. It is important to make an effort to capture past work, apply as much reuse as possible, and work smartly. This thesis covers four areas where reuse is important in visual simulation: rendering, data display, device/user interfaces, and network interface software.

1.3 Rendering Overview

In the past, some efforts have been made to reuse rendering software in the Graphics Lab (Bru91:Chs 1-5). In this thesis, rendering will refer to the process of showing the world to the simulation participant in the most realistic way possible. The rendering reuse efforts attempted in past years had some success. In fact, much of the existing code in the lab to implement rendering in visual simulation was based on a suite of code and tools called GDMS (Graphical Database Management System). However, new software methods to accomplish rendering (SG92) and building three dimensional models (SS93) created the need for more work in this area. Much of the impetus for this effort came from improvements in vendor software which alter the way simulations are developed (SG92:Ch 1). The Iris Performer software library provides a powerful way to create simulation software for Silicon Graphics platforms. Major software improvements in three-dimensional modeling software have also changed the way simulations are created (SS93:Ch 1). Past reuse libraries in the Graphics Lab have been tailored to much less capable rendering and modeling tools than now exist. Figure 2 is an illustration of the past process.

In this past year, the Graphics Lab at AFIT has begun to use a popular tool called Multigen and its associated model format to create models and scene databases. Multigen is one of a new generation of tools for creating three dimensional polygonal models and building scene databases from them (SS93:Ch 1) A *scene database* refers to a collection of three dimensional models that form a scene in the simulation. An entire scene database can now be constructed using Multigen. This is a departure from past methods used in the Graphics Lab. Multigen's "Flight" format for three dimensional models is an emerging standard for storing object representations.

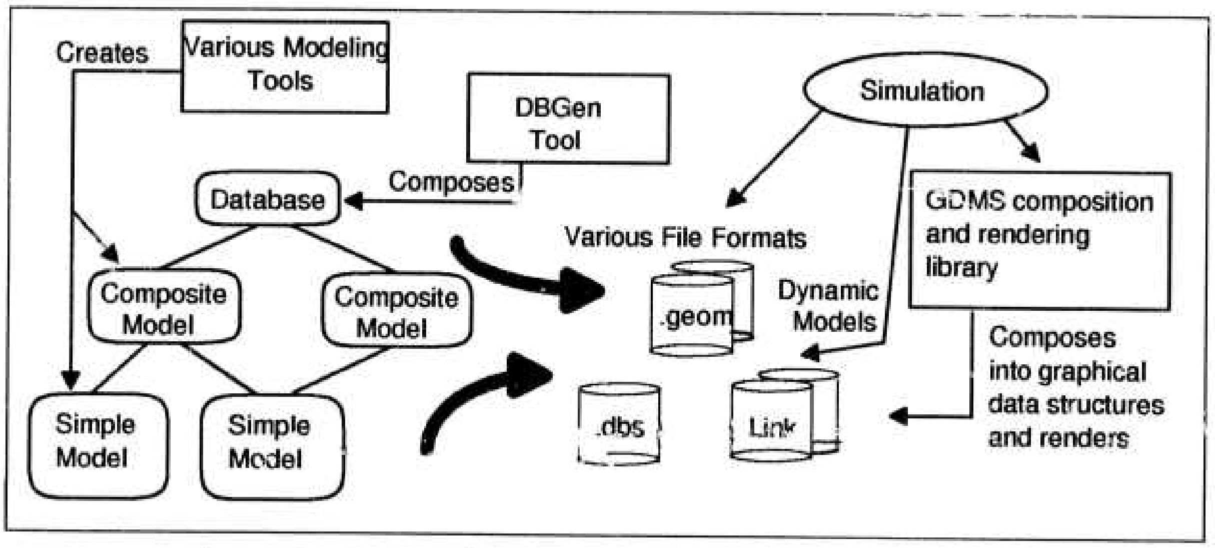


Figure 2 - Developing a Simulation with Past Tools

Since the release of Multigen, the vendors at Silicon Graphics have released their Performer software library, which is a collection of high level C language routines to use the Iris workstation efficiently. (SG92:Ch 1) This library is also well suited for using the databases created by Multigen in visual simulations. Together, these advanced software products define a new way to create and populate visual simulations for the Silicon Graphics workstations. The simulation development process with these tools is illustrated in figure 3.

The newer method of creating the simulation involves more work with the Multigen tool to create the database hierarchy, and a corresponding reduction of complexity in the rendering and database management side of the software. Other than the Performer library of C routines, no software was available in the Graphics Lab which could be easily reused for application building with these new techniques. Since the Performer library offers no class based C++ interface, it did not provide enough of a standard interface to support commonalty of design for software using the library. Also, Performer still leaves the simulation design up to the application developer. To get the maximum reuse from Graphics Lab software development efforts, the lab needed to use common designs as much as possible. This thesis describes the *application framework* approach I applied to facilitate a common design for the simulations.

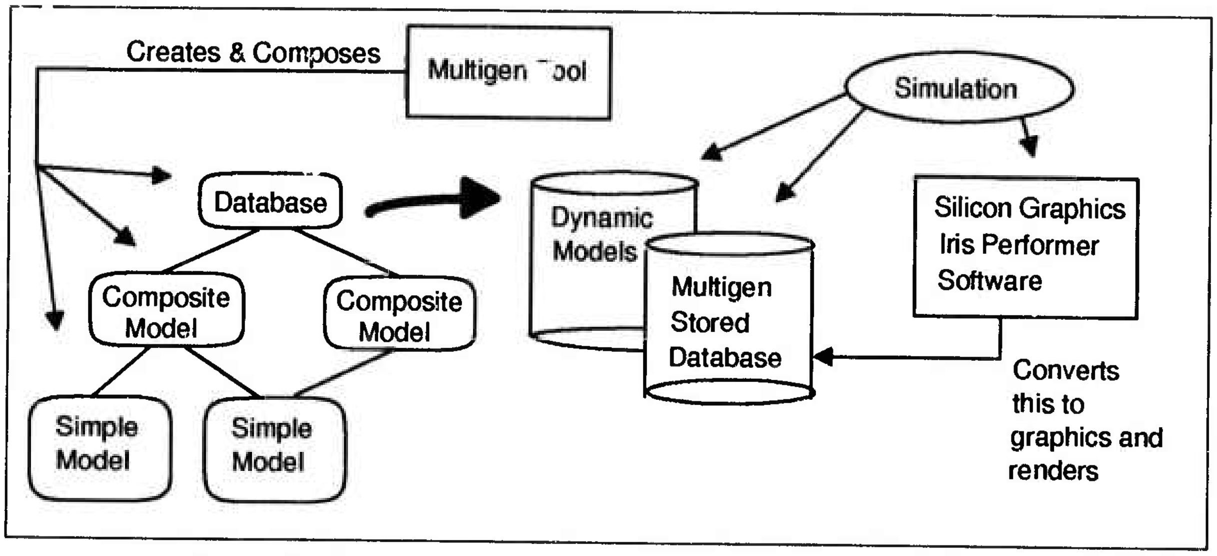


Figure 3 - Developing a Simulation with New Tools

1.4 Data Display Overview

A second major area of work for visual simulation developers is software to display information to the participant. Displays report information on the status of the simulation back to the simulation user. They include instruments in a cockpit, position, speed and orientation displays for a fly-through type simulation, data displays showing information about entities in the simulation, and any other visual element used to report simulation status to a user. They are sometimes a part of the scene database, but often are implemented separately because of their complex textual and graphical requirements. Display requirements can comprise a substantial portion of the work involved in writing visual simulation software. This makes them an ideal candidate for software reuse. Figure 4 shows some typical display requirements for a visual simulation.

In a virtual reality environment, where the simulation user may wear a head mounted display, this problem of reporting status is difficult to solve. Consider the problem of displaying cockpit instruments to a pilot of a simulated cockpit wearing a head mounted display. As the pilot turns her head, the instruments should not turn with her, but should remain in their correct location relative to her. One solution is to render the instruments in the scene database as dynamic simulated objects, as discussed above. This solution tends to introduce too much complexity into the scene database to allow for fast screen updates. This problem is typical of the requirements for immersive simulations, which place the system user into the scene, often through the use of head-mounted displays or other non-screen based methods.

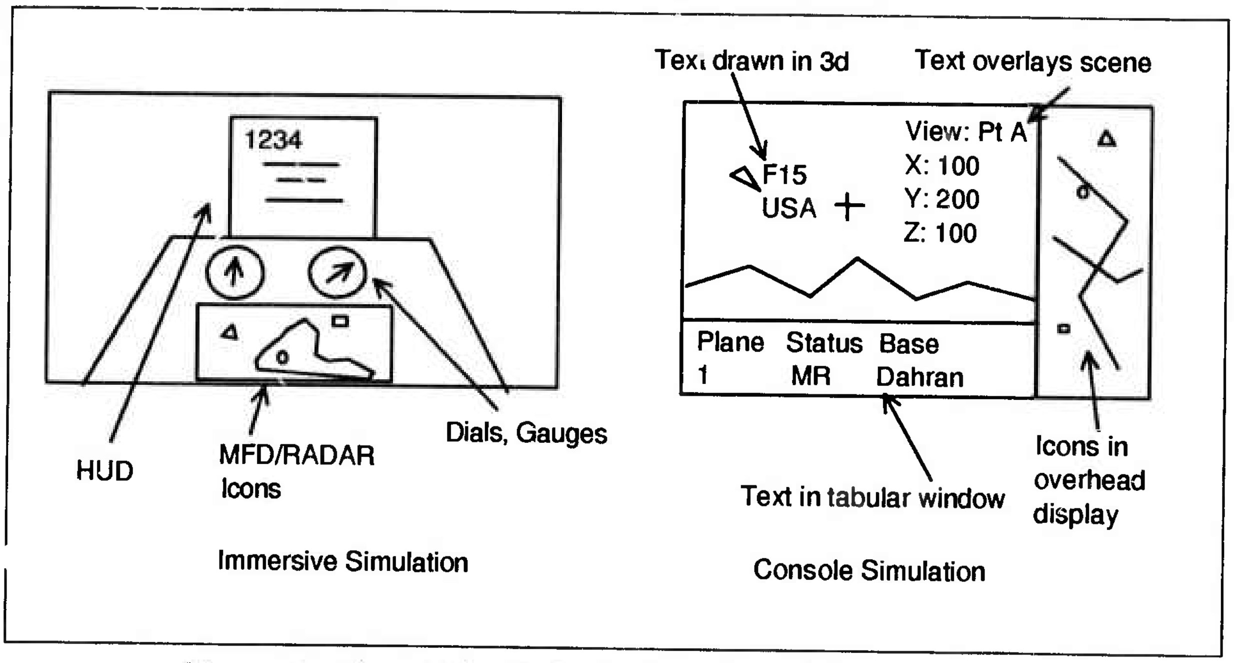


Figure 4 - Visual Simulation Display Requirements

Another problem in an immersive simulation is the necessity to simulate real, complex displays. A modern weapons systems has a very graphically oriented interface. To properly create any man-in-the-loop simulator requires the ability to display status to the weapons system operator in a reasonable, usable, and believable format. Display requirements will quickly escalate. A toolkit of reusable display components, comprising both rendering and a useful interface to the underlying simulation, is almost a necessity for much serious work in this area.

For screen based simulations (*console* simulations), the display requirements are also heavy. Console simulations typically display information to the user using common techniques such as tabular displays or displays overlaying the graphics in a scene. Data may appear in tabular format, or in an overhead view with icons. These display techniques are not commonly used with immersive simulations, but are more common to such familiar systems as command and control software. Typically, console simulations require extensive window management and data presentation software to be written. This is another area to concentrate on when considering a reusable solution for simulations.

1.5 Device/User Interface Overview

User input comprises a third major area of software efforts for visual simulation. User inputs include devices such as head tracker, spaceball, mouse, and keyboard. They may also involve traditional user interface areas such as buttons, sliders, etc. Past thesis efforts have created some reusable code for device interfaces, and much of this is in use in the lab. Figure 5 shows some representative user interface requirements for visual simulation.

For console simulations, user input may come from spaceball, mouse, or, commonly, graphical user interface (GUI) toolkits such as Motif. This means a robust simulation design must allow for many different interfacing methods without breaking the design. Common problems here include user event handling approaches, windowing, and performance impacts from 'expensive' GUI calls and functions.

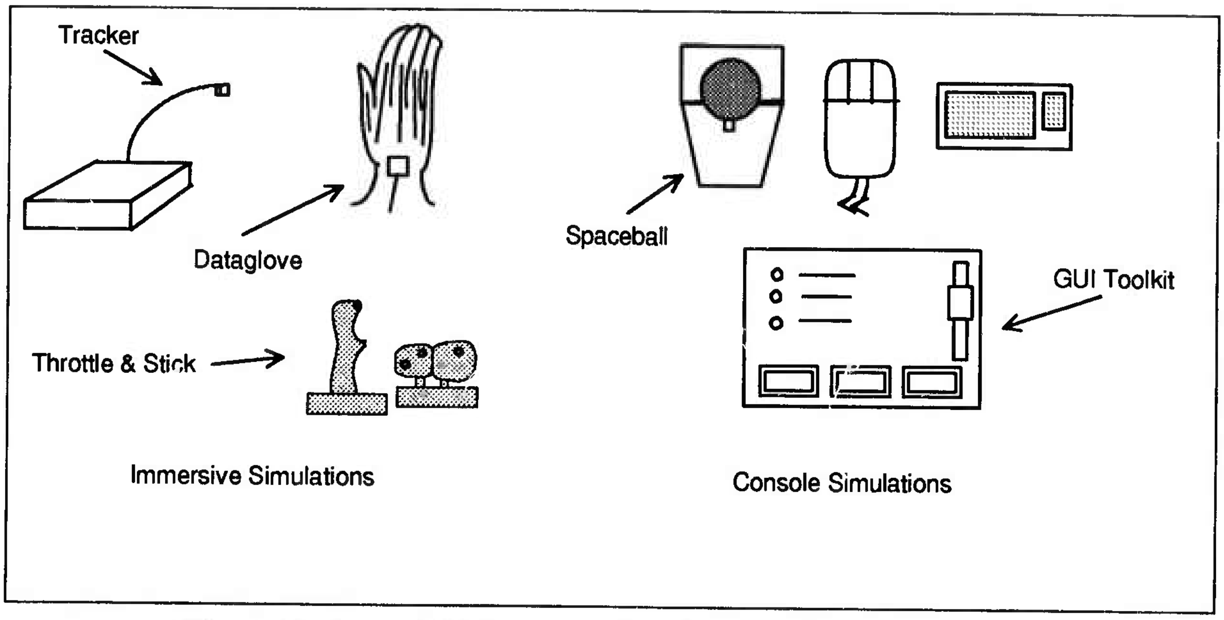


Figure 5 - Visual Simulation User Interface Requirements

For immersive simulations, the common interfacing techniques change dramatically. Since an immersive environment attempts to give the user as realistic an experience as practical, common techniques used to bridge the gap between human and computer are not enough. Typical interface challenges for these simulations include data gloves, voice commands, head tracking, boom devices, and platform specific interfaces, such as throttle and stick. Again, a robust design and reuse can reduce the software workload significantly.

1.6 Network Interface Software Overview

The *network interface* for a visual simulation refers to the software required for the simulator to function with other simulators over a network. For the Graphics Lab work, the simulators are designed to send and/or receive the Distributed Interactive Simulation (DIS) protocol, which is a standard network interface format used heavily within the Department of Defense. The network interface for a simulation has two main components, send and receive. The send portion of the interface formats information about the sending simulator into *protocol data units* (PDUs) which are sent out over a network. The receive portion of the interface reads these PDUs from the network and displays the entities and/or information about them in the simulation.

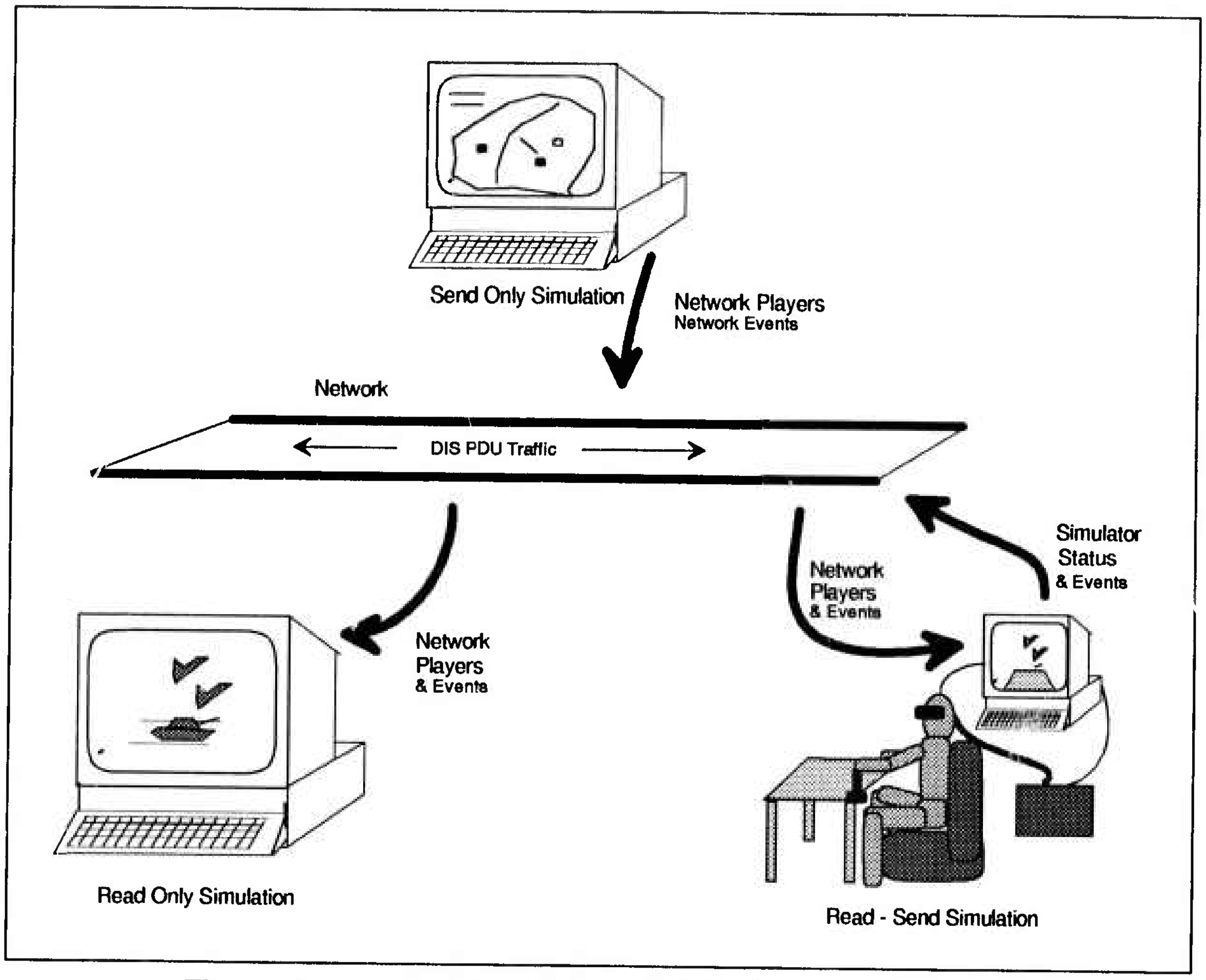


Figure 6 - Network Interface for DIS Simulators

A successful and efficient network interface comprises many different problem solutions. A network interface will have to manage coordinate conversion from network standard coordinates to simulation coordinates, maintaining fast network read rates, maintaining smooth entity appearance when PDUs are not continuous, maintaining a large database of entity 3D models, and sending/receiving events (e.g.; detonations). This research investigated methods to integrate these various tasks into an understandable, object oriented network interface capability for DIS simulations.

1.7 Research Overview

In this research, I have applied software engineering and reuse to the visual simulations in the AFIT Graphics Lab. I analyzed the requirements for visual simulation in the areas of rendering, displays, device interfaces, and network interface as a basis for the reusable design model I created. I created corresponding implementations using C++ to provide a basis for several visual simulation research projects. I also designed and implemented useful reusable software pieces to fit into this design, and supervised their application across multiple visual simulation projects. I collected a set of metrics on performance and reuse effectiveness to validate my approach.

1.8 Process

There are many powerful tools available to aid in creating simulations, but their introduction into the Graphics Lab at AFIT must be managed. Tools introduced too quickly, or without a thorough understanding of their power and intended use, may not help software development. Also, much of the work in the Graphics Lab is based on some prior reuse efforts, and may need to be migrated to use the new software and models. Migrating to use new technologies while not having to start over is one important challenge of this research and of the graphics lab software engineering effort.

A development environment must allow software developers within it to reach their goal. The environment's success is defined by how well it facilitates those goals. For the Graphics Lab visual simulations, these goals are influenced by two major factors: the parent project and the transience of the developers (students).

The research sponsor gives direction, goals, and funding to the Graphics Lab to support an ongoing effort in visual simulation. This large scale simulation project involves a concept known as Distributed Interactive Simulation (DIS). DIS allows multiple simulators in geographically separate locations to interact and wargame with each other. The Warbreaker program is an effort to apply DIS to a multifaceted battlefield environment to research training and weapons system effectiveness by using multiple, independently developed simulation projects. AFIT is developing several simulation research efforts as a part of the Warbreaker program. Because of the strong sponsorship provided for this research, the resulting simulations are expected to meet goals for performance, capability, and extendibility. Also, they need to be incrementally developed to allow demonstrations and a fast feedback loop for refinements.

In the Graphics Lab, the research is performed by students working on a Master's thesis, largely during a single year. This means each student has a limited amount of time to actually make a research contribution and produce or enhance a simulation. Also, students typically learn only the basics of the programming language used during their thesis cycle. As a result, the language is misused and underused in much of the existing work. The development environment should aid new student developers in learning quickly and in performing quick design and implementation of their projects.

1.9 Scope

This research is more than just conceptual. This means an important criterion for success is that the resulting software contributes significant time savings to the work in the Graphics Lab. The research in the Graphics Lab is sponsored and has specific goals it must reach, as discussed above. Any software engineering concepts used must immediately contribute to the construction of a working simulation, or risk losing acceptance by the simulation developers and sponsors.

To exploit the new performance and modeling software now available, I created an *application template*, or reusable design, for visual simulations created with the new software technology. This software provides an abstract method for writing new simulation software, with a clearly defined programmer interface to enable new simulations to be written. This work builds on previous reuse efforts in the Graphics Lab, while extending them to encompass newer technologies like Performer and Multigen.

I used ideas from Model Based Software Engineering (DIp89:1) to study the effectiveness of these concepts in the Graphics Lab. The application framework, called ObjectSim, provided an implemented version of an abstract meta-model (reusable system model) for the particular simulation environment I am working in. These concepts are covered more in later sections.

I also researched methods of *parameterizing* the reusable display and simulation software. Parameterizing means providing information to software to allow it to configure itself to solve a problem. I used methods of parameterization to provide a flexible enough simulation framework to encompass diverse simulation requirements. I studied the effectiveness, efficiency, and ease of use of some various parameterization methods in the context of working with developers using ObjectSim.

A major goal of this research was to leave the school with a toolkit it can use to create simulations and add capability to them. The goal was to improve development time, maintainability, and overall design for the simulations in the lab. To that end, I have made several attempts to pass the knowledge I have about customizing, adding to, and using ObjectSim. These include the ObjectSim Application Developers Manual (Appendix A) and several example programs to help get developers started (Appendix B).

The ObjectSim design is a reusable analysis model for visual simulations. The initial implementation was targeted for Silicon Graphics machines. Both hardware and toolkits will change over time. There are several approaches I discuss for migrating work and removing machine dependencies.

1.10 Research Approach

To design and write good reusable systems, the first step is a well thought out design. I used the Rumbaugh Object Modeling technique to capture the analysis and design steps of the Software Engineering process. In addition, I used some of the concepts of designing software based on models outlined by D'Ippolito(DIp89:3) to construct models of the simulation application template and other software that can be instantiated to build simulations. *Model* in this sense refers to a method to describe the behavior and structure of the system, not a three-dimensional model. Instantiating a model means filling in the specifics needed to define a solution to a particular problem. The ObjectSim design model evolved as the system was built, and reflects the structure of the completed software.

Since the Graphics Lab uses C and C++ for it's current software, I used C++ as the implementation language for the software. This provided some advantages. I used the C++ features which support objects, data abstraction, and inheritance to make as direct a translation from the application model into executable software as I could. I concentrated on preserving data abstraction with the language, since managed complexity is an important key to creating reusable software that is accepted by future users.

A large part of this research involved searching out technologies and existing solutions that can be used in the Graphics Lab. An example is the Silicon Graphics Performer library. Another area of research was working with the Wright Lab Avionics Division to reuse some of their flight simulator software on our Virtual Cockpit project. I also surveyed past work from the Graphics Lab to identify reusable software.

Since work has been done commercially and within other government and academic circles, the Graphics Lab naturally stands to benefit if it can profitably use this work.. The Graphics Lab software engineering team, as the reuse and software engineering element within the lab, was responsible for managing tool and technology introduction into the lab. In this sense, the group functioned as a *tools group*. Tools groups operate in a software development environment to facilitate technology introduction, create advanced designs and test them, and perform other advanced engineering tasks. Since tools groups are only loosely tied to a particular project schedule, they can "blue sky", or investigate technology too risky to incorporate in an important development before being validated. For this research, a testbed application was developed and maintained to develop and validate new capability without perturbing existing simulation projects.

Finally, the team attempted to foster the use of some basic software engineering concepts throughout the Graphics Lab. This included working from designs, reviewing and sharing each other's work, and the aforementioned tools group to foster technology introduction and advanced designs with broad applications.

1.11 Research Goals

This research studied the effectiveness of parameterized software architectures on the creation and maintenance of visual simulations. The result was a set of reusable software models, designs, C++ classes, and sample applications that provide the basis for current and future Graphics Lab work in visual simulation. These reusable pieces should contain components for a software toolkit to support rendering, displays, devices, and network interface in visual simulation. Students using a reusable design like this should be able to begin writing simulation software in much less time than before. Since students are only at AFIT a relatively short time, tools like this should really help more to get accomplished in the lab. Also, this research provides a good case-study analysis of the parameterization methods discussed above and their effectiveness in visual simulation software.

The practical goal for this research in the Graphics Lab was to provide the tools necessary for entry into a new stage in its simulation development work. By providing a high level toolkit designed for an environment such as this, researchers were able to focus on the advances their research provides, rather than detail work involved in solving routine simulation software problems.

1.12 Support

The support environment for this thesis was already in place. The lab has the machines, software, manuals, and compilers to provide enough tools to accomplish the tool building portion of the research. I had good access to Silicon Graphics vendor software and manual support, which the lab has. I also had access to several contacts who have experience in the latest software tools I used.

To support the overall goals of the Graphics Lab and the tools group, a form of integrated class library and configuration management was put into use to provide a common set of development tools for the lab. The tools group had the support of the research faculty and the acceptance of the other students, which aided greatly in accomplishing our goals.

1.13 Measurement and Validation

I studied the efficiency, maintainability, and usability of the ObjectSim framework from several perspectives. These include performance characteristics, storage and memory characteristics, usability studies and examples, code size and complexity comparisons with similar projects, and maintainability of the ObjectSim projects. The goal was a set of metrics which showed the effectiveness of this approach.

Any work in real-time simulation is performance bound. Often, this is viewed a drawback to using structured programming, object oriented programming, or any other 'inefficient' programming techniques. One intent of my measurements was to show the impact of using advanced C++ features like inheritance on simulation performance.

The more difficult aspects of the measurements are the benefits of our object-oriented approach. I have included development metrics and reuse data to attempt to show the benefits of the application framework on simulation development. These metrics are collected from four different thesis research projects which used ObjectSim as the basis for their design.

1.14 Document Overview

Chapter 2 of this thesis is a review of literature and industry practice, as it relates to this research. Chapter 3 reports on the initial analysis and background work I did to prepare for the main research. Chapter 4 presents the design of ObjectSim in depth and development notes for the different pieces. The emphasis is on the meta-model which ObjectSim captures and its aspects, dependencies, etc. Chapter 5 presents some sample instantiations of the model, including the Virtual Cockpit, Remote Debriefing Tool, and Satellite Modeller. Chapter 6 is a record of the validation metrics collected and a discussion of performance issues in visual simulation. Chapter 7 is a summary of the research accomplishments, along with recommendations for follow-on and future development of the architecture and its instantiations.

II Current Research Overview

2.1 Overview

This chapter will explore current research in the area of reusable architectures for building domain specific software systems. The study is motivated by the current effort in the AFIT Graphics Lab to design and implement a reusable architecture to support visual simulation. A *reusable architecture* is a reusable framework for designing and implementing a software system (Lowry91:647). *Domain specific* refers to the notion that many systems designed to automate or speed software development limit the class of problems they solve. For instance, a domain might be command and control systems, or banking systems. This notion of a reusable design has also been referred to as a *meta-model,* or reusable design model.

Reusable architectures provide an attractive method for the introduction of software engineering into new environments. This is because they help the system developers by giving them a reusable, customizable design for a class of problems, rather than a methodology or complex tools to master. This can mean relatively quick success for developers using this approach. Also, many developments can benefit from the work done on one reusable project.

For real-time interactive simulation systems, such as those created in the AFIT Graphics Lab, performance is an important criterion for the finished software. Any software engineering method applied in this domain must not significantly slow down the finished product. For this reason, I will include a look at some current literature about C++ and object oriented design and programming, since the majority of visual simulation software is written in C or C++ for performance reasons. Also, the C++ language gives a direct implementation of object oriented design, which is a useful method for constructing a reusable design model. Other object oriented languages, notably Ada9X, are also candidates for implementing an object oriented simulation design, provided their performance can be shown to be adequate for the high frame rates desired.

2.2 Method

I have explored the current state of the art for reusable systems first from the perspective of current academic papers and textbooks. This will provide some theoretical underpinnings for the research and explore some current efforts through a spectrum of software engineering topics related to this work.

I have also explored the state of reuse in current industry practice in this Chapter. This will be based on my own experiences in development environments, and some other references including periodicals and books. Because of the nature of this research (fusing advanced software engineering techniques with real developments), I feel a look at current practices of software professionals is also beneficial.

2.3 Academic Literature Reviews

Lowry (Lowry91) presents software architectures alongside many other current research areas in software engineering. He presents a likely path of evolution for software architectures, domain analysis, and other forms of automating the software process (Lowry91:646-650). This path includes various development paradigms software designers will follow as technology improves to encompass such methods as automated software design and code generation and expert system support for the software development process. He defines *megaprogramming* as the process of developing software with very high level components, and indicates that future system developers will use increasingly larger and more abstract components to build their systems (Lowry91:646). He also points out that late binding, a feature found in true object oriented programming languages (Str91:36), increases reusability but decreases efficiency of a software component. He writes that the late binding features of artificial intelligence languages are the basis for many current systems which support large scale reuse. These systems can be used for system prototyping, but do not generate efficient production software (Lowry91:648). He indicates that, in the future, software development systems will provide the ability to transform these inefficient but highly reusable systems into efficient programs via methods such as program synthesis (Lowry91:649).

Lowry also discusses two related notions which come closer to describing current industry efforts to create efficient, reusable software. *Application generators* use simple techniques, such as filling in software coding templates, to enable system developers to reuse designs and implementation (Lowry91:637). *Software architectures* are much more advanced. This area of research includes such topics as domain or application specific programming languages, automated transformations to produce efficient code, and encapsulating domain knowledge for use by designers (Lowry91:647).

Bhansali and Nii (Bha92) describe generic software architectures within the context of their Knowledge Assisted Software Engineering (KASE) system. KASE is based on the concept of *meta-models*, or reusable designs (Bha92:3). Some set of these reusable designs are stored in KASE, which allows the designer to pick an architecture matching his problem description and step through the design process to create a specific design for his problem.

The authors propose a formal description of a generic software architecture as a collection of modules which require customization to be useful in a specific solution. (Bha92:9). These modules, before being customized, represent a partial set of design decisions usable across the target problem domain (Bha92:10). These modules appear to be analogous to objects in an object-oriented design. Customizing the modules completely represents all of the design steps for the specific application, and can be accomplished with the KASE tool.

Wiederhold, Wegner, and Ceri (Wie92) describe a method for accomplishing *megaprogramming*, or very large component based reuse. They present a language which could be used to describe large component based reuse (Wie92:1), and show an example of this language in use. They describe *megamodules* as providers of services, autonomously operated, which encapsulate data, behavior, and knowledge (Wie92:2). As such, megamodule implementation and details are unimportant, as long as they can be accessed to provide their services. This is similar to current usage of such system components as databases, display toolkits, and operating systems.

In essence, the authors megaprogramming language generalizes the concepts behind these client-provider entities to arbitrary modules performing arbitrary functions. Since data will be communicated across these various megamodules, the megaprogramming language provides constructs to facilitate data conversion, or *transduction*, which migrates data from a form acceptable to a source megamodule to a form acceptable to a destination megamodule (Wie92:9).

These megaprogramming notions are a basis for the future reuse of large scale systems (Wie92:1). They have a direct application to software architectures because they describe standard methods for creating and using client-provider reusable components. A software design for a system of any size or usefulness will include the use of megamodule-like entities such as display toolkits and databases.

Stroustrup (Str91) examines the issues involved in creating software libraries in the C++ programming language. Class libraries store reusable code for use by other programs. They are the most common means of providing design reuse (software architecture) in C++ developments. Stroustrup, the designer of the C++ language, presents his ideas on the best method for using C++ to provide code libraries. These ideas include certain design guidance, such as language constructs to avoid, for writing C++ programs (Str91:370-375).

Stroustrup also describes his views of the software development process and how it relates to C++. He reiterates that the power of C++ for translating object oriented designs into executable code does not reduce the need for proper requirements specification or a thorough design process (Str91:365) According to Stroustrup, many software developments using C++ do not gain any from the power of the language, because the advanced features like inheritance and dynamic binding are wasted in confusing, poor, or inefficient designs (Stroustrup,1991:362).

D'Ippolito (DIp89) takes a different approach to the concept of reusable designs in software. He views the software development process as an instance of traditional engineering, and applies the concept of *models*, or abstract behavioral representations, to the software development problem (DIp89:3). His paper describes the benefits of designing software modules and systems as models to be instantiated for applications which need them, much as an engineer will select a power supply for an electronic component from available sources, rather than designing one himself (DIp89:3).

D'Ippolito points out that this model based view encompasses many of our other notions of how we reuse software, such as parameterized modules, domain specific architectures, code libraries, and client-provider relationships (DIp89:3-5). These are all activities intended to capture knowledge in a reusable form and apply it to other problems which can use it. This process of capturing knowledge, or science, has been performed in traditional engineering for many years. The author proposes we apply traditional engineering principles to our software components to facilitate their application to other similar problems (DIp89:3).

Harel (Harel92) expands on this concept of models as the basis for software system development. He describes an approach to modeling systems and their components which he terms the vanilla approach (Harel92:10). This approach involves abstracting a system or components function and behavior in a well defined, formal way (Harel,1992:11). These methods of abstraction must allow models to be structured and combined to form a system of models (Harel92:11). One interesting feature of his approach is that he concentrates on real-time, or reactive, systems. Real-time systems comprise some of the most difficult problems to effectively model in the computer systems domain (Harel92:10).

Once a system of models has been created, Harel points out that we can exploit various emerging capabilities to explore and improve the resulting system. These include the ability to *execute* our system model, or observe its behavior as it passes through states (Harel92:15-16). This execution can also form the basis for a *prototype*, or working model of the system (Harel92:17). The ultimate goal of this model transformation process is the ability to create a final, finished system from our model automatically, without the requirement for writing code in a high level language. (Harel92:18)

Biggerstaff and Richter (Big89) present some of the problems, challenges, and dilemmas of software reuse. They stress describe *generation* technologies, whereby technology is applied to make software creation automated and repeatable. Generation technologies work because or recurring patterns, either in code or in rules used to transform software from one form to another.(Big89:3) Composition technologies encompass a component based approach, where new systems are made from building blocks. The authors also lump object oriented technology into the composition style of reuse.(Big89:3);

The authors describe many problems associated with component based reuse in practice. These include understanding of component, locating the correct component, composing components, and modifying components. They point out that only with a formal, automated engineering effort can component based reuse progress and realize large gains. They then discuss the issues of design reuse, pointing out that, though this technique has much potential, research is needed to make it usable in practice.(Big89:5-8)

Finally, the authors indicate that general reuse, supported by representation of designs and by automation, will provide the major gains in the potential for reuse. They indicate that reuse systems of the future will provide for partial specifications, whereby a design will be represented in a form that allows for customization, broadening the generality of the design (Big89:12-13)

Curtis (Cur88) presents issues involved in reusing software from the standpoint of designer understanding, which he terms *cognitive* issues. He points out that different types of programmers approach design differently, depending on their experience. (Cur88:1-10) The different mental processes relate to the amount of pre-concieved design information the designer can bring to bear on the design problem. More experienced designers will have a more complete view of the system they wish to compose, while inexperienced designers will tend to start with little or no notions of the design. (Cur88:3-5)

These cognitive issues, the author points out, create the need for a classification system for reusable artifacts which supports different levels of designer experience. He relates some classification schemes for reusable software intended to provide varying views of the software to designers of differing experience. (Cur88:7-8)

2.4 *Current Industry Reuse Practice*

There are many examples of reuse proliferating in the commercial world. In general, reusable software can take several different forms, and the amount or reuse can range from code library level reuse up through entire system reuse. I will provide some examples I have researched and also seen in personal experience.

In one example, Borland (Bor92) provides an application framework designed to speed development for the Microsoft Windows PC environment. Recall an application framework is some set of code designed to be customized to solve a particular class, or domain, of problem. This framework, called ObjectWindows, contains "object oriented stand-ins" (Bor92:3) for Windows entities such as drawing windows, scroll bars, icons, etc. When using ObjectWindows, the developer need not master any of the details of programming the Windows interface, just the application framework. The result should be drastically reduced development time.

In experimenting with the software, I went from complete Windows and C++ novice to a finished, high quality graphics application in the course of 4 weeks using this library. I found the user of the library had to be comfortable with some advanced C++ programming language features to master the library. Once this was done, the library provided some abstract and powerful objects to build the Windows application.

In another project, this time from the Ada world, an Ada application framework was build on top of an X Windows interface for the MSS (Mission Support Segment) project at Peterson AFB. I was a member of a team which created several layers of the application framework on the MSS project. This application layered abstractions on top of the windowing environment to allow the developers to build interfaces from high level, logical pieces. These pieces were sufficiently general to allow quick development, but were not classical reusable components in the general sense of the term. They were not reusable outside of the application framework they were built to support.

Current industry uses several methods to *parameterize* software. Key information which customizes a software artifact is called parameters. There are several methods to parameterize software (Figure 7). I will explore some of these methods, including software coding templates and table driven software, which are discussed below.

Software coding templates refers to a method of parameterizing that provides software in uncompiled form to a programmer, along with specific, clear instructions on what to change to make it work. Using this type of software, a programmer will fill in clearly marked sections of code and then compile and link an executable system from these software modules. Software coding templates are often seen in Ada systems, such as the JMASS simulation software. (JMASS92)

Another method of parameterizing software is to write executable software that expects to read in information at the start of its execution to configure itself to solve the particular problem.

This approach has been called *table driven software*. The Ada Quality and Style Guide (SPC91) recommends this approach as an advanced way to build an Ada application.

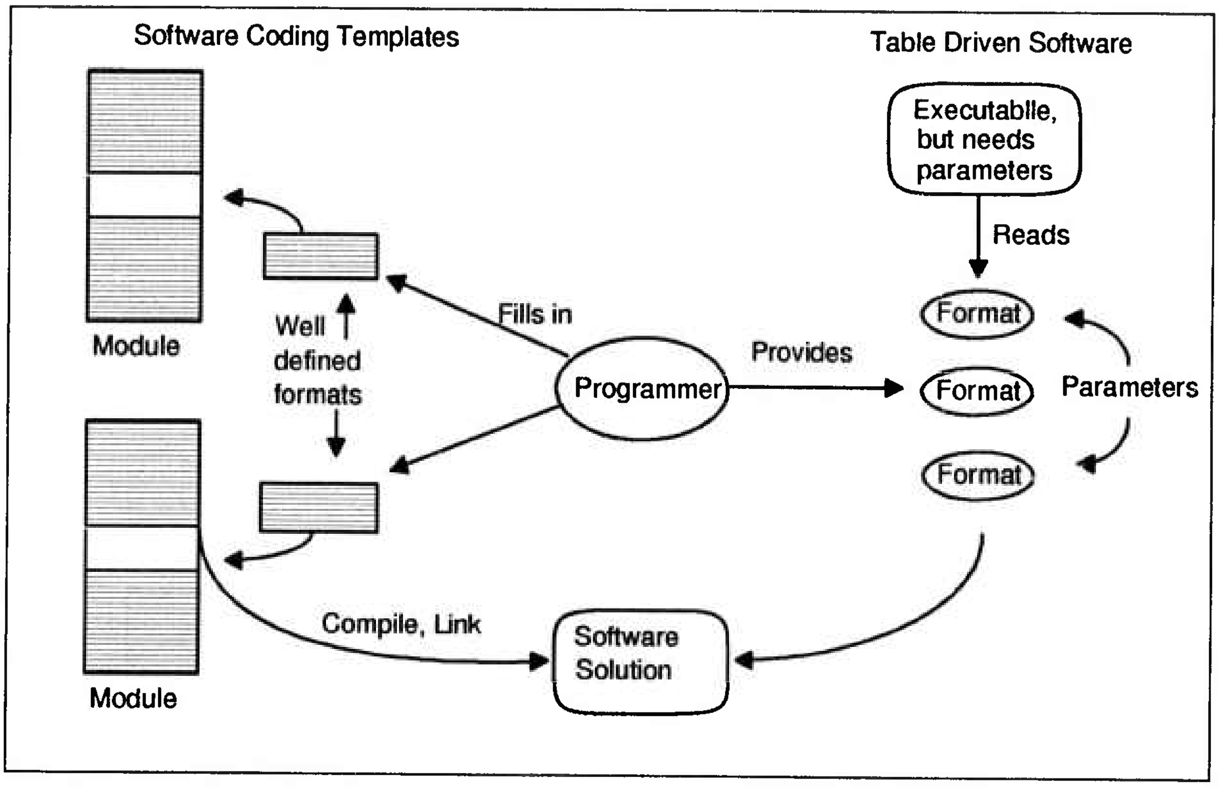


Figure 7 - Software Parameterization Methods

At Space Command, software organizations are exploring the use of entire reusable systems designed for a particular application domain. Using this approach, a system implementor uses a tool or tools designed to build the system from a set of parameters. In the Space Command case, the tool allowed a designer to interactively build command center applications using a graphical interface and underlying software designed to implement the application from the stored parameters created by the tool.

The commercial world has caught on to reusability with the advent of C++. As an example, books and articles have been appearing which attempt to show novice and non-professional programmers, as well as professionals, how to harness the power of object oriented programming

(OOP). OOP really builds on such concepts as data abstraction and information hiding commonly found in computer science curricula today.

Lafore (Laf92) shows how to use C++ in a reusable way. He covers the concepts of inheritance, dynamic binding, overloading, and class hierarchies in his book. These are all concepts included in C++ to provide the ability to extend and customize existing software. These capabilities are at the center of OOP's popularity.

Both Ada83 and Ada9X have powerful features to support reuse and customization. Ada provides its packages and generics as reuse mechanisms. In Ada9X, support for inheritance and dynamic binding is added to increase the functionality.(DoD92) Many repositories exist for Ada sofftware, allowing a developer access to a wide body of past work.

2.5 Conclusion

The notion that, as software engineers, we can reuse our prior work has been the basis for much of our methodologies and research for many years. In the last 5 years, research has begun to explore how we can build very large systems and networks of systems to exploit reuse. These methods include megaprogramming, generic or reusable architectures, and model based software engineering. These activities are all essentially concerned with the same problem - how do software analysts, designers, and developers capture knowledge and create an abstract description of it? How do they allow software to be used over many related problems, and ease the construction of software systems which use the knowledge?

Traditional component-based reuse is often inadequate to the task of constructing reusable systems. In a component based strategy, the designer must design the software architecture the components are to fit into. Though this type of reuse leads to savings and reduced time, a design expert is still required to complete the finished system.

I have covered several authors' ideas about how to approach architecture-based reuse. There is no agreement in the field currently on any one approach to this problem. Many of the research efforts show promising theoretical foundations for improving the future of this area. These efforts also show examples of successful application of these principles.

I have also taken a look at some industry practice in this area. Reuse is becoming popular and economical in industry, and it is approached from several different angles by practitioners. Many of these methods are based on newer object oriented or object based languages and concepts, such as C++ and Ada. Industry has much to teach the software development community about creating software, since industry is always seeking to save dollars.

It is this broad spectrum of application for these principles in both research and industry which makes architecture-based reuse difficult to study and quantify. However, discovering the principles and techniques behind these methods can lead to many different productivity gains in all phases of the software development process.

Reusable architectures don't impose a specific methodology or tool on the development organization using them. Because of this, they can be applied to software development environments fitting many varying descriptions. They can be applied successfully in environments ranging from the most rigorous, formal methodologies down to a single developer using a PC in his spare time. In fact, they have a potential to create significant savings across a broad spectrum by supporting analysis, design, and implementation reuse.

Reusable architectures and their application to various domains is an active research area. These methods are a good step to bringing large scale development under control. An evolving theoretical basis combined with encouraging results of applying reusable architectures in industry point the way to a bright future for these methods in the software engineering field

III The ObjectSim Concept

3.1 Overview

This chapter is a report on my first efforts to create a reusable analysis model and software design for visual simulation applications in the lab, and corresponding implementation in C++. The initial section is an analysis of the requirements and supporting software process in the Graphics Lab. Second, I discuss the method we chose to meet our goals in the lab. The result of the requirements analysis is expressed as a Rumbaugh object model of a rudimentary architecture for visual simulation. This chapter discusses the process used in the analysis and design stages. Initial projects which validated the extension through inheritance method of customizing the architecture are discussed.

3.2 Initial Analysis

The goals of the Graphics Lab visual simulation projects are to support sponsor research efforts involving Distributed Interactive Simulation (DIS). The lab is sponsored to develop several visual simulation projects which will interact over network lines with other simulators. Students develop the simulations either individually or in teams as research efforts. I analyzed the proces the lab used to reuse work and ensure continuity of these development to determine areas where software engineering techniques could help.

Reuse has been a goal of this lab for some time. Component based reuse, which emphasizes stand-alone classes or libraries, was used for many past projects.. However, as I examined the past projects and level of design sharing done for them, I noticed that past work had often been redone, or that existing components were not suitable due to subtle design differences. This led to a number of interesting effects, as shown in Table 1. Helping the development environment reduce these effects was a central goal of the architecture.

Problem	Impact
No one responsible for standard components	Past reusable software difficult to locate or classify/verify
No one responsible for improving common simulation capability or fostering common design	Lab duplicates effort it could reuse. Machine power underused. Students duplicating effort to achieve functionality
Existing components complex to understand	High learning curve. Students designing to lower level of components than necessary
No standard library locations, CASE, or CM to support large developments Libraries scattered	Students using word of mouth or global search to find past work
No design methodology adopted for projects, other than rapid prototyping / incremental development approach	No current, usable design representation for the work. Students reduced to maintainance of code only
Students don't have time to become well grounded in C++ or C.	Language features unused, poor code designs, little reuse or orthogonality in existing code
Research projects did not have stable requirements	Students implementing unneeded work, backtracking and redesigning on the fly

Table 1 - Graphics Lab Development Environment Problems

I surveyed the existing applications in the lab to attempt to determine both a robust set of visual simulation requirements and the level of reuse across projects. I discussed the designs of the Virtual Cockpit (VC) and Synthetic Battle Bridge (SBB) with the thesis students who had implemented them the year before. I found little commonalty in the implementations and designs of the systems, but much commonalty in the baseline requirements.

The Virtual Cockpit implements a simulation allowing the participant to fly through a visual scene in a simulated cockpit. The basic requirements include the ability to have a view attached to the cockpit, the ability to read tracking input and 'look around' at the scene, and the ability to receive and display participants from the network. For the last thesis cycle, all of the work done to look 'out the window' was done for the Virtual Cockpit project and was not reused. The network interface was developed for both the VC and SBB, but ended up being two versions because of different interface requirements. A central goal of any reusable architecture for the lab was the ability to support the Virtual Cockpit project.

The Synthetic Battle Bridge implemented a simulation allowing the participant to utilize different displays and interface devices to enter and move around an ongoing network simulation. The previous thesis work was done using GDMS. Since this effort and the VC were developed under different advisors by different students, there was no effort to create a common core for the two simulations except the network interface work.

After studying the VC and SBB, I surveyed the projected applications in the lab to determine if they would add requirements or constraints to the basic architecture. I discussed the Red Flag Remote Debriefing Tool (RDT) project, being developed to show ongoing and stored Red Flag exercises, with the responsible student. I also discussed a simulation requiring a space view, including satellites. From these discussions and time spent in the lab, I developed a set of high level basis requirements for the architecture. These are shown in Table 2 and Figure 8.

3.3 Iris Performer Overview

The first existing work I surveyed for this project was SGI's Performer. Our lab had an early version of this visual simulation library. Later, through contacts with SGI, we received a later release and a Programmer's Manual for the library. This overview of Performer is based on that manual and my initial experiments with sample programs..

Project	Requirement
All	Move viewpoint around simulation
All	Allow view direction to be based on device input
All	Varying window sizes
VC/SBS	Drive head mounted display and use head tracking
All	Display ongoing network traffic
VC	Send network information
All	Display configurable terrain
All	Display text and icons on screen window and in scene
All	Display correct world state (destruction, damage, time of day)

Table 2 - Basis Requirements for Graphics Lab Simulations

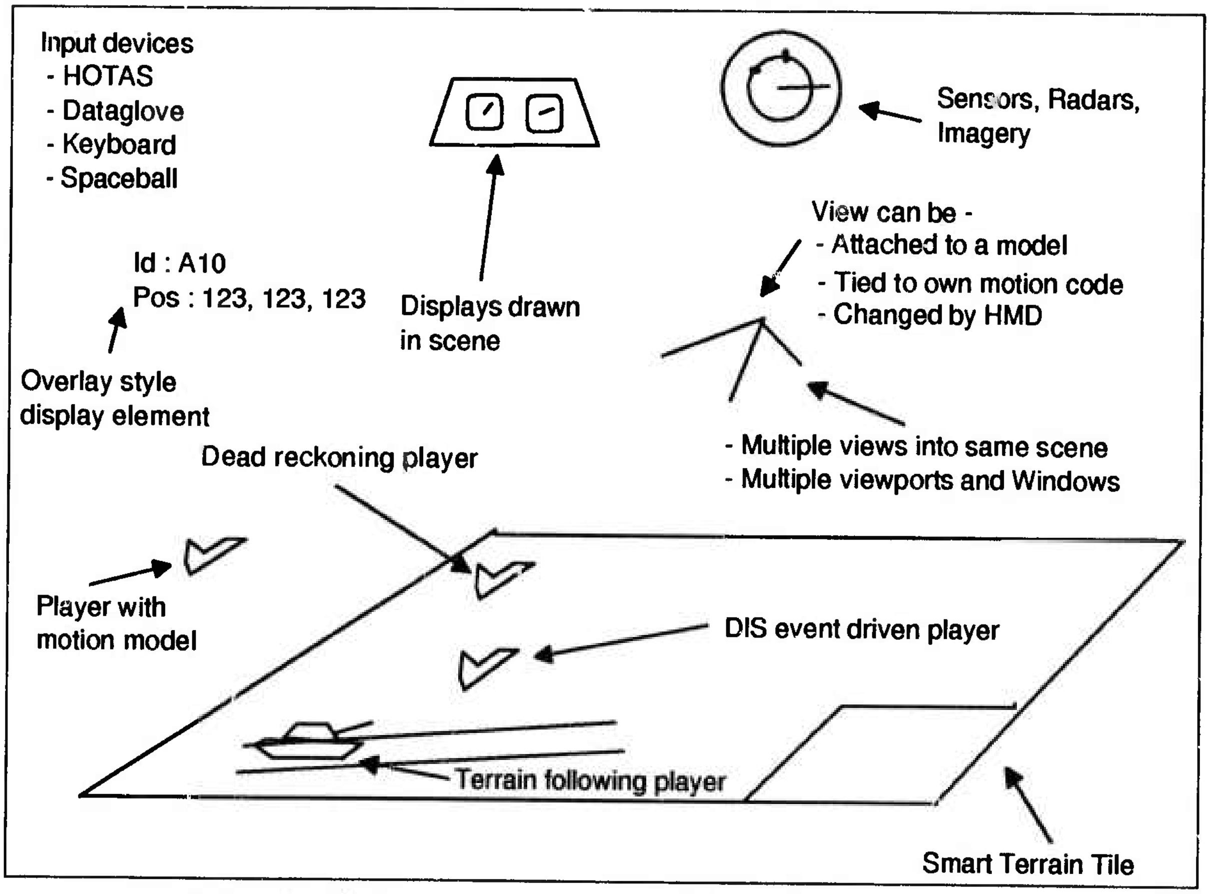


Figure 8 - High Level Graphics Lab Requirements

Performer is a library of C callable routines that provide a new interface to the SGI graphics pipeline. This interface improves vastly on the GL interface in many areas. This library abstracts many of the chores associated with programming the SGI, and captures much of the SGI expertise about programming their pipeline. In addition, Performer provides an easy upgrade to the RealityEngine platform from the VGXT, because the library hides much of the details of rendering geometry and the same Performer interface is used on the new machines. (SG92)

Performer maintains an internal geometry tree. This tree holds many of the common types of geometry used on the SGI. Geometry nodes, or *geodes*, can hold polygons, triangle mesh, light points, and many other types of primitives These geodes are the leaves in the geometry tree. The tree also hold information like level of detail switching nodes, animation nodes, coordinate systems, group nodes, and instance nodes (SG92:Ch9).

Since the geometry tree is not dependent on a particular external geometry format, any format which can be converted to the Performer internal geometry format can be used on the tree. This conversion is done through file reader routines (figure 9). Performer comes with a Multigen flight format reader, a reader for '.sgi' format geometry, and a reader for '.bin' format geometry. The flight reader allows all of the power and hierarchy expression available with Multigen to be seamlessly used in Performer applications with minimal effort. (SG92:Ch2)

Performer also maintains state information in the tree. These nodes called *geostates* hold information like materials, textures, transparency, lighting conditions, and other information normally part of the GL state. This allows Performer to easily handle multiple textures, transparency, etc, through its built in state management features.

Performer renders its geometry tree very efficiently. Some early test results showed very promising frame rates for our simulation prototypes. This indicated the limiting factors for efficiently rendering a set of geometry are dependent on geometry and scene management, not on rendering code efficiency, when Performer is used.

The other performance improvement with the Performer library is obtained through its multiprocess management features. Performer provides an abstract model for using multiple processors on a multiprocessor machine. This model dedicates one processor to drawing and culling per rendering pipe, and one to the application managing the scene. A straightforward callback model allows custom draw and cull processing for the simulation, such as is used for drawing display text, reading the keyboard, and similar tasks which need to be connected to the GL process. This is explained later in the section on architecture.

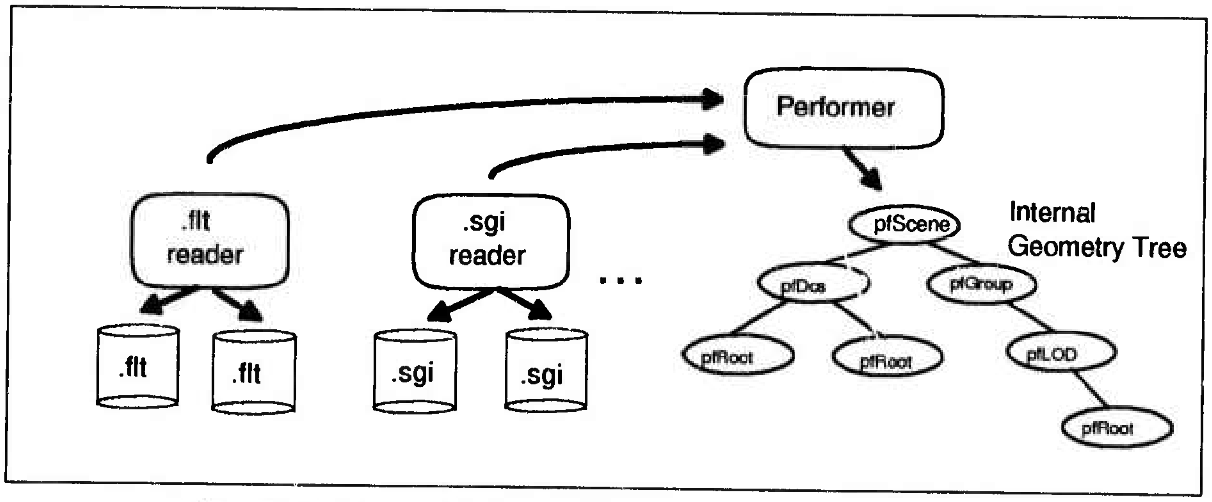
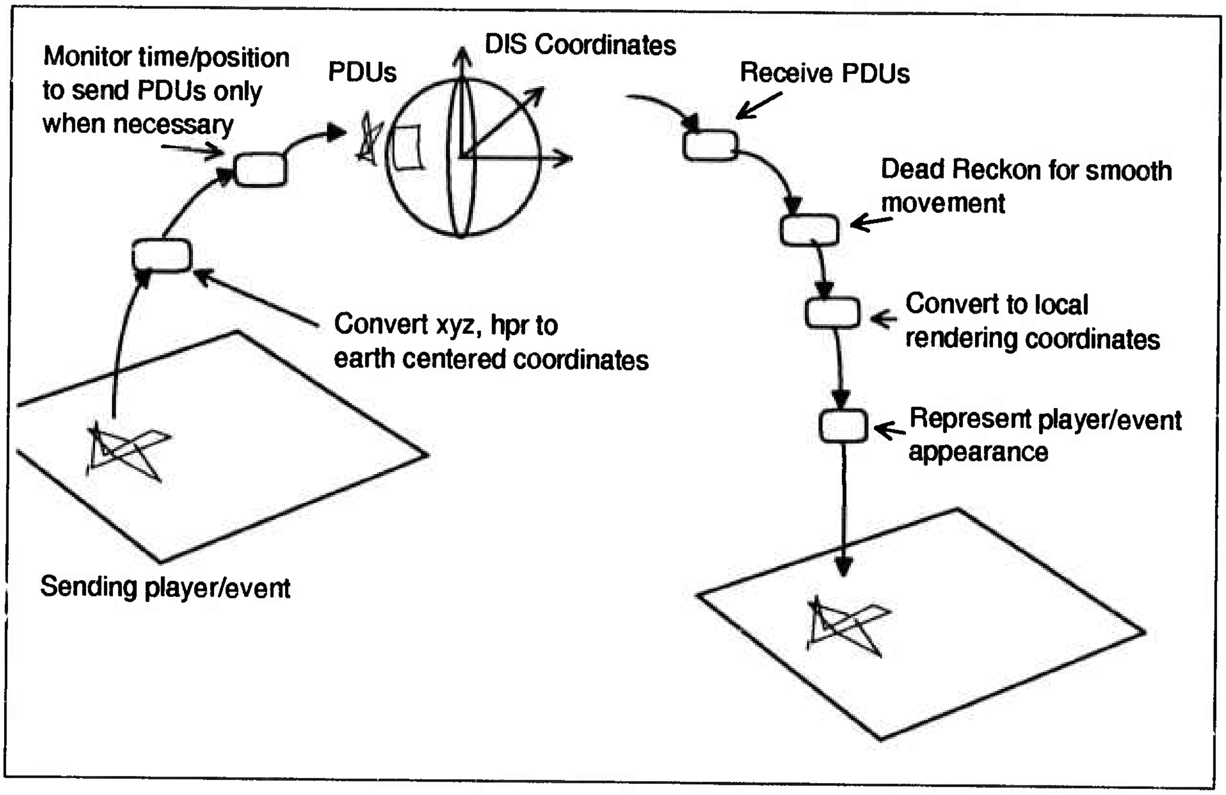


Figure 9 - Format Independent Performer Operation

Other features in Performer include multiple channels (viewports) into a scene, an easy to use viewing model based on Euler angles, and an extensive math library. Performer has built in collision detection features, intersection testing, and special effects processing (clouds, h[illegible] of day, earth-sky, etc). The version 1 release also contains a reader for Multigen Flight format which allows use with Multigen easily.

3.4 Object Manager

The Object Manager is an existing library which implements a network interface. As discussed in Chapter 1, a network interface manages the sending and receiving of network PDUs, enabling simulators to communicate. Figure 10 shows a high level view of a network interface's requirements.



Monitor time/position to send PDUs only when necessary
PDUs
DIS Coordinates
Receive PDUs
Dead Reckon for smooth movement
Convert xyz, hpr to earth centered coordinates
Convert to local rendering coordinates
Represent player/event appearance
Sending player/event

Figure 10 - Network Interface Requirements

The Object Manager handles the functions of position/time monitoring, dead reckoning, and sending and receiving the PDUs over the network through interfacing to a low-level network driver called a *network daemon*. In seeking to reuse the Object Manager, the developers were looking for ways to efficiently manage the processing shown in the diagram above. This processing can be particularly expensive for large numbers of entities being received and dead-reckoned. Table 3 shows a breakdown of the specific tasks required for the network interface.

DIS Simulation Requirement	Related Software Responsibility
Simulators need to show accurate state of external entities (type, position, orientation, damage, flaming)	Software needs functions to position network entities and reflect their state graphically and within any other required data or display structures. Software will map simulated entities onto correct graphical representation
Send simulators can cause events, which can affect other entities in the simulation	Software must be able to generate internal simulation events, broadcast them, and process them against the entities from the network to inform those simulators if they are affected
Simulators will process events in the simulated world, such as explosions and weapons fire	Software will allow events to be graphically shown from internal sources or over the network
Send simulators will change their state based on external or internal events and based on the simulations progress	Software will process both external and internal events against sending simulator(s) and broadcast and graphically show correct state
Send simulators will reduce traffic by limiting updates based on time or position	Software will not update state until some time or position threshhold is exceeded
Receive simulators will show a smooth entity progression through scene	Software will dead-reckon simulation entities between updates

Table 3 - Software Requirements for Network Interface

The Object Manager as implemented used a 'snapshot' interfacing method to communicate the received entity status to the application every frame. This means all entity data is copied into data structures for the requesting application. The application has responsibility to perform any coordinate transforms required, show the correct entity representation, and update the representation's position and orientation to show accurate movement.

3.5 Other Existing Work

Developers identified other existing libraries available for use in the Graphics Lab. These included device interface classes from previous years thesis work, some drawing classes for instruments and stroke text from Wright Labs, and a public domain user interface package with an interactive tool for constructing interfaces. For each reusable library, we either incorporated the library or devised a strategy of interfacing the common simulation design to the library.

3.6 Reusable Applications

After surveying the requirements in the Graphics Lab, I created a *meta model* or reusable design to base the simulation work on. Meta models and related notions of software architectures are referred to often in current literature. Lowry (Lowry91) describes both *application generators* and *software architectures* as software engineering techniques which abstract and reuse design knowledge and drastically shorten the time required to develop applications (Lowry91:641) Application generators are tools which allow a developer to supply the parameters to a reusable piece of code or application. The tool then generates a system fitting the parameters required. Generally these are useful in a narrow domain Software architectures take this approach one step higher. They seek to reuse designs across domains which share some commonalty. (Lowry91:648) The result in a system which is more general, but more difficult to customize, often required knowledge based techniques. (Lowry91:649)

In the C++ world, these same ideas are the basis for the notions of *application frameworks*, also called *application templates*. Often, these take the form of class libraries. Application frameworks differ from class libraries in that they support reuse at a much higher level than traditional component based software development. The reusable components in an application framework are *specifically designed* to provide a high level of services within an application *using the framework*. Outside of the framework, the reusable pieces may not provide any functionality, but within the framework, they can provide major savings.

3.7 Test Architecture

In the Graphics Lab at AFIT, I viewed an application framework as a good first step in solving the development problems encountered above. An application framework with suitable documentation would allow student developers to concentrate on the research parts of their simulation, while not redesigning the basic simulation repeatedly. The framework could provide a ready repository for storing reusable forms of solutions to problems developed by students. I will show more examples of this technique later.

After I decided on the application framework strategy, I began to incorporate previous reuse work into the architecture, beginning with Performer. As I progressed, I also incorporated other work in the form of device interfaces, network interfaces, and reusable classes developed by current students to fit into the architecture.

I designed the initial meta model to be C++ classes providing a high level stand-in for capabilities provided by the Performer library. This approach is similar to the ObjectWindows strategy discussed above. The initial Rumbaugh class diagram we designed to is at Figure 11. The simulation class is written for each application using this architecture. Each simulation class must have the same signature and perform specified jobs at each of its entry points (methods) to work with the architecture. When it is developed to meet these requirements, the developer can use the Renderer and Model classes' services to offload much of its detailed functionality, and concentrate on the functionality that makes his simulation unique .

Using this simple model as the guide for my efforts, I implemented the framework classes Model and Renderer. These provide an interface to functionality found in the Performer library. Once these were complete, I built two simulation applications to plug in to the framework, a geometry viewer and a fly-through simulator. Each application reused the Model and Renderer classes.

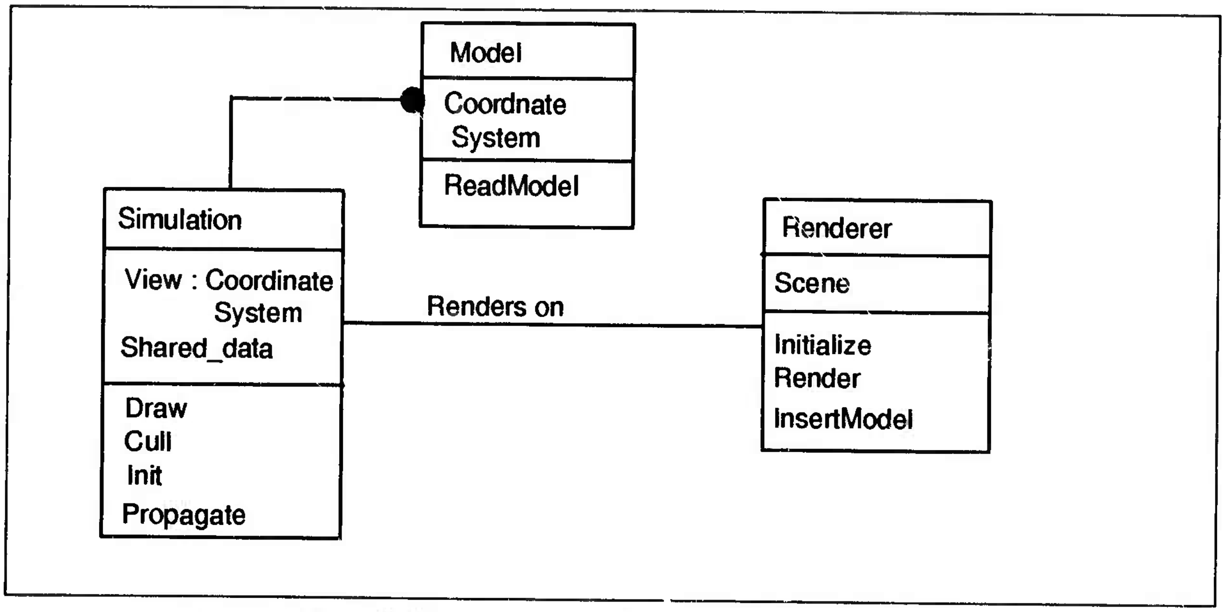


Figure 11 - Initial Reusable Simulation Model

In this much simplified model, a visual simulation consists of a renderer, an application which uses the renderer, and some collection of dynamic models manipulated by the application and rendered by the renderer. Every application has a view (in the basic case), which is updated by itself and is the view for the renderer. The application contains the callbacks for drawing and culling, to be called by the renderer. This structure allows the propagate, cull, and draw members to be called from different processes, which follows the Performer multiprocess technique. Finally, the application must maintain a special memory area for shared data between its draw, cull, and propagate members to accommodate multiprocessing. This area is allocated in the shared data area by the renderer if the renderer wishes to do multiprocessing.

It is important to note that only the application class needs to be written to reuse this architecture. The renderer and model class provide abstract services to application classes which are plugged in to this architecture. The application class provides entry points which the renderer calls in the form of C++ *virtual functions,* which allows any application subclass providing implementations for the virtual functions to function with the renderer. Two examples of simulations written using this architecture are given below.

3.8 Viewer Instantiation

The viewer provided a basic application to "shake out" the architecture and work with Performer. I started with an example program to implement a simple fly through of a scene represented as one flight format model. I first designed the architecture as shown above. Then I added functionality to improve the fly through, allow manipulation of the model, and display some data to the user as he used the program.

I used the viewer to explore the multiprocessing features of Performer. Since Performer provides a standard 3 process model for the simulation, I wanted the early design to be able to take advantage of this feature. Having access to the multiprocessor features was an important performance criterium for the architecture. Figure 12 illustrates the distribution of tasks to the application, draw, and cull processes and what the application class needs to do to accommodate this.

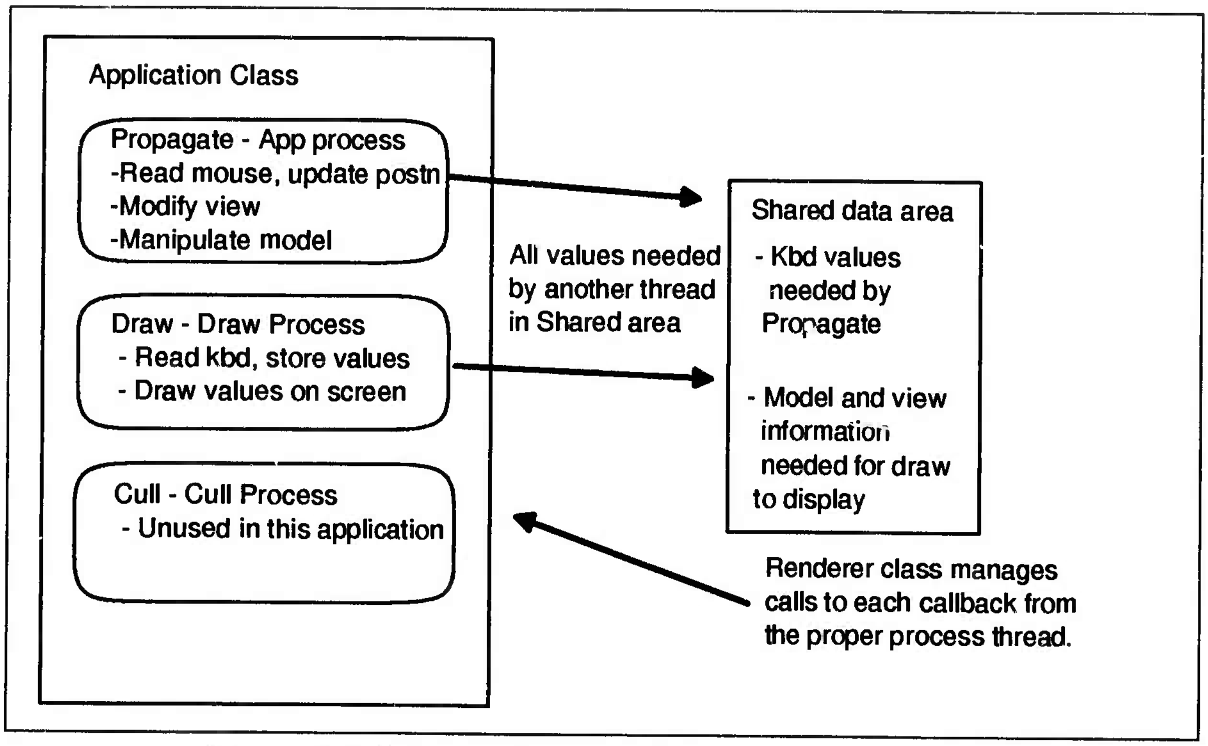

Application Class
Propagate - App process
-Read mouse, update postn
-Modify view
-Manipulate model
Draw - Draw Process
- Read kbd, store values
- Draw values on screen
Cull - Cull Process
- Unused in this application
All values needed by another thread in Shared area
Shared data area
- Kbd values needed by Propagate
- Model and view information needed for draw to display
Renderer class manages calls to each callback from the proper process thread.

Figure 12 - Viewer Multiprocessing

I was able to complete the viewer to work using the three processor, fully parallel mode in Performer. It reads in one command line argument, assumed to be a legal flight format file, and sets up a suitable view based on the model's size. It allows fly through and orientation manipulation of the scene, with some rudimentary statistics displayed to the user.

3.9 Flyer Instantiation

The second application I built to use the architecture was conceived as a platform to help the Graphics Lab develop displays and instruments for the Virtual Cockpit. It is a simple fly through application which allows the user to specify a flight file for terrain and another to fly through the terrain with. The user then flies through the terrain "in" the specified model. I built this application as a lead in to converting the Virtual Cockpit to a Performer application.

This application has become the standard test application for the ObjectSim architecture. This means that, as new capability is written for the architecture, it is first implemented, debugged, and tested with this application before being used on other Graphics Lab projects.

3.10 Conclusions

After implementing two very different simulations using extension through inheritance, I felt the idea had enough promise to become the main vehicle for the remainder of this research. I revisited the analysis work to construct a full scale model of the reusable simulation.

For this next step, I gave the system a name - the ObjectSim application framework for visual simulation development on SGI machines. ObjectSim was developed using a development model I will call the *necessity* model. In this model, successive applications are developed or converted to use the common architecture. As each is converted, worthwhile pieces of functionality are incorporated into the application framework, based on an overall design for the framework. In this way, during the conversion process, I incorporated previous solutions to particular problems and made them available to every simulation using the framework.

The requirements for ObjectSim were to support a large class of simulation applications in the Graphics Lab. These applications have requirements to communicate with and receive DIS simulation messages, drive head-mounted displays, provide flexible views into the simulation scene, and allow simulation through large geographical databases. In Chapter 4, I will present the model and class designs which combine to form a reusable visual simulation capable of satisfying these requirements..

IV ObjectSim Design

4.1 Design Process

The design process used for this framework was based on the analysis process described in the last chapter. Once I had the basic requirements listed and understood, I took the small framework I had created for the viewer and the flyer and added the other classes needed to support the thesis projects in the lab. Following the high level requirements I had identified for the Graphics Lab simulations, I came up with another high level diagram, which shows th[illegible]asic classes and division of responsibility for ObjectSim. This diagram is at Figure 13. This scheme for requirements separation in the architecture guided the design of classes, member data, and member functions.

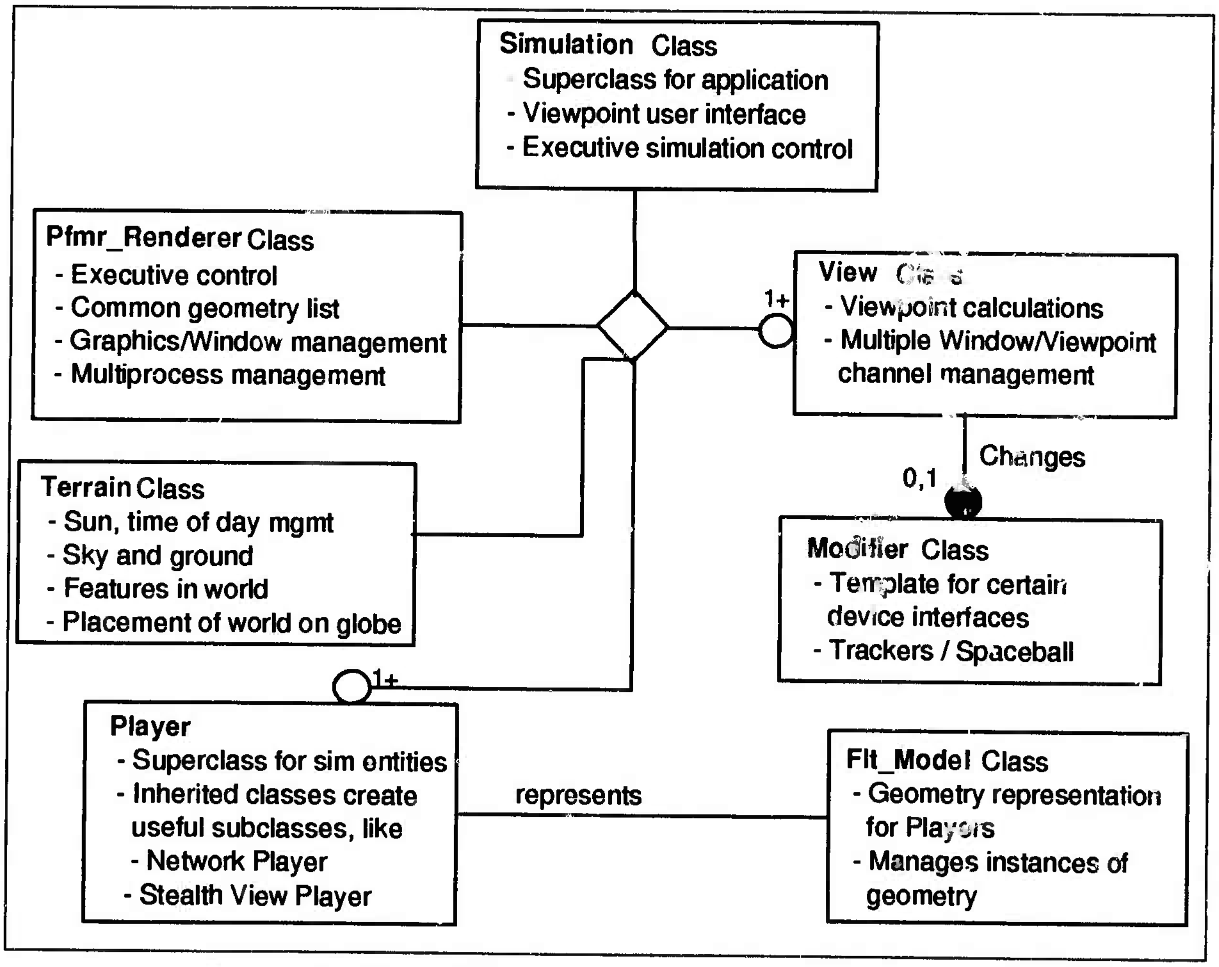

Simulation Class
- Superclass for application
- Viewpoint user interface
- Executive simulation control
Pfmr_Renderer Class
- Executive control
- Common geometry list
- Graphics/Window management
- Multiprocess management
View Class
- Viewpoint calculations
- Multiple Window/Viewpoint channel management
1+
Changes
0,1
Terrain Class
- Sun, time of day mgmt
- Sky and ground
- Features in world
- Placement of world on globe
Modifier Class
- Template for certain device interfaces
- Trackers / Spaceball
1+
Player
- Superclass for sim entities
- Inherited classes create useful subclasses, like
- Network Player
- Stealth View Player
represents
Flt_Model Class
- Geometry representation for Players
- Manages instances of geometry

Figure 13 - ObjectSim Requirements Separation

The second step in the design process involved creating C++ class descriptions for the basic classes. These descriptions contained methods to implement the functionality needed by each class and also the control points needed to make each class work in the multiprocess model.

The design process for ObjectSim was very much incremental and intertwined with the early implementations using the library. Overall, I would characterize the development as a rapid prototyping type of spiral model. Typically, I would concentrate my efforts on exploring high risk areas, followed by a sample implementation, usually in my Flyer test program. Once the piece of the library worked as I expected, I would aid the developers using the library in adding the capability into their simulation.

4.2 ObjectSim Documentation

The documentation for the classes appears in three forms. In this chapter, I document the main classes with Rumbaugh Object Models and Rumbaugh State Diagrams. The ObjectSim Application Developers Manual (Appendix A), contains a class reference which documents most classes contained in ObjectSim, along with a tutorial designed to aid a developer using the library. It is the developer's manual which is intended to be maintained as the reference to the library. The final design is found in the class header files These files contain the latest design of the library.

4.3 Class Simulation

This class is the superclass for all ObjectSim simulations. It was designed to provide the entry points needed to assign processing to different threads or times. In this way, the programmer can use a simulation class to provide an object oriented executive for the simulation. The calls into this class simply tell the simulation that the graphics portion of the program is ready for the simulation to compute another frame step of simulation 'movement' or propagation, or to draw or poll devices on different threads.

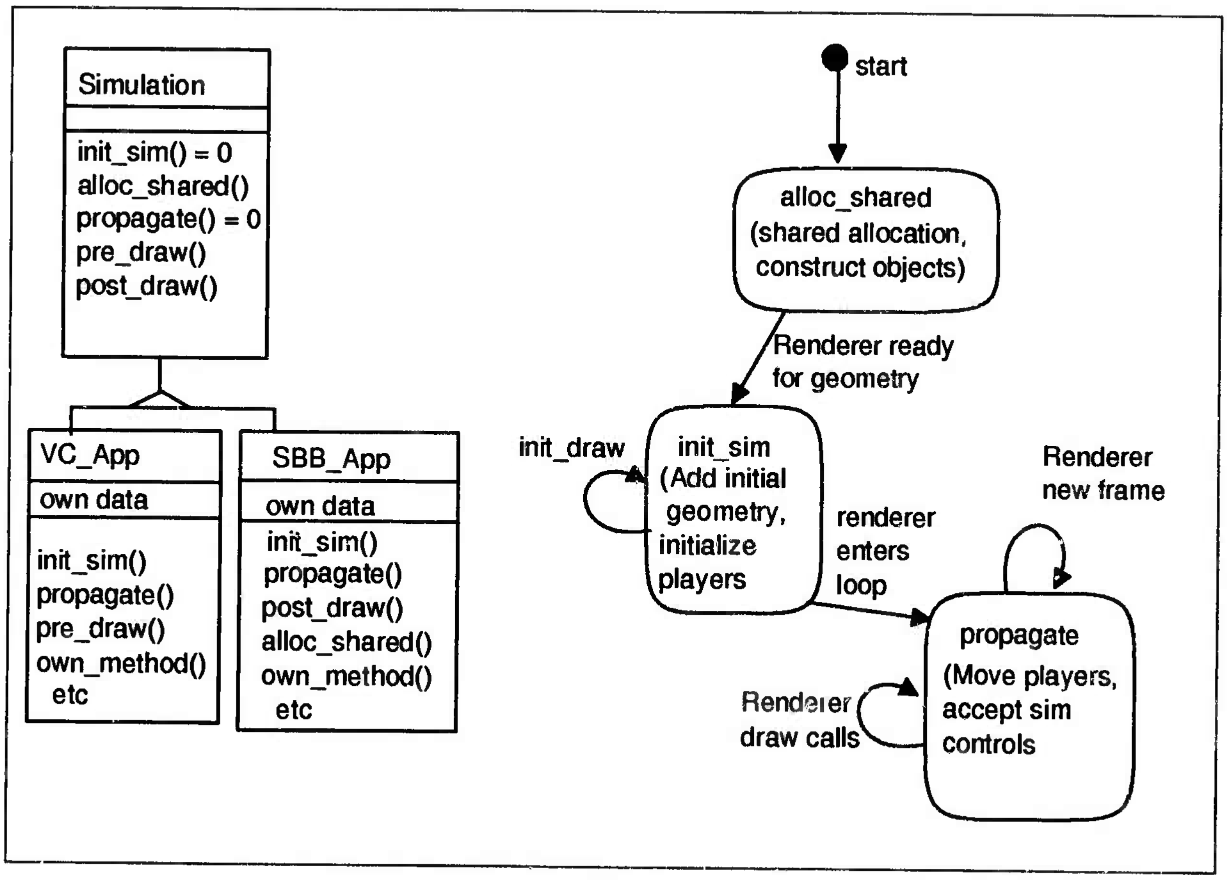


Figure 14 - Simulation Class Diagrams

Figure 14 shows a diagram of the simulation class, along with an example of the way subclasses must override certain operations and can override others, if needed. The state diagram shows the various states the simulation can be in, with the transitions equating to calls on the objects entry points.

Each simulation should contain a constructor which consists of calls to initialize the various ObjectSim objects. The constructor performs the functions described in the alloc_shared state above. After constructing objects, the simulation enters init_sim state as described above, which performs any geometry building needed for the simulation. The propagate state, encompassing all calls to the simulation, is a continuous loop at the frame speed of the machine. To effect a 'real time' simulation, a time call can be used to key simulation or player processing to time passage.

4.4 Class View

This class encapsulates a single view into the simulation. A view has a channel where it is rendered, and takes its viewpoint from the player it is attached to. Calculation of the viewpoint is discussed below. The view class holds the reference to the channel's scene and the Performer matrices used to translate the scene graph to move the terrain and players' geometry relative to its viewpoint. A simulation can maintain multiple views, which allows multiple channel simulations.

Views take their viewpoint from a player. A player is some entity in the simulation. From the view's perspective, the only important feature of the player is that it has a location, an orientation, and may have geometry associated with it. When a view is attached to a player, the view always takes that player's position and orientation. If the player does have geometry associated with it, the architecture takes the additional step of performing 'jitter' removal. The problem we noticed was when rendering geometry close to the viewpoint the geometry would jump from frame to frame, causing a jittery effect, This intensified as the viewpoint moved away from the origin. Setting the viewpoint to 0,0,0 didn't resolve the problem, so we traced it to floating point error in the geometry pipeline caused by the geometry being translated from its actual position to the position around the viewer at the origin. The solution was not to draw the geometry and translate it, but to substitute a copy of the geometry actually drawn at the origin. Therefore, a viewpoint inside geometry actually is set at the viewpoint offset rotated by the player's heading, pitch, and roll vectors. The 'zeroed' geometry is rotated and drawn, and the player's xyz location is used to scroll the world by (Figure 15).

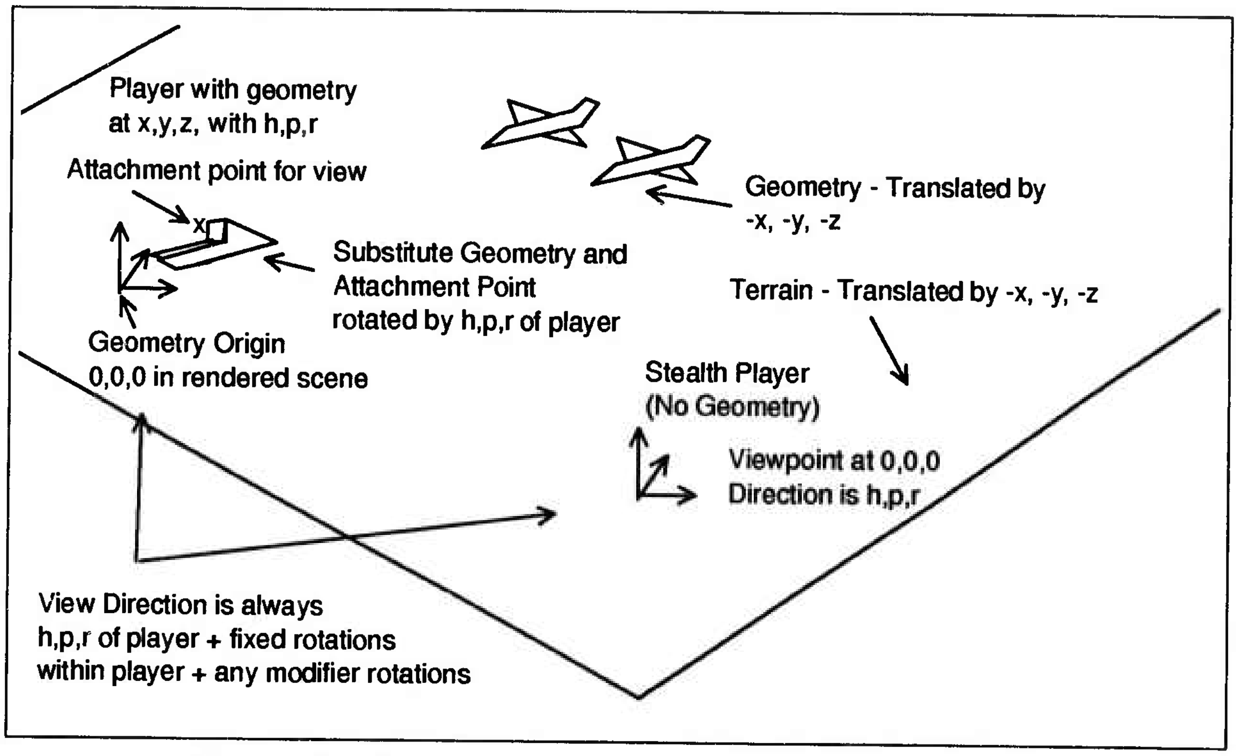


Figure 15 - View Computation and Jitter Removal

The viewing calculation was designed to allow for two other factors required in a simulation, positioning within the player and modification based on device input. The location of the view within the player is accomplished by a rotation and location vector expressed in the local coordinate system of the player. For instance, a location for a flight simulation might indicate the head position of the pilot, while the rotation might indicate the default gazing direction for the pilot. Figure 16 shows the View class design.

4.5 Class Player

This class is used for entities and stealth views in the simulation. Generally, a simulation will consist of one or more players which implement stealth or active viewpoints and/or other active entities in the simulation. For active entities, the player can be used to encapsulate the functions of modeling behavior, updating geometry, any drawing associated with the player, and handling any input required. For stealth views, the player's can handle moving the view, drawing, and input. For remote DIS entities, the player can serve as a capability designed to accurately reflect the position and appearance of the remote entity.

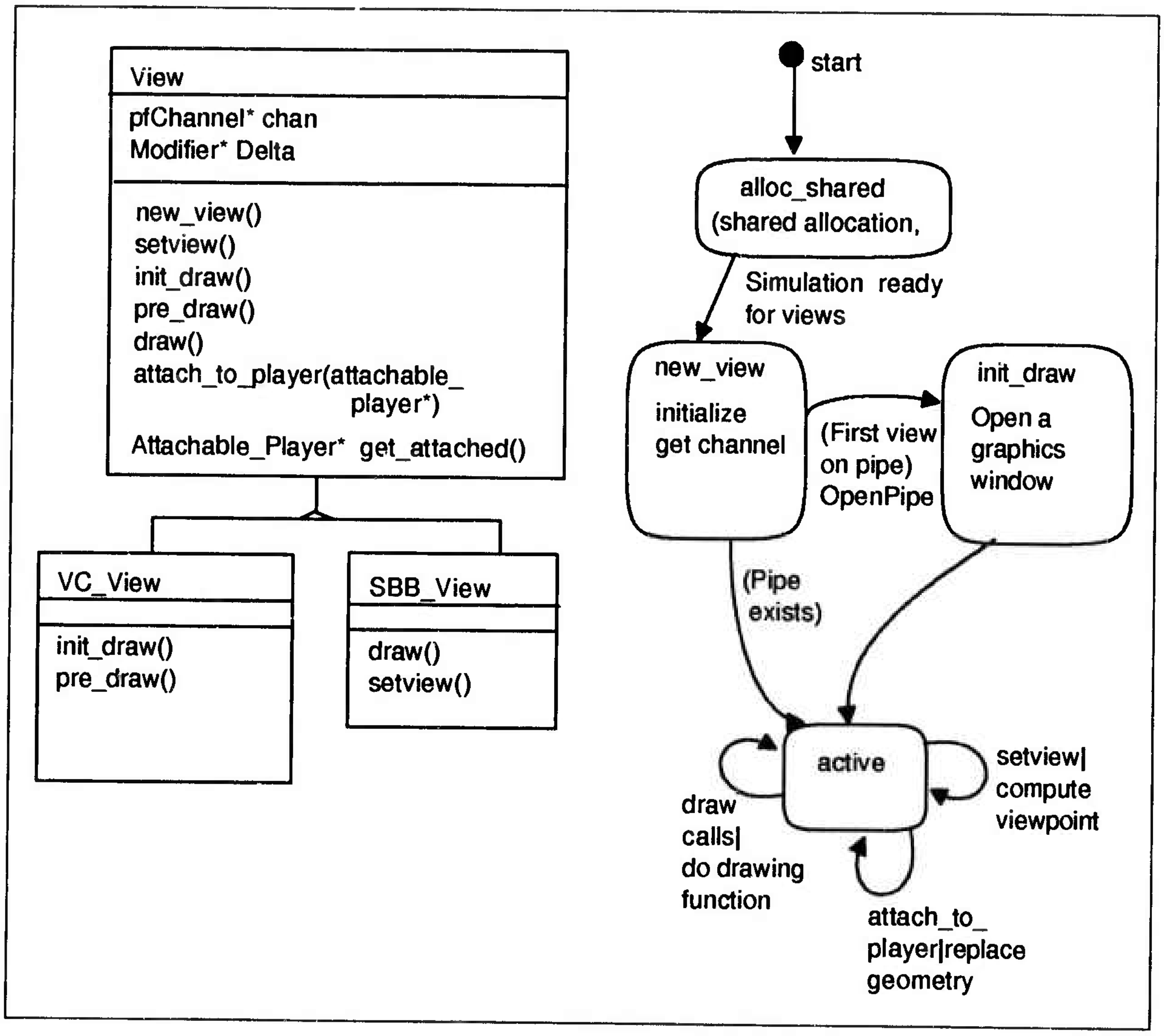

View
pfChannel* chan
Modifier* Delta
new_view()
setview()
init_draw()
pre_draw()
draw()
attach_to_player(attachable_ player*)
Attachable_Player* get_attached()
VC_View
init_draw()
pre_draw()
SBB_View
draw()
setview()
start
alloc_shared (shared allocation,
Simulation ready for views
new_view
initialize get channel
(First view on pipe) OpenPipe
init_draw
Open a graphics window
(Pipe exists)
active
draw calls| do drawing function
setview| compute viewpoint
attach_to_ player|replace geometry

Figure 16 - View Class Diagram

Players are the way the simulation accomplishes its purpose. Any entity being modeled can be thought of as a player in the simulation. The simulation presents its geometric display via attaching the views to players. Other players in the simulation may appear in the view, if they maintain a geometric representation and are in the right location to be seen by the player currently attached to the view. Figure 17 shows documentation for the player class.

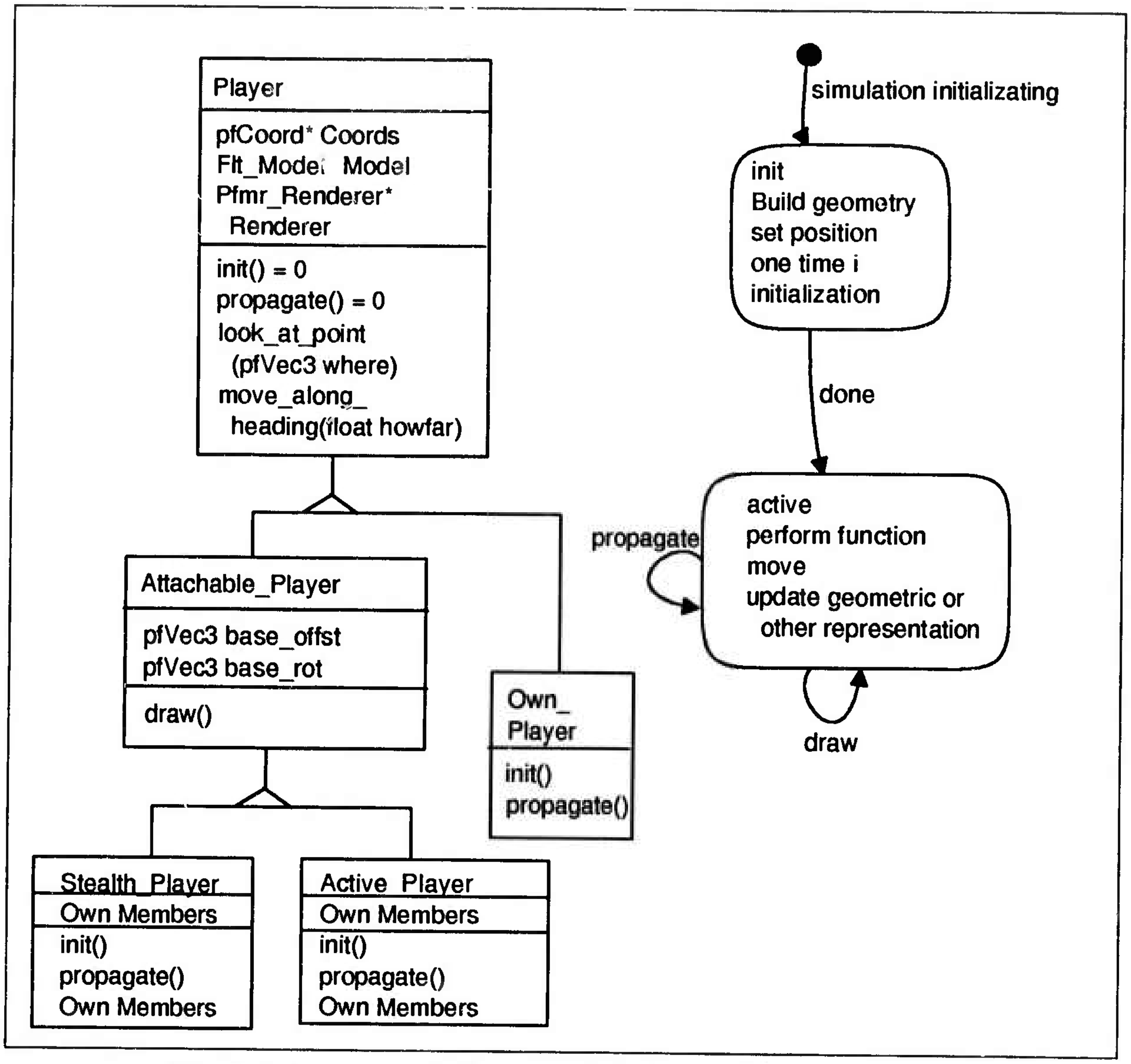


Figure 17 - Player and Attachable_Player Diagrams

4.6 *Class Flt_Model*

This class was designed as a way to allow simulations to handle their geometric representations in an abstract manner. The process of reading a representation from a file involves a utility called a 'loader' which translates the geometry into a format suitable for rendering. In the case of

ObjectSim/Performer, the loader translates the geometry into the Performer tree format. This class provides methods fo· abstracting this loading process.

In a large simulation, thousands of entities may be in the rendering tree at one time. Many of these will have common geometric representations (a battalion of tanks, for instance). Since these models take memory, applications seek to load only one copy of each unique geometry. This is accomplished through instancing. This class also maintains a data structure for instancing the models. Each model is identified by an integer index, which becomes its unique identifier. Any other load requests using that index will be given a handle to the already loaded model. This presumes, of course, that the applications utilize this class to load the geometry.

This class is a natural repository for methods which operate during the loading process to perform some specialized action. This might include: building a complex model from several files, reorienting models described in a non-compatible coordinate system, or handling requirements for articulated parts, etc. Figure 18 presents the design for the Flt_Model class.

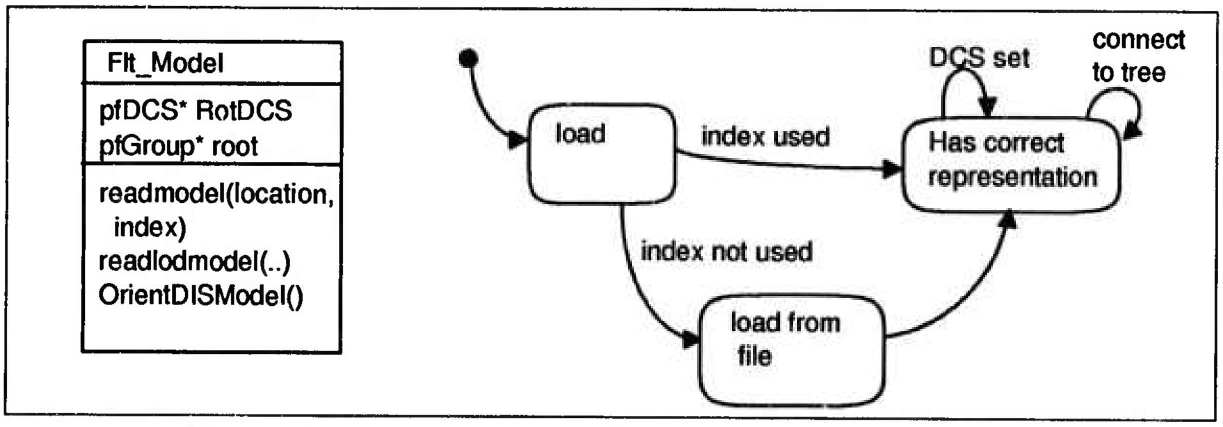


Figure 18 - Flt_Model Class Diagram

A Flt_Model is used to put representations into the Performer tree. It must be connected to the tree and the RotDCS must be connected to the root in order for the geometry to be seen. In this sense, the Flt_Model serves as a data structure for holding the needed hooks for a correct representation in the scene graph.

4.7 Class Pfmr_Renderer

This class encapsulates the simulation main loop and collection of views. It provides member functions to initialize the simulation's rendering, obtain graphics resources to open windows on, and to perform the simulation's inner loop. The inner loop manages the Performer three process model, issuing calls to the Simulation and views on the main application, cull, and draw threads, if they exist. The Pfmr_Renderer class is designed to hide the C callback nature of the Performer interface, instead issuing calls to the ObjectSim objects to provide processing on the various threads and preserving the object oriented simulation design. The Pfmr_Renderer class also holds the common part of the Performer scene graph used to share the player geometry across views. Figure 19 shows a design for the Pfmr_Renderer class.

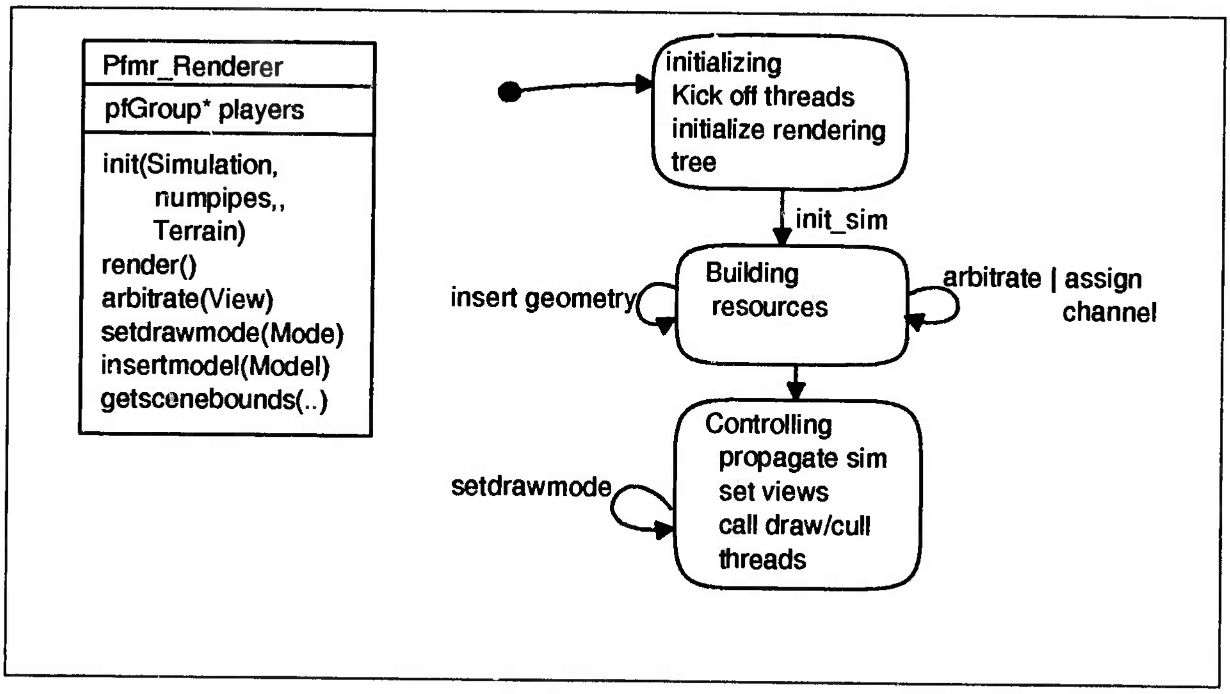

Pfmr_Renderer
pfGroup* players
init(Simulation, numpipes,, Terrain)
render()
arbitrate(View)
setdrawmode(Mode)
insertmodel(Model)
getscenebounds(..)
initializing Kick off threads initialize rendering tree
init_sim
Building resources
insert geometry
arbitrate | assign channel
Controlling propagate sim set views call draw/cull threads
setdrawmode

Figure 19 - Pfmr_Renderer Class Diagram

4.8 *Class Terrain*

The Terrain class was designed to encapsulate the environment of the simulation. This includes the ground (terrain database), the sky, the time of day, the weather, and other environmental variables. The terrain would thus become the simulated world where the players and interact. This class is responsible for both the simulated view of the environment and proper rendering of it in the graphical scene.

A terrain database can be generated many ways. Perhaps the simplest is to read in a model representing the terrain, similar to the way player representations are read in using the flight model class. Other ways exist to build terrain. These include building terrain representations using elevation data and cultural feature data, and generating dynamic terrain which is 'smart' enough to only render terrain near the viewpoint in high detail. The terrain class is designed to provide the environment and terrain database to the simulation independent of the way it was generated (by using subclasses for the various representations).

The Simple_Terrain class was the first terrain subclass developed for ObjectSim. It contains a sample implementation for the case when the terrain is just a simple model. Terrain implemented this way can be as simple as a single textured polygon or as complex as a multi-layered database including many detail levels embedded in the single file.

No matter how the terrain is implemented, ObjectSim requires that it be able to be translated to effect the proper viewpoint placement in the simulation. For the single model terrain, the terrain is translated just like any other dynamic object in the simulation. For terrain implemented in different ways, the terrain could be translated via pushing a matrix before drawing, or by attaching the terrain to a DCS node and then to the Performer tree, if this is possible.

The terrain class also contains the Round_Earth_Utils object for each application. This is an object which is configured when the terrain is built to provide flat to earth-centered coordinate transformations to the applications. This object is contained in the terrain class because the center of the terrain patch being used defines the correct transformation. The Simple_Terrain class has the Build_Terrain methods to read in terrain defined different ways. The Terrain class design is at figure 20 .

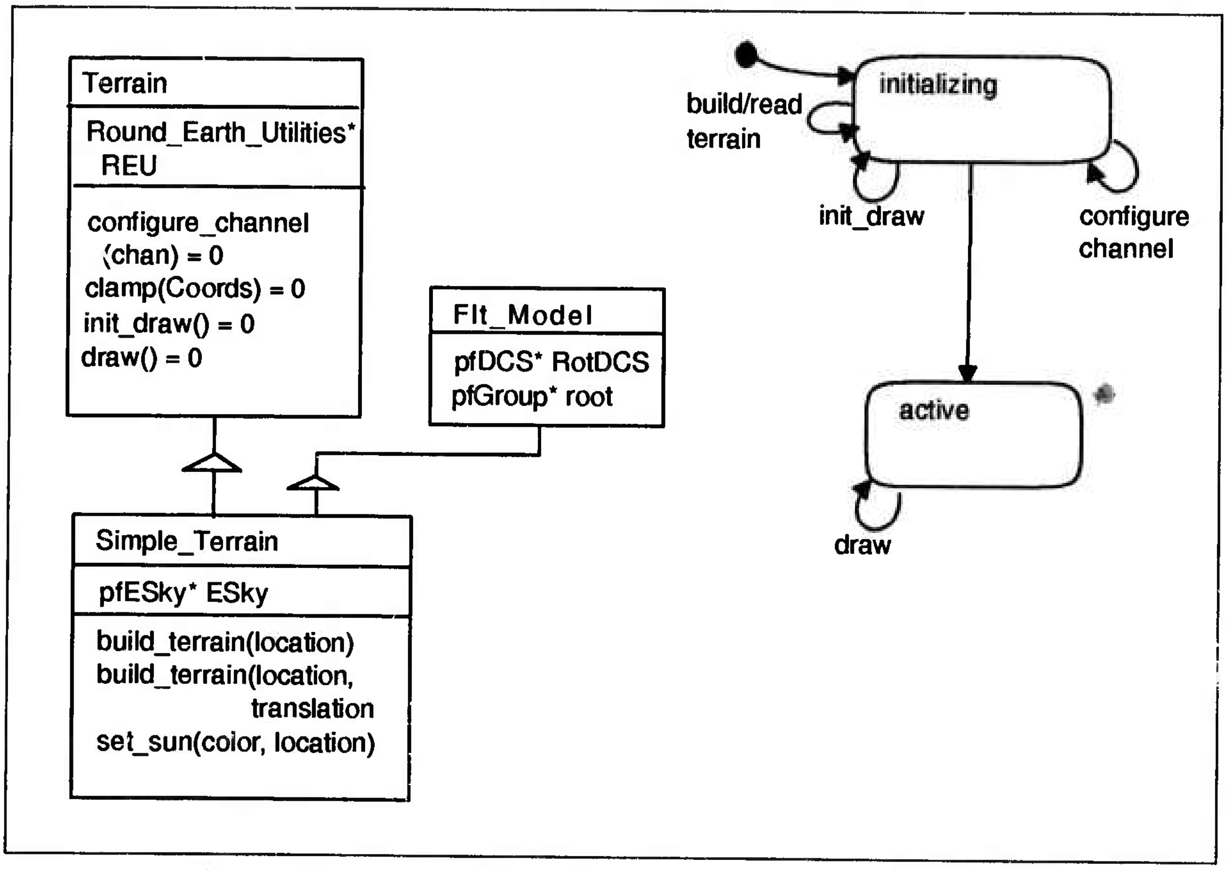


Figure 20 - Terrain Class Design

4.9 Class Modifier

Modifier is a superclass for a standard type of device interface in ObjectSim. A visual simulation often requires device inputs in the form of coordinates, both position and orientation. This type of input is used for spaceballs, head trackers, joysticks, and other devices. It can be used to change the viewpoint or move it, and also for other uses in the simulation. Modifiers abstract this type of device input, making possible interchangeable devices in a simulation. The view class includes a pointer to an abstract modifier, which is applied to the viewing calculation before the scene is rendered.

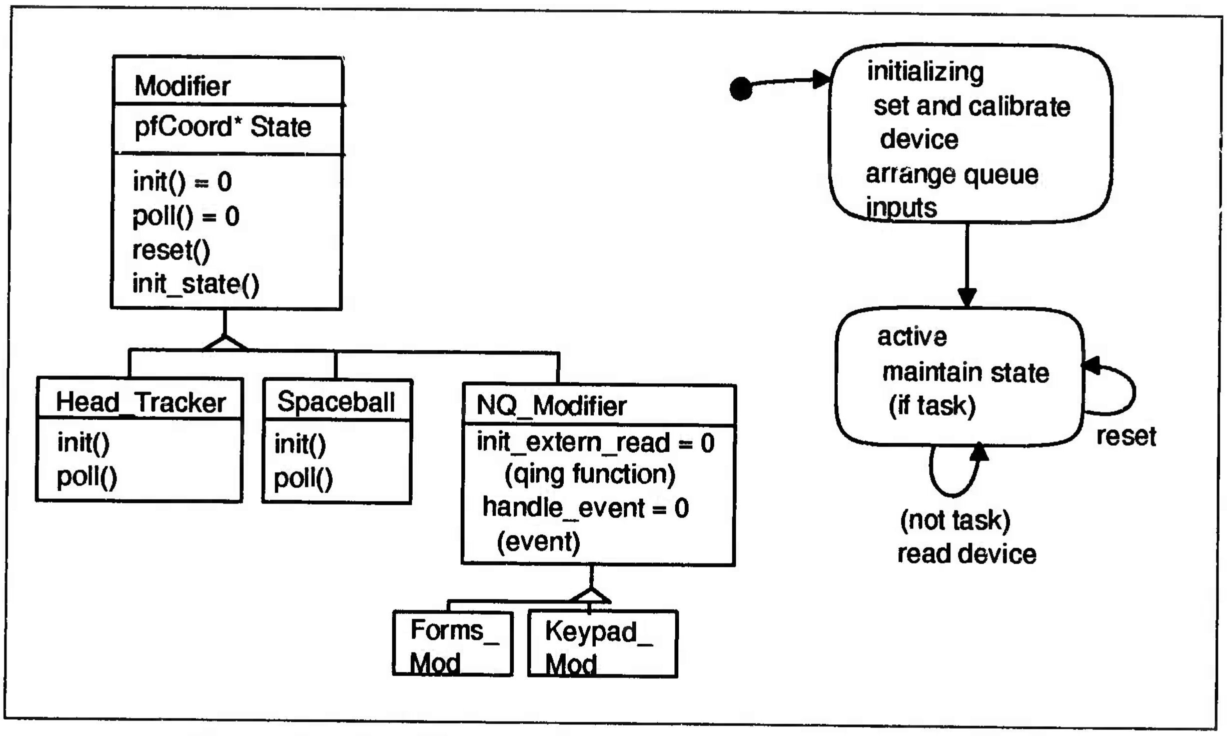


Figure 21 - Modifier and Subclasses Diagram

Each modifier is required to maintain a state, which is its current coordinate values. These values represent the value of the device inputs. These values can be read during a call to poll the device, or maintained in a local variable which is copied during polling into the visible data structure. In this way, the device input can be set up to work as an autonomous task.

Modifiers were designed to allow substitute capabilities for inputs which are difficult to set up in a development environment. For instance, a keyboard can provide the same state information as a head tracker, but the keyboard is much easier to use while a simulation is being developed.

For ObjectSim, a subclass of modifiers was developed for use with applications which maintain a queue of user inputs. The NQ_Modifier class gives a specification for a modifier which can work from input on a device event queue. Modifiers like this do not directly access the queue, but have a method which can test a particular queue input for applicability to that particular device. Figure 21 shows the modifier class design.

4.10 Initial DIS Interface

To interface to the Object Manager (Chapter 3), the Graphics Lab students developed a strategy in which each application had a network interface which returned a view of the world customized for that application. The Object Manager contained a short method for each application which returned the state of each entity every frame. The remote network entities in each application are called network players. Each application also maintains a class to manage the list of network players, called the network manager. The network manager interacts with the Object Manager via the method call for that application, bringing in the list of players for that application each frame.

Each network player is inherited from a common network player, the Base_Net_Player. The Base_Net_Player was designed to provide a basic network entity data structure. ObjectSim contains the Sim_Entity_Mgr class, which operates on Base_Net_Players and performs standard functions like appearance management on them. The Sim_Entity_Mgr class is also designed to manage appearance and network event handling for local non-network entities, such as terrain static features or local sending simulators. Because of this, the Sim_Entity_Mgr class also manages incoming network events for the application.

The Model_Mgr class was developed by one of the students this year. I incorporated it into the architecture as a method for managing the large model space found in a DIS application. Any of hundreds of different entity types can appear over the net. The Model_Mgr provides a way to list models by enumeration values in a separate file. This allows the DIS application to obtain the correct model for the enumeration value of the incoming entity and display it.

The Animation_Effect_Player was developed to show network events. It provides a class-based method for building and displaying an animation to model a weapons effect in a DIS simulation. Animations can be mapped to different weapons types or detonations, allowing a simulation to show a more realistic world view to the user.

The network interface subsystem developed for ObjectSim in shown in Figure 22. It shows a class model of the different parts of the interface and their relationships. Each application developed using ObjectSim uses a network interface like this to get its view of the world. The Object Manager is responsible for passing a dead-reckoned list of entities back to the application every frame in earth-centered coordinates, which each application has to convert into rendering coordinates. Also, entity statuses are passed whether they changed or not. In preserving the Object Manager as an abstract entity manager, this interface introduces inefficiencies into the process of receiving players described in Chapter 3. This led to an improved network interface design for ObjectSim.

4.11 Improved DIS Interface

The goals of the improved DIS interface are to maximize throughput of players from the network. I analyzed each portion of the pipeline players go through in coming off of the network, being dead-reckoned, undergoing coordinate conversions, and being represented in the scene. The results of this analysis are fully discussed in Chapter 6, but the improved design is here.

The major inefficiency in the network interface was the dead-reckoning of entities in earth-centered coordinates. This forced each application to convert each entity's coordinates into rendering coordinates each frame. Since a conversion involves 64 floating point multiplications, this became a major bottleneck each frame when the number of entities exceeded 100 or so.

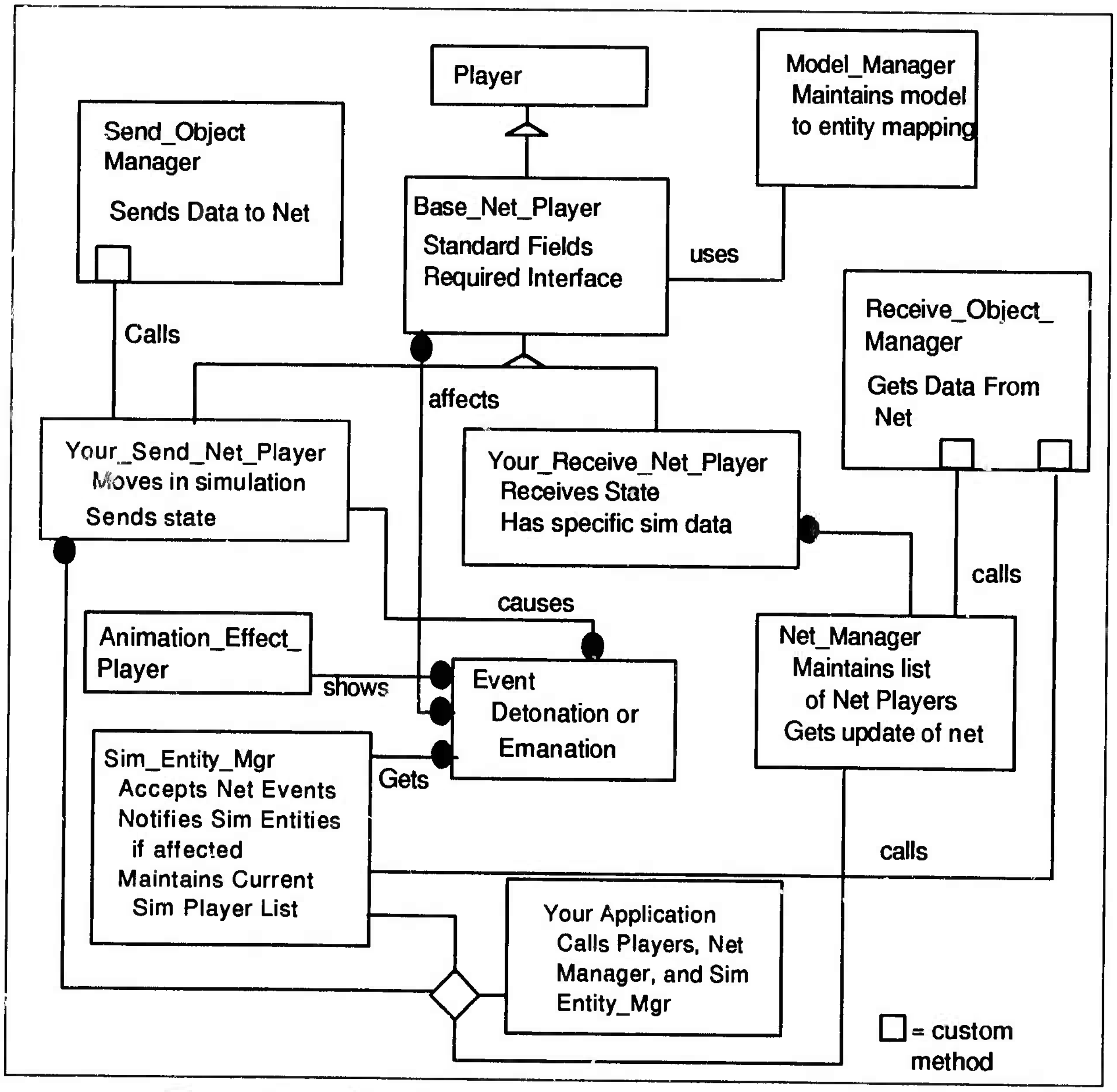

Player
Model_Manager Maintains model to entity mapping
Send_Object Manager Sends Data to Net
Base_Net_Player Standard Fields Required Interface
uses
Calls
Receive_Object_ Manager Gets Data From Net
affects
Your_Send_Net_Player Moves in simulation Sends state
Your_Receive_Net_Player Receives State Has specific sim data
calls
causes
Animation_Effect_ Player
shows
Event Detonation or Emanation
Net_Manager Maintains list of Net Players Gets update of net
Sim_Entity_Mgr Accepts Net Events Notifies Sim Entities if affected Maintains Current Sim Player List
Gets
calls
Your Application Calls Players, Net Manager, and Sim Entity_Mgr
□ = custom method

Figure 22 - Initial Network Interface Subsystem

In order to solve this, I designed a network interface which only communicates entity status when a new PDU is received for an entity. Then, I created a new class, Base_Net_Remote_Player, which has dead-reckoning (in rendering coordinates) built in. I designed the new player to allow the offending coordinate conversions to be performed on another thread, only when necessary due to a new PDU. I also built a new standard network manager class designed to work with the network players. This worked substantially better (see Chapter 6), and the network interface performance was improved.

The other problem was the design of the old network interface was not extensible. It was structured and not object oriented. The network managers and Sim_Entity_Mgr performed all of the processing for each player, rather than allowing the player to perform its own processing. In effect, the network players' methods were never called, so the network player was nothing more than a structure. The improved design has methods for the Base_Net_Player and the Base_Net_Remote_Player which perform the appearance management, dead-reckoning, and frame critical process reduction inside of the player, rather than having other classes responsible for these functions. This new design is more reusable and extensible, but further work can improve it more (Chapter 7). Figure 23 shows a design for the improved DIS interface.

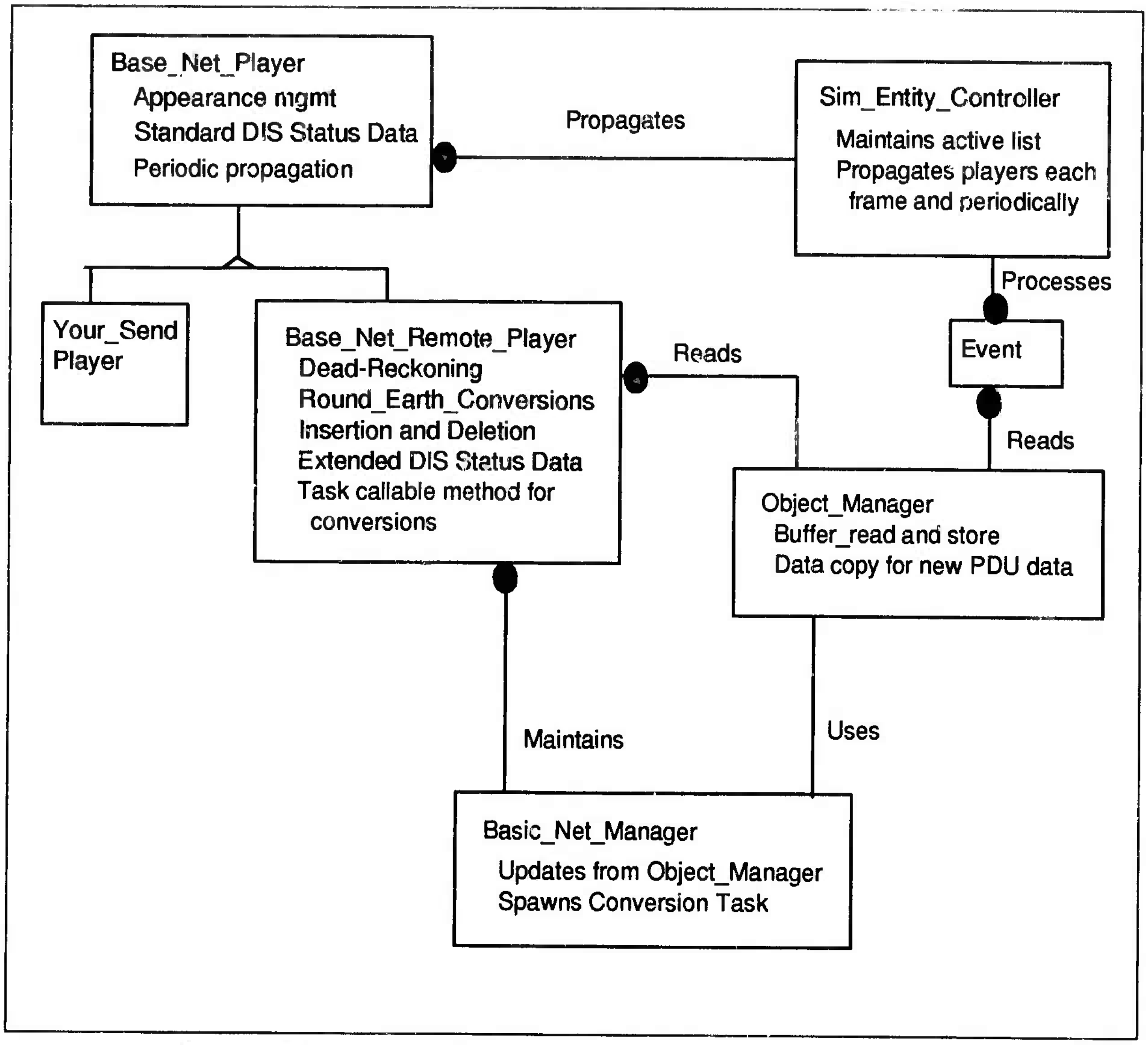


Figure 23 - Improved Network Interface

4.12 Overall Design

Simulations designed using ObjectSim all share the design information just presented. The high level model for all ObjectSim simulations is identical. Each simulation consists of a Pfmr_Renderer, a Terrain, at least one View, and at least one Attachable_Player. The simulation can also have one or more Modifiers. Players can have representations in the rendered scene (Flt_Models). If a simulation is a DIS compatible simulation, it will have a network manager and a collection of network players it maintains. Figure 24 is a high-level diagram showing this composition.

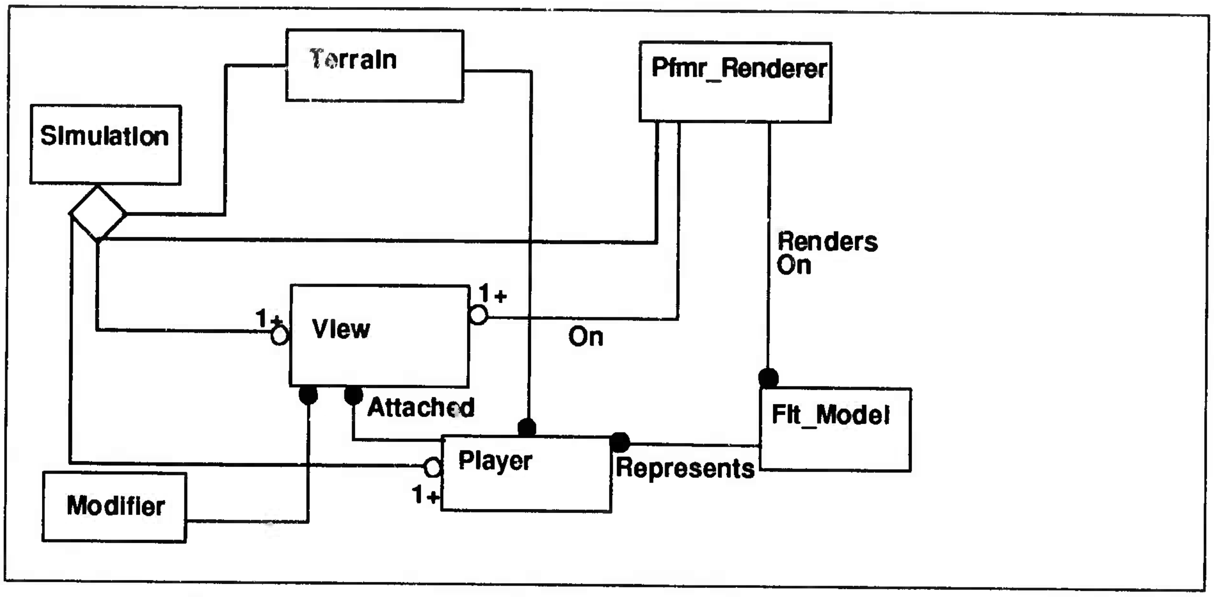


Figure 24 - High Level ObjectSim Composition

The next chapter discusses the process of creating instances of this model. These instances include four major thesis projects and some example and test projects. I will discuss how the model fared in use on these various projects, and show how the simulation designs are inherited from the design given above.

V *ObjectSim Applications*

5.1 Introduction

This section documents the process of aiding developers in using ObjectSim to create their simulations. Several student projects used the reusable design and created some quite successful research efforts using this approach.

Each research project was performed by one or more Masters Degree students, largely during a single year. This means each student had a limited amount of time to actually make a research contribution and produce or enhance a simulation. As the research goals for Graphics Lab simulations grew over several years, students faced a difficult task in developing capability to produce work which met sponsor research goals. The architecture developed in this research project attempted to provide the advantage of a proven design and components to these students. The various projects implemented using the architecture are listed in Table 4.

Project	Description - Research Goals
Virtual Cockpit	Immersive Flight Simulator - Research inexpensive alternative to domed simulator - Build man-in-the-loop DIS platform - Study modeling of advanced weapons system
Synthetic Battle Bridge	Immersive/Console based Commanders Eye view of battlefield - Research immersive interface techniques for Commander - Study expert computer situational analysis - Study situational representation techniques for battlefield - Study user interface techniques for effective user view control
Satellite Modeller	Immersive/console simulation for analysis of satellites - Represent single orbits or constellations - Study immersive interface into satellite simulation - Interface satellite data onto DIS simulations
Red Flag Display Tool	Console based debriefing/display tool for Air Force exercises - Study user interface for debriefing system - Interface live or recorded exercise data onto DIS

Table 4 - ObjectSim Projects In Graphics Lab

I will present the most basic ObjectSim simulation, implementing a fixed viewpoint over some terrain. Then I will present the instantiations for the Virtual Cockpit and the Satellite Modeller. I will also show a simplified object diagram for each, and discuss the level of reuse achieved and the corresponding student effectiveness.

5.2 Basic ObjectSim Simulation

This section will introduce the most basic ObjectSim simulation. This simulation will place the viewer in an upright position viewing a patch of terrain. The objective is to show how the combination of inheritance and the application framework provide functionality to the developer easily. The first block of C++ code shows the parts of ObjectSim we are using.

```
// Performer includes
#include <pf.h>

// ObjectSim classes
#include "pfmr_renderer.h"
#include "attachable_player.h"
#include "view.h"
#include "simple_terrain.h"
#include "simulation.h"
```

Each simulation must declare a class derived from the **Simulation** class, as discussed previously. For our example, the subclass will only be concerned with the following function entry points:

```
class Test_Sim : public Simulation
{
public:

  Test_Sim::Test_Sim();
  void init_sim();
  void propagate(int& exitflag);

  // Superclass Members
  // Terrain* Ter;
  // Pfmr_Renderer* Renderobj;

  Stealth_Player* Stealth;
  View* MyView;

};
```

As the design chapter discussed, most ObjectSim functionality is provided by instances of the class called **Player** and it's subclass **Attachable_Player**. Next is the basic stealth view player (no representation in scene), whose job will be to give a viewpoint into the simulation::

```
class Stealth_Player : public Attachable_Player
{
public:
 Stealth_Player();

  // Superclass members:
  // Flt_Model* Model;
  // pfCoord* Coords;
  // static Pfmr_Renderer* Renderer;
  // static Terrain* terrain;
  // pfVec3 base_offst;
  // pfVec3 base_rot;
  void init();
  void propagate(){};
};
```

These two inherited classes are needed to build the basic simulation. The Test_Sim class contains the ObjectSim objects which will be created and linked together to form the basic ObjectSIm model discussed in Chapter 4. The work is done in the constructor for the Test_Sim:

```
Test_Sim::Test_Sim()
{
  Ter = new Simple_Terrain();

  Renderobj = new Pfmr_Renderer();

  MyView = new View();

  Stealth = new Stealth_Player();

  // These must be done for one of the players
  // only.  They are static data members
  Stealth->terrain = Ter;
  Stealth->Renderer = Renderobj;

  // Also done for only one view
  MyView->Renderobj = Renderobj;
  // Called on each view
  MyView->alloc_shared();
}
```

With this constructor, the basic model is formed. Test_Sim has instantiated its Terrain as Simple_Terrain (one flight file). It has one View and one Player. The constructor is called by the main program, which also instructs the Pfmr_Renderer to perform the initialization and to enter the main loop. The code for the main program is:

```
int main (int argc, char *argv[])
{
  // The simulation
  Test_Sim* MySim;

  // Initialize Performer
  pfInit();

  MySim = new Test_Sim();

  // Cause the muultithreads to be kicked off and
  // Opens the default window. Calls init_sim()
  MySim->Renderobj->init(MySim,1,MySim->Ter);

  // Cause the main loop to execute.  Will not
  // return.  Calls propagate()
  MySim->Renderobj->render();

  exit(0);
}
```

Test_Sim will customize the model for its own purposes by fulfilling the abstract function calls init_sim and propagate. In this case, the initialization code looks like:

```
void Test_Sim::init_sim()
{
  int found;

  // Read the terrain file and place in
  // Flt_Model number one
  Ter->readmodel("terrain.flt",1,found);

  // Initialize the player functionality here
  Stealth->init();

  // A new view on pipe 0.  Cannot be done until
  // init_sim (after Performer fork)
  MyView->new_view(0);

  // Connect this view to the stealth player
  MyView->attach_to_player(Stealth);

}
```

Test_Sim has initialized its View and attached it to its single Player. The Simple_Terrain had been read in and is ready for rendering. The player has been initialized. Now, propagate will be called once per frame to perform the simulation. The code is:

```
void Test_Sim::propagate(int& exitflag)
{
  // This call will propagate the attached
  // player regardless of which player is
  // currently attached to the view.
  MyView->get_attached()->propagate();
}
```

The only processing done each frame (by Test_Sim) is to propagate whatever player is attached. The viewpoint location and movement are entirely defined within the Stealth_Player attached to the View. The Stealth_Player code is:

```
Stealth_Player::Stealth_Player()
{
  //Location, Orientation member
  Coords = new pfCoord;
}


void Stealth_Player::init()
{

  // Set initial position and orientation
  PFSET_VEC3(Coords->xyz, 0.0f, -300.0f, 100.0f);
  PFSET_VEC3(Coords->hpr, 0.0f, 0.0f, 0.0f);

  // These are for an attachable player.  They
  // define a position and orientation within
  // the object where an attached viewer is placed
  PFSET_VEC3(base_offst, 0.0f, 0.0f, 0.0f);
  PFSET_VEC3(base_rot, 0.0f, -15.0f, 0.0f);
}
```

The propagate member was declared null in the class definition. The Stealth_Player will constantly sit in one place and do nothing. The Test_Sim simulation will just show a fixed view of the terrain.

That completes the implementation of the basic simulation. Notice that the programming complexity and level of effort required to get started was small. The combination of Performer and ObjectSim requires the developer to master little of the complexity of the design before starting to work. To make a more interesting simulation, the developer must provide more interesting players to go into the simulation, but the basic architecture stays the same.

5.3 Virtual Cockpit Instantiation

The Virtual Cockpit was implemented as a player incorporating an aerodynamic flight model with various display modes and weapon systems. Figure 25 presents the design of the subclasses of ObjectSim classes used for the simulation.

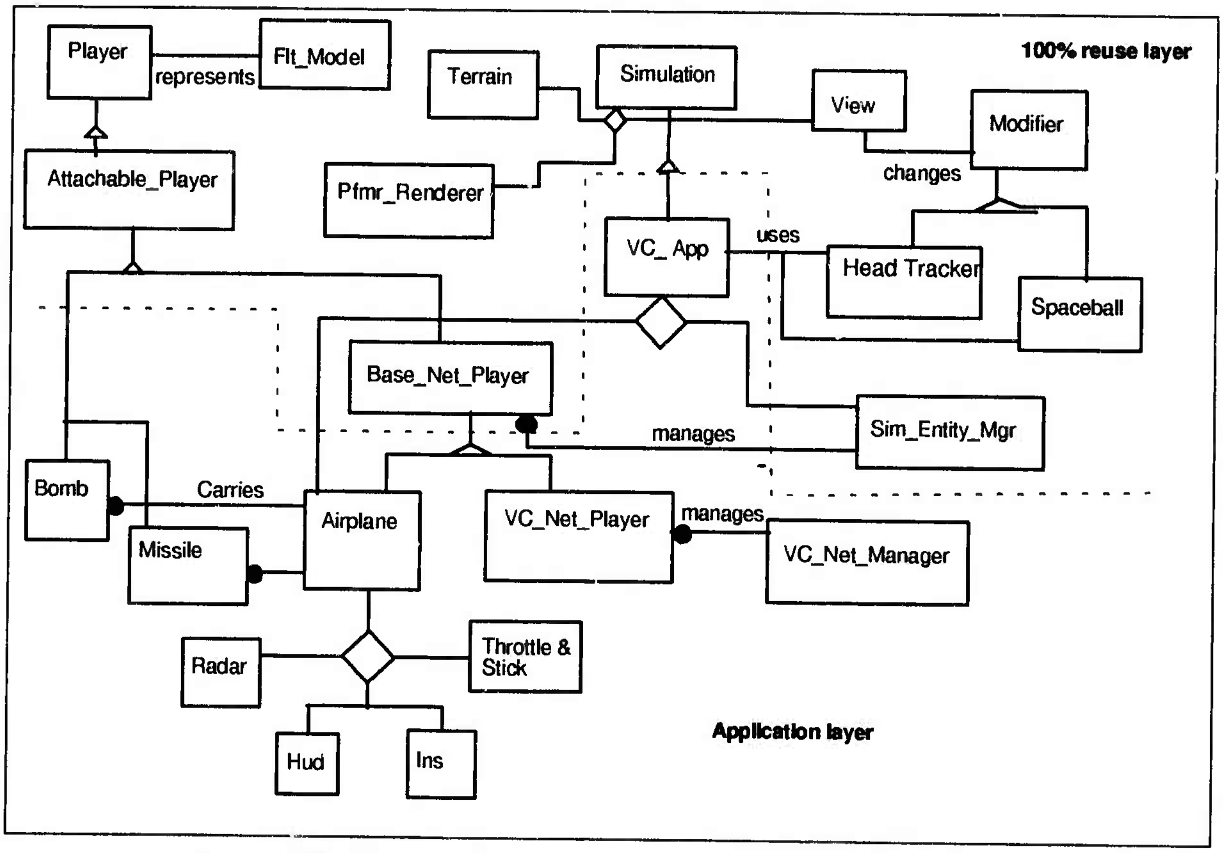


Figure 25 - Virtual Cockpit Partial Design

Figure 25 is not intended to show complete design information, but rather to show the types of problems faced by the Virtual Cockpit research students. The diagram shows that the Virtual Cockpit designers were able to concentrate on modeling an airplane and its weapons and subsystems, rather than writing a simulation design from scratch and re-implementing common functionality. Below the dotted line on Figure 25, the Virtual Cockpit designers were responsible for designing and implementing the classes. Above the dotted line, they were simply reused. Besides the basic ObjectSim model, the VC also reused the network manager subsystem for receive network entities. The level of reuse achieved with the ObjectSim design allowed the thesis students to focus on such problems as accurately simulating a modern weapons system, and to spend less time on non-research-specific details.

The code example below shows the class definition for the **Airplane** class, which encapsulated the other aircraft subsystems

```
class Airplane : public Base_Net_Player
// Base Net Player is used for remote and local
// DIS players
{
 // Application thread calls for airplane
 void init();
 void propagate(); // Move one tick
 // Draw thread calls for airplane
 void init_draw();
 void draw();  // Do any drawing for one frame

 // Base Net Players must provide a
 // function to accept damage from remote
 // weapons
 void accept_damage(pfVec3 location,
                    munition_warhead weapon);

 // The aerodynamic Model
 State* AcftState;
 // Throttle and stick class
 // (Hands On Throttle And Stick)
 HOTAS* MyHOTAS;
 // The Heads Up Display
 HUD* MyHud;
 INS* MyINS;
 Weapons_Controller* MyWeapons;
  .
  .
};
```

The above class definition shows how the Airplane must provide functions to perform certain processing at certain times. This is the *contract* each player has to fulfill to exist with the rest of the architecture. The init, draw, draw, and propagate functions are the calls used by the architecture to control the player. Many of these calls may be left out, if there is no need for them. The superclasses for the player contain default functions. This illustrates one of the best complexity management techniques found in Object Oriented languages. A superclass can provide a standard set of functionality, and the subclass can override this if necessary. Therefore, an architecture like ObjectSim can provide an almost complete application, if the application needs little customization. As the application needs to customize parts of the architecture, it can do so by declaring custom methods as appropriate.

Note how the designers were able to utilize a top to bottom object oriented design for the airplane. The ObjectSim architecture was designed to facilitate object oriented simulations. This is a key to the complexity management features of ObjectSim.

5.4 Satellite Modeller Instantiation

The Satellite Modeller was designed as a simulation allowing a set number of satellites to be propagated around the earth in either real time or time step based propagation. It includes a subclass of Terrain called Space_Terrain, which implements the terrain as a rotating earth model with the moon, stars, and a realistically positioned sun light source. The propagator code was based on code from the Space Operations program at AFIT.

The interface requirements for the Satellite Modeller were for a highly interactive GUI allowing user flexibility. The simulation is configurable for number of constellations, number of satellites, and viewing modes. Also, the simulation is designed to take advantage of the device-independent nature of ObjectSim by providing interfaces to a head tracker for an immersive operation mode. To accomplish this, an off-the-shelf GUI toolkit was used to design and build the interface. The ObjectSim simulation was designed to take the interface state and control the simulation, viewpoint, and other factors.

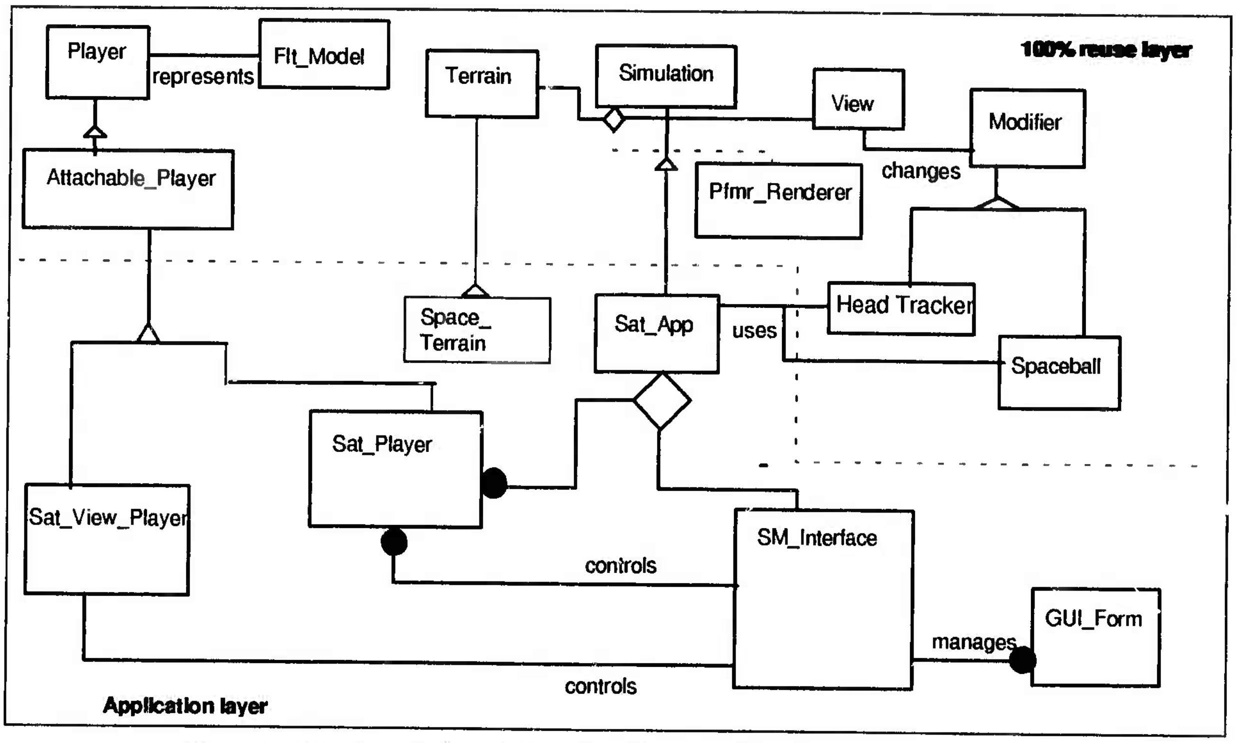


Figure 26 - Satellite_Modeller Partial Design

Notice the similarity of Figure 26 to Figure 25. Each ObjectSim application will follow the same basic design, with differences showing up in the unique processing required for each. The ObjectSim layer provides a template for each simulation and objects for each simulation to use.

5.5 Architecture Growth

Each of the four major applications implemented with ObjectSim added to and stretched the breadth of the requirements the library covered. The multiple channel simulations (RDT, Virtual Cockpit), required the library to correctly manage the shared scene graph and call the threaded member functions correctly. The RDT implementation paved the way for integration of the GUI interfaces into the applications. For each new problem solved, the major challenge was to fit it into the architecture while preserving a good design and not perturbing the existing code too much.

Several project developers completed reusable pieces which were able to be successfully used by other parts of the architecture. The RDT contributed the design method for integrating the forms, which were later used on the Satellite Modeller and the SBB. The SBB developer contributed the Model Manager, a way to manage the large model sets found in large simulation applications. The SBB also saw the design of a set of reusable interface objects which use innovative transparent representations to make them useful for immersive/console simulations. The Virtual Cockpit project contributed the GraphText implementation, which provides a nice stroke vector capability for in-scene text drawing. The Satellite Modeller project pioneered the use of a new loader for geometry as an alternative to the Multigen Flight format.

5.6 Effectiveness

The architecture provided considerable help for some of the development problems observed in our initial analysis. Table 5 enumerates some of the effects of our object oriented approach

Problem	Observed Effect
No one responsible for std components	ObjectSim provides a toolbox of components designed to fit together
No one responsible for evaluating and integrating new outside code	ObjectSim developer performed analysis in this area
Existing components hard to understand	Students had to understand ObjectSim approach, but less detail than before
No standard library locations, CASE, or CM to support large developments	ObjectSim captures a lot into one library, but does not solve CM problem
No design methodology adopted	ObjectSim facilitated a reasonable Object Oriented approach to simulation design
Students don't have time to become well grounded in languages or design methods	ObjectSim provides design, reduces the amount of code necessary for success
Simulation projects didn't have stable requirements	ObjectSim allowed quicker maintenance turnarounds; simulation design stable

Table 5 - Development Improvements with ObjectSim

This isn't to assert that merely building a library will solve all development problems in a particular setting. However, the existence of the application framework did help the lab's process. By using the application framework strategy, the developers could actually build rapid prototypes of their applications which became the base for final implementations. Since the final versions of each application also use the library, typically these prototypes were not throwaway, but became the basis for the final software.

With this kind of ability, developers were able use a risk-based approach to manage their development. They could approach the most risky parts of their development first, knowing that an architecture and its inherent engineering knowledge would provide much of the standard low-risk capabilities their application needed. This, I think, is the basis for the positive effect ObjectSim (and Performer) had on this years research prototypes.

The flexibility and success ObjectSim had owes much to the capabilities developed by the Iris Performer Group at SGI. Performer does provide a lot in terms of library calls and an object based C interface into a visual simulation. The contribution ObjectSim made was to take the abstraction of Performer one level higher, and to package the capabilities Performer provides into a reusable simulation.

The projects using ObjectSim and Performer have amassed quite a track record this year. The Virtual Cockpit has run on the console, in a Head Mounted Display at SigGraph, and on a Barco Multi Screen Display at the ARPA Sim Center. The Synthetic Battle Bridge has run at the console, at SigGraph in a boom, and at the ARPA Sim Center on a big wall. The RDT has been taken on the road to Nellis AFB and has demonstrated that a single AFIT student working with some good tools can replicate years of effort by contractors on a viewing console. The Satellite Modeller has impressed its user community and provided some striking visual images at the ARPA Sim Center and at AFIT demos.

In support of the DIS research goals and the Warbreaker program, AFIT has provided a man-in-the-loop flight simulator with multi-mode operation, a stealth console with intuitive interfacing capabilities, Red Flag DIS data on the network and a console to view it on, and satellite data on the network with a simulator to generate and view the satellites in orbit. Just the DIS applications in the AFIT lab alone form a good testbed for some of the concepts DIS and Warbreaker are meant to explore.

5.7 Lessons Learned

All of these developments weren't without some problems. One major hurdle was the C++ language expertise necessary to successfully use the architecture. Students were not prepared for full object oriented programming and using the methods of customization required for the architecture. Although the amount of code necessary to customize the design was small, deciding where to fit the code in was not as easy. Second, errors resulting from dependencies in the code and from the Iris Performer basis for the code were not always intuitive. Third, short timelines for development meant the developers often did not have adequate time to learn the architecture before pressing ahead.

As noted earlier, the lack of Configuration Management still plagued the Graphics Lab. Large simulations require data files, models, libraries, and many other parts. Often, these all didn't work together as they should, because everyone working in the lab didn't maintain a baseline.

C++ on the Silicon Graphics machines provided us with some interesting moments. For most of the thesis cycle, we struggled to find a way to make the objects behave correctly when they were allocated from shared memory using the SGI shared memory features. The run-time did not preserve the virtual function table for member functions when the parent was allocated by casting a shared memory allocation call back to the object type. Using a call to the Performer allocation routine, we tried:

```
class My_Player : public Player
{
  void init()   ///VIRTUAL FUNCTION IN PARENT
  void propagate()'
}

My_Player* Test;
.
.
---- in some function ----
// ALLOCATE TEST FROM SHARED MEMORY
Test = (My_Player*)pfMalloc(sizeof(My_Player),
                            pfGetSharedArena());
.
.
Test->init();   //BOOM - Segmentation Fault
```

As a result of this problem, the simulations using shared memory for input or output interfacing requirements had to allocate shared memory structures, or treat many of their objects as structures instead of true objects with callable member functions. Designs evolved with classes intended as member functions for other classes, to get around the problem. Needless to say, the cleanness of the object oriented designs began to suffer under these restrictions.

Two students finally came up with a solution to this after working on the problem for a while. The key turned out to be using a constructor for the object which handles the shared memory allocation. The new code segment looks like this:

```
class My_Player : public Player
{
  My_Player()   ///CONSTRUCTOR
  void init()   ///VIRTUAL FUNCTION IN PARENT
  void propagate()'
}

My_Player* Test;
.
.
My_Player::My_Player()
{
  this = (My_Player*)pfMalloc(sizeof(My_Player),
                              pfGetSharedArena());
}

---- in some function ----
// CALL CONSTRUCTOR
Test = new My_Player();
.
.
Test->init();   //OK, DOES NOT FAIL
```

So, the final work done for the architecture uses this style of shared memory object rather than the old style of shared memory substructures.

One other lesson I learned was the importance of the development environment to the success of a project. Many times in the lab we hurt our cause because of lack of change control, lack of tools on different machines, and lack of CM for the many components of a large simulation. A typical problem might see a developer change a header file another developer depended on. When an application was linked which contained two libraries built with two versions of a particular header file, the result was usually a non-intuitive crash. A development environment with as many people and machines as the Graphics Lab at AFIT needs good procedures to keep occurrences like this from slowing productivity.

The next section contains some measurements I made of the software and the development process in the Graphics Lab. I will show metrics involving performance and reuse and discuss the software work in the Lab relative to these issues.

VI Measurements and Performance

6.1 Graphics Lab Metrics

Any software effort requires some measurements to quantify the work developed. Metrics can answer questions about reliability, maintainability, performance, testing success, or development effort expected or expended on a project.

With the developments in the Graphics Lab, the most important concerns are performance and reusability. Performance has always been important in real-time, interactive simulations. Slow performance translates to slow visuals, which can significantly hinder a simulation's believability. Reuse is always desirable because it translates into improved productivity. However, to be effective, students must be able to take the reusable pieces without redeveloping them, or they will bog down in unnecessary detail. The measurements I took were mostly in these areas of performance and reuse.

I took the philosophy of measuring those things which I felt would yield some good data for analysis and recommendations. I included with the measurements recommendations based on any conclusions which can be drawn or inferred from the measurements.

6.2 Reuse Metrics

Reuse metrics have always presented a difficult problem to software developers. Typical approaches will attempt to show the lines of code reused versus the lines of code developed for a particular problem. When trying to consider the effectiveness of reuse on a particular problem, this can be inadequate.

ObjectSim can be thought of as a composition system. It uses the composition medium, the C++ language, to specify components in the visual simulation domain. These simulations are also composed of other reusable pieces, such as math libraries, Performer, a GUI library, string and file utilities, and other assorted pieces. Each simulation really can be thought of as an integration effort to bring all of these pieces together. In a setting like this, metrics can be especially difficult. I chose some high level metrics which I felt would show the reasons behind the Graphics Lab success this thesis term.

The first set of metrics (Table 6) for our Lab attempts to quantify the effectiveness of the ObjectSim approach on *developer focus*, or the percentage of design, implementation, and testing time the researchers applied to *research specific* problems while creating their ObjectSim applications. Research specific problems are those which are not concerned with the mechanics of the simulation (e.g.; how do I put my view here?), but more concerned with the domain the simulation is a part of (e.g.; how do I model a radar?). These numbers are based on discussions with the developers, who attempted to quantify the time they had spent on their thesis efforts.

Simulation	% Design Time research specific	% Coding Time research specific	% Lines of code research specific	% Testing time research specific
Virtual Cockpit	80	70	75	65
Satellite Modeller	80	90	85	60
Synthetic Battle Bridge	70	75	70	75
Red Flag Remote Display Tool	90	80	75	90

Table 6 - Developer Time Expenditure Survey

Past students were not available to interview about their time expenditure. In the Virtual Cockpit and Satellite Modeller cases, judging from the functionality the developers were able to achieve this year versus last year, my best estimate is that the developers were significantly more domain focused this year. Whatever the improvement, the student developers do feel the combination of ObjectSim and Performer has freed them up to focus most of their time on the specifics of their research.

The next set of metrics show the level of cooperative development. They show to what level developers of multiple ObjectSim simulations were able to share work. The numbers are percentages of code developed for one simulation and reused on another, either by use as a design and code template or by directly using a class. We are not including reuse from sources other than other ObjectSim simulations. Table 7 presents another set of data which is difficult to pin down, because I developed ObjectSim by incrementally implementing the applications and capturing reasonable pieces of reuse from them as they were developed. The applications are presented in the order of their implementation using ObjectSim. Percentages are developer estimates combined with other developer data to assess the overall reuse effectiveness.

Simulation	% of code contributed	% of code reused from others
Virtual Cockpit	50	10
Red Flag Remote Display Tool	30	50
Synthetic Battle Bridge	15	50
Satellite Modeller	5	70

Table 7 - Software Sharing in Graphics Lab

These metrics, which indicate some effectiveness of the ObjectSim approach, are presented with the caveat that the developer of ObjectSim was around to aid development of these first four applications. To realize this types of benefit, a developer must have a good understanding of the architecture. It thus becomes important to provide design information and, preferably, a User's Manual for developers. Also, a responsible party for overseeing and guiding development, such as a faculty member or reuse specialist, will facilitate this kind of reuse. Appendix A is a User's Manual I developed for the ObjectSim framework.

The last set of reuse metrics does not really involve numbers. What I will attempt to show is the way in which each successive ObjectSim application contributed functionality to the library which was picked up by another application. I will do this by listing classes or problem solutions developed for one application and then reused by another. Minor subclasses are not listed .

Application	Classes/Capabilities contributed	Classes/Capabilities Reused
Virtual Cockpit	Simulation, View, Flt_Model Simple_Terrain, Pfmr_Renderer, Modifier Polhemus_Modifier, Sim_Entity_Mgr, Player Base_Net_Player, Event, Animation_Effect_Player, Round_Earth_Utils	Model_Mgr, Multichannel display, Network Manager
Red Flag	Multichannel Display, Network Manager, Independent Scaling, Spaceball (SGI), Forms bridge, Trails queue	Simulation, View, Flt_Model, Simple_Terrain, Pfmr_Renderer, Player, Base_Net_Player, Model_Mgr, Event, Fonts Animation_Effect_Player, Round_Earth_Utils
Synthetic Battle_Bridge	Model_Mgr, Button, Multichannel boom display, Locators, Fonts,	Simulation, View, Flt_Model, Simple_Terrain, Pfmr_Renderer, Player, Base_Net_Player, Event, Animation_Effect_Player, Sim_Entity_Mgr, Network Manager, Round_Earth_Utils, Forms bridge
Satellite Modeller	Space_Terrain, Stars	Simulation, View, Flt_Model, Pfmr_Renderer, Player, Round_Earth_Utils, Forms bridge, Locators, Fonts, Trails queue

Table 8 - Capabilities Contributed or Reused

This listing really shows two important facts about the Graphics Lab developments this year. First, the visual simulations do share a common design, as evidenced by their reuse of the

ObjectSim architecture. Second, this common design fostered working together. The various developers helped each other many times, because they were all working on similar developments. The developers collectively created a capable set of engineering solutions for visual simulation problems.

The lab lacked tools for measurement of the software. We were able to compute some lines of code (LOC) measurements to give an idea as to the size of the projects. The LOC counts varied for the developments using ObjectSim from around 5000 to over 20000. For the Developer's Guide simulations, the average LOC for the instantiations was 900 lines. The complexity is found mainly in the understanding required for inheritance based C++ programming. The actual algorithms in the architecture are not complex. The code mainly consists of function calls (to other ObjectSim objects, to Performer, or to other libraries). Some of the applications contain more complex algorithms, but these are concerned with their research specific problems, not their simulation interface.

6.3 Performance Measurements

The simulations in the Graphics Lab have many factors which determine their performance. I will present some measurements of frame rate, and relate these back to the various factors which contributed to the speed of the applications. These factors will include geometry management, overlay GL drawing management, device handling, multiprocessing, non-frame-critical processing, network interface efficiency, shared memory locking, compiler optimization, debug logging, and C++ language constructs.

The method I followed to obtain these performance measurements and recommendations was to maintain a controlled testbed. This was the ObjectSim application I used to implement each new piece of ObjectSim functionality into the library. As the functionality was implemented, I instrumented the application, usually with standard Performer performance data, to see the affects of my changes. In this way I determined the performance characteristics of various methods of doing things in a simulation. More precise data would have come from profiling the application, but I will leave that to future work.

6.4 Geometry Management

Geometry management is the process of reducing the number of polygons sent through the graphics pipeline while maintaining a realistic simulation. In performance measurements we did, this factor always impacted performance more than others. Geometry management has many facets, mostly dependent on the hardware capabilities. Hardware capabilities are mostly outside the scope of my software library, so I established some baselines for the tests. The performance measurements were taken on a four processor, single Raster Manager Onyx, which means that texture in the geometry was not a factor. Each simulation used the standard three processors for the Performer threads. Frame rate control was set to free run, without waiting for frame boundaries. Each application ran in the same size window. On other machines, the frame rate will vary depending on the hardware pipeline, the texture in the geometry, and the number of processors, and the number of Raster Managers in the machine. In a test with an unsophisticated simulation (just moving a viewpoint around with a rudimentary interface and little computation), I obtained the measurements in Table 9 as I varied the parameters.

These results illustrate the first performance result we have found in the lab: the geometry management, and specifically the reduction of polygons, is the biggest determinant on the performance of an application. This doesn't mean an application is guaranteed good performance if it does this, because applications can fall into many other pitfalls. It means that if an application doesn't manage geometry complexity, it will suffer a performance penalty. Geometry management is primarily achieved by structuring terrain and models with levels of detail which decrease to low-polygon representations as the viewpoint goes away from them.

Geometry Parameters	Performance results
2000 poly, no LOD, no tub	30hz steady
9000 poly, No LOD, no tub	18hz low, 30 hz high, dependent on view
2000 poly, No LOD, viewer in tub	15hz low, 25hz high, dependent on view
9000 poly, No LOD, viewer in tub	12hz low, 25hz high, dependent on view
LOD Terrain, no tub 100 polys in farthest LOD, LOD cells 10 km on a side with 4 LODs	20hz low, 25hz high, frame rate more steady
Same as before, in tub	15hz low, 20hz high, frame rate steady.

Table 9 - Geometry Complexity Performance Effects

The second point illustrated above is the fact that an immersive simulation with geometry drawn around the viewer will cause a penalty. This is because geometry around the viewer is pixel intensive (each polygon drawn covers more pixels). This has the affect of increasing the drawing time of the application. Immersive geometry should be used only when necessary to the realism of the simulation.

6.5 Overlay GL Drawing

Overlay GL drawing involves rendering two dimensional or three dimensional GL polygons or text after the Performer geometry database has been rendered. This type of drawing can also impact performance, and this section will discuss ways to manage this impact.

The most important point about drawing is that any resources consumed by the GL drawing add to the time required to render geometry, and geometry drawing is typically the bottleneck for visual simulation. Therefore, drawing should not involve expensive calculations done on the draw thread. Instead, the drawing parameters should all be computed and stored, so all the d[illegible] thread has to do is the GL rendering. This storage technique typically works well with a protected data structure, since two different tasks will be accessing the data.

In our tests, 2D GL or text drawing typically did not affect the frame rate more than 1-3 Hz. The exception to this was when an application used Forms, a GUI package based on GL. Applications typically slowed down by over 5 hz when a complicated form was displayed. All of the forms processing, including the non-drawing background data structures, is performed on the draw thread. Even so, the Forms applications maintained a respectable frame rate despite the GUI overhead. The RDT, with a complex form and two separate drawing channels and over 100 moving entities still maintained a steady 15hz frame rate. These numbers reflect the effect from Forms drawing, not Forms event handling (user input).

6.6 Device Handling

Device handling refers to the process of obtaining user input from a device. In the Graphics Lab, we have used many input devices, including the keyboard, mouse, spaceball, head tracker, boom, and throttle/stick. This section will discuss the possible performance penalties of these devices, and the benefits of using tasks to avoid long reading delays in time critical threads.

The keyboard, SGI spaceball, and Forms GUI all operate using the standard device queue available on SGI machines, which can only be read by the draw process. This implies that all input must be taken by the draw process when these interfacing techniques are used. For performance purposes, expensive actions based on these inputs should be deferred to the application process. Also, the queue handling is important. A performance penalty can result if an application tries to requeue events it reads, so a later queue read can use them. This can result in many useless events being requeued, and long event reading loops. A better design is to centralize the queue read in one place, and pass the events around to each interested object in the program, so they can act on them if desired.

Devices such as the head tracker or the throttle and stick present different problems. Many times these devices can be slow to respond to requests for their value. In one test, the throttle/stick used for the Virtual Cockpit accounted for 18% of the processing on the application thread. To solve this, these device interfaces can be implemented as tasks, which operate asynchronously and maintain a state which the application then reads.

6.7 Multiprocessing

In discussing multiprocessing, I am referring to partitioning the application up into tasks which operate asynchronously or with synchronization. The processes are designed to avoid bottlenecks which can slow the throughput of geometry. Multiprocessing is a common technique to attack complex computational problems like visual simulation.

Each ObjectSim application inherits a design intended to take advantage of the Performer three process model for running a simulation. The simulation is divided into application, cull, and draw threads, a design based on the time-critical nature of the drawing process, which must keep the hardware geometry pipeline full as much as possible. This level of multiprocessing is available to even the simplest ObjectSim application.

The more complex an application becomes, the more need there may be to add more multiprocess threads. For DIS applications with a receive network interface, the part of the application which reads the incoming message buffer is typically a task, since messages are lost if the buffer is full when they arrive. Device reads are other typical areas for tasks, since they can involve blocking while waiting on a read. Expensive calculations can often be multiprocessed, providing a speedup for time-critical processes which use the results of the calculations.

6.8 Frame-Critical Processing

Every frame a visual simulation performs many necessary functions to compute a viewpoint. Often, the correctness of the simulation depends on certain calculations, and these must be done each frame. An example is computing a time step of the Virtual Cockpit and moving the airplane. A common mistake, however, is to try and handle all processing for a particular thread in one thread of control on the application thread, even when the processing could wait for more time to be available. The problem can occur when expensive floating point calculations or device reads are used. Multiprocessing was one solution to this problem. This section will explore other solutions.

One solution to this problem is to only do the offending calculations periodically. If the periodic calculations can be tasked, this might work. Sometimes, though, the periodic calculation involve data structures which are only visible on one thread, such as the Performer tree, which is only accessible to the simulation on the application thread [SGI, p]. But, if the expensive work is done periodically on the application thread, during the work a slow frame occurs, resulting in a jerky appearance.

A better solution we came up with in our lab was a technique called subdivision. Subdivision is a method of dividing a set of work over n iterations by only doing 1/nth of the work each frame for n frames. This works well for arrays of objects which must have some calculations performed, if the calculations are not critical to the realism of the simulation. In a test of subdivision with radar calculations, the Virtual Cockpit subdivided over 10 frames intersection testing and coordinate conversion for 300 objects in its radar. The result was an improved frame rate, from 10hz to 15hz, just from this one change.

6.9 Shared Memory Locking

Shared memory is necessary when using multiprocessing to communicate between the tasks. Often, this memory needs protection from mutual update or reads of incomplete data structures. On the SGI machines, applications use locks associated with their shared memory for this purpose. A common technique is to allocate a large shared memory block for sharing between two processes and to lock it when one or the other wants access. A problem arises when one process is slow during its access, because the other one may be waiting on the lock for too long. Again, if one of the processes is time critical, the all-important frame rate can suffer.

Many times, the whole shared memory area does not need to be locked for the concurrent processing. If the concurrent processes are iterating over some array or list data structure, only accesses to the same element must be locked. This can be accomplished by assigning different semaphore locks to different portions of the data structure, or by dividing some number of semaphore locks over a large array. Then, the concurrent tasks can avoiding blocking each other out unnecessarily.

The other point about locking is that, if a time critical process cannot wait, it should test the lock first before trying to wait on it. This means it might wait a frame or two for an update from the shared memory area, but this is often superior to waiting on a lock that will slow down the frame rate.

6.10 Network Interface

The network interface design is particularly important from a performance perspective. This is the portion of time a DIS application spends maintaining the correct state of the entities coming over the network. This usually involves reading the network buffers, accepting or rejecting the entity as important to the current simulation, dead reckoning the entities, converting the incoming coordinates into the simulation coordinate system, and correctly displaying the models. When the number of entities increases into the hundreds or thousands, the efficiency of these steps becomes very important.

The original network interface was what I will refer to as a thin-wire interface. The Entity Object Manager had the job of reading the network buffers, storing the entities, dead-reckoning them, and communicating their state to the application each frame when the application asked for an update. It was configured to perform the buffer read and dead reckoning on a separate task, so it could keep up with the incoming network buffer. In DIS, however, network entities broadcast their position and orientation in earth-centered coordinates, and the application was converting these two vectors into simulation coordinates each frame, whether or not any new information had arrived, since the Object Manager was dead-reckoning. Since a coordinate conversion involves 64 floating point multiplications per vector, the computation blocked the application thread in a hurry when the number of entities went up. Another inefficiency was the fact that the Object Manager copied its entity state into the application data structure for each entity each frame, whether any new PDU had been received or not. The net result was an interface that did not scale well for large numbers of entities.

To redesign this for performance reasons, I first created a field where I could store the information if a new PDU had been received. Then, I only performed copying on those entities. I only did the earth-centered-simulation coordinate conversion when I received a new PDU, and I put this calculation off in a task. To protect the coordinate values during these conversions, I assigned each remote player object a semaphore lock from a set of locks I was maintaining, to avoid the problem I discussed above with one lock for a large set of data. Only converting when a new PDU was received meant my application now had dead-reckoning responsibility, so I added the dead reckoning calculations on the application thread, which ensured smooth movement. The net result of these changes was the frame rate went from 3hz with 400 smoothly moving entities to 15hz with 400 smoothly moving entities.

Two other important points about network interface are geometry management and culling entities based on their interest to the simulation. When many vehicles from the network are displayed, the polygon count goes up. To manage this, network entities should have low polygon LODs so they will not draw many polygons when they are not close to the viewpoint. Similarly, a cull step in the network interface will not process entities which are not of interest to the current simulation. For instance, the Virtual Cockpit can selectively decide which entities it cares about based on its radar setting and possibly other parameters. So, in its network interface, it can test incoming entities to see if it needs to maintain them in its processing loop.

6.11 Compiler and C++ Language Issues

This last section of the performance chapter will discuss the impact of the so called expensive C++ language constructs (virtual functions and object oriented design) and compiler optimization on the performance of our AFIT visual simulations. I will discuss a test I performed involving virtual functions in particular, and briefly discuss compiler optimization of C++.

The performance concern for C++ comes about when a virtual function call is performed, and the run-time must make a decision about which subclass will handle the call, based on the type of the object. Fast C++ runtimes implement this as a fixed offset into a table of function addresses based on the type field of the object, commonly referred to as a virtual function table. For my test, I surmised that a virtual function call would add little overhead to a simulation, in comparison to floating point calculations, device reads, etc. I set up a test where I performed virtual function calls on network entities each frame - an average of one call per entity. I linked in a set of simple geometry to reduce the effect of geometry on the result. I performed basic processing during each call. I found that as the number of calls went up, the frame rate degraded only slightly. For 10 entities, I recorded 20hz. When I increased to 100, I still recorded 18 hz. 400 entities reduced the rate to 15hz. I believe that about half of the degradation is due to the virtual function overhead. Based on this, my feeling is that visual simulation applications can successfully employ this technique to improve code structure and reusability.

The C++ translator and C compiler on the SGIs provide different levels of optimization. All of our best performance results were obtained using the second level (O2) optimization switch on the compile line. To further improve performance (and get the benefit of register optimization), the applications could explore the use of the third level (O3). I believe this level will remove virtual function overhead that can be statically determined at compile time (where no truly dynamically determined heterogeneous function calls are being performed). However, this changes the configuration management, since libraries now cannot be linked in as object code, since they must be optimized also.

6.12 Conclusion

The developments in the Graphics Lab benefited from the reusable architecture approach of ObjectSim. The developers were able to share classes and reuse problem solutions. The numbers we collected were based on the assessments of the developers using the architecture, and indicate that the object oriented approach of ObjectSim is an effective way to accomplish reuse in an environment such as the Graphics Lab at AFIT.

A visual simulation application can run into many pitfalls which can impact its performance. This section has presented some of them and has discussed my experience in the Graphics Lab with improving the performance of simulations. I discussed several areas of performance improvement. The real key to performance is to understand what the simulation is doing every frame and answer the questions: Does it block? Does it do expensive floating point? Does it need to be done every frame? Answering these and tailoring the simulation code will improve performance problems in many visual simulations.

The final chapter will discuss the results achieved in the Graphics Lab this year, relative to the initial goals of the research. Also, I will discuss the future direction for this work, and present some recommendations for the future of DIS at AFIT now that an architecture like ObjectSim is available.

VII Conclusions and Recommendations

7.1 ObjectSim

ObjectSim was the name for the reusable simulation architecture this research created. It was a tool which allowed simulations to be written at a higher level than they had previously been done at AFIT. In its present form, ObjectSim provides a good, but not complete, set of capabilities to a simulation developer looking to get an application going quickly. It can be improved in several ways. This chapter summarizes the research accomplishments and then provides recommendations for improvement.

7.2 Accomplishments

This research met many of the goals outlined in Chapter 1. ObjectSim was successfully used on four different thesis projects, and became the backbone architecture for visual simulations in the Graphics Lab at AFIT. By using an object oriented architecture, the student developers using ObjectSim were able to bypass much of the complexity of writing a visual simulation and concentrate on research-specific simulation topics. ObjectSim provides a reusable renderer, several device interfaces and a device-independent interfacing strategy, modes for data display, and many elements of a reusable DIS interface.

The projects using ObjectSim were quickly and incrementally developed. ObjectSim is a rapid-prototyping capability for visual simulation, with a robust enough design that the prototypes don't need to be rewritten to be useful as final products. The existing simulations in the Graphics Lab now share a common, proven design, which should increase the maintainability and usability of the research work accomplished in the lab this year.

Each of the ObjectSim projects reached impressive levels of capability while validating the inheritance-based extensibility on which ObjectSim was designed. The Virtual Cockpit can run in three different viewing modes, including an HMD and a multi-screen cockpit at the ARPA Sim Center. The Satellite Modeller improved substantially on previous years' work, and provides a very impressive, visually striking simulation. The Synthetic Battle Bridge has innovative user interface techniques which explore new ground in immersive simulation. It also runs in several modes. The Red Flag Remote Debriefing Tool is a powerful display console for Red Flag data which was deployed on-site to test its effectiveness and received praise from users. All of the simulations were a success. The reuse we achieved with our object oriented approach is a big reason for the success in the lab this year.

I have several recommendations for improving and extending this year's work. Many of the capabilities developed for the applications this year could be captured and made into reusable pieces without too much effort. This would result in a more powerful reusable simulation with a greater amount of engineering expertise built in.

7.3 Architecture Problems

Much of the library was implemented before I discovered the shared object solution discussed in Chapter 5. The applications would be a lot cleaner and more understandable if the whole library were converted to use constructors for the various objects, and to eliminate shared data structures inside the ObjectSim code. Then the simulation class could have pointers to the Pfmr_Renderer, Terrain, View, and a dummy Player declared in the superclass header. A default class constructor could initialize the simulation and assign most of the pointers together, making the application subclass responsible only for customizing the simulation, and not the messy object assignments it does now. Using this same technique, the applications could be converted to have cleaner designs, and to get away from most of the shared data structures they maintain now. The example programs found in the Application Developer's Guide are designed in much cleaner ways than the actual instantiations.

The Terrain and Simple_Terrain classes are lacking in their abstraction. Specifically, I didn't completely specify the functions the Terrain class should require of its subclasses to be abstract. The Simple_Terrain as originally implemented also had the static features defined inside, and this is questionable abstraction. A better design would be to have another class responsible for the dynamic aspects of the world, and to have the application instantiate one of these if it needed it.

The Sim_Entity_Mgr class, part of the network interface, is a conglomeration as originally implemented. The functions it does are reading network events, testing them against players, model switching, and maintaining a current list of local/remote simulation entities. Much of this work is done using a subdivision technique as discussed in Chapter 6. A better version of this class is needed which farms the processing out to the appropriate players and to an event manager class. This would make the Sim_Entity_Mgr only responsible for maintaining the current remote/local player list and managing the subdivision of it.

7.4 Geometry Management

ObjectSim still uses a simple scene graph for its players. Most player geometry hangs off of one pfGroup, which can create long calls for insertion and deletion of players. A better method is to explore the new Performer 1.2 node types which provide a geometric grid into which geometry can be placed. This would allow a faster cull and faster intersection tests with the grid.This would be a part of the Pfmr_Renderer class, and also might involve the Terrain class in determining the grid. Also, an application can create its own scene graph structure to manage its own partitioning of the player geometry. No work was done on optimizing the geometry organization.

ObjectSim right now has no abstract support for intersecting a segment with the player list. Applications must iterate through the list testing their vector against each player's geometry. Again, a better method would be to write an abstraction which would quickly give back a player id based on a segment, using the fastest method if intersection testing it could implement. This would probably be best implemented as part of the Pfmr_Renderer class, since this class is responsible for the player list.

7.5 Future Enhancements

ObjectSim now provides a platform of many reusable pieces which could make future simulations easy to implement. We have already created a reusable Performer simulation which can be instantiated many ways. In this section I will present some of the ways I think the architecture could be extended, and larger reusable pieces captured into it.

The Terrain class should be extended to implement terrain that does not come from a Flight file. It should give the developer the option to read in DTED data or other elevation formats and create the terrain database on the fly. The cultural features aspect of terrain should be made available as an abstraction.

The SBB interfaced ObjectSim to the FakeSpace Boom. The Virtual Cockpit interfaced ObjectSim to a set of Barco projectors and to a Head-Mounted Display These various view configurations should be captured into classes and provided as standard functionality to a simulation developer. The associated modifiers (Boom movement or Head Trackers or calibration keystrokes could then be made a part of these classes, and each application could merely create an object of the appropriate type and run in various modes.

More Modifiers should be created to provide easier interfaces. A Modifier class which takes the viewpoint and moves around a point based on a radius should be created. This class should use an Attachable_Player as the look point, and provide the appropriate offset and view direction to move the viewer around the Attachable_Player the view is attached to. Other Modifiers to use standard Forms or the SBB's buttons should be implemented and made available as standard pieces.

The network interface we have used is not reusable. Right now, each applications has a method which returns the network status based on its requirements. A better solution is to incorporate a DIS translation capability into ObjectSim as a part of the Base_Net_Remote_Player class. Then, the task reading the PDU buffer could call the appropriate object with the PDU and let that object handle the translation into the player data structure. Then, if a player subclass required the ability to override the default PDU processing, it could see the incoming PDUs and handle them as appropriate, before passing them on for default processing.

The send network interface should be similarly handled. A Base_Net_Send_Player class should be created, with the responsibility of building and sending a PDU when the application is ready to send one. This class could have the knowledge to interface to the send daemons, and provide a standard method to configure the outgoing PDU from information in the default player structure. Then, if the application's player needed to specially configure the PDU before sending it, the player could modify the PDU before actually releasing it to the network.

The network interface also needs to model the incoming entities with more precision. The current interface does not handle articulated parts, fire, appearance management beyond damage, smoke, plumes, and other DIS appearance fields. This management could be added into the Base_Net_Player or new Base_Net_Remote_Player class as capability in the propagate_periodic call, which does appearance management.

The simulation class itself could have functionality added to reuse the work of configuring the simulation. A standard method could be written to parse the command line and instantiate the proper objects as necessary, based on the requested configuration. Then, the main programs could all share much of the same code for configuring the simulation for different modes or viewing devices.

This thesis didn't define the reusable display objects I once envisioned. Although the Virtual Cockpit did some work in this area, this remains a fruitful area for expansion. The SBB did create a reusable class of buttons which made its way into the library, as did the stroke vector classes from Wright Labs. The Pfmr_Renderer class gave some functionality for drawing in or over the scene, but was limited to mode changes for the drawing pipeline. A true class of reusable display objects, such as dials, guages, MFDs, etc., would make even quicker and better simulations possible.

For a full term thesis effort, much of the work involved in building these simulations could be done with a tool. Rather than writing code from scratch, the programmer could sit down with a tool and build a description of his simulation design. This might include the type of terrain desired, the viewing modes desired, the device interfaces needed, whether or not a DIS interface was required, and individual or collector classes of players. The tool could generate an ObjectSim template which required the programmer to fill in the implementations for the players. Then, the tool could automate the process of building a makefile and creating the actual executable. Finally, this tool could allow the developer to link in the proper models or directories to help specify the model lists, model managers, etc. needed for the simulation to function.

Another avenue of research for this architecture is to explore implementing the architecture in Ada or Ada9X. This research has indicated that traditional bugaboos about object oriented design and C++ may not be as important as once thought in determining frame rate. Certainly AFIT has several successful visual simulations with fully object oriented designs. A researcher with a good knowledge about performance issues in visual simulation could show similar conclusions about software written in Ada or Ada9X.

An Ada9X implementation of ObjectSim would provide a good test of the new object oriented extensions to the Ada language. The cornerstone of ObjectSim is extensibility through inheritance, an exclusive feature of object oriented languages. To successfully implement ObjectSim or a similar model in Ada9X would provide a good test of the new language.

ObjectSim currently is dependent on the SGI platform and the Performer library. To make the architecture platform independent would require a tree-based rendering abstraction similar to the Performer tree. Many of the areas where the architecture takes advantage of the Performer features could have been implemented without Performer, at a cost of increased development time. This would also be a good goal for future research on the architecture.

7.6 DIS at AFIT

AFIT now has several DIS applications which use ObjectSim. AFIT has the capability to put a Virtual Cockpit into the DIS environment. AFIT can also put satellites and a Red Flag exercise onto the network. The SBB is a highly capable stealth viewer which can run in several modes, and the RDT is an impressive console for Red Flag exercise data. The Graphics Lab is a multi-mode DIS testbed, with the capability to rapidly develop additional DIS simulations using ObjectSim and Performer.

With ObjectSim and Performer, and especially with a fully integrated and reusable network interface, the Graphics Lab should have the ability now to reduce the development cycle for DIS simulation applications significantly. A simulation which a year ago would have been a two-month project should now just take two weeks. I believe AFIT has a unique and powerful capability in this respect, and I believe the concept of an object oriented application framework is central to this capability being available.

The Graphics Lab should now be able to support the overall goals of the Warbreaker program - to explore the use of large-scale distributed interactive simulation for weapons analysis, training, and combat simulation. The projects are ready for the next stage - integrating the simulations with a larger simulated wartime environment. One possibility is to include simulating the Global Positioning System constellation and using the simulated positions to navigate the Virtual Cockpit. Another interesting project would be to create some manned or semi-manned threats in the environment and to take a Virtual Cockpit mission against the threats to test mission planning or simulated weapons effectiveness. I believe the logical next step is to begin integrating the AFIT simulations into the larger picture of the DIS future in the military. ObjectSim should provide a fine tool for the process.

References

DoD92	Ada9X Project Report.. Office of the Under Secretary of Defense for Acquisition, Washington DC, 1992
Big89	Biggerstaff, Ted and Richter, Reusability Framework, Assessment, and Directions.
Cur88	Curtis, Bill, Cognitive Issues in Reusing Software Artifacts.
Bru91	Brunderman, John A. Design and Application of an Object Oriented Graphical Database Management System for Synthetic Environments. MS thesis AFIT/GA/ENG/ 91D-01. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
SG92	Silicon Graphics. Iris Performer Programmer's Guide. Mountain View, CA: Silicon Graphics, Inc. 1992.
SS93	Software Systems. Multigen Modelers Guide. Software Systems, Inc, 1993
Rum91	Rumbaugh, James, and others. Object Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall, 1991
DIp89	D'Ippolito, Richard S. Using Models in Software Engineering. Software Engineering Institute. Pittsburgh: Carnegie Mellon University, 1989.
JMASS92	J-MASS Architectural Technical Working Group. Software Structural Model Design Methodology fot the Modeling Library Components of the Joint Modeling and Simulation System (J-MASS) Program, J-MASS Program Office, Wright Patterson Air Force Base, OH, 1992
SPC91	Software Productivity Consortium. Ada Quality and Style, Guidelines for Professional Programmers, Herndon, VA: Software Productivity Consortium, 1991.
Laf91	Lafore, Robert. Object Oriented Programming in Turbo C++. Mill Valley, CA: The Waite Group Press, 1991.
Lowry91	Lowry, Michael R. Automatic Software Design, MIT Press, 1991

Bha92	Bhansali, K., and Nii, H. Penny, "Software Synthesis Using Generic Architectures", Knowledge Systems Laboratory, Stanford University, 1992
Wie92	Wiederhold, Gio and others, "Toward Megaprogramming: A Paradigm for Component Based Programming", Computer Science Department, Stanford University, 1992
Str91	Stroustrup, Bjarne. The C++ Programming Language 2nd Ed. Addison-Wesley, 1991
Harel92	Harel, David, "Biting the Silver Bullet - Toward a Brighter Future for Systems Development" , IEEE Computer, January, 1992
Bor92	Borland., ObjectWindows for C++. Borland, Inc, 1992
DIp91	D'Ippolito, Richard and others Model Based Software Development, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1991
Rum91	Rumbaugh, James and others, Object Oriented Modeling and Design. Prentice Hall, 1991.

APPENDIX A

ObjectSim Application Developers Manual

1 Introduction

1.1 Visual Simulation

Visual simulation is an area with diverse and complex software requirements. A simulation might be immersive, involving HMDs and head tracking, or console based, with a complex user interface. Typically, software written to implement simulations must solve problems of interfacing to various devices, rendering the simulation scene, interacting with other simulators, and correctly processing and displaying the user's view based on all of these factors.

Some representative types of visual simulations include *platform* simulations, in which the user operates a simulated platform, such as a helicopter or airplane, and *stealth* simulatons, which provide capabilities for the user to view a scene and observe movement of entities without interacting with them. Though the two types of simulation have different requirements, they have much basic capability in common.

When a simulator interacts with other simulators, it must have the capability to send and/or receive a *network interface protocol*. This allows a simulation to be compatible with other simulations which 'speak the same language'. For visual simulations in the military world, common protocols include DIS and SIMNET.

Taken together, all of these requirements make for a daunting software task. The next section will examine some approaches to solving this complexity problem to allow for shorter development times.

1.2 Reusing existing work

Visual simulation developers often try to reuse work. Software like network interfaces, device interfaces, and other work is often reused. One of the common problems with reuse is the difficulty of making reusable software interface with other reusable software. For instance, consider the definitions below:

```
typedef struct
{
  x: float;
  y: float;
  z: float;
} Position;

class Network Entity
{
  Position* get_position();
  .
  .
  .
};
```

```
typedef float[3] Vec3;

void Set_Draw_Position(Vec3 Drawpos);
```

Now, to set the drawing position to render the entity, the developer has to convert from the *Position* structure to the *Vec3* array to accomplish the task. This assumes, of course, that the coordinate systems for the entity and the rendering are in synch. If the network interface was developed separate from the rendering library, the developer of the simulation has no choice but to convert from one format to the other, or to reimplement the functionality to get around the problem. In designing a large system (and attempting to reuse functionality), these types of interface problems can multiply.

High level languages like C++ were invented to solve problems like this by allowing *abstract* data structures which provide constructs to manage complexity. However, a quick survey of some simulation code will reveal that complexity ir reduced, but not much. Too often, C++ is just used as a more capable C, and not as a high level language with its powerful features.

1.3 Iris Performer Overview

Recently, Silicon Graphics, Inc, (SGI) has attempted to solve some of the complexity associated with developing visual simulations with their Performer library. This library is an effort to create a standard library for visual simulation. This section provides an overview, but the Performer Programmer's Manual is recommended for a more in-depth view of the library.

Performer is a library of C callable routines that create a new interface to the SGI graphics pipeline. This interface improves vastly on the Graphics Library (GL) interface in many areas. While GL provides a good low-level interface to rendering polygons, Performer adds a higher level of library routines for managing an entire visual simulation. This library abstracts many of the chores associated with programming the SGI, and captures much of the SGI expertise about programming their pipeline. In addition, Performer provides an easy upgrade to the RealityEngine platform from the VGXT, because the library hides much of the details of rendering geometry and the same Performer interface is used on the new machines.

Performer maintains an internal geometry tree. This tree holds many of the common types of geometry used on the SGI. Geometry nodes, or *geodes*, can hold polygons, triangle mesh, light points, and many other types of primitives. These geodes are the leaves in the geometry tree. The tree also holds information like level of detail switching nodes, animation nodes, coordinate systems, group nodes, and instance nodes, all of which are intermediate nodes providing the capability for stored maniputations of the geometry in the tree. In effect, the tree allows the programmer to specify behavior or groupings of the geometry with the intermediate nodes.

Since the geometry tree is not dependent on a particular external geometry format, any format which can be converted to the Performer internal geometry format can be converted and placed in the tree. This conversion is done through file reader routines (figure 1). Performer comes with a Multigen flight (.flt) format reader, a reader for '.sgi' format geometry, and a reader for '.bin' format geometry. The flight reader allows all of the power and hierarchy expression available with the Multigen modeling tool to be seamlessly used in Performer applications with minimal effort.

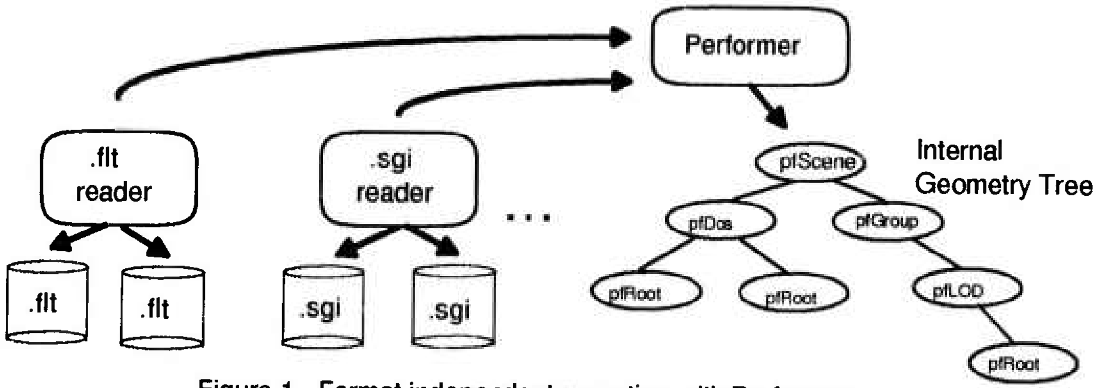


Figure 1 - Format independent operation with Performer

Performer also maintains state information in the tree. These nodes, called *geostates*, hold information like materials, textures, transparency, lighting conditions, and other information normally part of the GL state. This allows Performer to easily handle multiple textures, transparency, etc, through its built in state management features. Geostates are placed in the tree and apply to geometry under the geostate in the tree.

Performer renders its geometry tree very efficiently. According to Silicon Graphics, Performer maximizes frame rates for the Silicon Graphics geometry pipeline. This indicates the limiting factors for efficiently rendering a set of geometry are dependent on geometry and scene management, not on rendering code efficiency, when Performer is used.

Another performance breakthrough in Performer is obtained through its multiprocess management features. Performer provides an abstract, easy to use model for using three processors on a four processor machine. This model dedicates one processor to the drawing, one to culling, and one to the application managing the scene. This feature is also extensible to machines with more than one rendering pipeline. The multiprocess management features will manage the creation of separate cull and draw processes for each pipeline in the simulation.

Other features in Performer include multiple channels (viewports) into a scene, an easy to use viewing model, and an extensive math library. Performer has built in collision detection features, intersection testing, and special effects processing (clouds, haze, time of day, earth-sky, etc).

1.4 ObjectSim

Though Performer does provide a higher level interface to build visual simulations, much more work is necessary to write the software for one. Performer does not provide for standard device interfaces, network interfacing, complex view management, and other high level simulation requirements. Performer, as a powerful rendering library, requires expertise to successfully use. The Performer manual lists hundreds of Performer calls provided in the library. To manage this complexity, another level of encapsulation is needed.

ObjectSim helps manage the complexity problem by providing a series of *high level simulation objects* which interact to provide simulation functionality needed in a visual simulation application. Figure 2 shows some typical requirements found in visual simulation applications and support provided by ObjectSim to satisfy them.

ObjectSim provides ease of development for Performer simulations by:

- Encapsulating and abstracting these high level simulation capabilities with C++ classes.
- Providing the basic framework and design for a visual simulation
- Providing *abstract classes* which enforce interfaces between ObjectSim components

A simulation gains access to the ObjectSim functionality in two ways - by *inheriting* capability and by using *Object instances* which provide capability. The library does not provide everything needed to create a simulation, but it does give a good head start in getting a simulation running quickly with a proven design and with a small amount of code to maintain.

1.5 What's in this Manual

Because ObjectSim is an *application framework* and provides a different method for developing simulations, this manual contains a lot of explanation and examples. It has several chapters:

- **Chapter 2**, *Learning ObjectSim* gives a discussion of principles of writing a Performer simulation and includes a tutorial which will walk through some simple ObjectSim applications.
- **Chapter 3**, *ObjectSim Input Management* introduces the constructs for handling device and user input in ObjectSim, and presents some strategies

for managing user input from the keyboard or from a GUI package such as **Forms**.

- **Chapter 4,** *ObjectSim Drawing and Graphics* presents ObjectSim and Performer techniques for drawing text, graphics, and objects into the scene and overlaying the scene.
- **Chapter 5,** *ObjectSim DIS Simulations* presents some classes and ObjectSim constructs for managing the unique requirements for a DIS compatible simulation.
- **Chapter 6,** *ObjectSim Reference and Customization* includes design information and a class reference for the ObjectSim capabilities. Also included is a list of simulation requirements ObjectSim can be expected to aid with, with an indication of how to approach them.

ObjectSim and this manual will not provide everything you need to begin writing simulations. This manual does not:

- Provide extensive background on C++. For a good introduction, read (LaFor92) or another good reference.
- Provide complete Performer understanding. Read the *Performer Programmer's Manual* for this.
- Provide GL or GL windowing background. Read man pages or SGI relevant documentation.

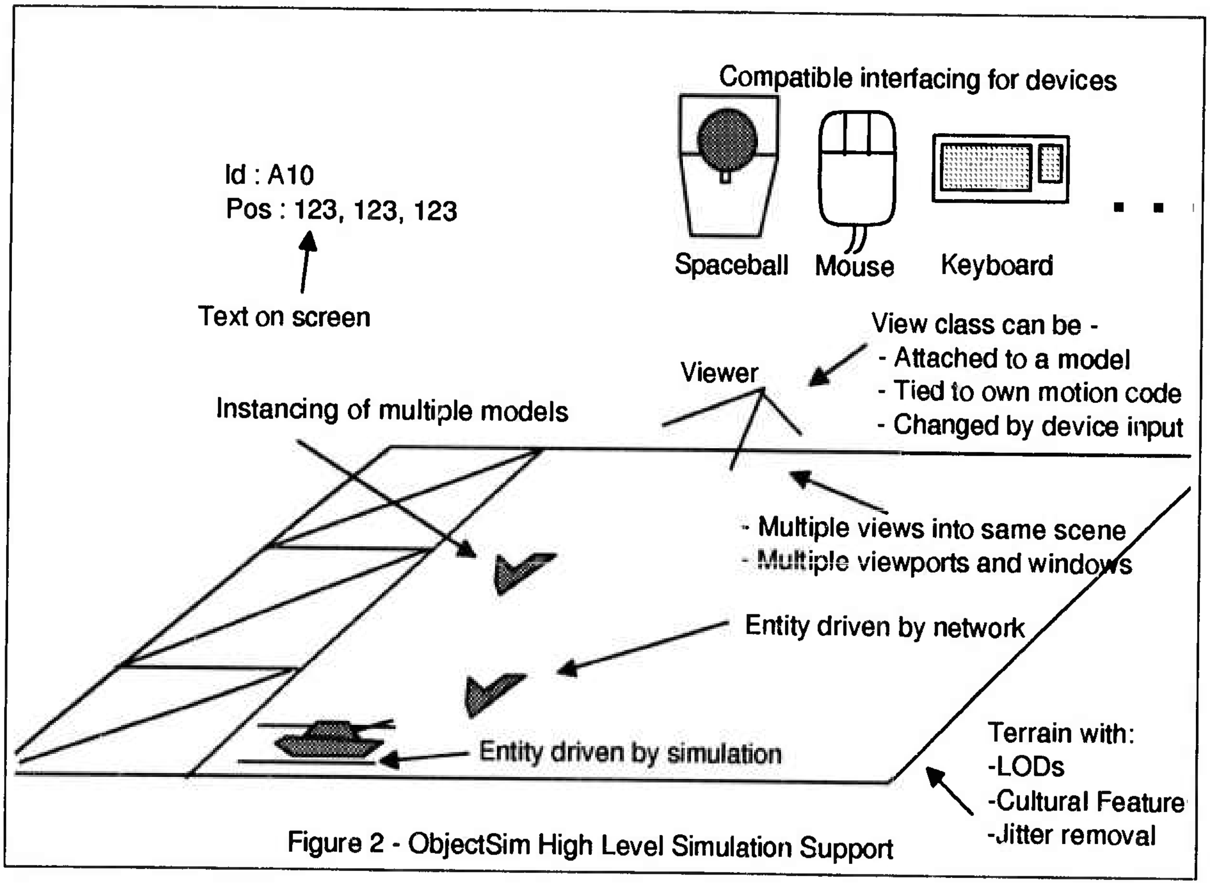

Compatible interfacing for devices
Id : A10
Pos : 123, 123, 123
Spaceball
Mouse
Keyboard
Text on screen
View class can be -
- Attached to a model
- Tied to own motion code
- Changed by device input
Viewer
Instancing of multiple models
- Multiple views into same scene
- Multiple viewports and windows
Entity driven by network
Entity driven by simulation
Terrain with:
-LODs
-Cultural Feature
-Jitter removal

Figure 2 - ObjectSim High Level Simulation Support

For this manual, a **boldface** class name indicates a class whose entry can be found in the class reference (Chapter 6) Code notes are found in the left margin.

2 Learning ObjectSim

2.1 Object Oriented Performer Applications

ObjectSim provides an *Object Oriented* interface into the Performer rendering library. This reduces greatly the amount of complexity a simulation developer needs to master before beginning to write visual simulation applications. ObjectSim follows the Performer three-process callback model to allow an application to split up its processing into three parts in an intuitive way. It manages much of the complexity of using Performer's process callbacks, giving the developer an intuitive interface into the objects which perform the simulations functionality. The objects hide the details of channel management, terrain and model instancing, view management, terrain translation, and the simulation inner loop, providing a standard method for implementing the specific functionality that makes each simulation unique.

Another benefit of this object oriented approach is the ability to create ObjectSim objects which are reusable across simulation projects. With just a little extra effort, ObjectSim objects can be developed as separate, reusable software pieces which can be 'plugged' in where needed to build different applications.

2.2 ObjectSim Environment

ObjectSim is built on top of the Iris Performer library. To compile and run with ObjectSim, the machine must have Performer installed. Also certain ObjectSim classes are designed to work with certain devices or network software. These dependencies are spelled out in Chapter 6.

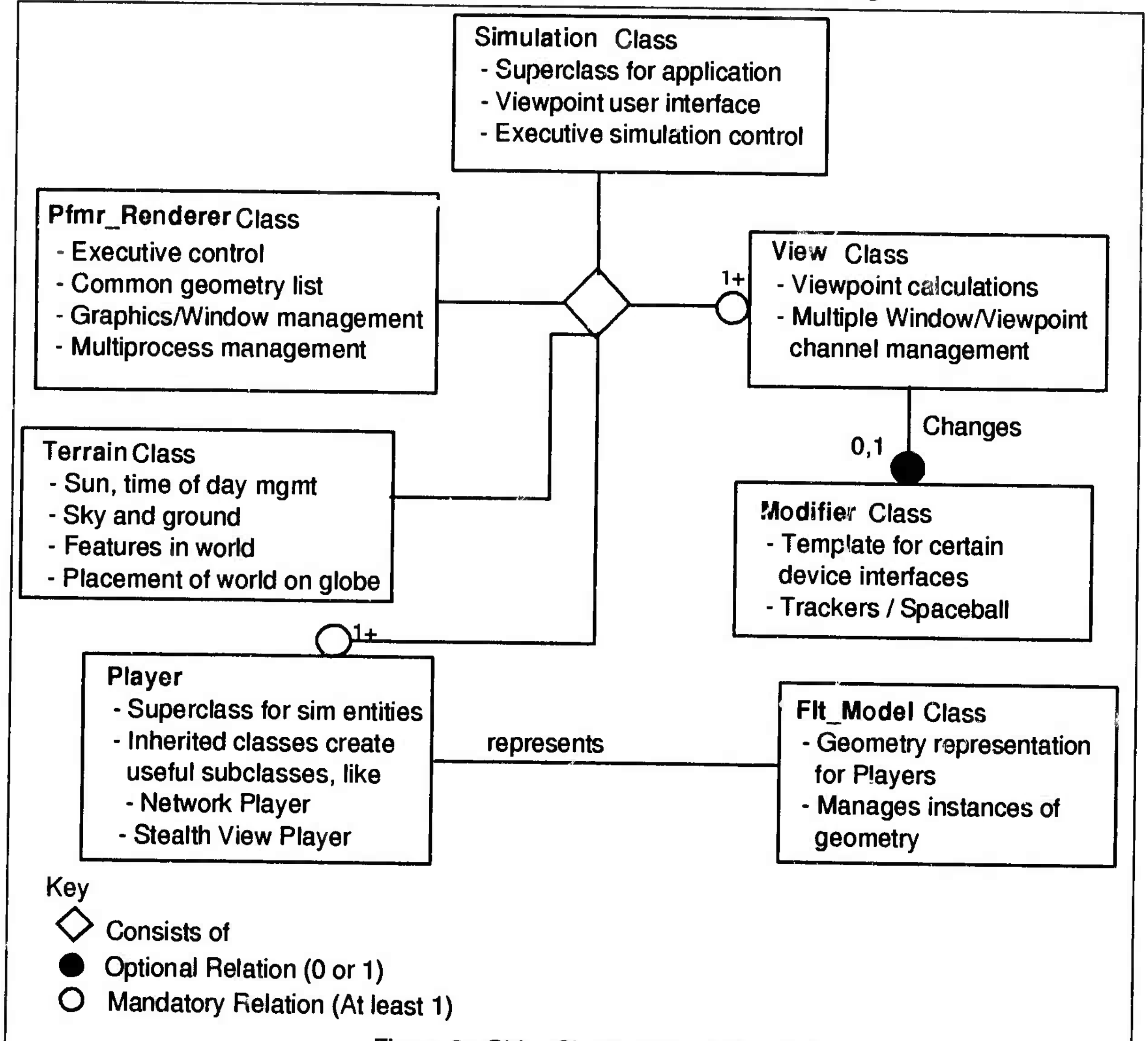
A typical ObjectSim development environment is organized by first setting up a directory for the ObjectSim work. Then, the ObjectSim 'include' directories and 'lib' directories are referenced in a makefile for your simulation, along with the Performer libraries and other standard libraries. The example programs found in the `ObjectSim/examples/pguide` directory contain sample makefiles for ObjectSim applications.

Geometry files are another dependency of all visual simulations. Some simulations, notably those which receive and display network entities, need access to hundreds of geometry files which will represent the entities. ObjectSim provides the **Model_Mgr** class for managing an extensive geometry context such as this. The example programs illustrate a method using a header file for simpler geometry dependencies. A good tip to remember is that if a program may change dependencies from time to time (change geometry files), either use the **Model_Mgr** flat file method for listing the geometry files, or use Unix symbolic links, which can be easily changed to point the application to different geometry.

2.3 *ObjectSim Design Overview*

This section provides a brief high level overview of the basic design of each ObjectSim program. In Chapter 6, this manual provides complete class reference information.

The design is shown in a simplified object model format in Figure 3. It is intended to show how the various objects are put together to form a working simulation.



Simulation Class
- Superclass for application
- Viewpoint user interface
- Executive simulation control
Pfmr_Renderer Class
- Executive control
- Common geometry list
- Graphics/Window management
- Multiprocess management
View Class
- Viewpoint calculations
- Multiple Window/Viewpoint channel management
1+
Changes
0,1
Terrain Class
- Sun, time of day mgmt
- Sky and ground
- Features in world
- Placement of world on globe
Modifier Class
- Template for certain device interfaces
- Trackers / Spaceball
1+
Player
- Superclass for sim entities
- Inherited classes create useful subclasses, like
- Network Player
- Stealth View Player
represents
Flt_Model Class
- Geometry representation for Players
- Manages instances of geometry
Key
◇ Consists of
● Optional Relation (0 or 1)
○ Mandatory Relation (At least 1)

Figure 3 - ObjectSim High Level Diagram

Each simulation written using ObjectSim will use instances of the above objects or subclasses of the above objects. Therefore, each ObjectSim simulation will inherit the basic design, and code written to interface to the design can be used with other simulations. Each class is documented in Chapter 6.

2.4 *Using The Tutorials*

The remainder of this chapter and chapters 3-5 contain tutorial stype examples of ObjectSim applications. The examples come from the **examples/pguide** directory under the ObjectSim directory on the machine. Consult the system management personnel for information as to where the software is installed.

To do the tutorials, copy the entire **pguide** directory and all of its subdirectories into a personal location. Edit the Makefiles to point to the location of the include and library files for ObjectSim. Then, compile and run the examples or experiment with them. They can also be used as templates to build visual simulation applications, if desired.

The example simulation applications will use more classes than discussed above in the overview diagram of ObjectSim. The discussion will explain the class being used and why, but for complete class information, use the Chapter 6 reference entry for the particular class.

2.5 *Hello World*

This section will introduce the most basic ObjectSim simulation. This simulation will place the viewer in an upright position viewing a terrain patch. As the basic pieces are completed, this guide will show how to add functionality to the simulation using ObjectSim. The code for this example is in the **ex1** directory.

The first step is to include the objects and abstract class headers. For the basic simulation, we need the following includes:

```
// Performer includes
#include <pf.h>

// ObjectSim classes
#include "pfmr_renderer.h"
#include "attachable_player.h"
#include "view.h"
#include "simple_terrain.h"
#include "simulation.h"
```

These files are found in the ../ObjectSim/include directory of the ObjectSim installed software

Attachable_Player is a subclass of **Player** which provides necessary members for a view to be attached to the player. View attachment will be discussed below. **Simple_Terrain** is a subclass of **Terrain** which implements a terrain patch which is a single file.

Now that the program has the proper files included, we can declare some classes to perform our simulation. Each simulation must declare a class derived from the **Simulation** class which spells out the requirements for a simulation. For this example, we will declare the following subclass:

The init_sim and propagate members are called by the Pfmr_Renderer to control the simulation.

Stealth_Player defined below

```
class Test_Sim : public Simulation
{
public:

  Test_Sim::Test_Sim();
  void init_sim();
  void propagate(int& exitflag);
  Stealth_Player* Stealth;
  View* MyView;

};
```

Keep in mind that the Test_Sim class also has members it inherited from the **Simulation** superclass (see Chapter 6). Therefore, the member functions init_sim and propagate can refer to any of these. The constructor is defined as follows:

```
Test_Sim::Test_Sim()
{
  Ter = new Simple_Terrain();

  Renderobj = new Pfmr_Renderer();

  MyView = new View();

  Stealth = new Stealth_Player();

  // These must be done for one of the players
  // only.  They are static data members
  Stealth->terrain = Ter;
  Stealth->Renderer = Renderobj;

  // Also done for only one view
  MyView->Renderobj = Renderobj;
  // Called on each view
  MyView->alloc_shared();
}
```

The constructor for Test_Sim has 'made' the basic set of objects which comprise the simulation. The Renderobj, defined in the superclass, is a **Pfmr_Renderer.** It controls the simulation's inner loop, handling calls from Performer to distribute processing to the application, cull, and draw processes. The MyView object is a **View,** which encapsulates one channel, or viewport, into the simulation. This simulation has declared its Ter object (class **Terrain)** as an instance of the **Simple_Terrain** subclass. Also, this simulation will contain one player, the Stealth object, defined below.

Next, is the implementations for the init_sim member function of Test_Sim. This function is called one time, after the Renderobj is ready for additions to its geometry tree and for viewing channels to be defined. This method is called after the standard Performer multiprocess fork, and before the simulation loop is kicked off. Therefore, this function is called on the application process, and one time application process functionality should be done here.

Notice the application is expecting the terrain (or a symbolic link to it) to be in the same directory as the executable with the name terrain.flt

```
void Test_Sim::init_sim()
{
  int found;

  // Read the terrain file and place in
  // Flt_Model number one
  Ter->readmodel("terrain.flt",1,found);

  // Initialize the player functionality here
  Stealth->init();

  // A new view on pipe 0
  MyView->new_view(0);

  // Connect this view to the stealth player
  MyView->attach_to_player(Stealth);

}
```

The only remaining function to be defined is the propagate function for the Test_Sim. This member is called once per frame on the application thread before any views are set or drawing is performed. Its function is to call any players or other objects which need to perform processing once per frame.

```
void Test_Sim::propagate(int& exitflag)
{
  // This call will propagate the attached
  // player regardless of which player is
  // currently attached to the view.
  MyView->get_attached()->propagate();
}
```

The **Simulation** subclass each application declares is the controller for the simulation. It defines the structure of the application and the objects which comprise it. Most of the simulation's functionality is provided by instances of the class called **Player** and its subclass called **Attachable_Player**. Players are entities which are in the simulation. They may be controlled by the simulation, by a network interface, or by the user. They may or may not have any 3D representation in the scene. The next section will declare the Stealth_Player, whose job will be to manage a viewpoint into the simulation:

The init and propagate functions are abstract in the superclass

```
class Stealth_Player : public Attachable_Player
{
public:
 Stealth_Player();

  void init();
  void propagate(){};
};
```

```
Stealth_Player::Stealth_Player()
{
  //Location, Orientation member
  Coords = new pfCoord;
}

void Stealth_Player::init()
{

  // Set initial position and orientation
  PFSET_VEC3(Coords->xyz, 0.0f, -300.0f, 100.0f);
  PFSET_VEC3(Coords->hpr, 0.0f, 0.0f, 0.0f);

  // These are for an attachable player.  They
  // define a position and orientation within
  // the object where an attached viewer is placed
  PFSET_VEC3(base_offst, 0.0f, 0.0f, 0.0f);
  PFSET_VEC3(base_rot, 0.0f, 0.0f, 0.0f);
}
```

Now all of the class methods are declared. The last part of this simple simulation is the main program. The main program in an ObjectSim application will knit all of the Objects together by establishing *relationships*. The relationships are implemented by assigning pointers between the objects. All ObjectSim main programs follow the same basic pattern:

- Initialize Performer
- Call simulation constructor
- Call **Pfmr_Renderer::init** and **Pfmr_Renderer::render** to perform simulation

```
int main (int argc, char *argv[])
{
  // The simulation
  Test_Sim* MySim;

  // Initialize Performer
  pfInit();

  MySim = new Test_Sim();

  // Cause the muultithreads to be kicked off and
  // Opens the default window
  MySim->Renderobj->init(MySim,1,MySim->Ter);

  // Cause the main loop to execute.  Will not
  // return
  MySim->Renderobj->render();

  exit(0);
}
```

The 'Hello World' simulation is now complete. To review, this example is a simulation with two inherited classes, a subclass of **Simulation** called Test_Sim, and a subclass of **Attachable_Player** called Stealth_Player. These classes define the simulations unique behavior. The main program called the Test_Sim constructor, which created the simulation objects and assigned pointers between them to establish relationships. Then, the **Pfmr_Renderer** object was initialized and its render method was called to perform the simulation. The member functions of Test_Sim and Stealth_Player perform the unique simulation processing. Run make to build an executable, and type testsim to view the simulation. The program will open a standard resizable window and then display the corner of the terrain.

2.6 Adding Moving Objects

These next sections will explore adding various basic functionality to an ObjectSim application. The code is contained in the **ex2** directory. First is a pair of moving objects.

In ObjectSim, the **Player** and **Attachable_Player** superclasses are a standard data structure for holding objects in the simulation. Since the **Player** class has certain functionality and required data members, it provides a nice way to ensure all players have a common look and interface. To implement the moving objects, we will declare the following (added to our previous simulation):

```
class Balloon : public Attachable_Player
{
public:
  Balloon();
  void init();
  void propagate();
};

Balloon::Balloon()
{
  Coords = new pfCoord();
  // This model will have a representation
  Model = new Flt_Model();
}

void Balloon::init()
{
  int found;

  // BALLOON & BALLOON_FILE #defined in 'sim_models.h'
  Model->readmodel(BALLOON_FILE, BALLOON, found);

  // This call will configure the RotDCS part of the
  // Flt_Model and add the object so it appears in
  // the scene
  Renderer->insertmodel(Model);

  // Define the viewpoint up along the z axis
  PFSET_VEC3(base_offst, 0.0, 0.0, 3.0);
  // And looking slightly down
  PFSET_VEC3(base_rot, 0.0f, -8.0, 0.0);
)

void Balloon::propagate()
{
  move_along_heading(4.0);

  // Ensure heading is between 0 and 360
  Coords->hpr[PF_H] -= 0.5;
    if (Coords->hpr[PF_H] < 0.0) Coords->hpr[PF_H] =
      360.0;

  // This call actually changes the DCS so the object
  // moves.  Moves the geometry in the scene
  pfDCSCoord(Model->RotDCS, Coords);
}
```

These statements define the moving objects behavior. The *model* for the objects is, again, a 'flt' format model. To allow a simple method for specifying and changing models used in a simulations, a header file with the proper information to allow the simulation to locate its models can be included. The code below is in the example directory in the file 'sim_models.h'

```
#define BALLOON 100
#define BALLOON_FILE "../models/yf22+1_a.flt"
```

The **Flt_Model::readmodel** method, called above to read the balloon, performs *instancing*. The software will only load one copy of the 'balloon.flt' file, and make copies of it at different locations if more than one is needed. Below, the players are initialized so we can watch them fly. The changes are made to the 'Test_Sim' class member functions as follows:

```
class Test_Sim ...
  ...
  Balloon* Bal[2];
  ...
Test_Sim::Test_Sim()
  ...
  Bal[0] = new Balloon();
  Bal[1] = new Balloon();
  ...

void Test_Sim::init_sim() ...
{
  ...
  //Initialize each balloon
  Bal[0]->init();
  Bal[1]->init();

  // Set their positions to be two points on a circle
  // And their directions so one will follow the other
```

```
   // around a spiral.
  PFSET_VEC3
    (Bal[0]->Coords->xyz, 3005.0f, 3030.0f, 105.0f);
  PFSET_VEC3
    (Bal[0]->Coords->hpr, 356.0f, 15.0f, 15.0f);
  PFSET_VEC3
    (Bal[1]->Coords->hpr, 0.0f, 15.0f, 15.0f);
  PFSET_VEC3
    (Bal[1]->Coords->xyz, 3000.0f, 3000.0f, 100.0f);
  ...
```

The next step in this example will show how to switch between players. The stealth player will get the added capability to look at a point (the front moving object). To do this, we give the Stealth_Player a pointer to any **Attachable_Player** so we can tell it where to look. The we add the code to lock at the player. We no longer just null the propagate member, and call a standard **Player** member function to orient the player toward a point:

```
class Stealth_Player ...
...
  void propagate();
  Attachable_Player* Look_Player;
...


void Stealth_Player::init()
  ...
  // Set initial position and orientation
  PFSET_VEC3(Coords->xyz, 3000.0f, 2800.0f, 100.0f);
  ...

void Stealth_Player::propagate()
{
  // Change the hpr to always point toward the
  // front balloon
  look_at_point(Bal[0].Coords->xyz);
}
```

Finally, this simulation needs the capability to attach the view to multiple players at different times. In this way, the user will be able to ride the back balloon or view both balloons from the stealth player's position. Since we have multiple players, we won't use the MyView->get_attached()->propagate, but will explicitly propagate all players:

```
void Test_Sim::propagate(int& exitflag)
{
  long buttons;

  Bal[0]->propagate();
  Bal[1]->propagate();

  Stealth->propagate();
```

```
        // Read the mouse
        buttons = ((getbutton(LEFTMOUSE)    ? 0x04 : 0) |
                  (getbutton(MIDDLEMOUSE) ? 0x02 : 0) |
                  (getbutton(RIGHTMOUSE)  ? 0x01 : 0));

        switch (buttons)
        {
        case 0x04:
          MyView->attach_to_player(Stealth);
          break;
        case 0x02:
          MyView->attach_to_player(Bal[1]);
        }
    }
```

That concludes the code for this example. To review, the simulation contains two *classes* of players (Balloon & Stealth) and three *instances* of player objects (a stealth object and a pair of balloons). The simulation moves the balloons around in a spiral and allows the viewer to switch between 'riding' the back balloon and viewing from a stealth position.

This chapter introduced the basic "look and feel" of an ObjectSim application. In the next chapters, this manual .vill cover device interfaces, DIS simulations, and drawing text and graphics into the scene.

3 ObjectSim Input Management

3.1 Input Methods

In a visual simulation, user input can take several forms. Figure 3-1 presents different types of interfacing done within various simulations. Within ObjectSim, the class called **Modifier** is used to provide a certain amount of device-independent interfacing. This chapter will show the use of modifiers as well as examples of other types of user input done within simulations using Performer and ObjectSim.

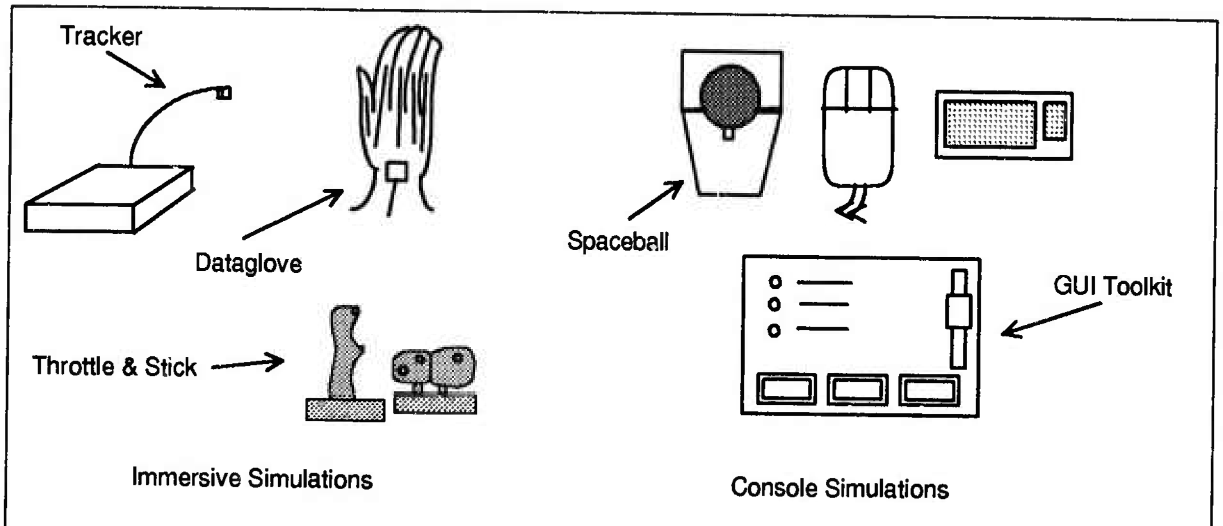


Figure 3-1 - Visual Simulation User Interface Requirements

3.2 Shared Memory

Before the next example, a brief discussion of shared memory in a multiprocessor application is needed. Shared memory is necessary because all drawing (and some device input) is performed on the drawing process thread, while the simulation processing is done on a different thread. Shared memory can be dynamically allocated in a structure or an entire object can be allocated out of shared memory, if its data members are needed on a different thread. The key point is that a call to `new` will allocate the storage for the object from non-shared storage. In order to allocate the memory out of shared storage, the class needing to use shared memory can declare the area it is interested in sharing as a substructure:

```
// Interface structure for My_Player
typedef struct
{
  int Current_Mode;
  float Speed;
} Shared;
```

```
class My_Player : public Player
{
   .
   .
  // Declare my_player's interface structure
  Shared* IfStruct;
};

// Declare an object of class My_Player in the
// simulation subclass
class Some_Sim : public Simulation
...
  Some_Sim();
  My_Player* TestPlayer;
...
```

The other requirement is that the shared memory be allocated before the call which forks off multiprocessing. The example subprograms show the constructor for the **Simulation** subclass being called after pfinit() and before the **Pfmr_Renderer::init** call, which performs the fork. During the constructor, an ObjectSim application could issue a call like this, which uses the standard Performer arena (very large) for shared memory:

```
Some_Sim::Some_Sim()
...
  Test_Player = new My_Player();
  TestPlayer->IfStruct = ( Shared* ) pfMalloc
   (sizeof(Shared), pfGetSharedArena() );
...
```

Now the `My_Player` class can use the members of it's shared structure on any thread. This is useful for drawing or interface purposes. All pointers, structures, or variables needed on two or more threads must be put in a shared memory area before they can be accessed.

A more clean design is to allocate the entire object out of shared memory, if most or all of its members are needed over multiple threads. However, if the object is allocated the above way, problems can result with virtual function calls on the resulting object. Instead, the following technique works:

```
class My_Player : public Player
{
  ...
  My_Player();
  // No more shared structure
};

My_Player::My_Player()
{
  this = (*My_Player)pfMalloc(sizeof(My_Player),
          pfGetSharedArena());
}
```

Now, when the constructor for My_Player is called, all of its member data will be allocated from shared memory and used on multiple threads.

Note that these techniques show how to gain multiple process visibility for shared data, but they do not address protection from shared updates and other issues. Simulations must handle these potential problems and be aware that their data is being accessed on multiple threads. A good approach to this is to develop the simulation and introduce protection as required when multiple update problems begin to occur. Many shared data accesses will require no special handling, but some may require locking or copying to prevent undesirable effects, such as partially correct vectors being used in calculations. Performer provides the pfDataPool functions for declaring and using shared memory locks.

3.3 Modifiers

Modifier is the class name for an input capability designed to translate or rotate the viewpoint. As such, modifiers cover such input as head tracking in an immersive simulation and using the spaceball to turn the viewpoint or move it. Figure 3-2 presents a conceptual view of the modifier.

In ObjectSim, a modifier represents rotations and translations added into the view calculation before the view is computed. Modifiers are state-based, which means they are asked for their current value before each view is computed. In ObjectSim, their is currently support for head tracking, spaceball modifiers, a keypad modifier, and a mouse modifier.

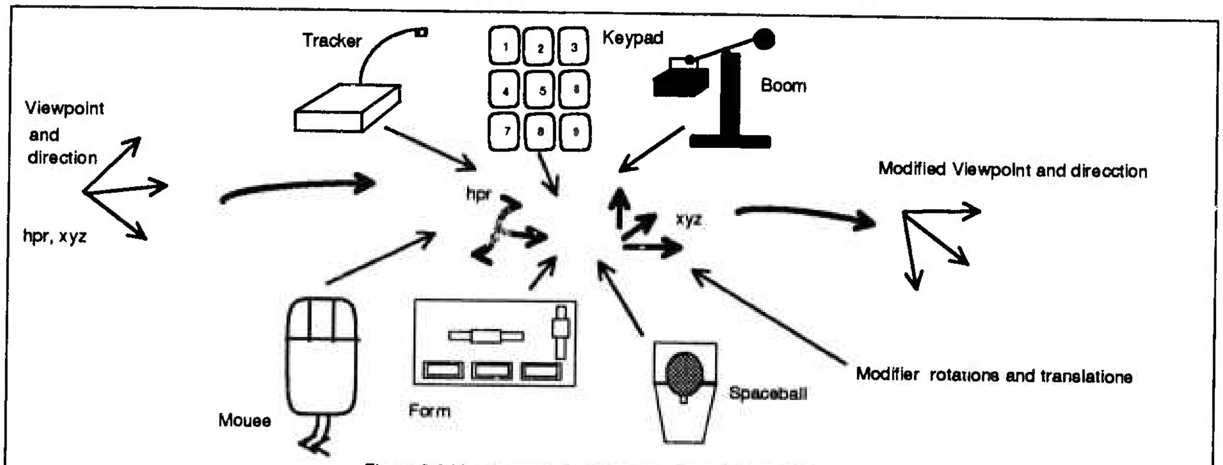


Figure 3-2 Modifier Role in ObjectSim Visual Simulations

3.4 Event Handling Approaches

An important part of visual simulation design is user event handling. This is simply the method by which the program gets and handles user input. For Silicon Graphics machines, often the user input is assigned to the same proccessor as the graphics drawing. Since, in multiprocessor applications, drawing is often assigned to a different thread than the main body of simulation, Performer and ObjectSim have built-in ability to assign event processing to the drawing thread. In ObjectSim, this is accomplished via methods in the **Simulation** and **View** classes. These methods allow the application to process user events, perform drawing, and other thread-specific activities. This is typically done for keyboard input and for input from a GUI, such as **Forms**.

Example 3 demonstrates the use of keyboard event handling and modifiers to assign to the viewpoint. The first step is to add some **#includes** and new objects in the Test_Sim class. We also override two draw thread methods so our application can perform some functions on the draw thread.:

```
.
.
// View modifier (std device functionality)
#include "mouse_mod.h"
#include "nq_keypad_modifier.h"

// For GL device queue
#include <get.h>
#include <gl/device.h>
#include <stdlib.h>
#include <stdio.h>

class Test_Sim ...
...
  void init_draw_thread();
  void pre_draw();

  // Modifiers
  Modifier* attached_modifier;
  Mouse_Mod* MouseMod;
  NQ_Keypad_Modifier* KeyMod;

  // Input data fields
  // Player I'm attached to
  Attachable_Player* attached_player;

  // Whether input was received
  int changed;
  int Exitkey;
...
```

The next step is to enable the draw thread processing discussed above. By declaring an **init_draw()** method and a **pre_draw()** method for the Test_Sim class, the application tells ObjectSim it wants to be called initially on the draw thread and once before any other draw thread processing each frame. Also, the simulation needs to have its data members allocated from shared memory so it can use them in the new functions. The new Test_Sim member functions are:

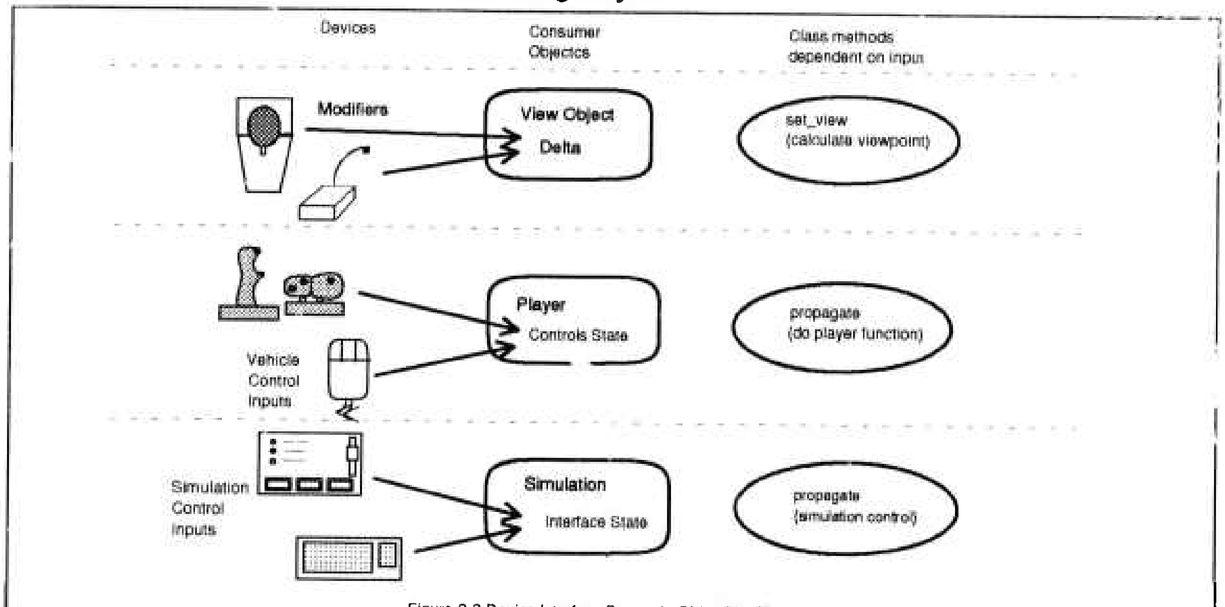
```
Test_Sim::Test_Sim()
{
  // Allocate all members from shared memory
  this = (Test_Sim*)pfMalloc(sizeof(Test_Sim),
          pfGetSharedArena());
  ...
  ...
  KeyMod = new NQ_Keypad_Modifier();
  KeyMod->init_state();

  MouseMod = new Mouse_Mod();
  changed = 1;
}

void Test_Sim::init_sim()
...
  MouseMod->init();
  MouseMod->attached_view = MyView;
  MouseMod->trans_rate = 0.3;
  attached_player = Bal[1];
  attached_modifier = MouseMod;
  changed = 1;
  Exitkey = 0;
 ...

void Test_Sim::propagate(int& exitflag)
...
  if (Exitkey)
    exitflag = 1;
...
  if (changed)   // Act based on input
  {
    MyView->attach_to_player(attached_player);
    MyView->Delta = attached_modifier;
    changed = 0;
  }
  MouseMod->poll();
...

void Test_Sim::init_draw_thread()
{
  // Tell the keypad that the standard device
  // queue will be feeding events to it
  KeyMod->init_extern_read((DEV)(&qdevice));
  KeyMod->turn_rate = 2.0;
  KeyMod->trans_rate = 1.0;

  // Queue up the keypresses I am interested in
  qdevice(BKEY);
  qdevice(SKEY);
  qdevice(MKEY);
  qdevice(KKEY);
  qdevice(ESCKEY);
}
```

```
void Test_Sim::pre_draw()
{
  // Read the standard device queue
  while (qtest())
    {
      // Queue is not empty
      short value;
      long  but = qread(&value);
      if (value)
      {
        // Pass the event to the keyboard modifier
        KeyMod->handle_event(but);
        // Handle the event
        switch (but)
        {
          case BKEY:
            attached_player = Bal[1];
            changed = 1;
            break;
          case MKEY:
            attached_modifier = MouseMod;
            changed = 1;
            break;
          case SKEY:
            attached_player = Stealth;
            changed = 1;
            break;
          case KKEY:
            attached_modifier = KeyMod;
            changed = 1;
            KeyMod->reset();
            break;
          case ESCKEY:
            Exitkey = 1;
        }
      }
    }
  // Ensure Performer draws to right window
  winset(MyView->WindowId);
}
```

When this example is run, the four keys queued above will allow the user to select both active modifiers and to switch between the players. This demonstrates how ObjectSim treats modifiers as standard capability, regardless of the source of the user input. The above simulation could have used a head tracker to modify the viewpoint instead of the mouse or keyboard. Having the mouse or keyboard interfaces allows for easy testing of immersive simulations, without having to hook up all the devices first.

3.5 Other Device Interfaces

Other devices or user inputs, such as throttle and stick, dataglove, GUI, etc, can also be used in a Performer/ObjectSim simulation. One key is to remember that the device read and device state check may or may not be in the same block of code. They may even be on different process threads. However, the functionality for the device should be encapsulated within a class, either a reusable device class or in the object using the input. When designing input capability, the easiest design to use is to maintain a conceptual interface state which the application depends on and make the state independent of the source of the particular input. This breaks the connection between the *producer* of the input (the device or input functionality) and the *consumer* of the input, the application. Figure 3 shows instances of this design style.



Devices
Consumer Objects
Class methods dependent on input
Modifiers
View Object
Delta
set_view (calculate viewpoint)
Player
Controls State
propagate (do player function)
Vehicle Control Inputs
Simulation Control Inputs
Simulation
Interface State
propagate (simulation control)

Figure 3-3 Device Interface Design in ObjectSim Simulations

3.6 Forms

Forms is a public domain user interface package which provides a Motif-like interface on Silicon Graphics machines. It incorporates an interactive forms builder which generates CC code suitable for compiling. Forms provide a useful capability to build and maintain a GUI for a simulation running at the console. This section will show how to interface to **Forms** in an ObjectSim application. The first step is to identify the relevant interface state information and design the class member data or structure to hold the input state. This code is in the **ex4** subdirectory. For clarity, the examples now have the Stealth_Player and Balloon classes declared in separate files.

This example will allow the user to interactively enter the viewpoint for the stealth player. The code will change in the Test_Sim class members as follows, to use the forms event handling instead of the device event handling:

```
void Test_Sim::init_draw_thread()
{
  // Initialize forms
  fl_init();
  // Tell KeyMod to use forms queue instead
  KeyMod->init_extern_read((DEV)(&fl_qdevice));
```

```
    KeyMod->turn_rate = 2.0;
    KeyMod->trans_rate = 1.0;

    Stealth->init_extern_read((DEV)(&fl_qdevice));

    fl_qdevice(BKEY);
    fl_qdevice(SKEY);
    fl_qdevice(MKEY);
    fl_qdevice(KKEY);
    fl_qdevice(ESCKEY);
  }

  void Test_Sim::pre_draw()
  {

    FL_OBJECT* obj;
    int form_event = TRUE;

    // Causes event queue read and callback invocation
    obj = fl_check_forms();

    while (obj != NULL)
      {
        if (obj == FL_EVENT )
        {
          short value;
          long  but = fl_qread(&value);
          if (value)
          {
            KeyMod->handle_event(but);
            // Pass event to stealth also
            Stealth->handle_event(but);
            switch (but)
            {
             ... Unchanged switch on simulation events
            }
          }
        }
        // Get another Forme event
        obj = fl_check_forms();
      }
    // Ensure drawing goes back to Performer window
    winset(MyView->WindowId);
  }
```

The stealth player now contains all references to the form, the callback for the form, and the event which brings the form up. This example has bound a part of its interface to a particular object, instead of having the entire interface state in one place and visible to all. The stealth player class definition has the following additions:

```
class Stealth_Player ...
...
```

```
    // Queues events this player is interested in
    // Assumes forms are initialized
    void init_extern_read( void (*qfunc)(long) );

    // Tests if this player is interested in the current
    //input event
    void handle_event(int EVENT);
  ...
```

And the member functions are defined in the .cc file as follows:

```
// Forms and the form for this object
// Form created with fdesign tool in file TSTinput.h
// and with callbacks TstOK and TstCancel, with
// creation function 'create_the_test_form',
// and with name of TSTval
extern "C"
  {
    #include "forms.h"
    #include "TSTinput.h"  // Generated by tool
  }

// This pointer is for the callbacks to access
static Stealth_Player* ThePlayer;

...
...

Stealth_Player::Stealth_Player()
{
  // Could allocate entire player from shared memory
  // if needed
  Coords = (pfCoord*)pfMalloc(sizeof(pfCoord),
                              pfGetSharedArena());
  ThePlayer = this;
}

void Stealth_Player::init_extern_read
                    (void (*qfunc)(long) )
{
  // Call the function pointer passed in to queue
  // My event
  (*qfunc)(VKEY);
  // Call creation routine
  create_the_test_form();
}

void Stealth_Player::handle_event(int event)
{
  if (event == VKEY)
    fl_show_form(TSTval, FL_PLACE_CENTER,FALSE,NULL);
}
```

```
// Callback for the OK button
void TstOK(FL_OBJECT* obj, long item)
{
  char* coordstr;

  // Read the input and store it
  coordstr = fl_get_input(TSTXinput);
  ThePlayer->Coords->xyz[PF_X] = atof(coordstr);

  coordstr = fl_get_input(TSTYinput);
  ThePlayer->Coords->xyz[PF_Y] = atof(coordstr);

  coordstr = fl_get_input(TSTZinput);
  ThePlayer->Coords->xyz[PF_Z] = atof(coordstr);

  fl_hide_form(TSTval);
}

void TstCancel(FL_OBJECT* obj, long item)
{
  fl_hide_form(TSTval);
}
```

That is all the code needed to put the form into the simulation. The form was created using the **fdesign** interface builder and told to have the named functions and callbacks. The key is that any data needed by the forms must come from shared memory, or be written by the form to shared memory. This example presented a hierarchical approach to allocating and using shared memory - by object. It also demonstrated partitioning development into smaller subpieces to increase maintainability.

The interfacing examples presented here are a basic set of interfacing capabilities. There are also other interface techniques. The key to managing interface complexity is to assign the interface to the appropriate level object in the design and to separate the source of the input from the use of the input.

4 ObjectSim Drawing and Graphics

4.1 Drawing Requirements in a Visual Simulation

So far, the simulation examples in this guide have just used flight format models for the rendered scene. In a visual simulation, often more types of information is needed by the user. This information could be textual, symbolic, appear out in the scene, or could overlay the scene. Figure 4-1 shows some types of drawing output found commonly in a simulation.

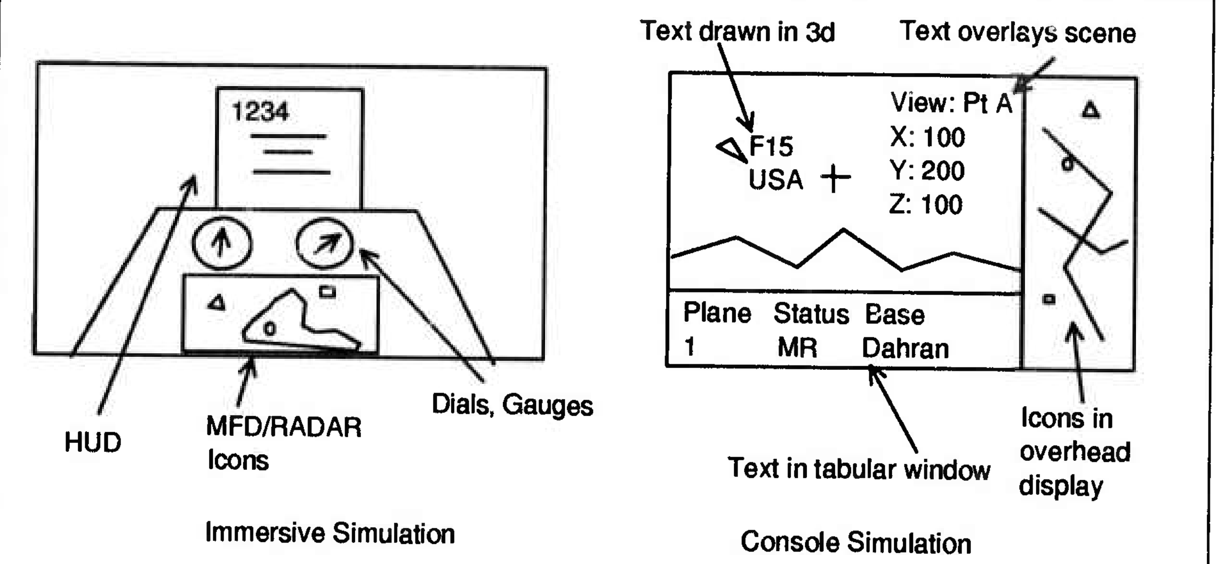


Figure 4-1: Visual Simulation Display Requirements

These differing display requirements can be met using varying techniques in ObjectSim/Performer applications. This chapter will present an example of doing these varying types of displays. The section on Performer tree management will discuss various ways to add geometry into the scene, and will describe the scene graph ObjectSim uses for its rendering, and how to add to and customize the scene.

4.2 Drawing Examples

The example in the **ex5** directory now contains three player classes used in the simulation. They are the Stealth_Player, Balloon, and Flyer classes. The Stealth_Player will show *in-scene* drawing of text based on the viewpoint.. The Balloon will show *screen overlay* drawing of text (or graphics). The Flyer shows the technique of drawing GL graphics as if they were part of a flt_model by using *callback drawing*. As the methods are presented, this manual will discuss its usefulness in simulations.

In the Test_Sim class, the changes to add the drawing are small. For a single channel simulation like ours, an ObjectSim program can just override the

Simulation::post_draw() virtual method. The post draw method is called when the proper tranformations are in place to draw text or graphics out in the scene (or overlaying the scene). So, the changes:

```
class Test_Sim ...
...
  void post_draw();
...
```

and, to actually call the player's draw method. The **Attachable_Player** class provides an abstract draw function:

```
void Test_Sim::post_draw()
{
  MyView->get_attached()->draw();
}
```

The only other main program changes are to add in the Flyer and to initialize the font manager one time for all the players. Note the main simulation could perform player-independent drawing at the same time it calls players for them to draw.

The Stealth_Player, for its drawing, now overrides the virtual **draw()** method in the player class. Also, it performs some drawing initialization in its `init_extern_read()` method, called already on the draw thread. The changes in the class definition are:

```
class Stealth_Player ...
...
  // Draw this players drawing
  void draw();
...
```

The changes in the .cc file are as follows:

```
// Array for drawing color
float Blue[3];

Stealth_Player::Stealth_Player()
...
  // Now allocate members from shared memory
  this = (Stealth_Player*)pfMalloc
     (sizeof(Stealth_Player),pfGetSharedArena());
...
void Stealth_Player::init_extern_read ...
...
   fmfonthandle   timesFont1;
   fmfonthandle   timesFont10;

   //Set up drawing
   timesFont1 = fmfindfont( "Times-Roman" );
   timesFont10 = fmscalefont( timesFont1, 10.0 );
   // Set font once, could be done each frame
   fmsetfont( timesFont10 );
```

```
    Blue[0] = 0.0f;
    Blue[1] = 0.0f;
    Blue[2] = 255.0f;
}

void Stealth_Player::draw()
{
  // Set mode for in scene drawing
  Renderer->setdrawmode(OVERLAY_IN_SCENE);

  // Set drawing color
  RGBcolor(Blue[0], Blue[1], Blue[2]);

  // Translate by negative viewpoint, since
  // Players are drawn relative to viewpoint in
  // ObjectSim
  cmov(Look_Player->Coords->xyz[PF_X] -
          Coords->xyz[PF_X],
       Look_Player->Coords->xyz[PF_Y] -
          Coords->xyz[PF_Y],
       Look_Player->Coords->xyz[PF_Z] -
          Coords->xyz[PF_Z]);

  charstr("YF-22");

  // Set mode for normal drawing
  Renderer->setdrawmode(NORMAL);
}
```

This completes the example for in scene drawing. The application can also perform GL drawing in the scene, using the same idea as the text example given above. In scene drawing is useful to overlay automatically scaled text over objects in the scene, such as numbers, identities, and so on.

The Balloon player demonstrates the overlay screen drawing capability. This looks much the same as the in scene drawing, but appears at a fixed 2D point in the channel where the drawing is being performed. The initialization code is similar. The only difference in in the actual **draw()** method for the player:

```
void Balloon::draw()
{
  char Outstr[10];

  Renderer->setdrawmode(OVERLAY_SCREEN);

  // Set color, font, and viewport
  fmsetfont(timesFont10);
  RGBcolor(Red[0], Red[1], Red[2]);
  ortho2(-0.5, 300 - 0.5,-0.5,300);
```

```
    // Print out data
    cmov2(10,22);
    charstr("X: ");
    sprintf(Outstr, "%4.2f", Coords->xyz[PF_X]);
    charstr(Outstr);

    cmov2(10,16);
    charstr("Y: ");
    sprintf(Outstr, "%4.2f", Coords->xyz[PF_Y]);
    charstr(Outstr);

    cmov2(10,10);
    charstr("Z: ");
    sprintf(Outstr, "%4.2f", Coords->xyz[PF_Z]);
    charstr(Outstr);

    Renderer->setdrawmode(NORMAL);
};
```

This code draws the player's position in the corner of the screen. This completes the example. The player could also draw 2D geometry on the screen. The viewport is always relative to the channel. This example is a single channel simulation. Below the guide discusses handling multiple channel simulations.

The last drawing example shows callback drawing. With this method, text or graphics are drawn in the scene as extensions of geometry in the rendering tree. This is done via a Performer callback assigned to the geometry the simulation wants to add to. The Flyer includes a rudimentary heads-up display which demonstrates this technique. The display is drawn using the coordinate system for the airplane - it is drawn in 'airplane space'. The following code handles the call to the heads-up display - again done on the draw thread:

```
// Typecaster for the draw callback
typedef long (*VPF)(pfTraverser*, void*);

HUD* Hudptr;

// Draw callback for the HUD
long drawfunc(pfTraverser* T, void* d)
{
  pfPushState();
  pfBasicState();
  Hudptr->draw_hud();
  pfPopState();
  return(0);  // Continue traversal
}

void Flyer::init()
{
 ...
  // Cockpit Drawing -- Set up callback
  // call drawfunc on root geometry, on draw thread,
    // after drawing the geometry
    pfNodeTravFuncs(Model->root,2,
                    NULL,
                    (VPF)(&drawfunc));
}

void Flyer::init_shared()
{
    .
    .
     // Set up the HUD
     Hud = new HUD();

     Hud->init_shared();
     Hud->init();

     // Assigned pre-fork, so visible on all threads
     Hudptr = Hud;
}
```

The heads-up display object contains code to draw the display. It uses GL drawing to draw the text and graphics. It also uses a stroke vector text class, called **GraphText**, which provides a nice way to do numbers and text which do not automatically scale. Hence these are good to use in an immersive simulation.

The above examples introduce techniques for augmenting the simulation with text and graphics. These are all implemented in the **ex5** directory. The next section will introduce the ObjectSim rendering tree, or *scene graph*, used to place players, terrain, and additional geometric objects into the visual scene. This next section is background information and does not have example programs for it.

4.3 ObjectSim Scene Graph

The term *scene graph* refers to the organization of the geometry in the visual scene. ObjectSim classes use different parts of the scene graph for different purposes. The *root* of a Performer scene graph is the **pfScene** assigned to the channel. In ObjectSim, the **scene** member of the **View** class holds the root of the scene graph. Terrain geometry is added as a child of this **scene** member. Moving geometry is added to the **playerlist** member of the **Pfmr_Renderer** class. The following diagram shows the relationship of the various ObjectSim classes to the underlying Performer tree.

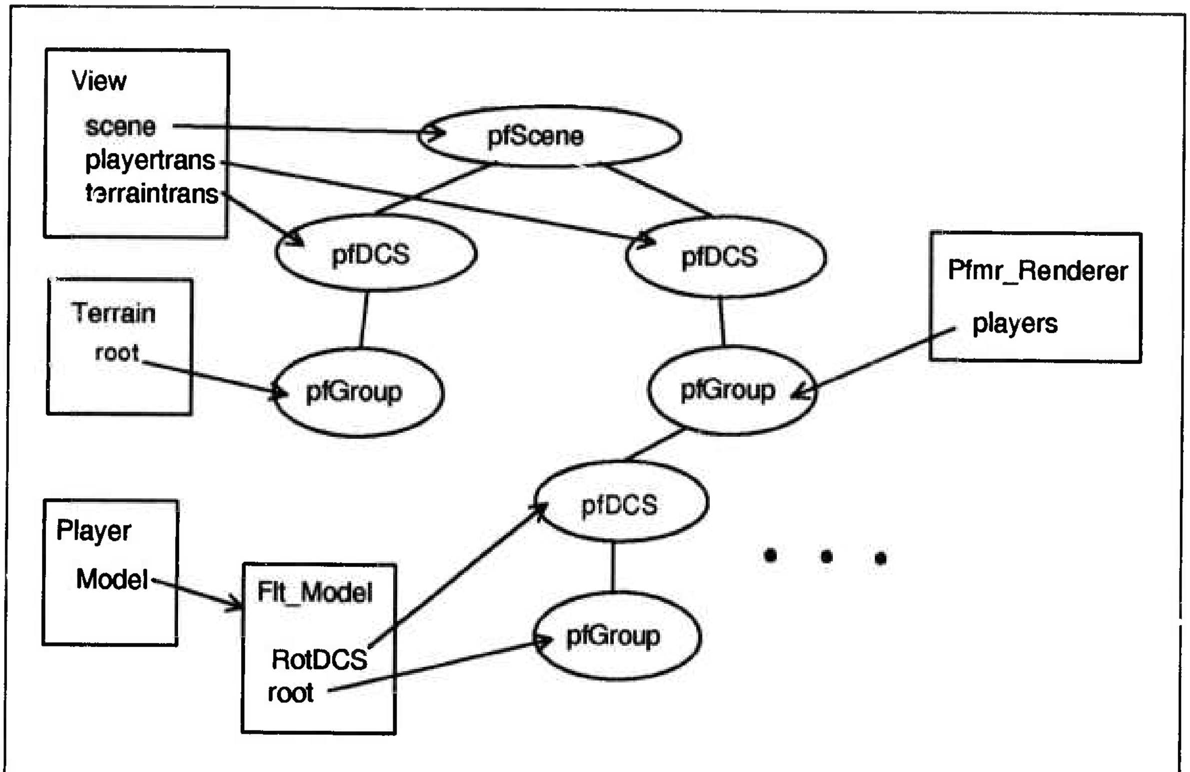


Figure 4-2 - ObjectSim Scene Graph

In Figure 4-2, the **pfDCS** nodes attached to the **pfScene** handle the translation of the terrain and players to account for 'jitter' removal. ObjectSim translates so the viewer is always at the origin, and the terrain and any players are translates to appear in the correct location relative to the viewer. For this reason, ObjectSim players must always attach their flight model RotDCS to the the **players** member of the **Pfmr_Renderer** or to another parent attached to the **players** member.

Figure 4-3 shows how the viewpoint is computed using this scene graph. This is shown to help the developer understand the viewing calculation. Keep in mind that all of the viewpoint computation occurrs automatically, and that the simulation can maintain player positions without regard to the translations being applied by the architecture.

4.4 Inserting Geometry

Geometry can be inserted in the rendering tree in a number of ways. This section will show some typical ways the rendering tree in ObjectSim is used to provide various effects, such as scaling, adding geometry to players (eg; plane carrying bombs), animations (eg; explosions), and instancing. Figure 4-4 will show examples of these various tree configurations.

To scale geometry in a Performer application, a **pfDCS** node must be inserted between the **Flt_Model's RotDCS** and **root** members. This **pfDCS** node can be used to scale the model. If the Flt_Model::RotDCS were used, the coordinates used for the models position and orientation would also have the scale applied, and this is incorrect.

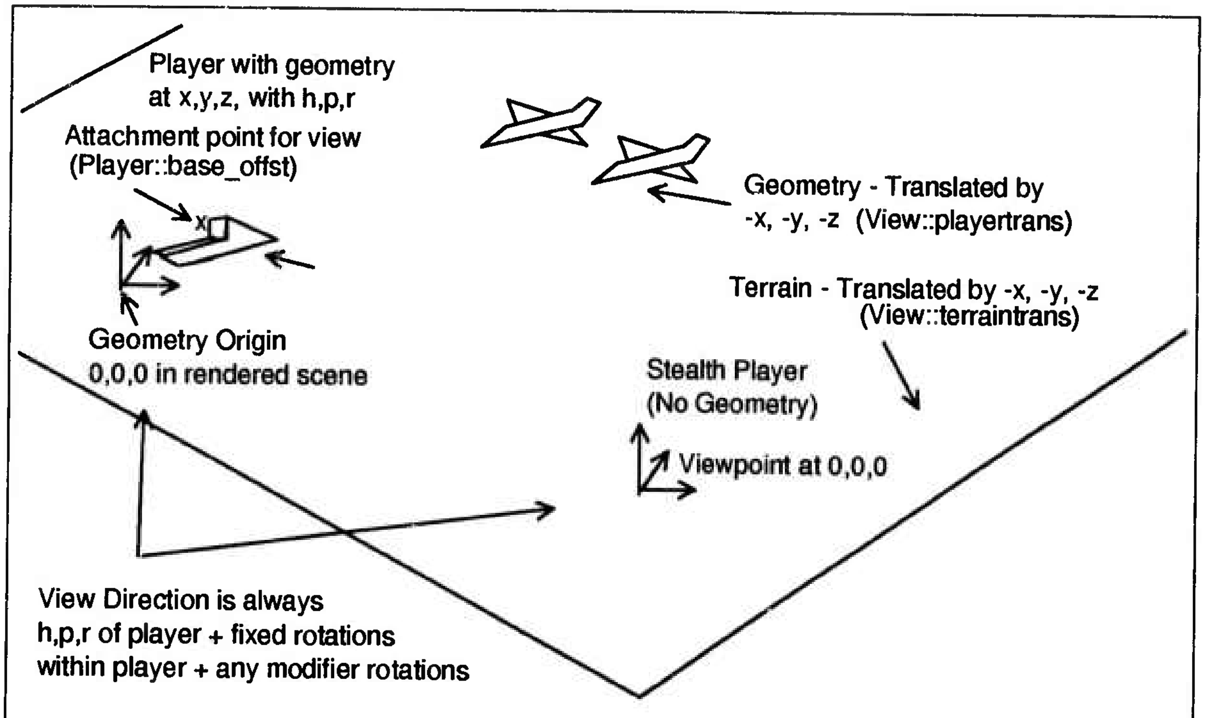
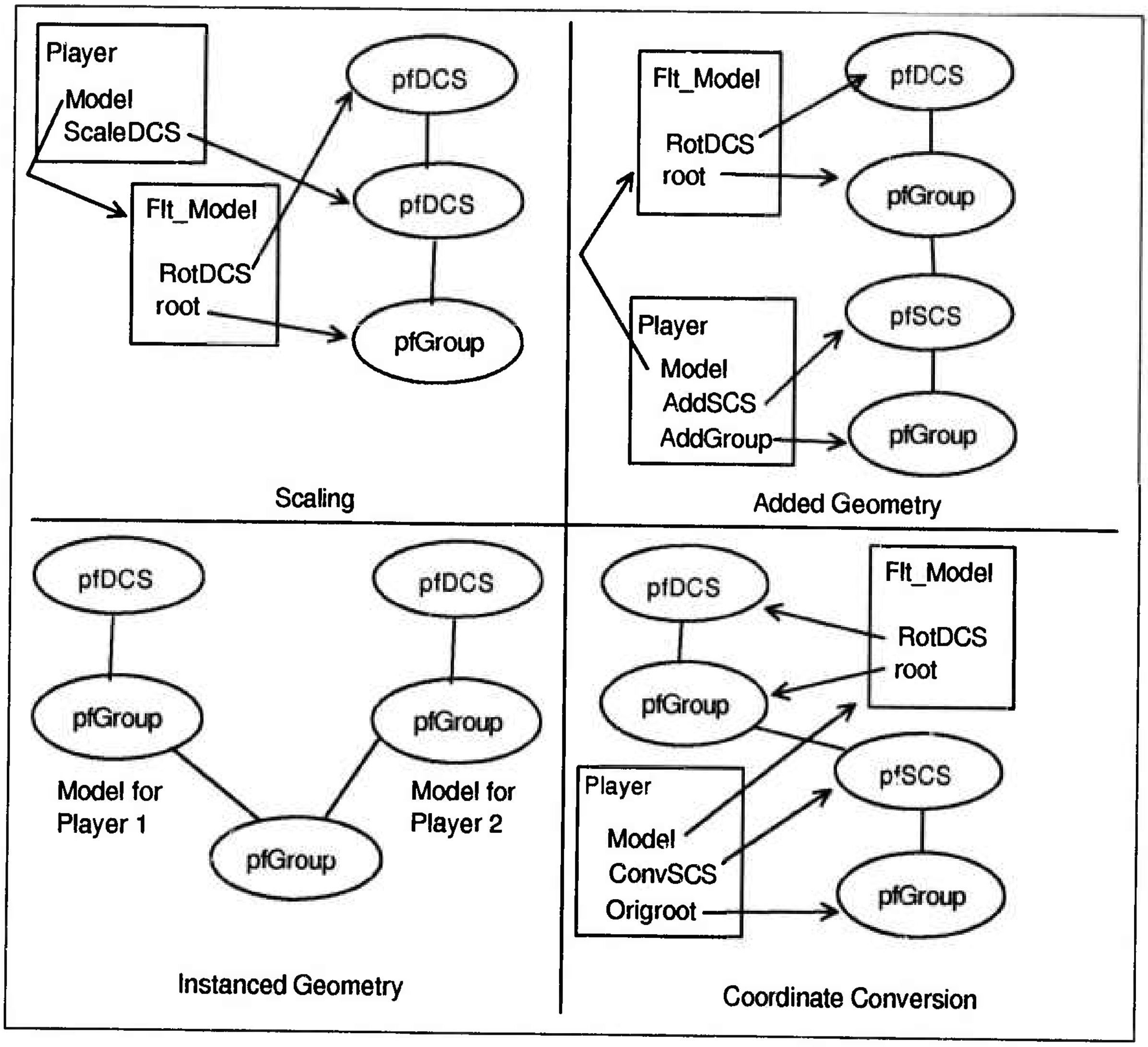


Figure 4-3 - ObjectSim Viewpoint calculation

To add geometry to a player. insert a **pfSCS** as a child of the **Flt_Model::root**, and add the geometry underneath. The pfSCS should be used to place the added geometry in the coordinates of the parent geometry.

Instancing can be used to create multiple copies of geometry in different places. Simply make the geometry be a child of multiple parents, and it will appear in multiple places. This is useful for such geometry as bombs, missiles, large geomatric objects used for visualization, etc. The Flt_Model::readmodel method automatically intances the geometry, so the simulation only maintains one copy of the geometry for each type of player (such as the F15 in the example).

A pfSCS can be used to convert models specified in some other coordinate system to the Performer coordinate system. If this is done, the simulation must take care to add any additional geometry to the proper level of the tree, so it appears correctly. If the geometry being added is in the original coordinate system, it should be added to the original root with the pfSCS set in original coordinate space. If the geometry is in Performer coordinates, it should be added to the transformed root at the Performer coordinate offset.



4.5 Multiple Channel Simulations

Multiple channel simulations include any simulation which provides two separate viewpoints into the scene. Performer supports basic multiple channel simulations by allowing multiple channels to be opened on one or more rendering pipes (see Performer Programmers Guide). In ObjectSim, a program can have multiple **View** objects attached to one or more players. In this way a simulation can maintain both an overhead and an in-scene view, for instance. Also, multiple channels are used for stereo vision applications, and can be used for multiple pipe rendering, where a simulation's view is spread across multiple screens.

This example will work as follows. The simulation will be able to have either two or three channels. The initial mode will be three channels, all attached to one player. This is the way multiple screen simulations are done. For the two side channels, we will use a fixed **Modifier** to hold the rotations tosteer these viewpoints to the side:

```
class FixedMod : public Modifier
{
  public:
    FixedMod(pfVec3 Loc, pfVec3 Offsets);

    void init(){};
    void poll(){};
};

FixedMod::FixedMod(pfVec3 Loc, pfVec3 Offsets)
{
  init_state();

  PFCOPY_VEC3(State->xyz, Loc);
  PFCOPY_VEC3(State->hpr, Offsets);
}
```

Now the simulation can use FixedMod as the **Delta** for any **View**. Next, the simulation needs to declare a subclass of **View** to handle opening the window and drawing the overlaying data on a per-channel basis:

```
class Fixed_View : public View
{
  public:
  // Override init draw to change window opening
  void init_draw();

  // Override draw to draw player's drawing
  void draw(pfChannel* chan);

  FixedMod* Mod;
};


void Fixed_View::init_draw()
{
  prefposition(0,1000,200,700);
  noborder();
  WindowId = winopen("ObjectSim Multi");

}

void Fixed_View::draw(pfChannel* chan)
{
  get_attached()->draw();
}
```

Now, Fixed_View will allow the attached_players drawing to be done for whatever player is attached. Next, the simulation needs three Fixed_View objects to be declared:

```
class Test_Sim : public Simulation
{...
  // No More MyView
  // Declare a set of three
  Fixed_View* LeftView;
  Fixed_View* MiddleView;
  Fixed_View* RightView;
...
  Attachable_Player* attached_player_left;
  Attachable_Player* attached_player_right;

  // Offset Vectors for grouped views
  pfVec3 LeftRots;
  pfVec3 RightRots;
  pfVec3 NoTrans;

  // Viewing mode switch
  int dualmode;
  ...
```

The simulation has three views and can attach to two players at the same time. The dualmode variable will be used to flag that we want side by side mode. Next, the constructor and initialization look like:

```
Test_Sim::Test_Sim
...
  LeftView = new Fixed_View();
  MiddleView = new Fixed_View();
  RightView = new Fixed_View();
...
  // Static member for view
  LeftView->Renderobj = Renderobj;

  // Must call for each view
  LeftView->alloc_shared();
  MiddleView->alloc_shared();
  RightView->alloc_shared();

  // Offset vectors for each view
  PFSET_VEC3(LeftRots, 45.0, 0.0, 0.0);
  PFSET_VEC3(RightRots, -45.0, 0.0, 0.0);
  PFSET_VEC3(NoTrans, 0.0, 0.0, 0.0);

  // No modifier for middle
  LeftView->Mod = new FixedMod(NoTrans, LeftRots);
  RightView->Mod = new FixedMod(NoTrans, RightRots);
...
}

void Test_Sim::init_sim()
...
  // A new view on pipe 0
  LeftView->new_view(0);
```

```
    // Middle
    MiddleView->new_view(0);
    // Right
    RightView->new_view(0);

    attached_player_left = Fly;
    attached_player_right = Bal[0];
    attached_modifier = KeyMod;
...
```

That completes the setup. The last thing needed is to set the viewports and turn the views on or off as appropriate for the viewing mode.

```
void Test_Sim::propagate...
...
  if (changed)
  {
    // Check for both attached to same
    if (!dualmode)
    {
       // Turn on middle view
       Renderobj->toggle_view(MiddleView, VIEW_ON);

       // Attach rotation modifiers
       LeftView->Delta = LeftView->Mod;
       RightView->Delta = RightView->Mod;

       // Attach all three to player on the left
       LeftView->attach_to_player
                 (attached_player_left);
       MiddleView->attach_to_player
                 (attached_player_left);
       RightVie  w->attach_to_player
              (attached_player_left);

       // configure Viewports for side by side
       pfChanViewport
          (LeftView->chan, 0.0, 0.33, 0.0, 1.0);
       pfChanViewport
          (MiddleView->chan, 0.33, 0.67, 0.0, 1.0);
       pfChanViewport
          (RightView->chan, 0.67, 1.0, 0.0, 1.0);
    }
    else
    {
      // Switch off middle view
      Renderobj->toggle_view(MiddleView, VIEW_OFF);
      // Don't attach fixed modifiers, but whatever
      // The other modifier is
      LeftView->Delta = attached_modifier;
      RightView->Delta = NULL;

      // Attach to separate players
      LeftView->attach_to_player
            (attached_player_left);
      RightView->attach_to_player
          (attached_player_right);

      // Set viewports to side-by-side
      pfChanViewport
          (LeftView->chan, 0.0, 0.50, 0.0, 1.0);
      pfChanViewport
          (RightView->chan, 0.50, 1.00, 0.0, 1.0);
    ...

////////////////' '//////////////////////////////////
void Test_Sim:     _draw_thread()
...
  fl_qdevice(OKEY);
  fl_qdevice(DKEY);
...

void Test_Sim::pre_draw()
...
            case OKEY:
              dualmode = 0;
              changed = 1;
              break;
            case DKEY:
              dualmode = 1;
              changed = 1;
              break;
...
```

Now the simulation can have two separate viewpoints attached two two different players or attach three viewpoints to one player. The other changes in **ex6** are administrative

Views can override draw operations in ObjectSim. In this way, a simulation maintaining multiple viewpoints can assign different drawing to each view (preferably with a player's draw method). In this way, a multichannel simulation can overlay different graphics for different channels. This is the only reason to override a **View's** draw method.

For specialized copies of geometry across multiple channels, a simulation can use masks to only allow geometry to be drawn on a particular channel. This, for instance, can provide view dependent scaling operations, by assigning multiple copies of geometry to a particular player, masked by channel.

The Performer documentation covers many basics of channels. See the Performer Programmer's Guide for more information.

An ObjectSim application can be run in different types of GL window. The **View::init_draw()** method contains the standard, resizable window opening call. Only one window is opened per pipe. To open a different size window, declare a subclass of **View** and override the init_draw function to open the window. The **View** class contains a WindowId member intended to hold the window id of the opening window.

5 ObjectSim DIS Simulations

5.1 DIS Overview

The simulations presented so far have been all internally generated. The simulation has generated all entities shown in the scene. In a *Distributed Interactive Simulation* (DIS) , entities are read over a network link and are involved in the ongoing simulation. A DIS simulation may be *read only* , *read and send,* or *send only.* Read only simulations allow the user to internally see players and events from the network, but not affect or communicate with them in any way. Read and send simulations allow the user to operate a player or players which broadcast status and send events, and may also affect other simulators (eg; damage them with a weapon). A send only simulation does not receive any network traffic, but does broadcast status of one or more players and possibly events. Figure 5-1 shows some various modes of operation for DIS simulators.

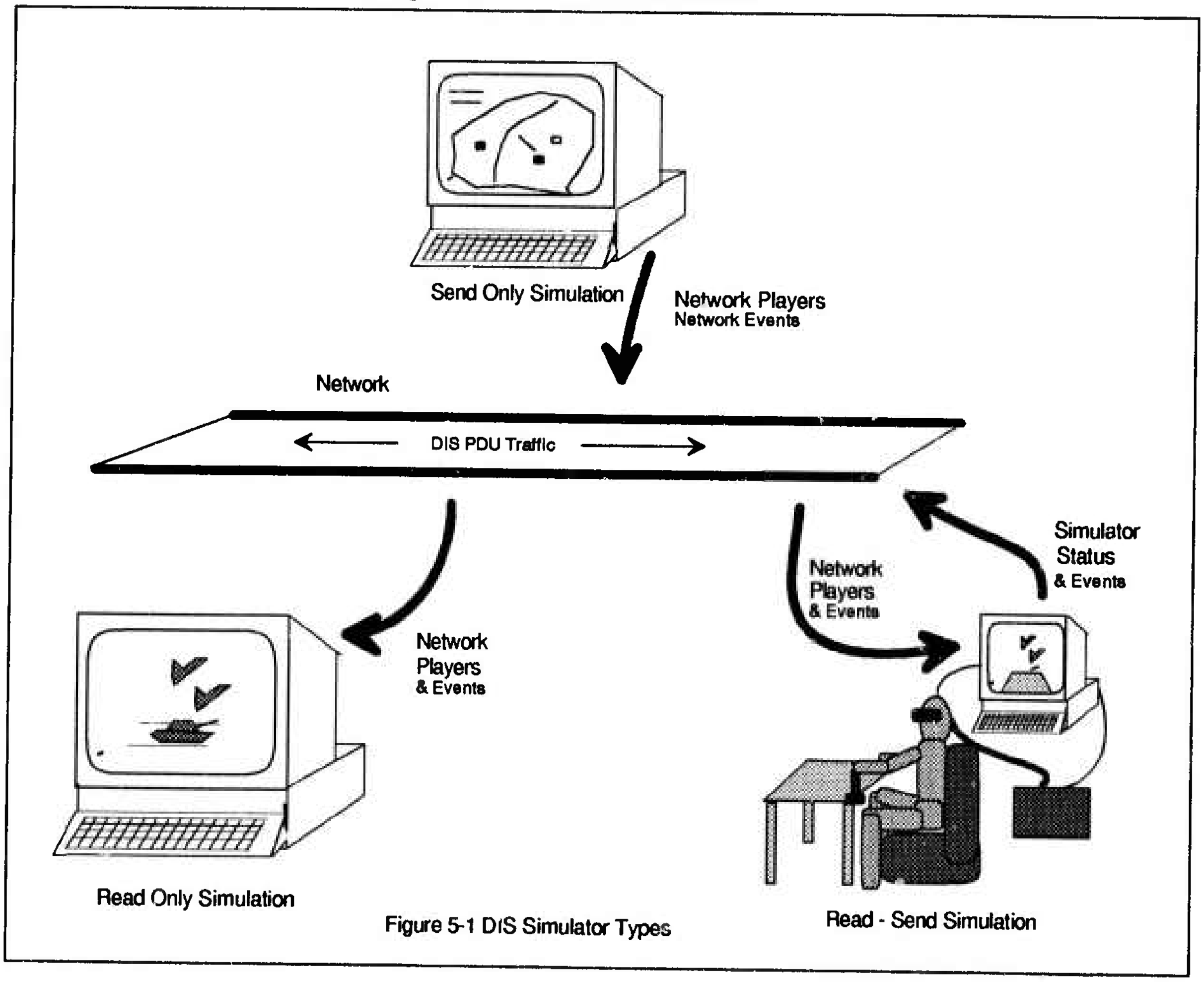


Figure 5-1 DIS Simulator Types

A simulator's communication with the network governs the classes required in the application to manage the interface. The special requirements of DIS also govern the functionality needed for a simulation, both read and read-send. Table 5-1 shows some of these requirements and the necessary software functions. The remainder of the chapter addresses how these requirements are met with a representative network interface.

DIS Simulation Requirement	Related Software Responsibility
Simulators need to show accurate state of external entities (type, position, orientation, damage, flaming)	Software needs functions to position network entities and reflect their state graphically and within any other required data or display structures. Software will map simulated entities onto correct graphical representation
Send simulators can cause events, which can affect other entities in the simulation	Software must be able to generate internal simulation events, broadcast them, and process them against the entities from the network to inform those simulators if they are affected
Simulators will process events in the simulated world, such as explosions and weapons fire	Software will allow events to be graphically shown from internal sources or over the network
Send simulators will change their state based on external or internal events and based on the simulations progress	Software will process both external and internal events against sending simulator(s) and broadcast and graphically show correct state
Send simulators will reduce traffic by limiting updates based on time or position	Software will not update state until some time or position threshhold is exceeded
Receive simulators will show a smooth entity progression through scene	Software will dead-reckon simulation entities between updates

Table 5-1 - DIS Requirements Breakdown

5.2 ObjectSim and DIS

ObjectSim has a DIS interface built into the library. This interface will manage a list of remote entities and present them to the simulation. The interface consists of three parts - the Basic_Net_Manager, which handles reading the PDUs from the network, The Base_Net_Remote_Player, which accepts PDU updates and models a network entity, and the Sim_Entity_Controller, which handles events in the simulation and provides stimuli to the Base_Net_Players to perform certain functions. The ObjectSim application must declare the Basic_Net_Manager and the Sim_Entity_Controller, then it will have a network interface with the remote players being represented.

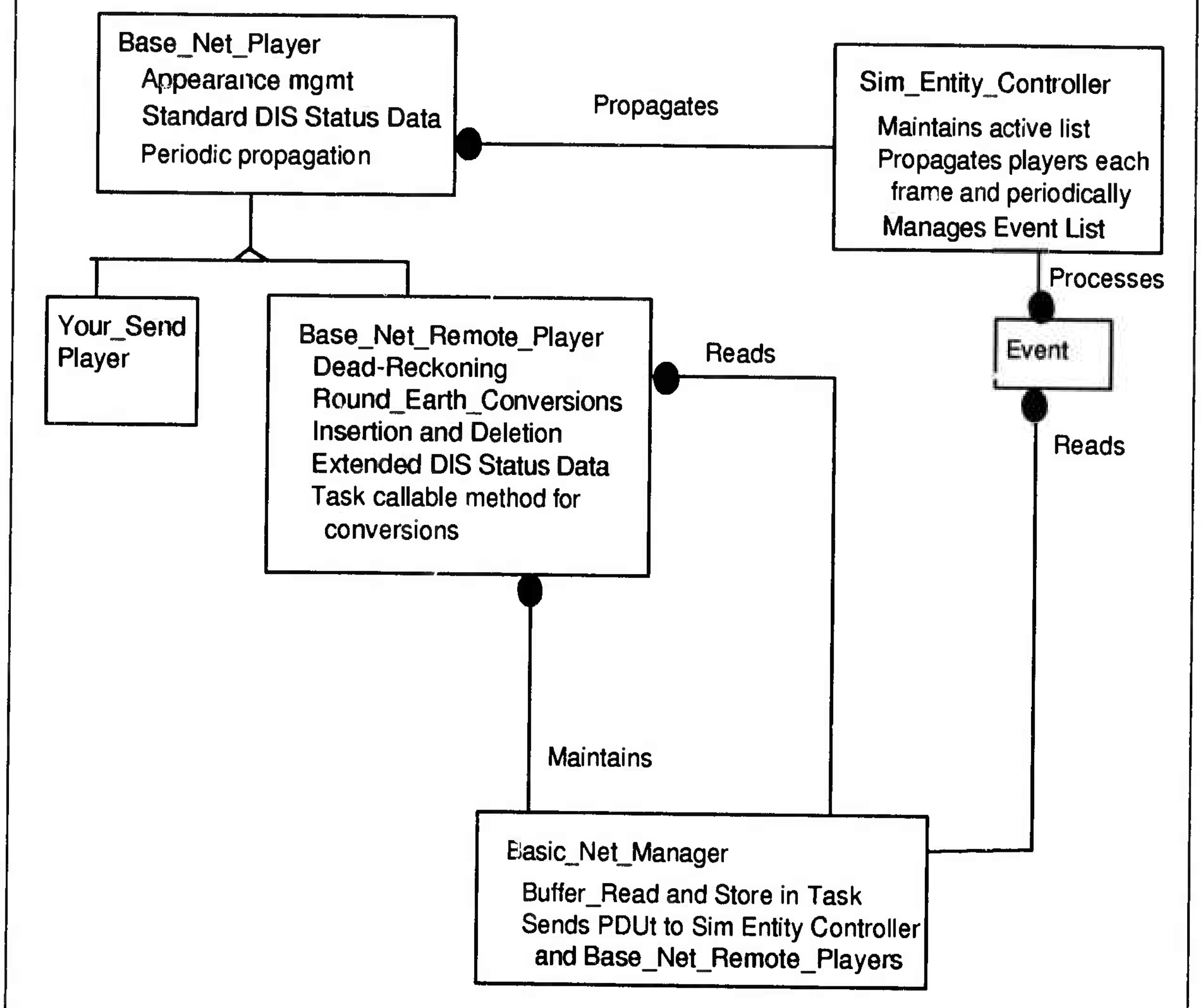


Fig 5-2 - ObjectSim DIS Interface

5.3 Round Earth

The DIS standard calls for entities to broadcast their position and orientation relative to a round earth. Since most simulators use a flat earth reference frame (and Performer is designed for this), The data from these entities must be converted from the round earth reference into the flat earth reference frame used by Performer. ObjectSim contains the class Round_Earth_Utilities for this purpose.

The terrain may also have to undergo a transformation to make it visible correctly in the simulation. If the terrain is defined in round earth coordinates (as earth skin, for instance), then it will need to be translated to appear correctly in the flat earth reference frame.. The transformations used for the terrain will affect the transformations used to send out information for send simulations. The Simple_Terrain class provides methods for specifying the type of terrain defined for the simulation. When the terrain is properly initialized, the

Round_Earth_Utilities will also be initialized to properly convert ingoing and outgoing DIS data.

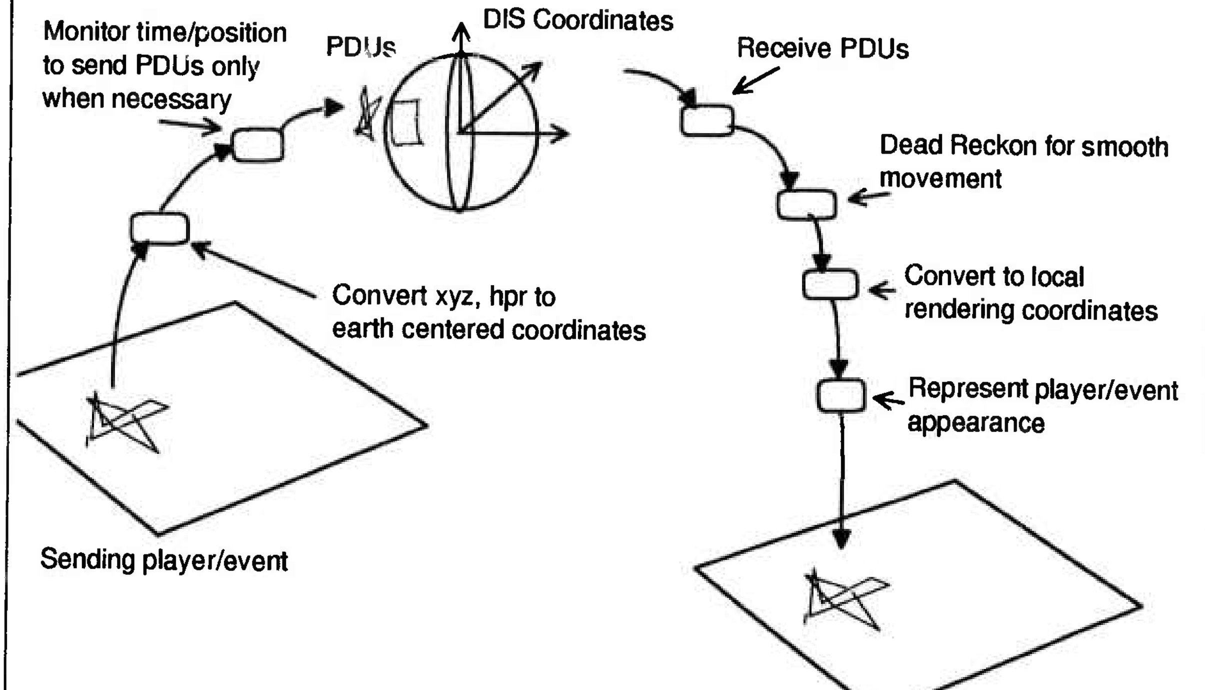

Monitor time/position to send PDUs only when necessary
PDUs
DIS Coordinates
Receive PDUs
Dead Reckon for smooth movement
Convert to local rendering coordinates
Convert xyz, hpr to earth centered coordinates
Represent player/event appearance
Sending player/event

Figure 5-3 - Round Earth DIS Interface

6 Class Reference

This section provides a class reference for the ObjectSim classes. The reference information is presented in inheritance order for the various classes. The description of the class covers assumptions, common usage, problems, and customizing/extending. The last section cross references the customization and problem sections to give a reference to these areas.

6.1 Class Simulation

Purpose

This class is superclass for all ObjectSim simulations. It provides abstract member functions which a simulation must provide and virtual member functions a simulation must override to assign processing to different threads or times.

Member Functions

init_sim(): Abstract function. Usually will initialize app thread processing, call init() on any players, and any other initialization (but not shared memory allocation, which must be done before Renderer::init()). Called by: ObjectSim (Pfmr_Renderer::init(...)). All applications will override this function.

propagate(): Abstract function. Called once per frame. Main application thread processing for application. Usually will call players propagate() functions, which update player state and rendering tree if applicable. Any other processing which affects rendering tree is done here. Input communicated via shared memory from draw or cull thread is processed on this thread. Called by: ObjectSim (Pfmr_Renderer::render()) (Main Loop). All applications will override this function.

alloc_shared():

init_draw_thread(): Virtual Function. Called as first processing on draw thread. Used to initialize devices or interfacing required on draw thread. Called by: ObjectSim (Pfmr_Renderer::init, after init_sim()). Application overrides if needed, otherwise defaults to empty function.

pre_draw(): Virtual function. Called once per frame before rendering the scene on the draw thread. Use to read or update interfaces on draw thread. Can also be used to customize graphics state before drawing. If used to read forms, use winset with the view's WindowId at end of this function to point rendering at correct window. Called by: ObjectSim (Pfmr_Renderer::render() while drawing frame in parallel with application). Application overrides if needed, otherwise defaults to empty function.

post_draw(): Virtual function. Called once per frame after rendering the scene on the draw thread. Use to perform GL drawing functions overlaying or in scene. May reset graphics state before drawing. Called by: ObjectSim Pfmr_Renderer::render() while drawing frame in parallel with application). Application overrides if needed, otherwise defaults to empty function.

Notes

Use the draw methods from this class for single channel simulations and for non channel-specific multichannel simulation processing. Use view class draw methods for channel specific drawing or graphics processing in multichannel simulations.

6.2 Class View

Purpose

This class encapsulates a single view into the simulation. A view has a channel where it is rendered, and takes its viewpoint from the player it is attached to. A views may also have a modifier, which add an additional translation and orientation into the viewing calculation. The view class holds the reference to the channel's scene and the DCSs used to translate the scene graph to move the terrain and players' geometry relative to its viewpoint. A simulation can maintain multiple views, which allows multiple channel simulations.

Member Functions

setview(): Called to perform viewpoint calculation for current frame. Will take any modifier state and the attached player's state and calculate the viewpoint and the scene graph's translations to correctly render the frame. Called by: ObjectSim (Pfmr_Renderer::render() after simulation is propagated, once for each view). Future subclass could override to set its own view calculation, but must pay attention to attached player code within here.

new_view(int desired_pipe): Called to initialize a view on the pipe specified. Opens the channel on the desired pipe and creates the necessary scene graph elements. Registers this view with the renderer. The first View initialized on each pipe kicks off the draw thread processing for that pipe. Called by: Application, during init_sim for each view. Could be overridden for a different scene graph structure if desired, but has not been explored.

init_draw(): Called to open a window for the view. Calls foreground() first to ensure window open does not force draw thread into background. Default window is user sized and resizable. Saves WindowId data member on winopen. Called by: ObjectSim: (View::new_view()) Application may override to open window with different characteristics (but must save WindowId if it uses it elsewhere);

pre_draw(pfChannel* chan): Override point for pre draw functions specific to one channel in a multi channel simulation. Used for graphics state changes prior to rendering. Called by: ObjectSim (Pfmr_Renderer::render() once for each view). Application may override, defaults to empty function.

draw(pfChannel* chan): Override point for draw functions specific to one channel in a multi channel simulation. Used for channel drawing after rendering. Called by: ObjectSim (Pfmr_Renderer::render() once for each view). Application may override, defaults to empty function.

init_cull, cull: cull thread overrides, similar to draw overrides. Init_cull not implemented.

alloc_shared(): Shared allocation for view. Called by: Application for each view it maintains, during simulation alloc_shared call.

attach_to_player(attachable_player* the_player): Used to attach the view to a player. Attached player defines viewpoint. If the player's Model->root = null, the player has no geometry attached, or else this method assumes it has attached geometry and takes appropriate action. Called by: Application. Assumes the player's Model has been created.

getwindowsize & getwindoworigin: application thread utilities for window attributes.

attachable_player* get_attached(): Retrieves a pointer to the attached player. Can be used on any thread. Called by: Application.

pfChannel* chan: Member holds the pfChannel for this view. The application can use the Performer channel attributes on this member defined to affect clipping planes, offsets, channel groups, or any other channel manipulations needed. Has valid value after new_view

Modifier* Delta: Member defines which view modifier is assigned to this view. Can be null. Set during application processing to define different view modification.

pfDCS* playertrans: A member (Performer pfDCS*) Used to translate scene graph for proper view rendering. Adding a group to this will have the effect of causing that group to be rendered after the main group of players, so long as the addition is done after new_view for the affected view. This technique is used for transparent objects.

static Pfmr_Renderer* Renderobj: A pointer to the Pfmr_Renderer. Set by application in main program for only one view per simulation.

Notes

Views can be used for stereoscopic views or for inset views. Whenever more than one channel is used, they share the terrain and the players. If the application needs multiple copies of geometry on separate channels, the simulation or players can have multiple copies of geometry, and mask the geometry for each on the unwanted channels.

6.3 Class Player

Purpose

This class is used for entities and stealth views in the simulation. Generally, a simulation will consist of one or more players which implement viewers and moving objects in the scene. For active entities, the player can be used to encapsulate the functions of modeling behavior, updating geometry, any drawing associated with the player, and handling any input required. For stealth views, the player's can handle moving the view, drawing, and input. For remote DIS entities, the player can serve as a data structure to hold the entity state in a standard format.

Member Functions

init(): Abstract function to perform initialization. If a player has geometry, it should be initialized here. All state initialization is done here. If player is being used as structure and not an object, this can be null, but some other function should perform appropriate initialization. Called by: Application (generally during init_sim()). All players override this function.

propagate(): Abstract function to propagate player through simulation. If a player has geometry, it should be manipulated with the Flt_Model::RotDCS here. If player is being used as structure and not an object, this can be null, but some other function should perform appropriate propagation. Called by: Application (generally during propagate() for simulation). All players override this function.

look_at_point(pfVec3 where): Utility function. Takes the coordinate system of the player and modifies the hpr to orient the positive Y axis of the entity coordinate system toward the point (where). Called by: Application (in player::propagate() as needed).

move_along_heading(float how far): Utility function. Takes the coordinate system of the player and modifies the xyz to move the player along its heading vector. Called by: Application (in player::propagate() as needed).

pfCoord* Coords: Structure contains two pfVec3s, hpr and xyz. This is used to hold entity's position and orientation. All other parts of program with access to this player will get position and orientation from this member. Must be allocated (either with new or from shared memory, if necessary).\

Flt_Model* Model: Holds the representation for this player in geometric scene, if any. Used to read in and manipulate the geometry associated with the player.

static Pfmr_Renderer* Renderer: Allows all players access to the Pfmr_Renderer. This is used to enter geometry into the scene, etc. Static, so application only must set this for one player.

Notes

Players are normally used in one of two ways: as an active entity dependent on some behavior model, and as a data structure to hold information about a simulation entity. When used the second way, the member functions are generally not used, although they can be (except when the player is allocated from shared memory, as noted earlier in the guide).

6.4 Class Attachable_Player : public Player

Purpose

Subclass of player used for players which can have views attached. Used for stealth viewers and other players where the viewer is directly attached to the player.

Member Functions

draw(): Virtual function. Perform draw process functionality for this player. Player can override this. Called by Application (usually during Simulation or view draw call).

pfVec3 base_offset: Used to hold an xyz value which represents the viewer's relative position to the origin of the player. Modifies the view calculation to place the view correctly. Set in init or propagate.

pfVec3 base_rot: Used to hold an hpr value which represents the viewer's relative orientation within the player. Modifies the view calculation to orient the view correctly. Set in init or propagate.

Notes

Attachable_Players are useful for stealth viewers and entities the user controls or 'rides' in the simulation. Multiple players can be attached in multichannel simulations, which allows for multiple viewpoints into the dadabase.

6.5 Class Flt_Model

Purpose

This class holds a player's representation in the Performer scene. It can also be used to hold geometry not associated with a player. Encapsulates any format/Performer loader dependencies into this class.

Member Functions

readmodel(const char* modfile, int modelname, int& found): The modelname is the integer index into the array used for instancing. If a previous model has been loaded with that name, this model will not load it again, but instance it..

Otherwise, calls the flight format loader to read in the geometry usiig filename. Found will report the outcome.

readlodmodel(const char* modpath, const char* modfile, int modelname): Uses a format where the LODs for a model are speciified in a separate file. Obsolete, and remains for historical purposes.

OrientDISModel(pfSCS* CorrectSCS, pfGroup* DISRoot): Inserts an SCS into a model to oriient a DIS model, defined with y out the nose and z down, into the Performer space of x out the nose and z up. Returns a pointer to the original root and to the SCS used in the correction. If another DIS model (non-oriented) is added as a child, it should be added to the SCS returned from this.

OrientDISModel(): Inserts an SCS into a model to oriient a DIS model, defined with y out the nose and z down, into the Performer space of x out the nose and z up. FfFlattens the resulting transformation to improve speed.

pfDCS* RotDCS: Use this DCS in Performer calls to manipulate the geometry. Also, add children to this, rather than the root, if the child is not a part of each instance of this geometry (e.g, some geometry only for one copy of the model, not all copies). When the root is added to this and this added to the Pfmr_Renderer::players, the geometry will appear in the scene (provided you use pfDCSCoord or some other call to set the DCS).

pfGroup* root: The actual geometry.

Notes

Each Flt_Model must be configured before use. The constructor must be called, followed by pfNewDCS on the DCS. If no geometry is desired, the root can be left null. Removing and adding the root to the RotDCS will perform a switch operation to turn the geometry on and off.

6.6 Class Pfmr_Renderer

Purpose

This class encapsulates the main loop and callls from the draw and cull threads back into an ObjectSim application. It contains the standard draw and cull thread callbacks, and calls to the simulation, terrain, and view classes to accomplish the simulation.

Member Functions

init(Simulation* theapp, int numpipes, Terrain* theterrain): Initializes the renderer. Calls the Simulation::alloc_shared(), followed by the Simulation::init_sim(). Processes are forked after this call. Requests a frame rate of 60 with free-run. Can be overridden in init_sim. Called by: Application (after all the relationships are built in the main program).

render(): Executes the main loop for the simulation. Will call Simulation::propagate, followed by set_view for each view, and then release the frame to be drawn. The cull and draw threads will return callbacks inside this class, which will then invoke the appropriate methods in the Simulation or Views, as discussed in thise class references.

arbitrate(View* theview, int desired_pipe): Will request a channel on the desired pipe. Causes the pipeline to be opened if it hasn't already. After this call, the View::init_draw will be called to open a window. Called by: ObjectSim (by standard View::new_view()) or by Application, if it overrides the new_view call (unexplored).

toggle_view(View* theview, int viewmode): Takes a mode of VIEW_ON, VIEW_OFF, or VIEW_MASKED. VIEW_ON is the default. When VIEW_OFF is the mode, the Pfmr_Renderer will not call the set_view method and the pre_draw and draw members of the view. When VIEW_MASKED is the mode, the Pfmr_Renderer will still call the draw member of the view. VIEW_OFF and VIEW_MASKED both will not render the scene graph. Called by: Application, to switch on or off a channel.

setdrawmode(int Mode): Sets the graphics pipeline to either NORMAL, OVERLAY_SCREEN, or OVERLAY_IN_SCENE. Used by application or view draw methods (or players) to set the mode for text and graphics drawing. OVERLAY_SCREEN is for text or graphics in the 2D viewport. OVERLAY_IN_SCENE is for text and graphics in the 3D scene. Note that OVERLAY_IN_SCENE drawing must be translated by the negation of the viewpoint because of the ObjectSim viewpoint convention. Called by: Application, as discussed above. Application must return the pipeline to NORMAL after finishing drawing.

insertmodel(Flt_Model* Model): Called to insert a newly read model into the scene. Will call pfNewDCS for the RotDCS, add the root as a child, and add the DCS to the players member. Call only when the geometry is to be added to the players group with no special nodes between the RotDCS and the root. Called by: Application.

getscenebounds(pfBox* box): gets a bounding box for the players. Selldom needed or used.

pfGroup* players: A pfGroup used to hold geometry in the scene. Use this to hold all geometry unless the application requires some geometry to be rendered after other geometry (see View class).

Notes

This class handles all standard callbacks for the draw and cull threads from Performer. Your application can run on up to 7 processors, plus any forks you do. Although it calls member functions in the ObjectSim classes for a nicer interface into the components of the application, it cannot free the application from using shared memory in the Simulation, View, and Player objects to share data across threads. The best design is to localize shared memory for these objects within the subclasses you declare, and so preserve the class modularization. This guide has samples of this technique.

6.7 *Class Terrain*

Purpose

This class encapsulates the environment the simulation runs in. This includes the lighting, ground the simulation is over, and the Performer earth-sky model used to present a background for the simulation. This is an abstract class which is independent of the format the geometry is in. Therefore, an ObjectSim simulation can use terrain read in as a model or generated in some way by the application.

Member Functions

configure_channel(pfChannel* chan): abstract function. Called by the View::new_view() function to configure the View's channel. If the simulation has multiple viewpoints which share the same earth-sky model, for instance, then a single configure_channel will configure each view. Called by: ObjectSim (View:new_view).

clamp(pfCoord* ClampCoords): Utility method to clamp a set of coordinates to the terrain. Does not modify the hpr at all, but changes the xyz to equal the terrain point directly under the x and y of the object. Could be rewritten to also orient the pitch and roll with the terrain.

init_draw(): Abstract function. Terrain maintains the light source for the sun, which provides the illumination for a scene. Light sources are maintained on the draw thread in Performer. This call allows the terrain to initialize its lighting source. Called by: ObjectSim, when each draw thread is initialized with the first View:new_view() on a pipe.

draw(): Abstract function. Called once per frame per channel to allow the terrain to maintain the lighting and turn it on for a scene. Called by: ObjectSim, during draw thread functionality for each frame.

Round_Earth_Utils* REU: A pointer to the Round_Earth_Utilities. These utilities are to be initialized to reflect the position of the terrain on the earth. Once they are properly initialized, they are used to convert coordinated (See entry for Round_Earth_Utilities class).

Notes

This class is intended as a base class for implementation of terrain in ObjectSim. It reflects the way terrain fits into an ObjectSim simulation. The subclasses will provide the implementation for the terrain geometry and will specify the lighting and earth-sky model used to simulate the terrain.

6.8 *Class Simple_Terrain : public Terrain*

Purpose

Imlpements the terrain class for the case when terrain is a single flight model. Also provides base functionality for the lighting, earth-sky, clamping, and round earth transformations.

Member Functions

build_terrain(char* filename, int modelindex, double origx, double origy, double origz): Reads in a single flight file which defines the terrain. The double precision coordinates define the location of the terrain's origin relative to the center of the earth. Assumes the terrain is described as earth skin. Orients the terrain to fit into the Performer earth-sky box and into the local ObjectSim coordinate system. Called by : Application (during init_sim() call). Can be overridden.

build_terrain(char* filename, int modelindex, double origx, double origy, double origz, pfMatrix Terraform): Like previous method except the matrix provides the transformation which would orient the flight file as earth skin, which assumes the terrain is not already on the skin of the earth.. In this way, the terrain can 'pretend' it's defined as earth skin when it isn't. Using this, a simulation can run over terrain defined in flat earth and send/receive round-earth network traffic.

configure_channel(pfChannel* chan): Provides an implementation for the abstract configure channel. Creates an earth-sky-ground model with default green ground plane. Override this to create a custom earth-sky, or directly modify the eSky member of this class after new_view is called on an initial view. Called by: ObjectSim (View:new_view()). Can be overridden.

set_sun(pfVec3 ambience, pfVec3 position): Sets the position and color of the 'sun' lightsource to the parameters passed in. Called by: Application (on draw thread, typically during a Simulation::pre_draw call).

Notes

This class also contains basic implementations for the abstract methods of its parent class. In the initial version, methods implementing a class of static features have also been used, but these should be removed and put in another class to manage them.

6.9 *Class Modifier*

Purpose

Modifiers are used to provide an implementation independent way to interface devices into the simulation. Modifiers represent devices whch return an offset and rotations in world coordinates. Views can have modifiers assigned to them, which makes the view calculation take into account the modifier's data value. Modifiers are typically used to represent spaceballs, head trackers, or similar devices.

Member Functions

init_state(): Allocates the state (pfCoord which represents the world coordinate values of the modifier) out of shared memory. Call if modifier must use shared memory. Called by: Application (in alloc_shared call).

init(): Abstract function. Each modifier class must define a way to initialize inself. Called by: Application (Either in init_sim() or init_draw(), depending on interface requirements.

poll(): Abstract Function. Each Modifier must provide a method to poll and get a new value. Could also write a modifier which operated as an autonomous task. Application (Either in propagate() or pre_draw(), depending on interface requirements.

reset(): Virtual function. Base capability just resets state data to zero. Called by: Application.

pfCoord* State: Coordinate system defines the current rotations and offsets for the modifier. This member is used by the View:setview() function to apply the modifications to the viewpoint.

Notes

When used with the view class, modifiers give offsets and rotations. They can also be used by players, for instance, to define control inputs, etc. The idea is that the device state is independent from the device implementation.

6.10 Modifier Classes

NQ_Modifiers

NQ_Modifiers are inherited from the class NQ_Modifier. Indtead of reading a device queue (like the standard GL device queue or a GUI device queue) they allow another part of the program to handle this. NQ_Modifiers use a function pointer to queue their events, and provide a routine which checks a single event against their list of events. For instance, the NQ_Keypad_Modifier will queue keypad events (init_extern_read) which it uses to modify its state. It then will test keypad events (handle_event) to actually modify its state. The NQ_Spaceball_SGI_Mod works the same way (since the SGI spaceball uses the event queue). NQ_Modifiers will work with any device queuing scheme which uses the GL device tokens. This guide presents examples with both the GL device queue and the forms device queue.

Other Modifiers

Other modifiers do not use the event queue. These include the Mouse_Mod (mouse inputs), the Spaceball_Mod (Dimension 6 spaceball) and the Polhemus_Mod (Fastrak head tracker). Other modifier class possibilities include modifiers based on forms (sliders/buttons) or the Fakespace boom.

6.11 DIS Classes

Base_Net_Player

The Base_Net_Player implements a standard remote or local network entity. It includes appearance fields, data fields for the round earth data coming from the network, and a virtual function (accept_damage) used to notify local entities when a network event affects them. Player classes can be inherited from this class. Base_Net_Player also includes a Model_Manager*, used so the entity can be model switched to maintain correct appearance.

Sim_Entity_Manager

This class handles several network interface functions. They include receiving network events and processing them against local entities, removing unneeded player references, and model switching local and remote entities. Local and remote entities are entered with the register_player method. They remain until their entity_state member becomes 'inactive', at which point the reference is removed. Players are registered as either LOCAL or NETWORK.

If a player calls register_player, the Sim_Entity_Mgr will perform several functions. It will model switch the player if the appearance fields change. LOCAL players will have appearance managed by this class if they are registered and they change the appearance fields in the Base_Net_Player portion of their member data. Sim_Entity_Manager will test events against all LOCAL players and call the accept_damage method when an event affects the player. It will also insert effects into the scene graph as appropriate.

The playerlist is available as a list of simulation entities. This is a shared memory array of pointers to players which reflects all registered players.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1993

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
OBJECTSIM - A REUSABLE OBJECT ORIENTED DIS VISUAL SIMULATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Mark I. Snyder, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-20

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/ASTO
3701 North Fairfax Drive
Arlington, Va 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research designed and implemented a reusable Distributed Interactive Simulation (DIS) visual simulation architecture for Silicon Graphics platforms. The goal was to research software architecture technologies and to create a design and implementation using these ideas. The architecture was designed using object oriented techniques to provide the ability to customize it via inheritance extension. The resulting design was implemented using C++ and applied to several DIS visual simulation projects in the Graphics Lab at AFIT. The architecture, named ObjectSim, was successful in its goal of providing a reusable core for the DIS visual simulation projects in the Graphics Lab at AFIT. It provides simulation developers reusable capabilities in the areas of rendering, data display, device interfacing, and DIS network interfacing. The projects designed and implemented with ObjectSim exceeded their research goals. Data on reuse effectiveness and several different performance areas was collected.

14. SUBJECT TERMS
Simulators, Distributed Interactive Simulation, Synthetic Environments, Computer Graphics, Object Oriented Design, Software Architectures, Software Reuse

15. NUMBER OF PAGES
168

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL