Architecture, Design, and Implementation of a Rapidly Prototyped Virtual Environment for a Military Spaceplane

Scott A. Rothermel

Architecture, Design, and Implementation of a Rapidly Prototyped Virtual Environment for a Military Spaceplane

THESIS
Scott A. Rothermel, 2Lt, USAF
AFIT/GCS/ENG/97D-17

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

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/97D-17

ARCHITECTURE, DESIGN, AND IMPLEMENTATION OF A RAPIDLY PROTOTYPED VIRTUAL ENVIRONMENT FOR A MILITARY SPACEPLANE

THESIS

Scott A. Rothermel, 2Lt, USAF

AFIT/GCS/ENG/97D-17

AFIT/GCS/ENG/97D-17

Architecture, Design, and Implementation of a Rapidly Prototyped Virtual Environment for a Military Spaceplane

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Scott A. Rothermel, B.S.

2nd Lieutenant, USAF

December 1997

AFIT/GCS/ENG/97D-17

ARCHITECTURE, DESIGN, AND IMPLEMENTATION OF A RAPIDLY PROTOTYPED VIRTUAL ENVIRONMENT FOR A MILITARY SPACEPLANE

THESIS

Scott A. Rothermel, 2Lt, USAF

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

[illegible]
Sheila B. Banks, Ph.D., Maj, USAF
Member

[illegible]
Keith A. Shomper, Ph.D., Maj, USAF
Member

[illegible]
Martin R. Stytz, Ph.D., LtCol, USAF
Chairman

ACKNOWLEDGMENTS

Although a Master's degree is a significant personal achievement, I would like to thank several people for their support. First and foremost, I would like to thank Capt John Lewis and Lt Troy Johnson for completing our triad and making my graduate experience both an enjoyable and successful experience. It was much easier to work on those long weekends and late nights with someone there to bounce ideas off and provide occasional diversions. If John, Troy, and I made a triangle, then LtCol Stytz provided the fourth point that brought everything into three dimensions (my thesis concerned graphics after all). His vision, drive, and continual raising of the bar made each day a new, exciting challenge. Since I am on a roll, I would like to thank the entire GCS-97D class for providing the texture that made AFIT more than just a place to get my Master's. I only hope the entire Air Force is full of the same friendly, motivated, all-around great people that I had the pleasure of spending the last 18 months with. Representing the high performance rendering hardware in this analogy I am developing, were Dan Zambon and Dave Doak. I would like to thank these gentlemen for all their hard work behind the scenes to make sure everything went smoothly whether it was getting machines working on Saturday mornings or getting HMD's fixed on a moments notice. And finally, I would like to thank my family for bringing everything into perspective. No matter where in the world the Air Force takes me, my heart will always be with you in South Dakota.

Scott Rothermel

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract

The new Global Engagement vision places increased emphasis on the Air Force's ability to control and exploit space. A military spaceplane combining reliable access to space, high operational tempos, and multi-mission capabilities is in conceptual stages of development. Virtual environment technology provides an opportunity to investigate system requirements and unconventional interface paradigms for this unique vehicle.

A virtual environment architecture and design based on support for a rapid prototyping development process, separation of concerns, and user interface development is presented. The rapid prototyping process allowed management of changing requirements via an evolutionary approach to implementation. Separation of the activities performed by the virtual environment into classes enabled high performance through computational distribution, prevented modifications from rippling through the system and impeding development, and promoted reuse of computation and geometric models. A technique was developed to reduce the flimmer induced by the large spatial extent of the virtual environment.

The architecture succeeded in providing a flexible framework for the AFIT Virtual Spaceplane. The Virtual Spaceplane is a large scale virtual environment within which an immersed user commands a military spaceplane through atmospheric and orbital regimes to complete several simulated missions via an unconventional virtual interface.

ARCHITECTURE, DESIGN, AND IMPLEMENTATION OF A RAPIDLY PROTOTYPED VIRTUAL ENVIRONMENT FOR A MILITARY SPACEPLANE

1. INTRODUCTION

The United States Air Force recently released a new vision statement to provide strategic guidance for the start of the next century; Global Engagement: A Vision for the 21st Century Air Force. Whereas Global Reach - Global Power, the previous vision, transitioned the Air Force out of the Cold War, the new vision leads the Air Force into the next millennia.

Global Engagement is unique in its emphasis on the Air Force's relation to space. Fifty years ago, the Air Force was created to take advantage of the military aspects of powered flight. As we enter the second fifty years of its existence, the Air Force is beginning to fully investigate the advantages and capabilities provided by the control and exploitation of regions outside the Earth's atmosphere. Global Engagement never refers to the Air Force as simply an air force; it is always an air *and space* force. In fact, it specifically states that "we are now transitioning from an air force into an air and space force on an evolutionary path to a space and air force" [DAF96]. Space operations span all the core competencies - rapid global mobility, precision engagement, global attack, air and space dominance, information superiority, and agile combat support - and their importance to the national security of the United States will only increase as we enter the next century.

One future embodiment of the Global Engagement vision is a military spaceplane. A military spaceplane would project force and support national interests around the world with a unit based in the continental U.S. Unlike reusable space vehicles designed for commercial or scientific applications, a military spaceplane would complete missions including transportation, force projection, reconnaissance, and deployment of space assets supporting the strategic and tactical goals of theater commanders. Reliable access to space will drastically change how military goals are achieved.

However, development of a capable, cost effective military spaceplane is fraught with difficulties. Very little doctrine is established for completion of direct space operations. Requirements for the military spaceplane are based on best guesses of the capabilities and applications of space assets. Unconventional interface paradigms are needed to filter and display tremendous amounts of data and enable a single spaceplane commander (and possibly co-commander) to manage the systems, maintain situational awareness, and accomplish mission goals.

Virtual environment technology provides a medium for investigation and experimentation to determine requirements and solutions for a conceptual military spaceplane. Development of experimental vehicles traditionally requires extensive time and money for the assembly and testing of models and vehicles. Virtual environments allow users to immerse themselves in a situation and directly determine the effects and results of a particular implementation or capability for the spaceplane being developed.

Unfortunately, the development of large, active, complex virtual environments involves its own set of difficulties and pitfalls. As with any complex software system, implementation and requirement changes can ripple through a system and require tremendous effort to maintain the integrity of the initial architecture and design. Many traditional design approaches combat this effect by trying to specify the problem and solution before implementation even begins. This approach is unacceptable for a virtual environment intended to support the design of a conceptual vehicle such as a military spaceplane - too many unknown factors exist at the beginning. A few of the unknown factors are issues such as:

1. How will the dynamics of the vehicle by simulated?
2. How will the user control the spaceplane?
3. What missions need to be supported and to what extent?
4. How can immersion in a virtual environment be used to enhance the situational awareness of a spaceplane commander?

Effective virtual environments are also confronted with demanding performance requirements, typical of real-time systems. Imaging systems must produce sufficiently realistic images at frame rates exceeding 15 frames per second, while hundreds of entities are concurrently simulated to achieve a dynamic and realistic environment. User interaction must be detected, processed, and its effect demonstrated in less than 100 ms

[Ellis91][Macedonia94][Stytz96]. All of these constraints must be met to induce the immersion of a participant into a virtual environment.

A key to meeting these challenges is the underlying software architecture that the virtual environment is based upon. The architecture and design must withstand continuous, concurrent implementation changes by multiple researchers during the development cycle. It must be flexible enough to allow requirements to be investigated, changed, added and removed without wholesale redesign and recoding effort. It must support multiprocessing capabilities needed to achieve demanding performance constraints. Unfortunately, little investigation has been conducted into architectures that can satisfy all these conditions.

1.1 Thesis Statement

Develop a virtual environment architecture that specifically acknowledges unknown and changing application requirements, high performance constraints, development of the user interface, and concurrent implementation and use the resulting architecture to rapidly prototype a virtual environment for a military spaceplane.

This research effort was conducted in close cooperation with two other Master's students, Capt John Lewis and Lt Troy Johnson. Although we cooperated to accomplish the overall goal of developing a virtual environment for a military spaceplane, each member contributed by investigating separate areas key to the success of the project.

Capt John Lewis concentrated on developing the user interface for controlling the spaceplane. Rather than relying on conventional cockpit paradigms using a throttle and stick to control the vehicle, Capt Lewis investigated techniques employing the capabilities possible with virtual environment technology and user immersion [Lewis97].

Lt Troy Johnson investigated techniques for modeling a multi-regime vehicle, like the military spaceplane, that operates both inside and outside of the Earth's atmosphere. No complete aerodynamic/astrodynamic model existed for the military spaceplane, so multiple, independently developed models simulating separate portions of the overall flight regime were acquired. Lt Johnson's research focused on the integration of the diverse models and their various coordinate systems, and the smooth, stable transition from one to another [Johnson97].

The primary focus of this research was the development of a virtual environment architecture that specifically attacks the problems of unknown and/or changing requirements, high performance constraints, lack of support for user interface development, and the control of concurrent changes that result from a rapid prototyping development process.

Because of the wide operational envelope of a military spaceplane, the virtual environment must accurately model an area of space over 400,000 km in diameter. This expansive region of interest creates problems with maintaining state information of entities and with rendering scenes in the virtual environment. As such, a secondary focus was the investigation of techniques for handling the enormous spatial extents in the military spaceplane virtual environment. These topics will be discussed in detail in the following chapters.

1.2 Overview and Terminology

Before providing an outline of this thesis document, several terms used throughout must be defined. The term *military spaceplane (MSP)* will be used whenever properties of the generic class of vehicles are discussed. In contrast, the term *Gryphon* will be used to describe the specific instantiation of a military spaceplane simulated in the virtual environment. This name arose during development and follows the military tradition of christening vehicles with suitable nicknames. The final term, *Virtual Spaceplane (VSP)*, will refer to the entire virtual environment encompassing the architecture, interface, all the entities of the environment, and the Gryphon.

The current chapter established the relevance of this research and defined the purpose of the VSP project and of this specific effort. Chapter 2 presents background material necessary for appreciating the research, while Chapter 3 defines the requirements for completion of the initial VSP. The Architecture chapter explains the development process, the architectural goals, and the final VSP architecture. Chapter 5, Design, describes how and why the architecture was instantiated into a set of object-oriented classes that provided the framework for developing the VSP. The Design chapter will be followed by a discussion of several implementation issues in Chapter 6 and the results of the initial VSP effort in Chapter 7. The document concludes with conclusions and recommendations for future work.

2. Background

The previous chapter set the stage for this research effort by demonstrating relevance to the Air Force, describing the purpose of the Virtual Spaceplane, and establishing the focus of this effort with regard to the companion efforts of Capt John Lewis and Lt Troy Johnson. This chapter provides background needed for understanding and appreciating the focus of this research. After opening with the military spaceplane, distributed virtual environment concepts and architectures will be characterized. The background chapter closes with an introduction of IRIS Performer and discussion of several existing large scale virtual environments.

2.1 Military Spaceplane

The Air Force Space Command (AFSPC) and Air Force Material Command (AFMC) Military Spaceplane Integrated Concept Team is currently investigating technologies and plans for developing a reusable launch vehicle for the military exploitation of space. This conceptual vehicle would be launched from earth (horizontally or vertically), climb out of the Earth's atmosphere into a low-Earth orbit, perform a variety of military operations, and then return to Earth.

The military spaceplane (MSP) would share some properties with NASA's Space Shuttle (Space Transportation System or STS), but it would be radically different in many aspects. While the STS was designed primarily for the transportation of scientific experiments to low-Earth orbit and launch and/or retrieval of satellites, the MSP's range of mission profiles would be more diverse. To support short notice reconnaissance missions, the MSP could take off and fly over any point on the Earth within 90 minutes. Space operations would include launch and retrieval of military assets, as well as rendezvous and docking missions with other orbital vehicles. At this time, it is expected that the MSP will be operated manned, unmanned, or virtually commanded.

The military spaceplane must be revolutionary in terms of pre-launch operations. The STS requires thousands of personnel to maintain and prepare it for launches that are typically separated by several months of downtime. The MSP will combine low maintenance and low turn around times (on the order of half a day) to support operational tempos and maintenance costs normally only associated with traditional aircraft. This capability will be supported by a standard containerized payload system for rapid mission reconfigurations [MSPICT97] [PLSTD97].

2.2 Distributed Virtual Environments

A Virtual Environment (VE) involves the use of various technologies to immerse a person into an artificial environment such that they act and respond as if the environment was real. Distributed Virtual Environments (DVE) extend this concept by using network technology to enable entities on multiple, geographically separated computer hosts to interact in the same environment. DVE research is a broad field that incorporates software architecture, 3D computer graphics, traditional simulation, human-computer interaction, networking, and artificial intelligence [Stytz96].

Various taxonomies have been proposed to describe the effectiveness and/or components of a distributed virtual environment. These taxonomies highlight the goals and purposes that should be considered when developing a DVE.

Zeltzer offered a cube model that illustrated the degree with which a virtual environment captures three non-orthogonal components [Zeltzer92]. His AIP-cube described a virtual environment in terms of three qualitative measures: autonomy, interaction, and presence. Autonomy is associated with the computational models used in the VE and their ability to react to events in the environment. Interaction is achieved by supporting user modifications to the environment and presenting the results of these modifications in real time. The final dimension of the AIP-cube relates to the user's perception that they are part of a real environment, not simply a participant in a virtual world. A system with complete presence would integrate input from all five senses and be indistinguishable from reality.

Zeltzer's AIP-cube is shown in Figure 1 with several corners labeled with example systems. The origin of the AIP-cube, signified by no autonomy, interaction, or presence, could be thought of as a program that simply prints out the text "This is a virtual environment" and ends. Complete presence would make an excellent amusement ride, and combined with autonomy would equate to an immersive theatrical experience. The ultimate goal is virtual reality with all three components fully integrated into a single system.

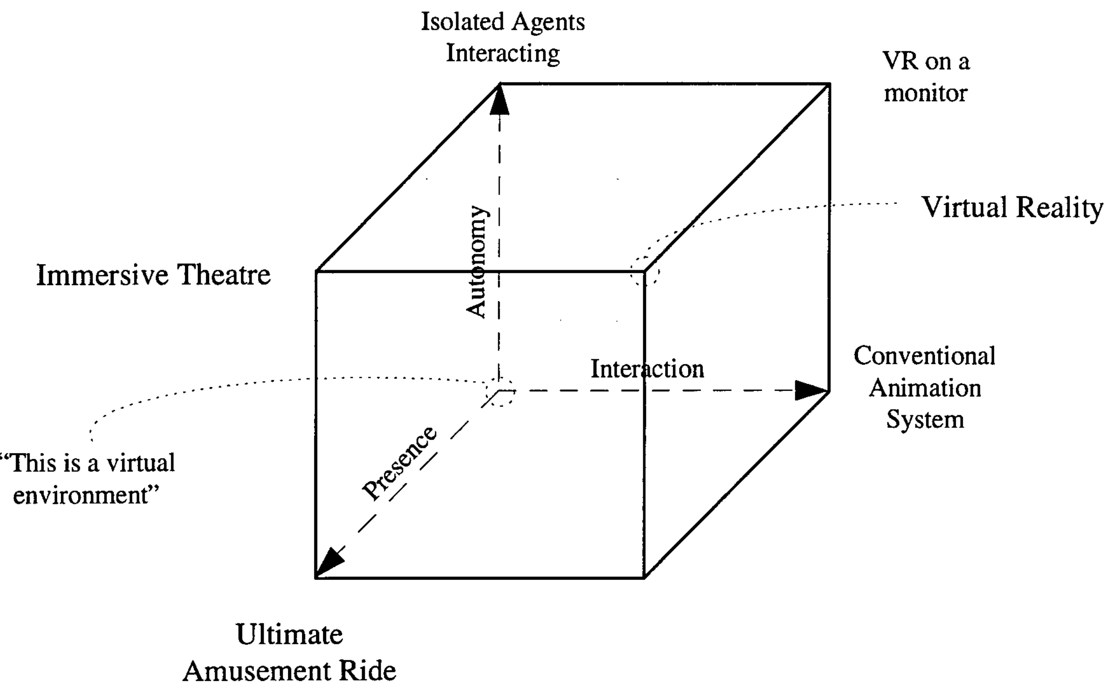


Figure 1 - Autonomy, Interaction, Presence cube.

Whereas Zeltzer's AIP-cube characterized a VE's use and effect on the user, Macedonia and Zyda provided a breakdown of the architecture of the DVE, particularly with respect to how the network, views, data, and processes are distributed [Macedonia97]. The network component of a DVE, as is typical of most network applications, is concerned with bandwidth, data distribution (broadcast, multicast, or point-to-point), reliability, and latency. In very large DVE's involving hundreds or thousands of entities, the network component is currently the primary limit to performance.

Views provide a visual representation of the virtual environment and are therefore intimately related to 3D graphics. Views are typically either synchronous or asynchronous. A system with multiple views, each of which is rendered on a separate machine and combined to form one coherent image of the environment, would utilize synchronous views. Most virtual environments use what Macedonia refers to as asynchronous views. In these environments, a separate view is associated with each entity and all the views are related with regard to the single environment within which they interact.

While the network component describes the communication and information flow between entities, the location of the state information of each entity is central to the data component. The four most common methods for data distribution are listed in Table 1 with example systems that utilize the method.

Table 1 - Data distribution techniques in Virtual Environments.

Data distribution	Example system
Replicated homogeneous	SIMNET, DIS
Shared centralized	Vistel, MUD's
Shared distributed, peer-to-peer	DIVE
Shared distributed, client-server	WorldNet, Massive

In systems using replicated, homogeneous distribution, each distributed host maintains a separate copy of the entire virtual environment and hosts use broadcast messaging to keep all the environments consistent. Shared, centralized environments, typical in text based MUD's (Multi-User Dungeon), employ a single database maintained on a central server. Users send messages to the server, which processes and redistributes the message to all the other current users. In the shared distributed paradigms, the environmental database is partitioned among the various hosts in order to reduce bottlenecks around a central server. In the peer-to-peer variant, reliable multicast communication is used to share state information between relevant hosts, while the client-server variant uses a central server to dynamically configure multicast groups and determine which host an entity exists on.

The final component in Macedonia and Zyda's taxonomy is the approach to process distribution. Most modern systems use multiple processes (on single or multiple machines) to increase the aggregate computing power and to reduce system latency and lag. However, some VE's combine all computation, interaction, and rendering in a single process. Typically, these systems must use domain specific customizations to maintain frame rates sufficient for immersion. An example is the *virtual colonoscopy* project at the University of New York at Stony Brook which required preprocessed optimizations based on the unique shape of the human large intestine to achieve interactive frame rates [Hong97].

The preceding section presented several taxonomies for describing distributed virtual environments. These taxonomies are useful for comparing and contrasting virtual environments with differing goals or capabilities. Zeltzer's AIP-cube describes the behavior of a VE and its affect on a user, while Macedonia's classification emphasizes the attributes of four architectural components of a VE.

2.3 DVE Architectures

A wide variety of software architectures have been developed to combat the unique difficulties inherent in building DVE's. Each approach addresses different aspects of the problems of user interaction, real-time performance requirements, communication (between entities and between processes) and scalability. This section will propose a virtual environment taxonomy based on a set of activities common to all virtual environments and a method for describing how a system organizes these activities into modules and tasks. The purpose of this taxonomy is to provide a common framework within which different architectures can be compared and analyzed. Several virtual environment architectures will be discussed and presented using this taxonomy.

An analysis of the architectures manifested in the MR Toolkit [Shaw92], Walkthru-CFAST [Bukowski97], Veridical User Environment [Appino92], and ObjectSim [Snyder93] virtual environments uncovered seven primary activities that a virtual environment must perform. These activities and their purpose are:

1. Acquire Input - gather input from the user
2. Process Input - interpret user input as changes in the virtual environment
3. State Computation - determine the current status or position of all objects in the virtual environment
4. State Representation - determine how the current status or position of objects will be depicted or presented
5. Output - present non-visual aspects of the virtual environment to the user
6. Rendering - present visual aspects of the virtual environment to the user
7. Sim Management - perform initialization, destruction, time management, and coordination activities for the virtual environment.

Although the Rendering activity is a form of output, it was separated from the other Output activities to illustrate the increased attention visual presentation of the VE must be given in terms of design and processing.

The other aspect of this taxonomy is describing how a specific architecture organizes these activities into modules and tasks. A module is an object or structure designed to perform a subset of the activities presented above. A task is an independent process that may contain one or more modules. A task may also perform a subset of the activities independently of any modules contained within the task. Data may flow from module to module or from task to task. Figure 2 shows the symbols used to represent the components of this taxonomy.

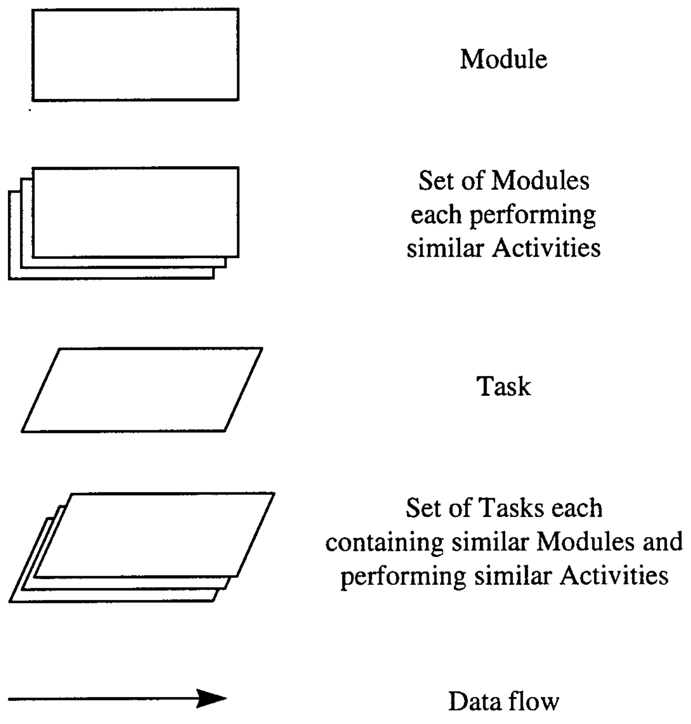

Module
Set of Modules each performing similar Activities
Task
Set of Tasks each containing similar Modules and performing similar Activities
Data flow

Figure 2 - Module and Task Symbology.

Given the seven activities, a specific architecture defines the organization and relationships between a collection of modules, tasks, and data flows that together complete all the activities.

2.3.1 MR Toolkit

The MR Toolkit, developed at the University of Alberta, was developed to support "distributed computing, head-mounted displays, room geometry, performance monitoring, hand input devices and sound feedback" [Shaw92]. The MR Toolkit used a *Decoupled Simulation* model composed of four components:

- Interaction components control input devices
- Presentation components present output to the user
- Computation components perform the traditional simulation function of calculating new state information for all the entities
- Geometry Models represent the data produced by the Computation components.

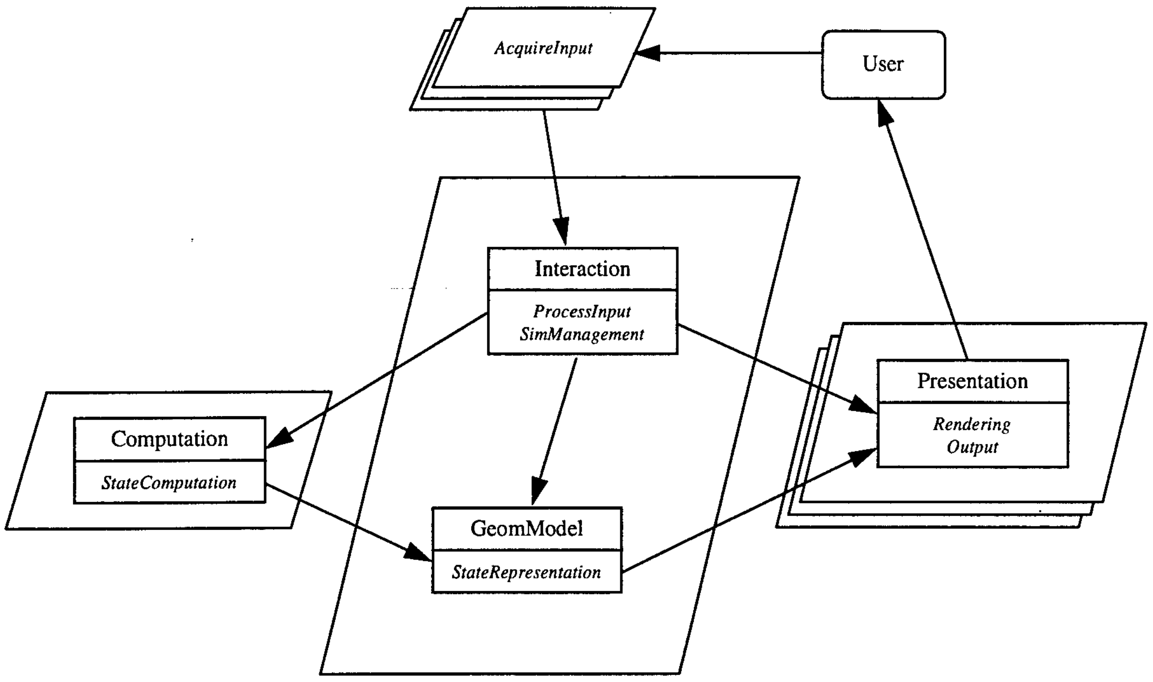


Figure 3 - MR Toolkit architecture.

Figure 3 illustrates the MR Toolkit architecture using the taxonomy described at the beginning of this section. Several components execute in their own processes (potentially on different machines) with communication accomplished via TCP/IP socket connections. Separate processes shared data using a synchronous producer/consumer model. A primary feature of the MR Toolkit was the ability to decouple the various VE components, allowing them to execute at their optimal rate and preventing changes in one component from rippling through the entire system.

2.3.2 Walkthru-CFAST

The Walkthru-CFAST architecture was the result of integrating two large, independently developed systems into a single virtual environment [Bukowski97]. The

Berkeley Architectural Walkthrough Program was designed to visualize "large (several million polygons), densely occluded building models" composed of a collection of rooms (cells) connected by doors (portals). The second component was the National Institute of Standards and Technology's CFAST fire simulator. CFAST simulates the chemical and physical reactions of fires in an environment composed of many volumes connected by vents. Bukowski refers to the systems as the *visualizer* and *simulator*, respectively.

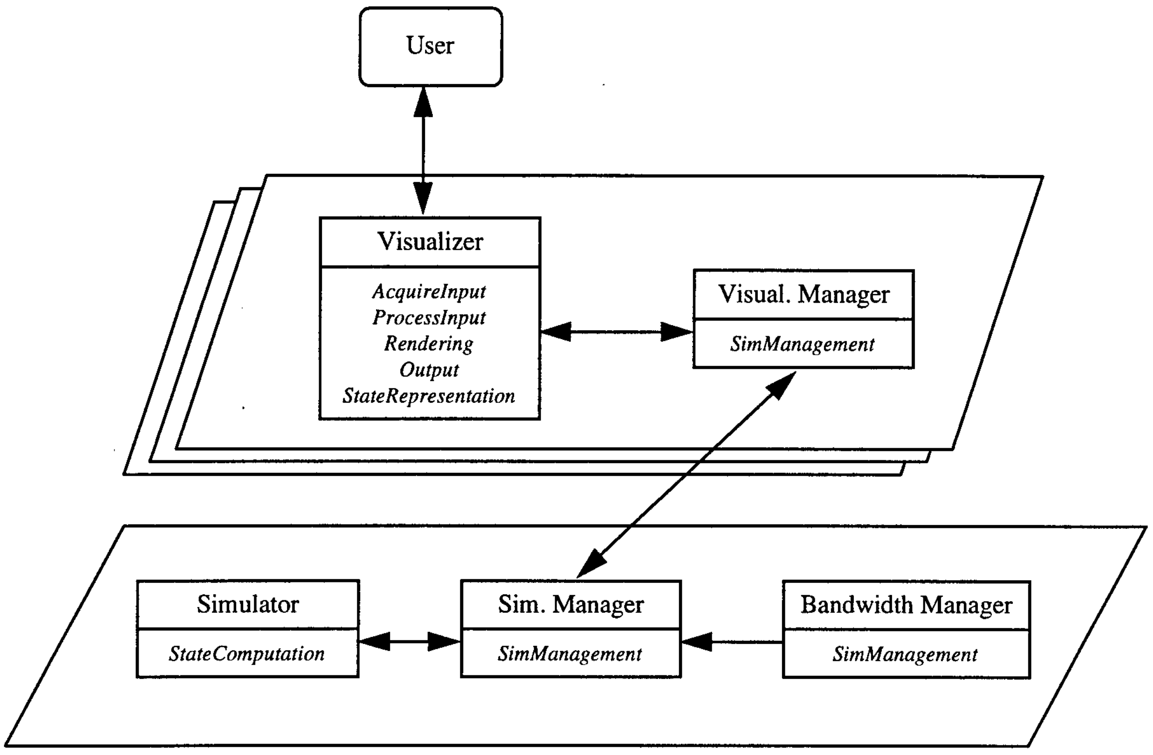


Figure 4 - Walkthru-CFAST architecture.

In the Walkthru-CFAST architecture (see Figure 4), each system executes on a separate machine (with potentially multiple visualizers accessing a single simulator) with a mapping scheme correlating cell/portal pairs to volume/vent pairs. The visualizer performs all rendering and user interface handling. The architecture utilizes several manager components (visualization, simulator, and bandwidth) together with a "just-in-time" data management philosophy to synchronize and arbitrate data flow between the visualizer and simulator.

2.3.3 Veridical User Environment

The Veridical User Environment (VUE) was similar to the MR Toolkit with its emphasis on partitioning the system into processes, but contained several features that distinguished it [Appino92]. The VUE used a collection of independent processes, each executing various portions of the virtual environment:

- Device Servers controlled input and output devices
- Application processes executed the computation models supporting the VE
- A central event driven Dialogue Manager passed messages from process to process.

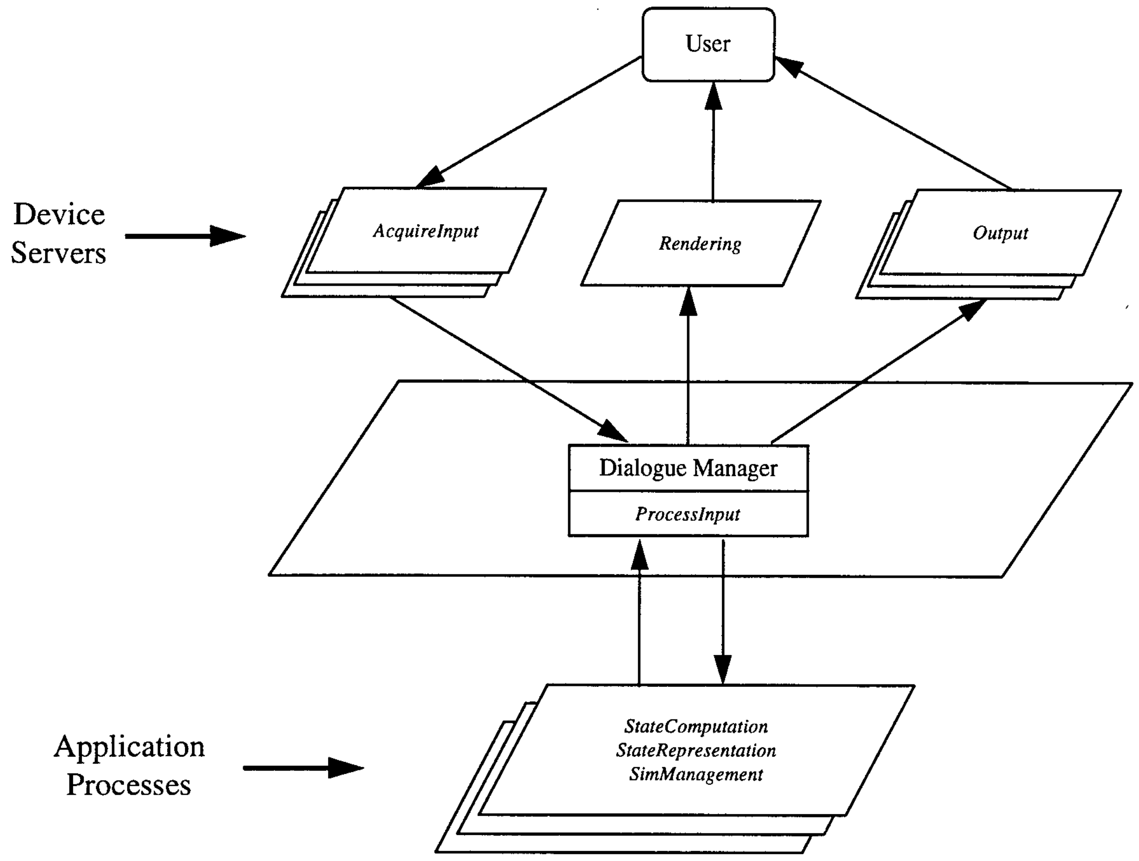


Figure 5 - Veridical User Environment architecture.

What distinguished VUE from many architectures was its asynchronous message handling and emphasis on user interaction. The VUE architecture, described using the module/task taxonomy, is shown in Figure 5.

2.3.4 ObjectSim

The ObjectSim software architecture was developed with the goal of providing an object-oriented design framework and corresponding set of classes to develop a variety of distributed virtual environment applications at the Air Force Institute of Technology [Snyder93]. By providing a common framework for development, it was hoped that the development time, maintainability, and design quality of the applications produced would improve. ObjectSim provided seven tightly coupled superclasses which individual applications modified to suit their specific needs. The classes and their functions are listed below with the architecture illustrated in Figure 6:

- The Simulation class exercised primary control of the simulation
- The Pfr_Renderer encompassed the majority of the rendering functions
- The Terrain class handled a flat-earth terrain representation
- The Player provided a base upon which simulation entities were derived
- Flt_Models abstracted and facilitated the reuse of geometric models
- The View class represented the viewpoint
- Modifier subclasses abstracted input devices
- No specific class had responsibility for processing input.

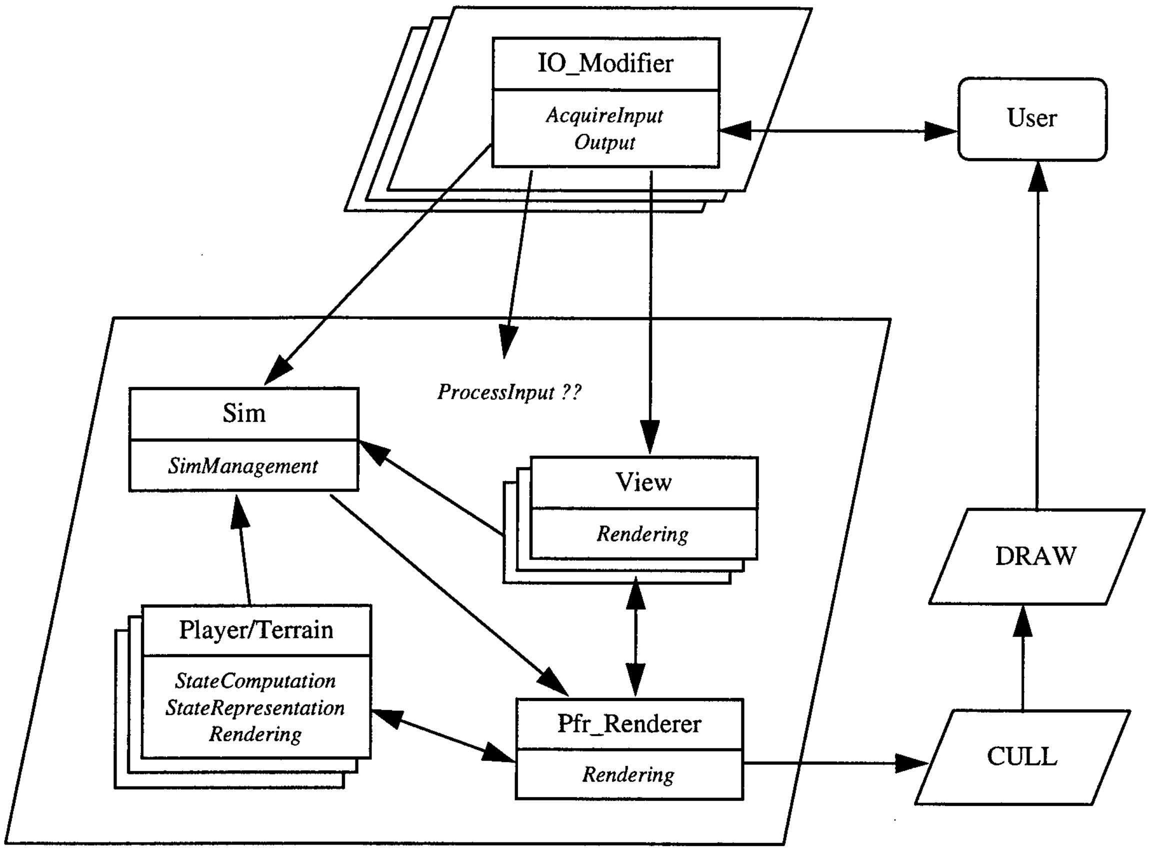


Figure 6 - ObjectSim architecture.

Several projects including the Virtual Cockpit [Erichsen93], the Synthetic Battle Bridge [Wilson93], and the Solar System Modeler [Kunz93] were developed concurrently with and using ObjectSim. The success of these applications demonstrated the usefulness of ObjectSim.

Kayloe described several weaknesses and problems associated with ObjectSim [Kayloe94]. ObjectSim was developed in conjunction with several other simulations, and in many instances was designed around these applications. Rather than forcing the applications to fit within the original ObjectSim framework, ObjectSim was modified to satisfy the applications. This approach resulted in an undesirable intermingling between the architectural framework and the implementation. Although ObjectSim was implemented using C++, it did not make use of C++'s powerful object-oriented features such as private sections of a class, polymorphism, and constructors. The result of these weaknesses was a highly coupled design that relied on global variables for communication and was subsequently difficult to understand and modify.

The preceding sections presented a taxonomy for comparing the software architectures of DVE's and several virtual environments were discussed using this taxonomy. The taxonomy describes a VE as a collection of interconnected modules and tasks that must perform a set of activities common to all virtual environments. This taxonomy will also be used to describe the VSP architecture.

2.4 Common Object DataBase

The Common Object DataBase (CODB) provides several capabilities to a virtual environment, enabling it to take advantage of multiprocessing and distributed resources required for effective execution of large, complex, distributed virtual environments [Stytz97]. Containers of data are defined and stored in the CODB. Through the use of a consistent interface, multiple processes on the same host asynchronously access and/or modify the contents of the containers. Semaphores built into the CODB protect the data from concurrent access and a double buffering scheme reduces latency due to blocking of data.

The CODB can be used in many ways to improve the execution of a virtual environment. For example, the CODB can eliminate processing stalls related to the serial polling of many input devices. VE's designed around the CODB take advantage of the computing power available in multiprocessor machines by forking off independent processes simulating specific components of a virtual environment. These processes communicate to the main process via containers in the CODB.

2.5 IRIS Performer

IRIS Performer is a suite of high performance libraries designed to build real-time 3D applications. Performer supports level of detail switching, billboards, anti-aliasing, texture mapping, illumination, intersection testing, vector and matrix mathematics, multiprocessing, and shared memory. Performer rendering is built upon two concepts; a hierarchical scene graph describing the rendered environment and a three stage rendering pipeline.

In Performer, scenes are built by creating an acyclic *scene graph* of nodes. Table 2 (based on a table in the IRIS Performer Programmer's Guide) lists some of the nodes available in Performer used while developing the VSP. The scene graph precisely describes the positions and orientations of all geometry that must be rendered. The IRIS Performer Programmer's Guide details the use of these nodes to describe a scene [IRIS95].

Table 2 - Subset of IRIS Performer scene nodes.

Node	Description
pfNode	Abstract base type of all nodes
pfGroup	Contains one or more pfNodes
pfScene	Root of the scene graph
pfSCS	Static coordinate transformation
pfDCS	Dynamic coordinate transformation
pfSwitch	Selects one of more of its children to be traversed
pfLOD	Level-of-detail node
pfLayer	Manages coplanar geometry
pfLightPoint	Points of light (no illumination)
pfLightSource	Provides scene illumination
pfGeode	Contains geometry specifications
pfText	Renders 3D text

After the scene graph is created, a three stage rendering pipeline produces 3D images in real-time. Performer obtains real-time frame rates by executing each stage of the pipeline in an independent process and by virtue of custom rendering hardware inherent in Silicon Graphics workstations. In the Application (APP) process, a program creates and modifies the scene graph based on results of the developer's simulation engine. The vast majority of code written executes in the APP process.

When the APP process has completed modifications to the scene graph, the graph is passed to the CULL process. The CULL process completes a graph traversal that determines which geometry is visible in the viewing frustum. The CULL traversal produces an optimized display list of the visible geometry and passes the list to the DRAW process. The DRAW process then renders the scene.

The pipeline architecture of Performer allows three sequential frames to be in various rendering stages at the same time, resulting in a potential sixfold increase in frame rate [Macedonia94].

Performer also supports a wide variety of external geometry specifications. Performer's flexibility allows geometry to be built using geometric modeling programs such as Coryphaeus' Designer's Workbench and loaded into the Performer scene graph at runtime. To reduce memory use and improve loading time, a conversion routine called *pfconv* is available to convert any Performer loadable format into the Performer Fast Binary (pfb) format.

Geometry can also be created in the APP process using pfGeoSets of points, lines, triangles, or polygons. While building complex models is simpler using external modelers, pfGeoSets of dynamic points and lines can be used to create many interesting effects not available in static modelers.

The VSP project relies on SGI rendering hardware and Performer software to efficiently render the virtual environment. The preceding section introduced basic Performer concepts needed to understand the VSP architecture with regard to how it integrates the Performer rendering structures.

2.6 Large spatial VE's

One of the challenges of simulating a vehicle that operates from the earth's surface to low-earth orbit, is the enormous range of positional data that must be maintained and the expansive scenes that must be rendered. Assuming that a simulation must portray objects extending from the earth's center to the apogee of the Moon's orbit, distances between objects can exceed 400,000 km. Although a MSP may only travel to low-earth orbits, the VSP must be capable of rendering objects anywhere within the Moon's orbit. Figure 7 illustrates this worst case scenario with several important orbital distances labeled.

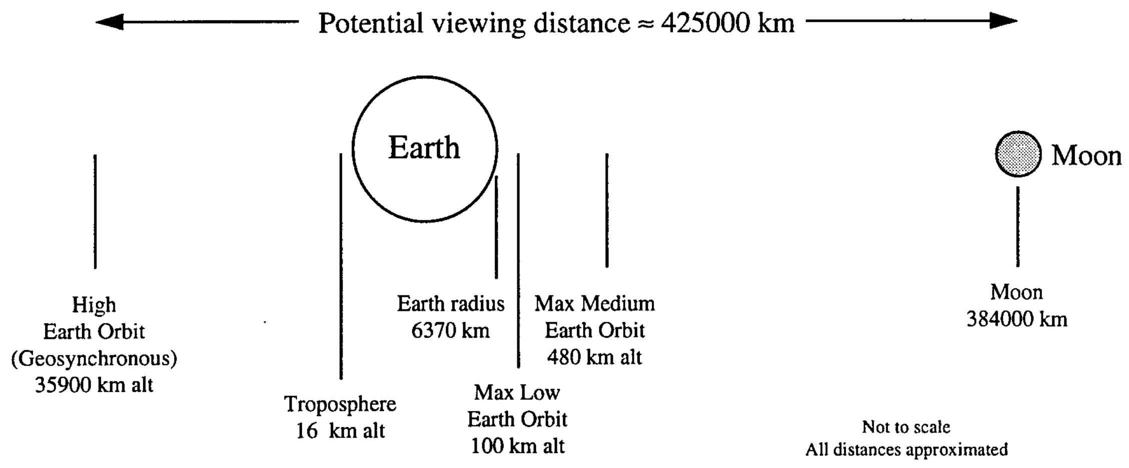

Potential viewing distance ≈ 425000 km
Earth
Moon
High Earth Orbit (Geosynchronous) 35900 km alt
Troposphere 16 km alt
Earth radius 6370 km
Max Low Earth Orbit 100 km alt
Max Medium Earth Orbit 480 km alt
Moon 384000 km
Not to scale
All distances approximated

Figure 7 - Virtual Spaceplane region of interest.

Since the MSP conducts precision operations such as orbital docking, a simulation must not only maintain large distances, it must maintain them with high accuracy. Double precision operations have the required range and accuracy for performing and storing these positional calculations. Table 3 demonstrates the accuracy differences between single and double precision floating point numbers.

Table 3 - Accuracy of single and double precision floating point numbers.

Distance from origin (arbitrary units)	Single Precision Positional Accuracy (23 bit mantissa)	Double Precision Positional Accuracy (52 bit mantissa)
1	1.19×10^{-7}	2.22×10^{-16}
10^3	6.10×10^{-5}	1.13×10^{-13}
10^6	6.26×10^{-2}	1.16×10^{-10}
400×10^6	3.2×10^1	5.96×10^{-8}

However, the limited precision available in the graphics hardware presents a rendering problem referred to as *flimmering*. During the rendering process, a technique called z-buffering determines the portions of the polygons that are closest to the viewer and are therefore rendered to the screen. The Silicon Graphics Onyx Reality Engine^2's used in the AFIT Virtual Environments Lab use 24-bit depth buffers to determine the depth levels of all polygons at pixel resolutions (the hardware actually computes depths at sub-pixel accuracy, but the concepts below apply either way) [Akeley93]. When two polygons are coplanar or very close to one another (relative to the viewpoint), the limited precision of the depth buffer prevents the rendering process from determining which polygon is in front. This inconclusive determination results in the hardware rendering different portions of each polygon from frame to frame. The net effect is flimmer.

The region of interest in the Virtual Spaceplane is from the surface of the Earth to slightly beyond the Moon (the Sun is modeled slightly beyond the Moon's orbit and is scaled appropriately). Therefore, at any time the rendering hardware must differentiate the depths of objects spanning a distance of over 400,000 kilometers. In addition, because all objects are modeled to scale, the depth resolution must be accurate to several centimeters. Unfortunately, many of the depth levels are used to model empty areas between the center of the Earth to the Earth's crust and between high Earth orbits to the Moon.

Several large VE's built in the past address and propose (directly or indirectly) solutions to the aforementioned problem. These systems and their applicability to simulating a military spaceplane are discussed below.

2.6.1 Solar System Modeler

The Solar System Modeler (SSM) was a virtual environment developed at AFIT that simulated the motion of many of the heavenly bodies within our solar system [Williams96]. An immersed user employed an interface scheme called the *Pod* [Kestermann94] to explore the planets, planetary moons, asteroids, interplanetary and earth orbiting satellites. The SSM accurately simulated the propagation and transmissions of the Global Positioning System (GPS) satellites, so remote entities could enter a virtual environment and, using a virtual GPS receiver, obtain position, velocity and time measurements.

The solar system is over 11 billion kilometers in diameter, so the SSM routinely manipulated enormous numbers. It approached the problem of rendering this space by uniformly scaling all objects and distances in the virtual environment by a factor of

1:10,000. Small objects such as the GPS satellites were scaled to make them visible from large distances. This was acceptable for the SSM because the purpose was to visualize and explore planetary and orbital motion, not represent the detailed interaction of a space vehicle with the environment and other entities in the environment.

2.6.2 Virtual Planetary Exploration project

The Virtual Planetary Exploration (VPE) project conducted at NASA Ames enabled geologists and engineers to virtually explore planetary terrain data [Hitcher93]. VPE was built using digital elevation data and projected image sets of Mars obtained from the Viking missions. This data was used to generate 3D polygon meshes that VPE visualized and the user explored using either a six degree of freedom *environment* mode or a two degree of freedom *panorama* mode. The VPE project also investigated the use of geometry culling (both viewport and object occlusion), level of detail switching, and texture mapping to improve the performance and quality of the visualized environment. The paper discusses the implementation, advantages, and disadvantages of these techniques.

Although Hitcher did not directly discuss the problem of rendering a large scale virtual environment (other than with respect to performance and data requirements), several implied observations can be drawn from his discussion. VPE used custom visualization software that employed a 32-bit z-buffer implemented in main memory. The software implementation combined with the fact that their region of visualization was smaller than the VSP (the radius of Mars is 3,398 km, whereas the radius of a geosynchronous orbit is 35,900 km and the apogee of the Moon is greater than 380,000 km), prevents flimmering.

2.6.3 Diamond Park's Locales

The final discussed approach to handling spatially large virtual environments is the *locales* concept developed at the Mitsubishi Electric Research Laboratory (MERL) and subsequently used to build a virtual world called Diamond Park [Barrus96]. The premise of locales was that the user of a virtual environment can only view and affect a small portion of the world at any given instance. Therefore, rather than building the entire world as a single environment (with a single coordinate system), the world was divided into individual locales and hierarchical relationships were built to specify which locales were connected and/or visible from another locale. A single transformation converts an entity's state from one locale to an adjacent locale.

Locales reduced problems associated with spatially large environments by propagating entities in their immediate locale and building the virtual environment such that the number and size of locales visible at a given instant was significantly smaller than the entire world together. The technique also provided regions that could be used to minimize communication between entities and increase culling efficiency.

Virtual environments that model large spatial extents must deal with the degradation of accuracy as entities in the environment retreat from the origin. Approaches applied by the Solar System Modeler, Virtual Planetary Exploration project, and MERL's Diamond Park were introduced and analyzed for their applicability to the VSP. This problem and a proposed solution will return in the Implementation chapter.

2.7 Background Conclusion

This chapter discussed several subjects needed to understand and appreciate the research completed in the development of the VSP and presented in the following chapters. The entire project revolves around the simulation of a military spaceplane, so an understanding of its purpose and capabilities is required. Discussion of example virtual environment architectures and descriptions of several VE taxonomies enables comparisons between these VE's and the VSP to highlight strengths and weaknesses of each. The chapter closed by presenting problems associated with large spatial VE and several solutions used by past virtual environments. The next chapter presents the requirements and goals for Gryphon, the AFIT Virtual Spaceplane.

3. REQUIREMENTS

Before proceeding into the Architecture and Design chapters, the requirements for the research product must be established. These requirements derived from the VSP project objective and evolved during the course of the development of the VSP.

The VSP requirements fell into five primary areas:

1. Simulated capabilities of the MSP
2. Supported missions
3. User interface
4. Virtual environment
5. Miscellaneous requirements.

The following sections describe each area. Because this research was conducted in cooperation with two other students, this thesis and the following chapters will only address a proper subset of the requirements. However, the entire set of requirements are presented to provide a global perspective of the VSP project.

3.1 Simulated Capabilities of a Military Spaceplane

The VSP must simulate the operation and characteristics of a hypothetical MSP. A MSP operates in atmospheric and space envelopes, so the VSP must support these modes as well as smoothly transitioning from one to the other. An interactive environment must provide the ability to manually modify the environment (in this case the spaceplane). Additionally, the VSP must provide capabilities for the system to accept higher level maneuvering commands and automatically perform the intended operation. However, since the VSP is an initial prototyping effort, these automatic operations need not be *intelligent* or validated; the VSP must only demonstrate support for these features. Because no MSP exists to validate the model against, no accuracy requirements were established. Table 4 lists the capability sub-tasks required of the VSP.

Table 4 - Capability Requirements.

ID	Requirement Description
	Flight Characteristics
1.11	The VSP shall simulate maneuvering on runways.
1.12	The VSP shall simulate flight through the atmosphere.
1.13	The VSP shall simulate maneuvering in space.
1.14	The VSP shall transition from one flight regime to another.
	Manual Operation
1.21	The VSP shall provide capability to manually operate MSP in the atmosphere.
1.22	The VSP shall provide capability to manually operate MSP in space.
	Automatic Operation
1.31	The VSP shall provide capability to automatically takeoff.
1.32	The VSP shall provide capability to automatically fly predetermined routes.
1.33	The VSP shall provide capability to automatically enter orbit.
1.34	The VSP shall provide capability to automatically modify orbital parameters.
1.35	The VSP shall provide capability to automatically reenter the atmosphere.
1.36	The VSP shall provide capability to automatically land.

The focus of this thesis effort with regard to the capability requirements was development of a flexible architecture that could accommodate a variety of propagation models. The propagation models and architecture should be designed to accept input originating from either users or internal agents.

3.2 Supported Missions

As discussed in the Background, the MSP will conduct a variety of missions intended to accomplish the exploitation of space to achieve military objectives. The VSP must initially only support a subset of these mission profiles as listed in Table 5. In particular, the Gryphon must be able to co-orbit with objects in low-earth orbit and deploy additional satellites into orbit.

Table 5 - Mission Requirements.

ID	Requirement Description
	Supported Missions
2.1	The VSP shall support co-orbiting with low-earth satellites.
2.2	The VSP shall support deployment of satellite.

This thesis will not address completion of these mission requirements. Interested readers should refer to Lt Troy Johnson's thesis for discussion and results [Johnson97].

3.3 User Interface

A primary goal of the VSP was to investigate non-traditional interface schemes for controlling a MSP and maintaining situational awareness without becoming overburdened with information. The requirements specifying the interface goals are listed in Table 6.

The facilities at the AFIT Virtual Environments Lab do not include reliable 3D pointing devices. Until these resources become available, all user interaction must be accomplished via a standard three button mouse. However, the interface design should consider future integration and use of 3D pointers such as the Ascension Flock of Birds™. Enhanced immersion via support for a head mounted display and associated head tracker, and configurability of the interface, specifically what information to display and where to display it, were also requirements. The interface had to present information describing the state of the Gryphon in the atmosphere, in space, and during entry/reentry, as well as providing mechanisms for changing the state of the Gryphon in all the operational regimes. The interface must assist the user in acquiring potential targets and displaying their relevant state information. The interface must relieve the user of nominal system management via diagnosis of problems and presentation of the status of consumable resources. Finally, the VSP needed to investigate the use of hyper-text paradigms and the interface had to minimize obstruction of the user's view of the virtual environment.

Table 6 - User Interface Requirements.

ID	Requirement Description
	Interaction Methods
3.11	All user interaction shall be performed via a three button mouse.
3.12	The VSP shall support a head mounted display (HMD) with head tracking.
	Configurable Cockpit
3.21	The user shall be able to display or conceal information interactively.
3.22	The user shall be able to modify the location of information displays interactively.
	Displayed Information
3.31	The interface shall display state information for the Gryphon in the atmosphere.
3.32	The interface shall display state information for the Gryphon during orbit entry/reentry.
3.33	The interface shall display state information for the Gryphon in the space.
3.34	The interface shall display status of consumables (propellants, life-support, etc.).
3.35	The interface shall display state information of potential targets.
3.36	The interface shall assist the user in locating/acquiring potential targets.
3.37	The interface shall assist the user with system management and diagnosis.
3.38	The VSP shall investigate hyper-text paradigms for display of information.
3.39	The interface shall minimize obstruction of the user's view of the VE.
	Controlling the Gryphon
3.41	The interface shall not utilize a throttle and stick for control of Gryphon.
3.42	The interface shall enable users to change the state of the Gryphon in the atmosphere.
3.43	The interface shall enable users to change the state of the Gryphon in space.

The focus of the this thesis was developing a flexible architectural framework that supported evolutionary prototyping of interface components. The exact properties, style, and mechanisms of the interface were initially unknown, so the architecture had to allow experimentation resulting in frequent changes to all aspects of the interface. The actual interface development is presented in Capt John Lewis's thesis [Lewis97].

3.4 Virtual Environment

An effective virtual environment encourages the user to make the mental transition from participating in a computer simulation to accepting immersion in the environment. This shift requires a virtual environment that looks and changes like the user expects or would accept as the actual environment. Entities must resemble their real world counterparts and react to the user or other entities in a believable fashion. Although ultimate immersion requires integration of all five senses, technological and time constraints restricted the VSP to concentrate on the visual aspects. The environmental requirements of the VSP are given in Table 7.

Table 7 - Environmental Requirements.

ID	Requirement Description
	Environment
4.1	The VSP shall present convincing terrain surrounding Edwards AFB.
4.2	The VSP shall present a convincing representation of the Earth, Sun, and Moon.
4.3	The VSP shall simulate multiple constellations of Earth orbiting objects.
4.4	The VSP shall portray the transition between day/night and atmosphere/space.

This thesis effort addressed completion of requirements 4.1 through 4.4. Requirement 4.1 included building a representation of the runways and other terrain features near Edwards AFB. Models of the Earth, Sun, and Moon provided a sense of familiarity and an intuitive mechanism to illustrate the passage of time. The Earth, Sun, and Moon should display such phenomenon as moon phases, seasonal changes in sunrise and sunset, and a visible day/night terminator when in space. An important aspect of a virtual environment is a population of dynamic entities the user can interact with; in the case of the VSP, a variety of earth orbiting objects. Visually modeling the atmosphere concentrated on performing a smooth transition from the light blue sky associated with the atmosphere to the star-filled black of space. As with previous requirements, the VSP design must support development of these requirements.

3.5 Miscellaneous

Several requirements levied against the VSP did not correspond to one of the previous areas. These requirements are listed in Table 8.

Table 8 - Miscellaneous Requirements.

ID	Requirement Description
	Miscellaneous
5.1	The VSP shall be able to exchange state information with remote entities via the DIS protocols.
5.2	The VSP shall transmit state information of the Gryphon via the DIS protocols.
5.3	The VSP shall operate at a mean rate of 15 frames per second on a 4 processor 250 MHz R4400 SGI Onyx with Reality Engine^2 graphics equipped with 16 Mbytes of hardware texture memory.

The primary mechanism for increasing the number of entities in the VSP was through the DIS protocol standard. Support for the DIS protocols enabled the VSP to join in large scale distributed environments and interact with more entities than would be computationally possible using only a local host. As such, the architecture and design must support a sufficient subset of the DIS protocols to accept state information of remote entities and to transmit the state information of the Gryphon. The final requirement ensures that the VSP maintains an interactive frame rate; users lose their sense of immersion if long latencies are introduced into the system.

3.6 Requirements Conclusion

The preceding chapter presented the overall requirements of the VSP and highlighted specific requirements addressed in the following chapters. These include requirements defining the capabilities of a MSP the VSP must simulate, types of MSP missions supported, functionality provided by the user interface, aspects of the environment modeled, and several miscellaneous requirements. The next chapter describes the development process, architectural goals, and resulting architecture for implementing the requirements of the Virtual Spaceplane.

4. ARCHITECTURE

The development of a distributed virtual environment is a challenging multi-disciplinary task beset by high performance requirements and unknown and/or constantly changing application requirements. A development environment such as this demands flexible processes and architectures designed to combat architectural and design degradation. The following sections in this chapter discuss the procedures and architectural goals used to guide the development the VSP. The chapter concludes by presenting and justifying the architecture chosen for the VSP.

4.1 Development Process

Although the specific architecture and design are vital to development of a complex system, the process by which the system is built is also of paramount importance. The process defines how requirements are developed and changed, how team members work together to achieve common goals, and how problems are detected and removed. The VSP used a rapid evolutionary prototyping process designed to produce functional prototypes built on the following principles [Stytz97]:

a) Employ Object Oriented Design techniques

b) Support undeveloped and/or changing requirements

c) Exploit Silicon Graphics hardware and IRIS Performer software for performance

d) Reduce dependence between components

e) Reuse components from previous efforts as applicable.

Projects conducted in the Virtual Environments Lab at AFIT are typically sponsored by organizations with high level goals and concepts, but few specific application requirements. In many cases, one of the Lab's purposes is to explore and define requirements. Rapid evolutionary prototyping accomplishes this exploration by defining baseline requirements using current knowledge and quickly building a functional prototype to aid in exploring these requirements. The prototype is used to refine and uncover more requirements, providing input to the next iteration of the prototype development process.

However, if this process is not managed properly, the initial architectural integrity could quickly degrade to chaos. Specific actions can be taken to combat this eventual decline by maintaining the architecture and design. Reducing dependence and coupling between components of the design is one such action. By minimizing coupling in the system, the inevitable changes in requirements and the resulting changes necessary to implement these requirements are localized to a few components. Standard object-oriented design concepts such as encapsulation and information hiding also combat the decline.

The Common Object DataBase supports the evolutionary prototyping process by providing a simple, uniform method for multi-process communication. As such, the CODB enables computational distribution with the resulting improvements in performance. It also improves maintenance of the initial architectural integrity by providing a standard interface for components of the virtual environment to communicate.

To manage the software development, the coding process had to enable concurrent development by each team member, but keep the code from diverging by using frequent integrations. The process allowed each member to pursue independent tasks concurrently by maintaining four versions of the VSP simultaneously. Each member modified and developed his own copy, while a fourth *read-only* version ensured that a recent fully functional VSP was always available for demonstration and testing. At least once a week, code integrations assembled the three diverging versions into a single functional program. Each team member then proceeded using the new integrated version. The frequent integrations ensured that problems resulting from concurrent changes or misunderstandings between team members about design or implementation goals were detected as soon as possible. Finding problems and misunderstandings early prevented the investment of large amounts of time and effort into code, based on incorrect assumptions, that must eventually be recoded or restructured. More significant changes warranted immediate integration to uncover potential problems as soon as possible. The use of concurrent development, frequent integrations, and continual testing is similar to the *synch-and-stabilize* method employed by Microsoft [Cusumano97].

The success of a rapidly prototyped project depends on finding problems and deficiencies before significant portions of the system become dependent upon the faulty component. Rather than using a dedicated testing phase in the development process, informal testing was done continuously in an attempt to find problems as early in the development as possible. Whenever a problem was encountered (regardless of the problem's relation to the individual team member or their current task), the symptom was documented for further investigation. Following code integrations, the team categorized and assigned pending problems to specific members for immediate resolution. However, because the VSP was an initial prototyping effort, it was not necessary for all problems to be found and resolved. More attention was focused towards investigating functionality and concepts in the virtual environment, than in producing reliable, error-free code.

4.2 Architectural Goals

Before the design and implementation of the VSP began, several virtual environments previously developed at AFIT (the Solar System Modeler [Williams96] and the Virtual Cockpit [Adams96]) were investigated and modified. These investigations uncovered many strengths and weaknesses of the architectures and designs. These findings combined with observations gleaned from literature reviews, produced several high level architectural goals for the VSP. The architectural framework must:

a) support the development process

b) emphasize separation of concerns

c) support user interface development.

The following sections discuss the importance of each of these architectural goals and the alternatives considered for realizing the goals. Past efforts at AFIT, the ObjectSim architecture, and architectures discussed in the background chapter were used as comparisons to make the architectural design decisions. The ObjectSim architecture was the primary source for comparison; readers unfamiliar with ObjectSim are encouraged to review Snyder's thesis and the ObjectSim programmer's guide [Snyder93].

4.2.1 Support Development Process

Complex systems such as the VSP can not be reliably built using ad hoc development processes. The development process described in Section 4.1 and used for the VSP follows the advice of Brooks in *No Silver Bullet: Essence and Accident of Software Engineering* [Brooks87] by *growing* the system using an iterative prototyping approach to software development. With this process, functionality and capability are gradually and concurrently integrated into the system by multiple developers. An architecture needs to consider this process. Otherwise, concurrently developed components may not work together and time and effort will be required to integrate the incompatible components.

Snyder designed the ObjectSim framework as a general purpose architecture for quickly building distributed virtual environments. It was object-oriented, Performer based, and enabled the rapid creation of initial prototypes so next generation requirements could be developed and explored. ObjectSim was used to develop several virtual environments (Solar System Modeler, Virtual Cockpit, Synthetic Battle Bridge) and was known to be stable and useful.

Unfortunately, it suffered from several weaknesses that lead to decreased productivity and limited exploration later in the development cycle. First, it achieved much of it generality through the use of pointers linking the components together. This high coupling increases the interdependence of the components and complicates the process of isolating and modifying problems and/or changes. Another weakness of ObjectSim was the absence of the CODB as an integral part of the design. Previous attempts at integrating the CODB into ObjectSim were characterized by replicated data and functionality. The replication was due to the ad hoc methods (typically use of Performer shared memory) required for interprocess communication in ObjectSim.

In contrast, decoupled architectures like the MR Toolkit and the Veridical User Environment would better support a rapid prototyping development process via inherently low component interdependence forced by the limited methods of interprocess communication. These limitations keep changes from rippling through the system and the decoupling improves performance on multi-processor systems. Ordinarily, the primary disadvantage to this approach was the time required to build and establish a stable system that incorporated multiple process communication. The capabilities inherent in the CODB and the rapid prototyping process alleviated these development problems. The CODB provides a standard, stable, and efficient mechanism for multiple processes to communicate so a system can take advantages of multiprocessing immediately.

4.2.2 Separation of Concerns

The separation of concerns emphasizes a system in which each class or object has a specific goal or purpose. Given the taxonomy proposed in the Background, these purposes correspond to the seven primary VE activities. Objects should not be responsible for completing several different activities of the virtual environment. Conversely, a single activity should not be spread across multiple components of the architecture. Separation of concerns prevents components from developing interdependencies that complicate modifications to the requirements and implementation. Changes to one component should not require significant changes to other components that require time and effort to implement. These distributed changes increase the complexity of the system and make it more difficult for other team members to understand and use the components.

The VSP is a distributed virtual environment composed of many individual entities (as opposed to VE's with few entities such as the Virtual Colonoscopy Project [Hong 97] or the Virtual Wind Tunnel [Bryson92]) that emphasizes the visual representation of these entities and state of a military spaceplane. As such, how the architecture organizes the State Computation, State Representation, and Rendering activities is paramount. The separation of these activities with respect to previous architectures is discussed below.

4.2.2.1 State Computation and Representation

ObjectSim approached entity abstraction by providing a Player class responsible for managing the geometry (State Representation) and propagation of entities (State Computation). Subclasses defined specific custom propagation or geometry models. Unfortunately, this approach bound the movement of an object to its physical representation. While this binding exists in the real world, in software it prevents components from being developed independently, complicates the code and structure of a component, and restricts the amount of code reuse available. For example, in space environments all entities essentially propagate identically. ObjectSim forced the replication of orbital mechanics code wherever different methods of managing the geometry were needed.

The Virtual Emergency Room (VER) employed a central *Motion Manager* that coordinated all propagation within the virtual environment [Garcia96]. This approach was sufficient for the VER because only the user initiated movement or a change within the environment (other than changes in the virtual patient). In the VSP, however, entities must propagate and interact independently.

The MR Toolkit separated the State Computation and State Representation aspects of the application. However, the MR Toolkit was not specifically designed for an entity based VE, so it aggregates all computation (and similarly geometric) functions into single processes. This aggregation prevented distribution of independent propagation models across multiple processors.

4.2.2.2 Rendering

The IRIS Performer rendering software was based on the concept of managing a rendering pipeline and a scene graph describing the environment. Because rendering is typically the key limit to performance and it was anticipated that the large extent of the VSP would require special techniques to correctly render the environment, determining how the architecture handled rendering was important for the performance, development, and maintenance of the VSP.

ObjectSim distributes the rendering responsibilities throughout the application with no central control. The Pfr_Renderer isolated the Performer pipeline from other architectural components. However, the tasks of building and updating the scene graph were shared by the Pfr_Renderer (who created the root pfScene), the Players (who inserted and updated their state information in the scene graph), and the View class (which handled the viewpoint and jitter removal). This organization made complex rendering modifications like jitter and flimmer removal difficult to track and modify and unnecessarily bound the framework to Performer.

Even if rendering was not completely decoupled from the rest of the application, as VUE and the MR Toolkit advocate, it made sense to isolate all rendering operations in a single component of the application. Isolation of rendering promoted the process goal of reducing interdependencies and took advantage of the resulting improvements in maintainability and flexibility.

4.2.3 Support User Interface Development

A primary goal of the VSP was the development of an unconventional, reconfigurable interface for controlling a military spaceplane. However, at the beginning of the research term the mechanisms and style of the interface were unknown. This objective demanded an architecture that specifically accommodated the incremental development of user interaction in its design.

The ObjectSim design was aimed at providing generality and quick functionality for a virtual environment. Unfortunately, these features did not extend to the user interface. Other than providing an abstract Modifier class for deriving input devices, ObjectSim did not address user interaction in any specific part of its design. Examples in the ObjectSim programmer's guide placed keyboard interaction in the Sim's pre_draw method and head tracking handled by a View class accessing an HMD_Modifier (a Modifier polling a sensor attached to a head mounted display). This inconsistent design made adding or modifying user interaction capabilities and developing a consistent user interface difficult.

The Virtual Emergency Room handled all user interaction in its Renderer class. This centralized approach was an improvement over the distributed manner that ObjectSim used. However, it was unclear why (other than ease of implementation) user interaction was completed by a component associated with an aspect of the virtual environment's output.

The approaches used by the MR Toolkit and VUE provided the most architectural integrity by providing a single specific place in the architecture for managing all user interaction. Centralizing user interaction also supported the goals of reducing component interdependence and establishing clear separation of responsibilities.

4.3 Virtual Spaceplane Architecture

There is one final aspect of the architecture that must be discussed before presenting the VSP architecture - the division of the system into a set of independent tasks or processes. The primary purpose for using multi-processing in virtual environments is the increase in performance it affords on machines equipped with multiple processors. Process distribution is also used to eliminate the lag produced by polling input devices and to increase the fidelity of a simulation by allowing computational models to update themselves more frequently then if they were synchronized with the application's main loop.

However, unnecessary separation of the system into processes introduces several problems. These include the limited forms of interprocess communication, maintaining the consistency and coherency of shared memory, potential communication lag if processes are distributed across multiple machines, and the overhead required to perform context switches if tasks outnumber physical processors. Therefore, it was decided that unless direct performance or fidelity gains resulted, the activities of the VE would be kept within the main application process.

The culmination of the goals presented above was the Virtual Spaceplane architecture, shown in Figure 8 using the Activity/Module/Task taxonomy described in the background.

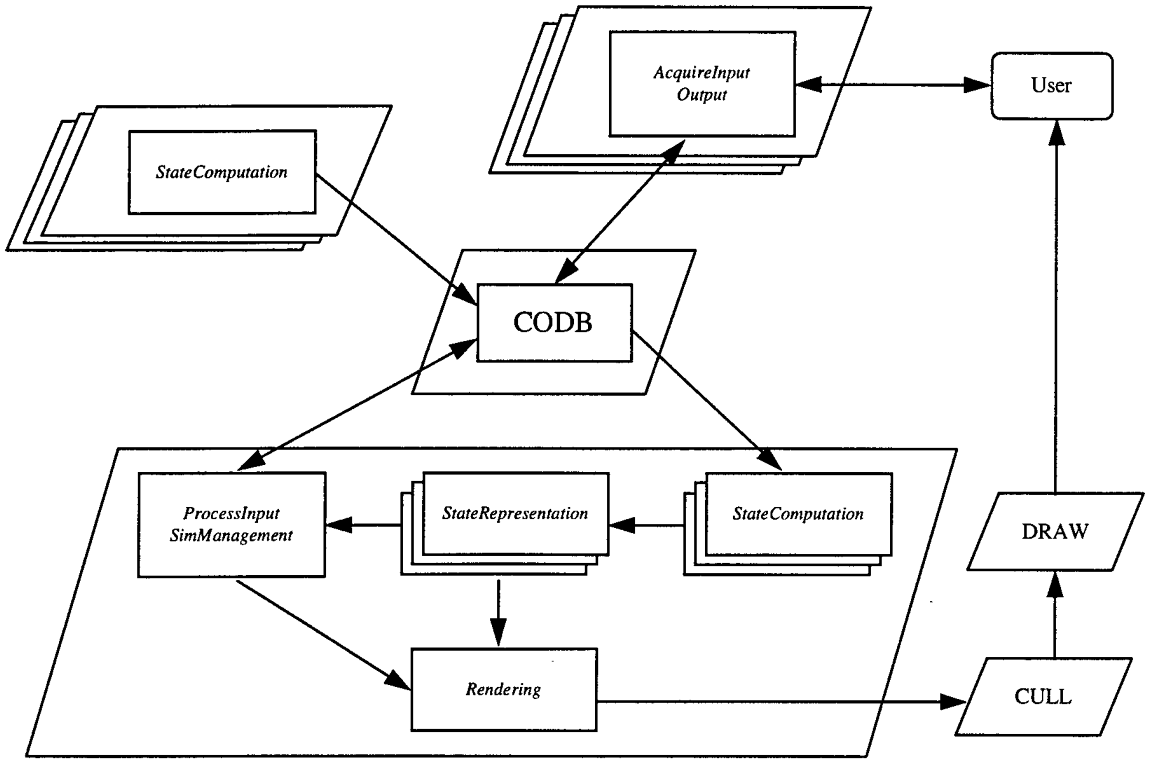


Figure 8 - Virtual Spaceplane architecture.

The VSP architecture was based on the following principles:

a) The ObjectSim framework was not used because of its tight component coupling, its lack of a specific place to perform input processing, the absence of a uniform method for safe interprocess communication, and the architecture's failure to recognize the separation of concerns. Instead, a custom design directed towards the unique requirements of the VSP was developed.

b) Emphasis was placed on separating different components of the application into distinct classes/components to reduce interdependencies and coupling.

c) The CODB was used to maintain architectural integrity and provide the capability to decouple input/output devices and computationally intensive propagation models from remainder of VSP.

d) Tasks performing the Acquire Input activities were forked off to eliminate lag produced by polling input devices.

e) Entity's State Representation and State Computation activities were separated into different classes, making each class simpler and, therefore, easier to test and maintain. The separation enabled more class reuse because general State Representation and State Computation modules could be developed independently and then paired to instantiate specific entities.

f) Both Performer pipeline and scene graph maintenance were isolated into a single module responsible for Rendering activities. The large spatial extent of the VSP required special processing to correctly render the VE. Isolation of Rendering activities shielded the remainder of the VSP from any changes or problems in the rendering process.

g) A single module performed Sim Management and Process Input activities. In an interactive environment such as the VSP, the user could potentially affect all aspects of and entities in the environment. Therefore, the module performing the Process Input activity needed access to all entities. In the proposed architecture, only a single module had this global knowledge which necessitated combining these activities in a single module.

h) The capability to fork off additional tasks to perform computationally intensive State Computation routines was supported. This capability resulted from the separation of State Computation and State Representation activities and the use of the CODB to provide safe interprocess communication.

4.4 Architecture Conclusion

This chapter presented the rapid prototyping process used to develop the VSP and the goals used to design the software architecture. The architecture was based on support of the development process, separation of concerns, and the ability to incrementally develop an unconventional user interface. The chapter concluded by presenting justification for the architecture using the taxonomy introduced in the Background chapter. With the architecture established, the following chapter will detail how the architecture was realized into the design of a collection of classes for implementing the VSP.

5. DESIGN

As discussed in the previous Chapter, the decision was made to combine lessons learned from a variety of previous virtual environment architectures and goals specifically derived for the VSP into an original design. This chapter describes the capabilities of a set of classes forming the framework of the VSP and presents justifications for breaking the system down in this manner. Rumbaugh diagrams [Rumbaugh91] will be used to show the methods of each class and relationships between classes. The architectural principle of separation of concerns was paramount in designing these classes. Each class had a specific goal or purpose and a concerted effort was made during implementation to keep this design integrity maintained.

Figure 9 is a modification of the architectural diagram shown in Figure 8. Figure 9 names the modules that perform each activity and shows the relationship between the high level modules used in the VSP. In the VSP design, each module was replaced by a C++ class that performed the appropriate activities and functions.

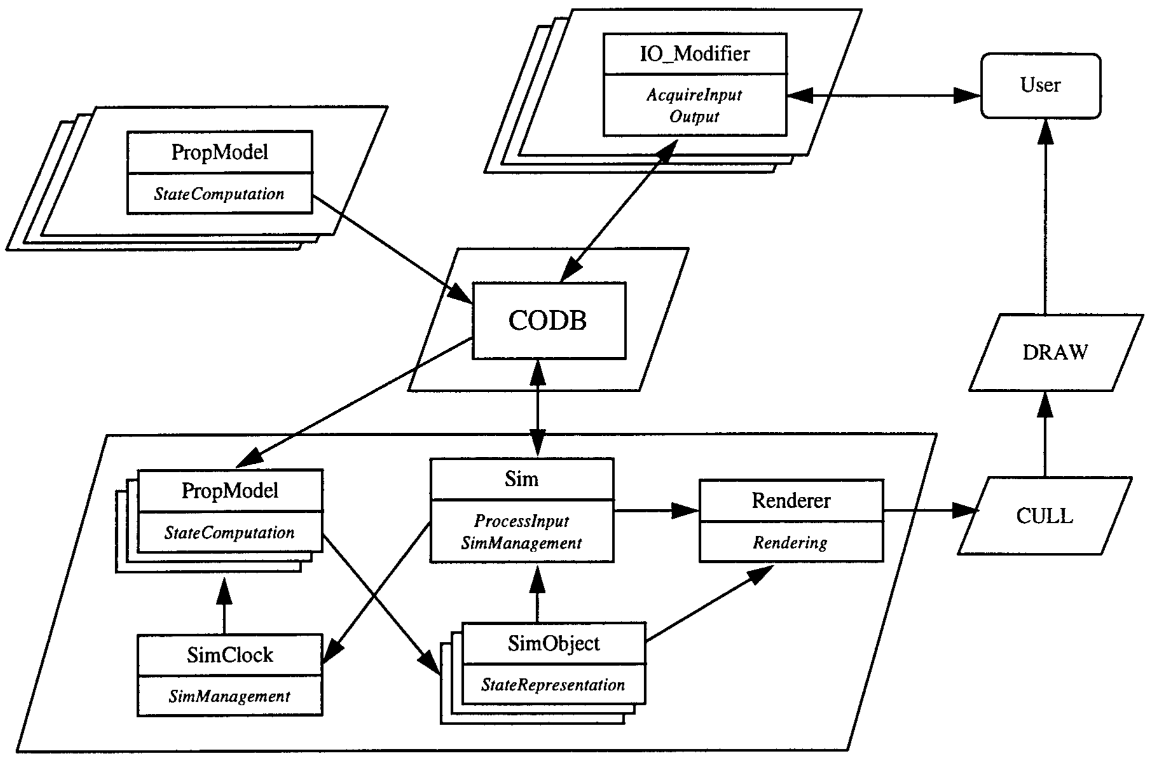


Figure 9 - Diagram of Virtual Spaceplane design.

The heart of the VSP is the Sim class that contains all the components of the virtual environment. Instances of the SimObject class represent entities in the virtual environment and are associated with a PropModel that models the dynamics of each entity. SimObjectManagers (not shown) manage and encapsulate multiple SimObjects with similar physical and propagation representations. The Renderer controls all rendering activities and a global SimClock maintains a consistent flow of time throughout the virtual environment. The I/O Modifiers provide abstract access to input and output which the Sim class manifests (via the CODB) as changes in the virtual environment. More detailed discussion of each class is given in the following sections. Although several classes provided bases for extensive inheritance (particularly the SimObject and

PropModel classes), these subclasses are not shown in Figure 9, but are discussed further in the relevant sections below.

5.1 Renderer

The goal of providing separation of concerns in the design led to the development of the Renderer class. This class had the responsibility for setting up the graphics hardware and then building and maintaining the Performer scene graph. With the exception of SimObjects creating and updating their local geometry, the Renderer class encapsulated *all* rendering functions. These functions included managing the IRIS Performer components (pfPipe, pfPWindow, pfChannel, and pfScene), the scene graph, the user's viewpoint, and jitter and flimmer removal. The encapsulation allowed jitter and flimmer removal (see Implementation chapter) to be completely localized to the Renderer, rather than spread throughout the simulation where changes to a specific implementation could require rework of other components besides the Renderer. Figure 10 shows the class structures related to the Renderer.

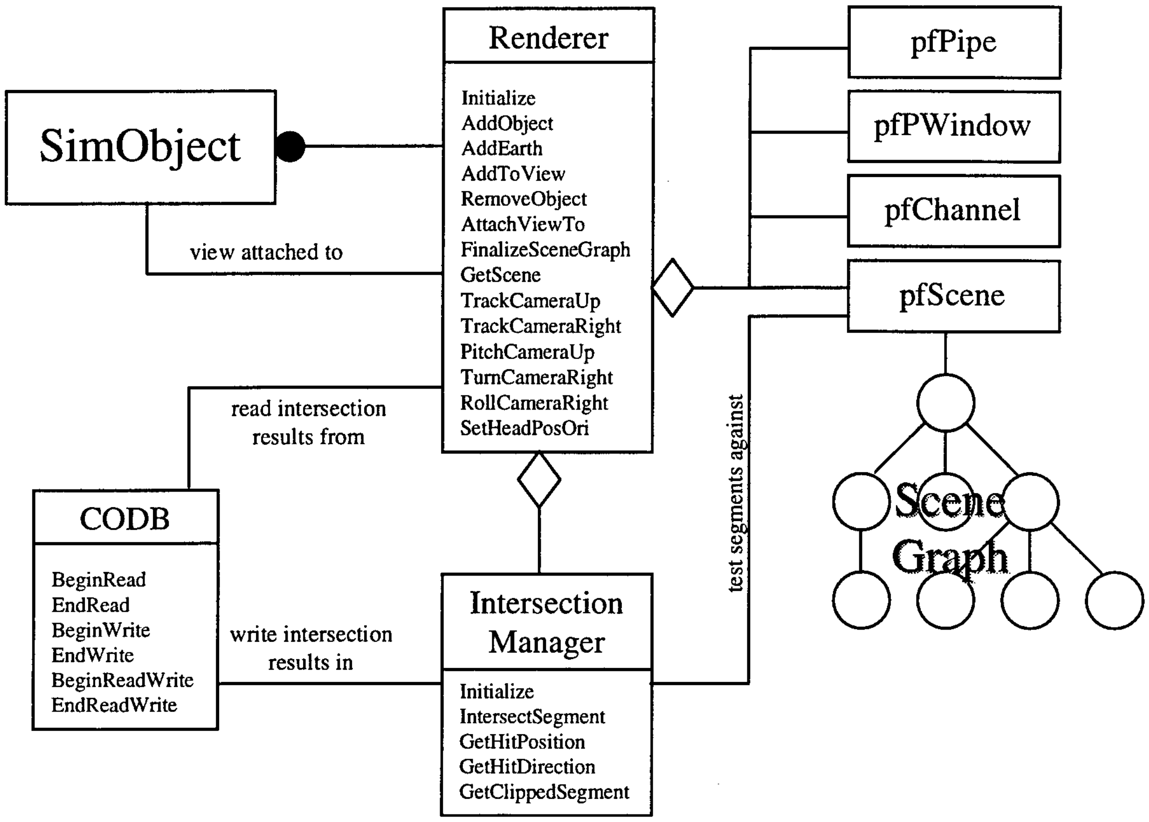


Figure 10 - Renderer class diagram.

The Initialize method of Renderer set up all the Performer objects required for rendering the environment, including the pfPipe, pfPWindow, pfChannel and pfScene [IRIS95][Rohlf94]. It also created the Intersection Manager discussed below.

Restricting other simulation components from the scene graph necessitated that the Renderer keep a list of all the active SimObjects in the virtual environment. This entity list enabled the Renderer to query SimObjects about their state and geometry information prior to finalizing the scene graph. Registering SimObjects by invoking the AddObject, AddEarth and AddToView methods added entities to the entity list. Most entities were registered via the AddObject method, except for the entity representing the Earth. The AddEarth method was needed because the Renderer was also tasked with transforming all entities into a single coordinate system. This transformation potentially involved correcting an entity's state for the rotation of the Earth; the Earth's rotation was obtained by querying the SimObject signified by the AddEarth method.

The Renderer also managed the location and orientation of the user's viewpoint in the virtual environment. The viewpoint had two independent aspects. The first aspect was represented by a logical camera that could be dynamically attached to any SimObject in the VSP. Each SimObject controlled a camera offset that specified the initial distance and orientation of the camera attachment and the Renderer provided several methods (TrackCameraUp, TrackCameraRight, PitchCameraUp, etc.) to modify the offset after the view was attached. The *TrackCamera* methods kept the field of view and the center of the attached SimObject aligned, while the remaining camera methods removed this restriction. The second aspect of the viewpoint was represented by the user's head position and orientation relative to the camera. This aspect of the view enabled a 3D head tracker to correlate the viewpoint with a user's actual head position by making calls to the SetHeadPosOri method each frame.

SimObjects registered via the AddToView method were always associated with the camera viewpoint; their position and orientation indicated offsets from the camera. The VSP cockpit is an example of an entity with this association to the viewpoint. The RemoveObject method provided for the removal of entities from the environment.

The Renderer's FinalizeSceneGraph method was one of the most important routines in the VSP. The method queried each entity for their state information, integrated the various coordinate systems used in the VSP, and placed the resulting position and orientation information into the appropriate pfDCS nodes in the scene graph. FinalizeSceneGraph transformed view modifications and offsets and applied them to the pfChannel. Operations for jitter and flimmer removal, discussed in detail in the Implementation chapter, completed scene graph finalization.

5.1.1 Intersection Manager

Performer provided considerable functionality for implementing intersection testing between geometry in the scene and line segments. Requirements for runway operations motivated the development of intersection testing in the VSP, particularly between the Gryphon and the terrain. Two primary concerns emerged when building the Intersection Manager for the VSP, where the Intersection Manager would fit into the design and how it would affect performance.

The Performer software contained all the code to calculate hit points between application specified line segments and geometry in the scene graph. However, the line segments must be specified in the coordinate system of the scene graph, which was only known by the Renderer. This restriction necessitated making the Intersection Manager an internal component of the Renderer.

By default, Performer tests for intersections in the application process by searching the entire scene graph for valid intersections. Informal observations showed this scene traversal added approximately 15 ms to the application process, making it the longest process and translating directly to a reduced frame rate. Fortunately, Performer allowed the application to fork off another process that performed intersection testing concurrently with the cull process. By instantiating an additional process, the frame rate penalty resulting from intersection testing was undetectable. Rather than using standard shared memory to communicate the result of the intersections back to the application, the CODB was used to help meet design goals and prevent memory contentions.

5.2 SimObject

The SimObject class provides a base class from which all simulation entities and methods for manipulating and managing the entities were developed. As mentioned previously, a primary goal of the design was to provide clean separations between activities in the simulation. This goal prompted the removal of all State Computation activities and methods out of the SimObject, into a separate PropModel class. Therefore, each SimObject was associated with a propagation model that described the object's movement in the virtual environment. Therefore, the SimObject's primary purpose is performing the State Representation activity for its entity; i.e. managing the entity's local geometry to ensure that it reflects the state of the entity as determined by its PropModel. This separation produced several capabilities. A SimObject could change the PropModel it was associated with and therefore change how it moved through the environment. Identical SimObject subclasses could also be instantiated with different PropModels resulting in different movement characteristics.

SimObjects contained two primary methods, Initialize and Update. These methods were generally overridden by subclasses to customize the SimObject's behavior. In the Initialize method, the SimObject loaded external geometry or created its own geometry using Performer structures. Prior to rendering each frame, the simulation invoked each instantiated SimObjects' Update method. In response, the SimObject dynamically modified its displayed geometry (with the option of making no changes) and notified its propagation model to calculate the SimObject's new position and orientation.

In addition to these functions, SimObjects maintained their camera offsets, provided the capability to draw and modify characteristics (such as the length and color) of trails behind themselves, and provided methods for accessing state information from their PropModel. The Renderer used the camera offset when attaching the view to a SimObject and the trails provided information about the entity's previous position. The flimmer removal technique described in the Implementation chapter required that all the geometry in the VE was sorted into logical bins. The PutGeomInBin method placed the entity's geometry in the bin specified by the input parameter.

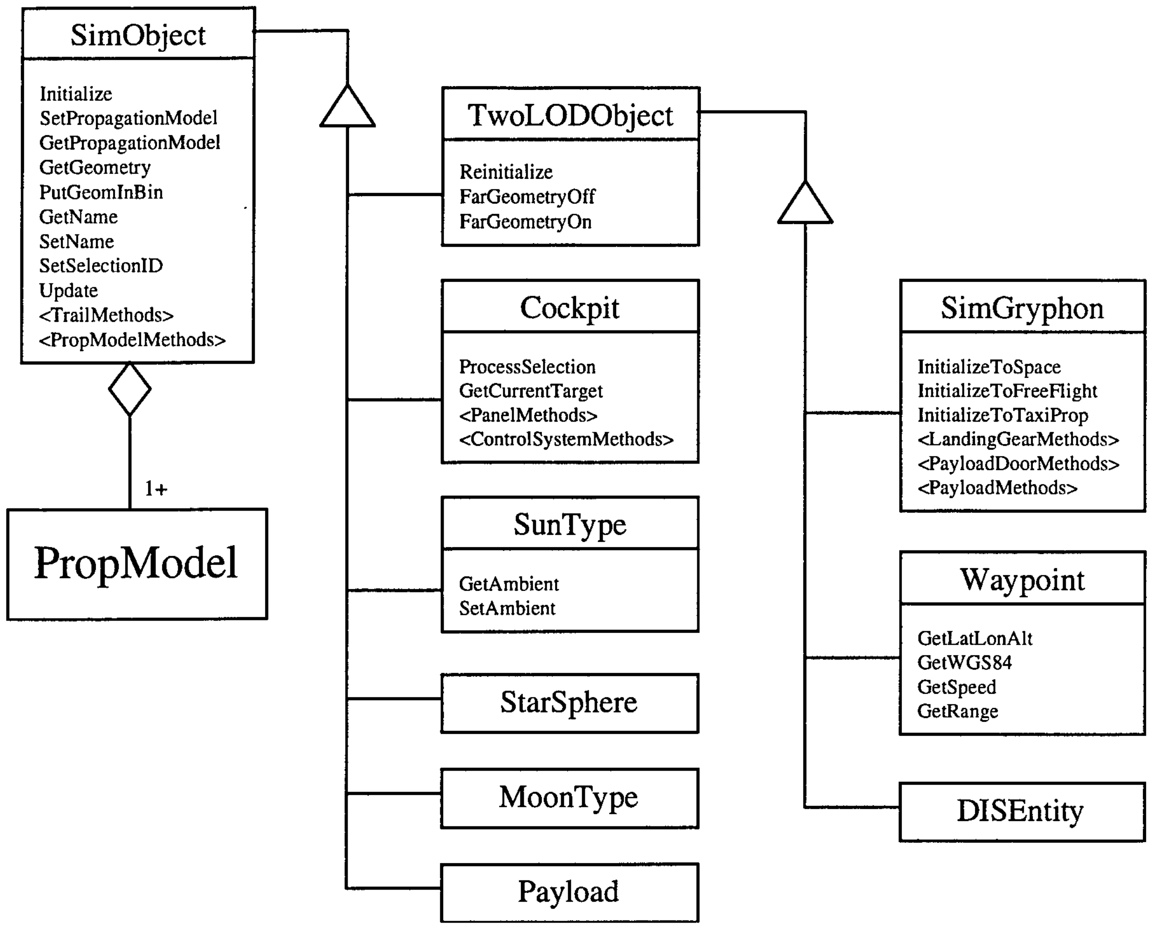


Figure 11 - SimObject class diagram.

Several subclasses of SimObject were created during development. These and several other subclasses are shown in Figure 11. A list of these subclasses and their purpose is given below.

- TwoLODObject represented an entity with two levels of detail (LOD). TwoLODObjects were initialized by specifying the geometry for the two levels and the distance from the viewpoint when the LOD's should switch. The Reinitialize method provided the ability to change the geometry or switch distance at any time.

This class was used primarily to implement entity locators - simple, color-coded representations designed to identify the entity from distances at which the entity would not be visible. The FarGeometryOff and -On methods enabled the locators to be turned off and on.

- The Cockpit subclass contained the geometry of the user interface developed for the VSP. The Cockpit used a collection of movable panels, buttons, and displays to both communicate and enable the user to change the state of the Gryphon and the environment. The Sim class invoked the ProcessSelection, Panel and ControlSystem methods to notify the Cockpit of user events or changes in the virtual environment that the user interface had to reflect. Capt John Lewis's thesis details the development and operation of the user interface encapsulated by the Cockpit class [Lewis97].
- Sol was represented by SunType, whose distinguishing feature was the addition of illumination to the environment. The SetAmbient method changed the ratio of ambient to diffuse lighting, providing the capability to artificially increase the lighting of dark areas such as the night side of the Earth.
- The StarSphere visualized over 32,000 stars visible in space or during the night. The subclass also included lines representing 81 major constellations. A description of how the stars and constellations were visualized is given in the Implementation chapter.
- MoonType represented the Earth's only natural satellite.

- The Payload subclass represented the payload within the Gryphon. The payload is currently limited to a stowed satellite, but could be extended to other potential payloads such as weapon systems or reconnaissance equipment.
- SimGryphon managed the external representation of the Gryphon spaceplane and the various propagation models used to propagate it. A detailed description of SimGryphon follows this list.
- A Waypoint signified a specific location in space that was relevant to the current mission. A set of Waypoints could constitute a route, in which case speed and range attributes defined a successful rendezvous with the Waypoint. A Waypoint was visualized as a spinning diamond in the VE.
- The DISEntity subclass was used in conjunction with the DISEntityManager, DISEntityProp and DISManager to model remote entities using the DIS suite of protocols [IEEE93]. The DISManager removed DIS packets from the network and placed the contents into the CODB. The DISEntityManager collected the state information out of the CODB and partitioned it out to the DISEntities and DISEntityProps. The DISEntity interpreted the entity type fields and selected an appropriate geometric model to represent the entity.

5.2.1 SimGryphon

SimGryphon was one of the most complex classes in the VSP. This complexity resulted from the Gryphon's need for dynamic geometry and use of multiple propagation models. Unlike most SimObjects, the Gryphon contained moving geometry representing the landing gear, payload, and payload door. However, most of the complexity resulted from managing multiple propagation models. Because Gryphon simulated a military spaceplane in the conceptual stages of development, no single propagation model for the entire flight regime of a military spaceplane existed. To overcome this obstacle, the SimGryphon class dynamically switched between separate propagation models calculating the atmospheric, hypersonic, exo-atmospheric, and orbital dynamics. At each transition the Gryphon's position, orientation, and velocity were synchronized so the immersed user had no indication that the transition actually occurred. The primary difficulty with this process was the conversion between the various coordinate systems used in the VSP. Lt Troy Johnson's thesis provides more detail on the procedures and problems associated with this approach [Johnson97].

5.3 PropModel

The abstract PropModel class provided a base to develop propagation models used by SimObjects and methods to consistently manipulate and query the PropModels. This separation between PropModels and SimObjects was chosen for several reasons. It made each class simpler, and therefore easier to code, maintain, and test, than if the SimObject and PropModel functionality were combined into a single class. It also provided more opportunity for reuse. For example, a single orbital mechanics model could be developed for all space entities and different SimObjects could be assigned to achieve different geometric models, two-line element sets, and dynamic behavior (established satellites do not change orbits considerably, while the Gryphon will do so frequently).

PropModels have two primary methods, Initialize and CalcNewPosition. Derived subclasses of PropModel overrode both methods to model the dynamics appropriate to each subclass. The Initialize method assigns a starting location, loads model specific data from an external file such as two-line element sets, or completes other appropriate activities. The CalcNewPosition method determines the entity's new position and orientation with respect to the entity's old position and/or the current simulation time. PropModels obtain the simulation time from the global SimClock.

Each PropModel was designed around an arbitrary coordinate system. This coordinate system was selected in each case to simplify the calculations performed by the PropModel. For example, the AeroModel (described below) assumes a flat earth to remove complications that arise from the curvature of the Earth, while the AstroProp model uses orbital mechanics and assumes a spherical Earth. However, to simplify interactions between other entities and components of the VSP, each PropModel provided routines to access state information in several standard coordinate systems. These included:

1. the WGS84 system - a non-inertial, right-handed coordinate system that rotates with the Earth and defined with its center at the center of the Earth, its z-axis passing through the North Pole and its x-axis passing through the point where the equator and the prime meridian intersect [DOD87]. The methods

GetWGSPosition and GetWGSOrientation return an entity's state in this coordinate system.

2. the Earth Centered Inertial (ECI) system - an inertial, right-handed coordinate system the Earth rotates within and defined with its center at the center of the Earth, its z-axis passing through the North Pole and its x-axis pointing towards the Point of Aries (where the WGS84 x-axis points at during the vernal equinox) [Bate71]. The methods GetGeocVEPosition and GetGeocVEOrientation return an entity's state in this coordinate system.

3. the Geodetic coordinate system - a flat-earth system defined by lines of latitude running parallel with the equator and lines of longitude intersecting at the Earth's poles. The methods GetGeodeticPosition and GetInstrumentOrient return an entity's state in this coordinate system.

To simplify the transformation of coordinates between these systems, the AFIT Coordinate Conversion Utilities (ACCU) were developed. They served the same purpose as ObjectSim's Round Earth Utilities (REU), but whereas the REU were meant to operate over a small specified region of the globe, the ACCU functions correctly anywhere on the globe [Johnson97]. The ACCU provided methods to convert between the WGS84, Geodetic, ENV (a local flat-earth system), and DIS coordinate systems. The ENVDelta_for_WGS84Move and ENVDelta_for_GeodeticMove methods compensated for the curvature of the Earth by calculating a new position based on a starting position (in the WGS of Geodetic systems) and a movement in the East, North, and vertical directions. PropModel also included standard methods for accessing the current speed, velocity, mach, ground speed, altitude, and percent throttle applied.

PropModels exploited polymorphism and a suite of standardized routines so other VSP components could consistently access state information of entities while ignoring the internal differences between the various models. Unfortunately, providing standard methods for input to the PropModels introduced complications because each PropModel often required unique types of input. For example, the orbital mechanic model used deltaV (change in velocity) inputs to calculate changes in orientation and orbital elements, the atmospheric flight model used throttle setting, pitch, bank, and roll commands, and the terrain following model only needed the offset from the ground to function correctly. Rather than building standard input methods, which would have been useless and/or confusing in many cases, each PropModel defined specific input routines. The GetPropModelType method allowed simulation components to determine the appropriate input by returning a unique enumerated value for each PropModel.

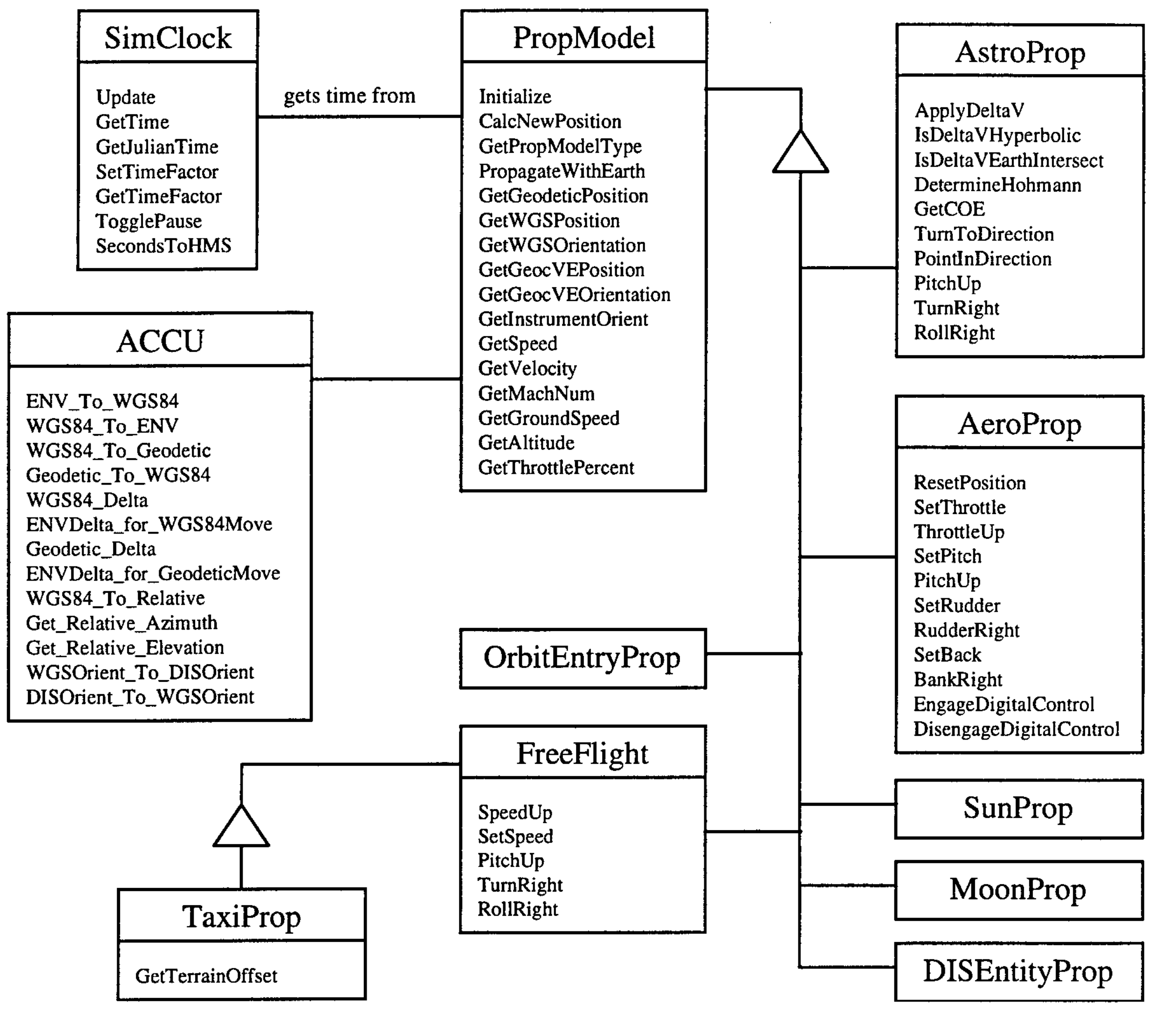


Figure 12 - PropModel class diagram.

PropModels describing different dynamic models were developed and reused throughout the simulation. Examples of PropModels developed for the VSP are shown in Figure 12 and listed below.

- FreeFlight ignored most physical laws of motion. It accepted commands to change its speed or orientation and performed the actions without regard for constraints such as inertia or aerodynamics. FreeFlight was also used to position static entities such as ground tracking stations and terrain patches in the environment.

- TaxiProp implemented terrain following between an entity and the Earth's surface. When TaxiProp was Updated, it would compare the current altitude above ground (as determined by intersection testing performed in the Renderer) with its target offset. It would then move itself up or down to maintain the specified terrain offset.
- SunProp, MoonProp, and EarthProp simulated the movement of the Sun, Moon, and Earth within the ECI coordinate system. These classes were based on orbital mechanics routines developed for the Solar System Modeler [Kunz93][Williams96].
- AeroProp contained the six degree of freedom aerodynamic model supplied by WL/FIGD (Wright Lab's Flight Simulation Branch) and initially integrated into the AFIT Virtual Cockpit [Adams96]. This model accepted stick, throttle, and rudder commands and was only accurate at altitudes below 30 km and velocities below Mach 2.5.
- AstroProp was based on an orbital mechanics model obtained from the Air Force Academy Astronomy Department. AstroProp accepted commands to change the entity's current velocity (ApplyDeltaV) or orientation (PitchUp, TurnRight, RollRight, TurnToDirection, PointInDirection). It also determined deltaV's required to change from one orbit to another (DetermineHohmann) and returned the set of classical orbital elements describing the current orbit (GetCOE). Methods were also available to test whether a potential deltaV would send an entity out of a stable orbit (IsDeltaVHyperbolic) or into the Earth (IsDeltaVEarthIntersect). AstroProp was used to simulate entities orbiting the Earth at altitudes above 100 km.

- OrbitEntryProp was an orbit entry/reentry model for transitioning from the Earth's atmosphere to space and back provided by ASC/XRD (Air Systems Center's Directorate of Development Planning, Design Branch). This model bridged the gap between the AeroProp and AstroProp models. OrbitEntryProp accepted a potential path (described as a plot in the altitude vs. velocity domain) together with data describing the aerodynamic and propulsion properties of a vehicle and returned the angle of attack, throttle settings and down range necessary to follow the specified path.
- The DISEntityProp subclass was used in conjunction with the DISEntityManager, DISEntity and DISManager to model remote entities using the DIS suite of protocols. Whereas the DISEntity determined the geometric representation, DISEntityProp uses DIS packet information and calculates the entity's position and orientation in the WGS84 coordinate system.

5.4 SimObjectManager

As the number of entities in the virtual environment increased, it became more tedious to manage and manipulate them. A SimObjectManager provided a consistent mechanism to manage multiple SimObjects that shared common geometric representations and/or PropModels (see Figure 13). A single method call from the Sim to a SimObjectManager resulted in the identical call being echoed to each of the Manager's SimObjects. For example, the VSP modeled 25 Global Positioning System (GPS) satellites. Without using a SimObjectManager, the Sim class would have had to explicitly manage all 25 satellites individually. A SimObjectManager aggregated all the GPS satellites into a single logical group that could be manipulated as a whole. Instead of explicitly calling the Update method for each satellite, Sim simply invoked the GPS SimObjectManager's Update.

In order to provide more capability, SimObjectManager's aggregated a collection of TwoLODObjects rather than simple SimObjects. In this sense, a more accurate name for the class would be TwoLODObjectManager.

The Initialize, Update, Trail and FarGeometry modification methods echo the method invocation to each of the TwoLODObjects the SimObjectManager controls. The SimObjectManager also acknowledges a specific SimObject as the current SimObject. The GetNext and GetPrev methods iterate through the Manager's list of SimObjects, changing the current SimObject along the way. GetSimObjectNumber returns the *i*th SimObject, where *i* is passed in as an argument and the GetNumOfSimObjects method returns the total number of SimObjects managed by the SimObjectManager.

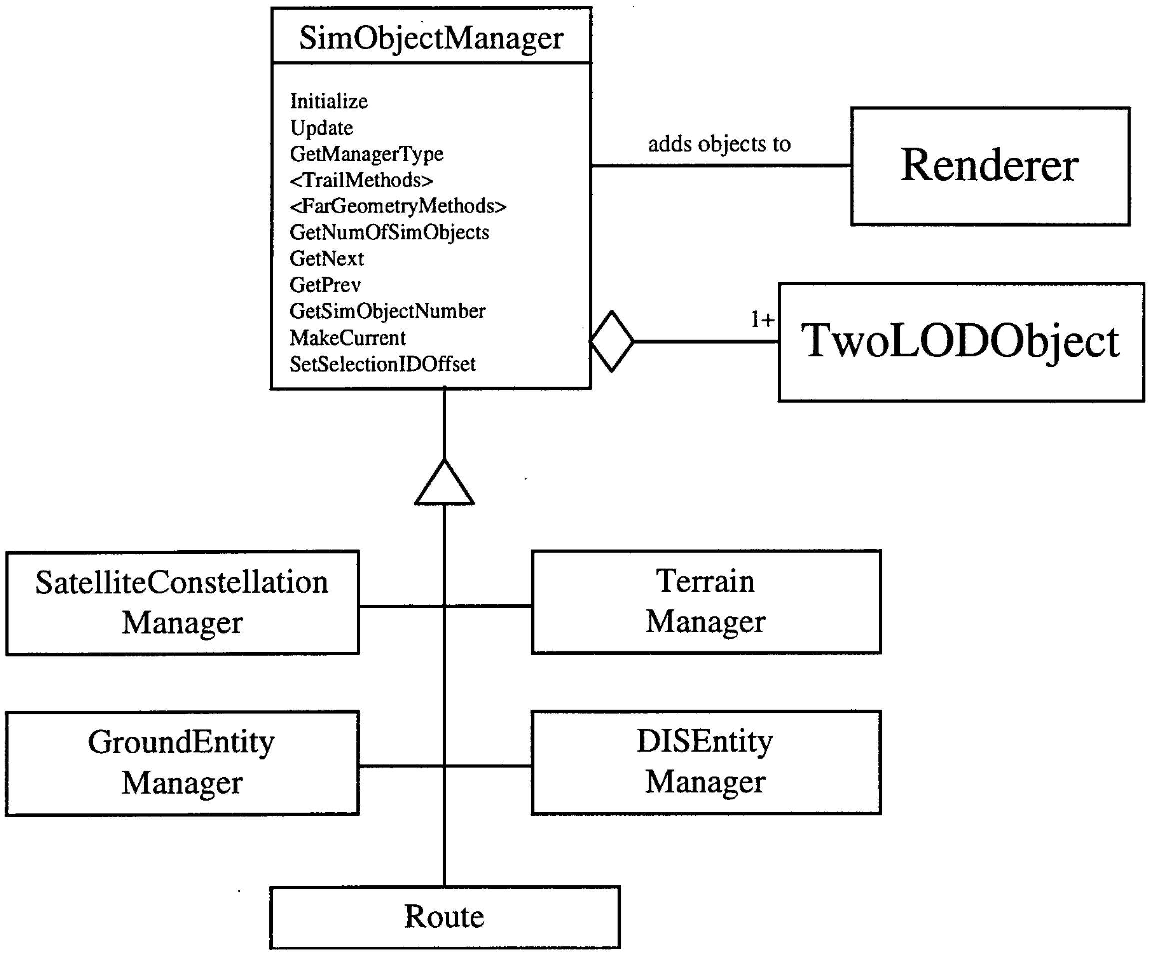


Figure 13 - SimObjectManager class diagram.

Subclasses of SimObjectManager were defined to capture certain characteristics of a group and are listed below with a description of their purpose.

- A SatelliteConstellationManager was instantiated for each constellation of satellites that shared common physical representations (GPS, DMSP, DSCSIII, TDRS, Molniya). It assigned each of its SimObjects an AstroProp model initialized with an appropriate two-line element (TLE) set.

- TerrainManagers loaded polygonal terrain patches over selected regions of the Earth. TerrainManagers used multiple levels of detail to balance detail and frame rate. More information concerning the creation of the terrain patches is given in the Implementation chapter.
- GroundEntityManagers created static terran entities that model reconnaissance targets or ground tracking stations.
- The Waypoint Route represented a collection of Waypoints constituting a flight pattern necessary to complete a goal.
- The DISEntityManager used the DISEntity class, DISEntityProp class, the DISManager, and the CODB to realize, manipulate, and access state information of remote entities via the DIS suite of protocols.

5.5 SimClock

The SimClock class controls the flow of time in the simulation. Previous virtual environments at AFIT have either used the operating system's clock or used a global data structure that all components were free to modify at will. The operating system clock was unacceptable because it provided insufficient control of time in the virtual environment, such as the ability to pause or change the rate at which time flows in the virtual environment. The global data structure technique was unacceptable because it violated the architectural principle of separation (and the associated isolation) of concerns. Any implementation changes to the clock would require modifying components throughout the system. By encapsulating all time management in the SimClock class, other design components were protected against implementation changes and a single modification to the SimClock would affect the entire environment. For example, all propagation models referred to the SimClock when calculating their next location, so a single change to the SimClock affected all SimObjects uniformly (Figure 12).

The SimClock provided several methods for controlling and accessing the flow and state of time. Entities requested the time in either the Gregorian date:hour:minute:second format or as the Julian Date, used in many astronomical algorithms [Meeus91]. The SetTimeFactor method allowed the simulation to proceed at an arbitrarily faster or slower pace then real time. It was anticipated that modifying the flow of time may be useful during lengthy automatic takeoff, landing, or orbital maneuvers. The TogglePause method stopped and restarted the flow of time in the simulation. The SimClock also provided several static methods for converting from one time format to another.

The SimClock was the only globally visible object used in the Virtual Spaceplane other than the CODB. Extensive use of the SimClock in the simulation necessitated this break in object-oriented design.

5.6 Sim

The Sim class was the heart of the Virtual Spaceplane. The Sim class, shown in Figure 14, contained all the entities in the virtual environment, a Renderer, a Selection Manager, and the input modifiers. The Sim's responsibilities included creating and destroying all the SimObjects, assigning a PropModel to each SimObject, creating the

Renderer, invoking the Update methods of the simulation components (SimObjects and/or SimObjectManagers), and translating all user interaction and events into specific commands to the various simulation components.

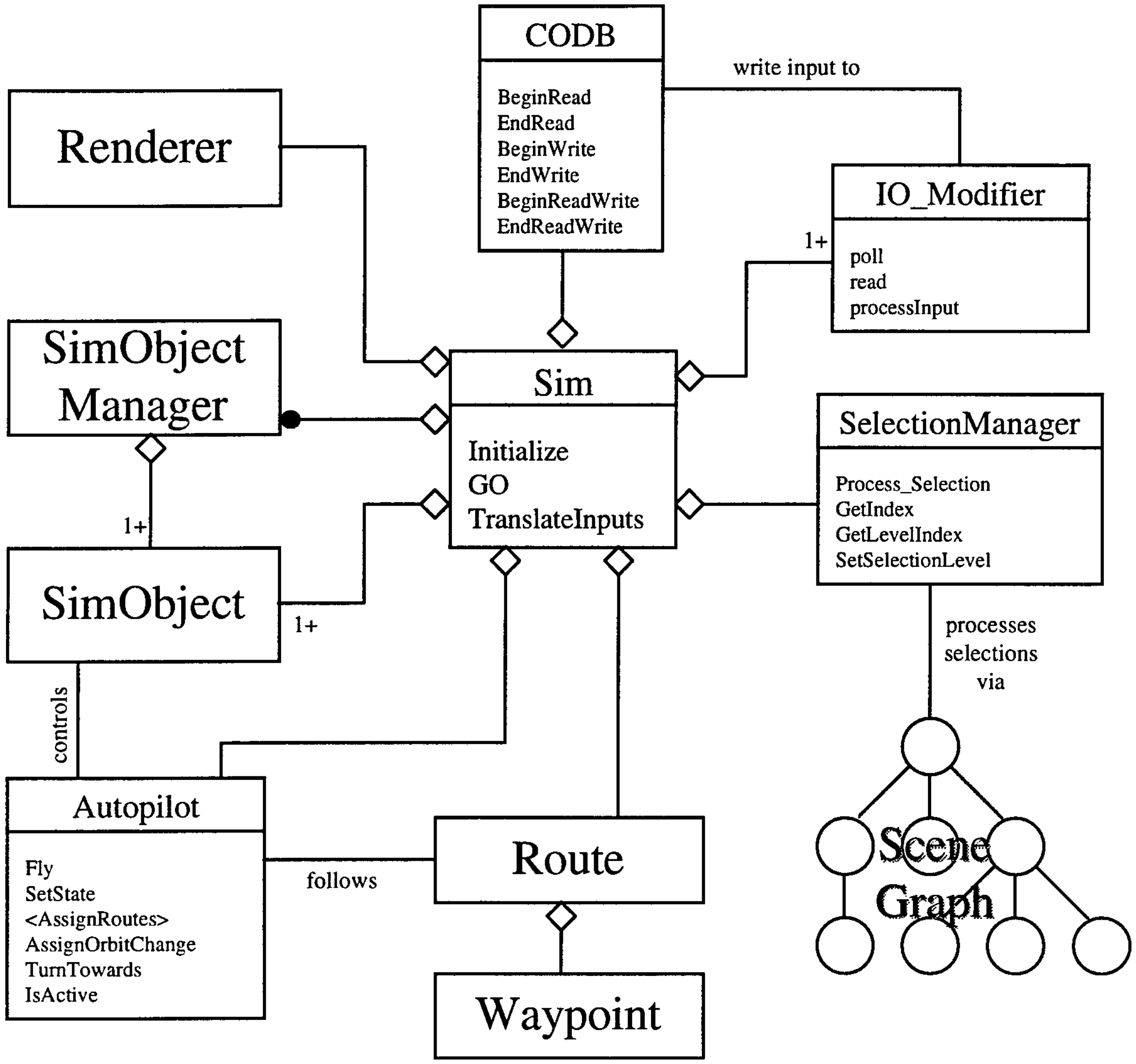


Figure 14 - Sim class diagram.

The majority of simulation execution occurred within the GO method of Sim. Invocation of the GO method initiated the simulation's primary loop until the user indicated the simulation should end. An abbreviated listing of the GO method is shown below.

```
void Sim::GO()
{
        // Complete Initialization

        while (notDone)
        {
                //Signal Input modifiers to update their CODB containers
                Keyboard->Poll();
                Mouse->Poll();
                .
                .

                TranslateInputs();
                // eventually sets notDone to false

                //Update all the SimObjects
                Gryphon->Update();
                theEarth->Update();
                .
                .
                theStars->Update();

                //the Renderer builds the scene graph
                //   for the current frame
                theRenderer->FinalizeSceneGraph();

                //Tell Performer to render the current frame
                pfFrame();
        }
} // end of Sim::GO
```

The main execution loop begins by polling all the input devices. All I/O devices were decoupled from the main application, but they needed to be signaled to update their

CODB containers with the most up to date input. Next, the Sim translated all user interaction into events that triggered changes in the virtual environment. All entities in the Simulation were Updated to modify their geometry and propagate themselves as appropriate. Finally, the Renderer built the scene graph before signaling Performer to render the current frame. This loop continued until the user terminated the virtual environment.

To provide the capability for various automatic control modes, the Sim had an Autopilot class that could be assigned to any SimObject (normally the Gryphon). Based on the Autopilot's state, the current PropModel of its assigned SimObject, and possibly a Route, the Autopilot directed the SimObject towards an objective.

The Sim also had a private method for broadcasting the state information of any SimObject using the DIS protocols. As with the DISEntity and DISEntityProp mentioned previously, this method used the CODB and Sheasby's DISManager (also referred to as World State Manager in [Stytz97]) to manage the network and packet issues.

5.6.1 User Interaction

A key design decision of the VSP was centralizing the translation of all user (and potentially intelligent agent) interaction in the Sim class. Although combining these activities deviated from the goal of separation of concerns (the Sim now instantiated all the VSP components *and* performed input translation), the translation was performed in TranslateInputs, a private method of Sim. This organization simplified the implementation because the Sim was the only component with access to all other simulation components. The centralization directly supported the development process because different interaction mechanisms could be investigated and functionality could be gradually introduced/modified by changing this single method.

To further support the development process, the translation of input into changes in the virtual environment was divided into two distinct phases. The first phase interpreted user input from the keyboard, mouse, Polhemus Fastrak, Ascension Flock of Birds™, intelligent agent, or other IO_Modifier, and generated a unique event. The SelectionManager facilitated this first phase by accepting screen coordinates (from mouse clicks or 3D tracker input) and returning the event associated with the selected geometry. The second phase interpreted events as modifications to the virtual environment via method calls to the appropriate entities or VSP components. The phased approach cleanly separated specific input devices from the effects they created, preventing modifications in one phase from affecting the other.

5.7 Design Conclusion

This chapter presented the design framework of the VSP. Discussion of each of the key base classes (Renderer, SimObject, PropModel, SimObjectManager, SimClock, and Sim) covered the capability and the class's relation to the architectural principles introduced in previous chapters. The next chapter will describe several implementation techniques and issues featured in the VSP.

6. IMPLEMENTATION

While the primary focus of this research was creating the architecture and design discussed in the previous chapters, many implementation issues arose during the development process. The techniques for jitter and flimmer removal and aspects of modeling the Terran environment not only demonstrate the flexibility and capability of the VSP's architecture and design, but also highlight several features of the Virtual Spaceplane.

6.1 Jitter removal

During the implementation of ObjectSim, Snyder discovered that geometry rendered far from the simulation's origin shifted positions chaotically from frame to frame [Snyder93]. This phenomenon was referred to as *jitter*. The cause of jitter was hypothesized to result from the limited precision of the rendering hardware.

ObjectSim's response to jitter was rendering the scene with the viewpoint at the origin by placing a pfDCS translation node at the root of the scene graph. By placing the negative of the viewpoint's position in this root translation, the scene effectively rendered around the viewpoint. Unfortunately, the translation relied upon single precision subtraction of large numbers, an imprecise operation. The imprecision became apparent in the Solar System Modeler by a return of jitter when the user attached to the outer planets or distant inter-planetary satellites.

Like ObjectSim, the Virtual Spaceplane combated jitter by rendering the scene with the view at the origin. However, instead of relying on Performer's single precision mathematics to perform the final translation, the Renderer translated positions in double precision. The translation was accomplished by calculating the position of the view (based on the position of the SimObject the view was attached to) and subtracting this view position from each of the other SimObjects' positions and placing these values into the appropriate pfDCS nodes. By using double precision, the Renderer better preserved small changes in SimObjects' positions regardless of the objects' pre-rendered position.

6.2 Flimmer removal

The key to removing flimmer (introduced in Chapter 2) lies in minimizing the distance between the far and near clipping planes of the viewing frustum, or more accurately, the ratio of the far to the near plane. Several methods were considered to minimize this ratio and solve the flimmering problem.

1. The Performer library provided several structures specifically designed to combat flimmer (pfLayer and pfDecal). Unfortunately, these structures were designed for flat surfaces known to be coplanar. In the VSP where non-planar objects are moving dynamically with respect to one another, these structures were not appropriate.
2. Rather than modeling the entire region from the center of the Earth to the Moon, the region could have been reduced to just beyond high Earth orbit. This approach would require modeling the Moon and Sun within this region and would have created several disadvantages in the process. Because the Moon and Sun would be modeled

so close to the Earth, their true position and apparent position would be significantly different. The perceptual errors resulting from modeling the Moon closer to the Earth were deemed unacceptable. This error is present when the Sun is modeled just beyond the Moon, but the resulting reduction in the modeled space outweighs the perceptual error.

3. At first it would seem the scaling approach used by the Solar System Modeler would help combat flimmer, when in fact it would have no effect. In a given depth buffer, there is a constant number of levels that can be differentiated regardless of the values assigned to the shallowest and deepest levels. If the geometry in the virtual environment is unchanged while the clipping planes are scaled down, the resolution of the depth buffer will be improved and flimmer will be reduced in the resulting scene. However, when the geometry is uniformly scaled with the clipping planes as the SSM did, the relative resolution of the depth buffer is unaffected. Uniformly scaling the environment has no effect on the resolution of the depth buffer, so it has no effect on flimmering.
4. The *locales* approach to partitioning an environment into smaller sections used in MERL's Diamond Park was an innovative technique that would reduce flimmer problems in many applications [Barrus96]. Unfortunately, the VSP failed to satisfy the premise the technique was based on; "...even in a very large virtual world, most of what a single user can observe at a given moment is nevertheless local in nature."
5. The final unacceptable approach involved rendering each frame in two cycles. To implement the two cycle render, all geometry is placed in the Performer scene graph

at a uniform scale. During the first cycle, the color and depth buffers would be cleared and the far and near clipping planes would be set such that only geometry far from the viewpoint would be rendered. During the second cycle, only the depth buffer would be cleared and the clipping planes would be reset to render the near geometry. Although this approach required two actual frames to render each image (for a peak frame rate of 30 fps), it was hypothesized that since less geometry was rendered each frame, the perceived frame rate would not be significantly reduced. However, the hardware's use of double buffering prevented this approach from working. Since the same frame buffers were always used for the near and far geometry, respectively, the frame buffer associated with the near geometry never had its color buffer cleared. Forcing the hardware to use only a single buffer would have fixed this problem, but the potential flickering was deemed unacceptable.

The method implemented to remove flimmer was conceptually similar to the final solution discussed above. Figure 15 and Figure 16 illustrate how the implemented method worked. Figure 15 is a profile of a viewing frustum with a simple scene and the desired image using no flimmer removal techniques. When the distance between the near and far clipping planes becomes large, the rendering hardware can not differentiate which object is closer to the viewer and flimmering occurs.

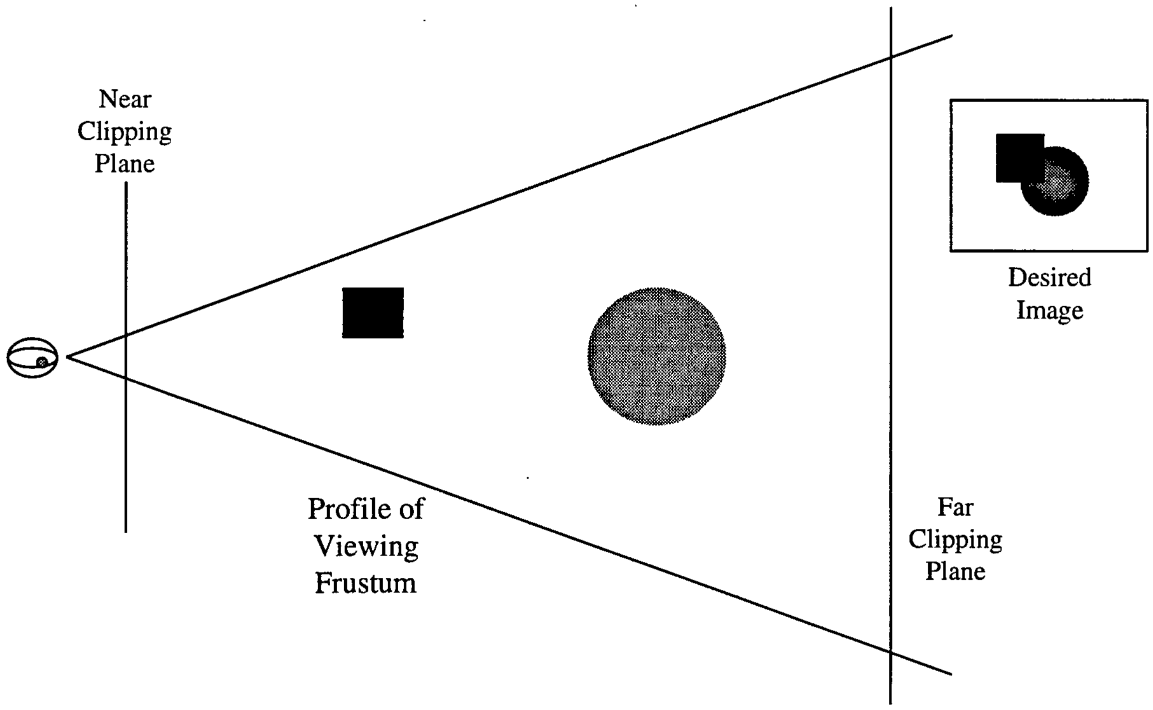


Figure 15 - Profile of Viewing Frustum with no Flimmer Removal.

The selected method (illustrated in Figure 16) divided the geometry into two groups; objects farther than a specified distance from the viewpoint (indicated by the near-far divider) and objects closer than this distance. The clipping planes were set for rendering the near geometry and the geometry located far from the viewpoint had its position and size uniformly scaled towards the viewpoint so it also lay within the clipping planes. As long as scaling occurred uniformly towards the viewpoint, no distortion would appear in the rendered image.

As is, the scaling still caused incorrect rendering of the scene, indicated by the image in upper left of Figure 16. Geometry located far from the viewpoint incorrectly overlapped geometry much closer, due to the position change. To correctly render the two sections and create one coherent image, the method sorted objects into *bins*, one for far geometry and one for near geometry. For example, in Figure 16 the square object would be placed in the near bin, while the sphere would be placed in the far bin. Each frame, the far bin was rendered, *the depth buffer* cleared, and then the geometry in the near bin rendered. Clearing the depth buffer ensured rendering of the near geometry in front of the far geometry. In the lower left corner of Figure 16, the resulting correct image is shown.

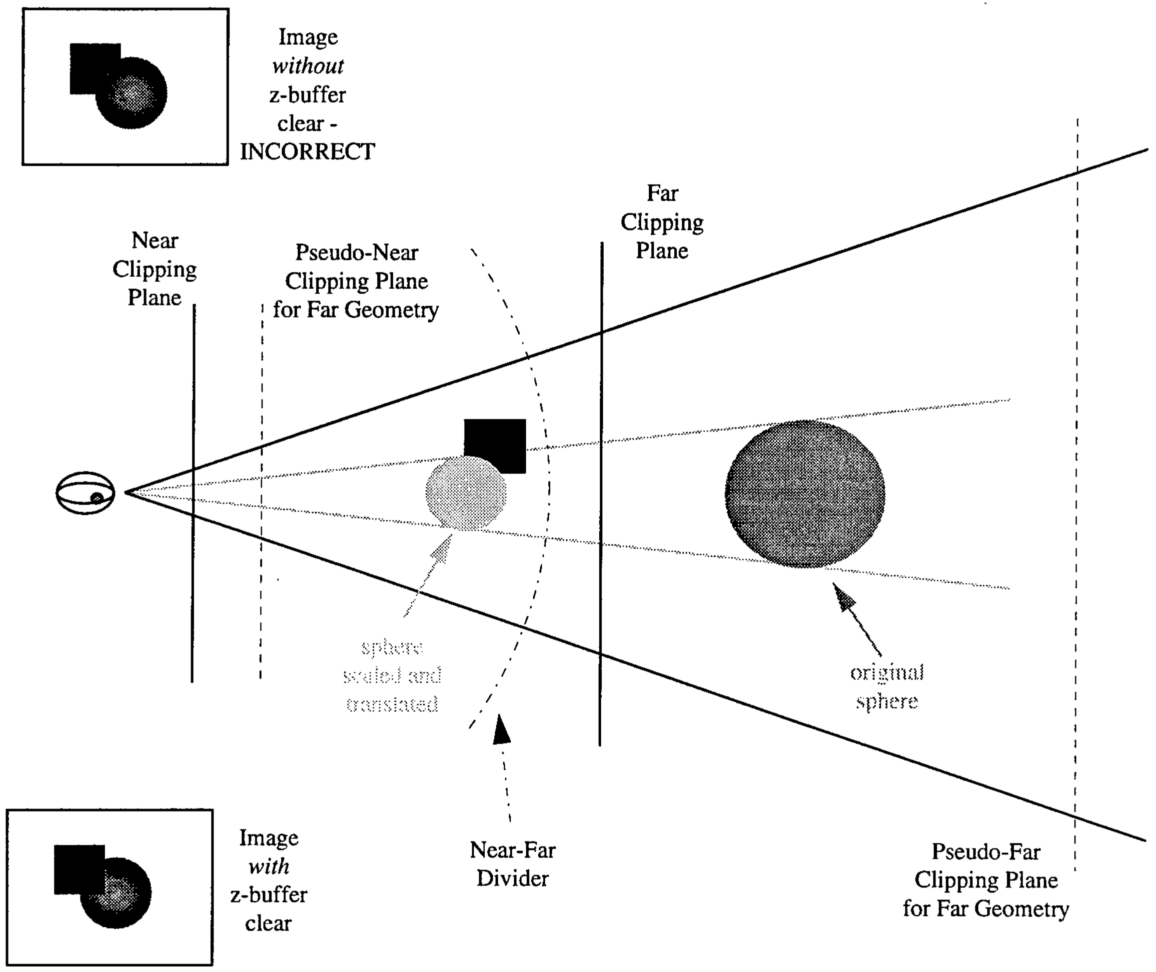


Figure 16 - Profile of Viewing Frustum demonstrating Flimmer Removal.

The final implementation actually used more than two bins to separate geometry. To improve performance, a permanent-far bin was used to hold geometry known to remain in the far bin. Examples included the stars, Sun, Moon, and Earth (terrain was separate from the Earth). To correctly render transparent geometry (used extensively in the interface), the transparent geometry must be rendered last. This ordering necessitated the creation of a transparent-near bin rendered after the (opaque) near bin.

Figure 17 - Figure 22 illustrate the separation of a scene into appropriate bins. Figure 17, Figure 18, Figure 19, and Figure 20 show the contents of the permanent-far, far, near, and transparent-near bins, respectively. The composition of these bins without the z-buffer clear is shown in Figure 21. The correct image resulting from a z-buffer clear between the far and near bins is shown in Figure 22.

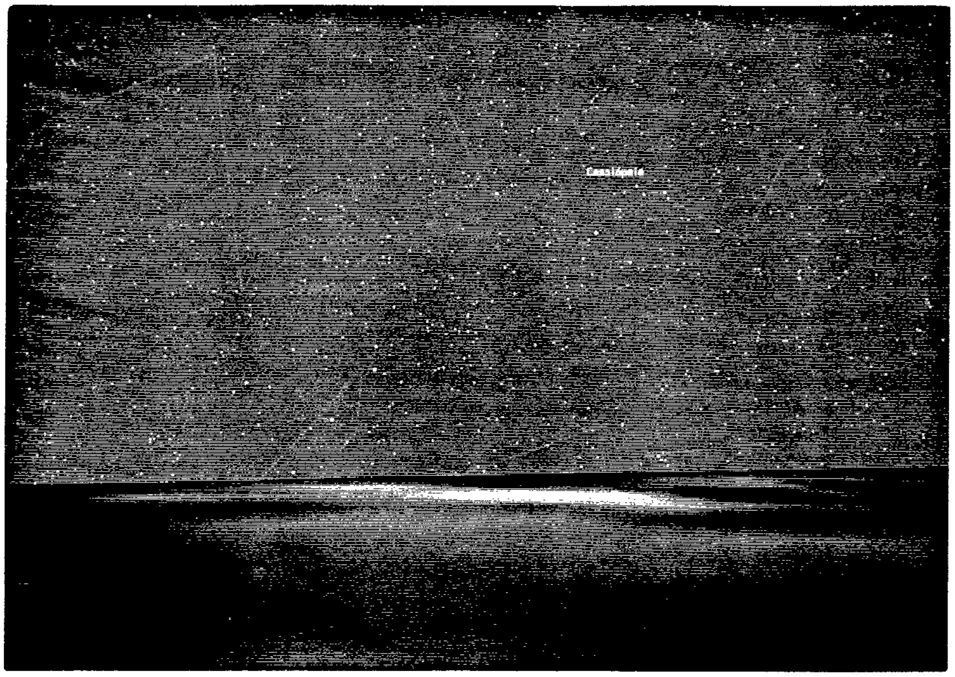


Figure 17 - Contents of Permanent-Far bin.

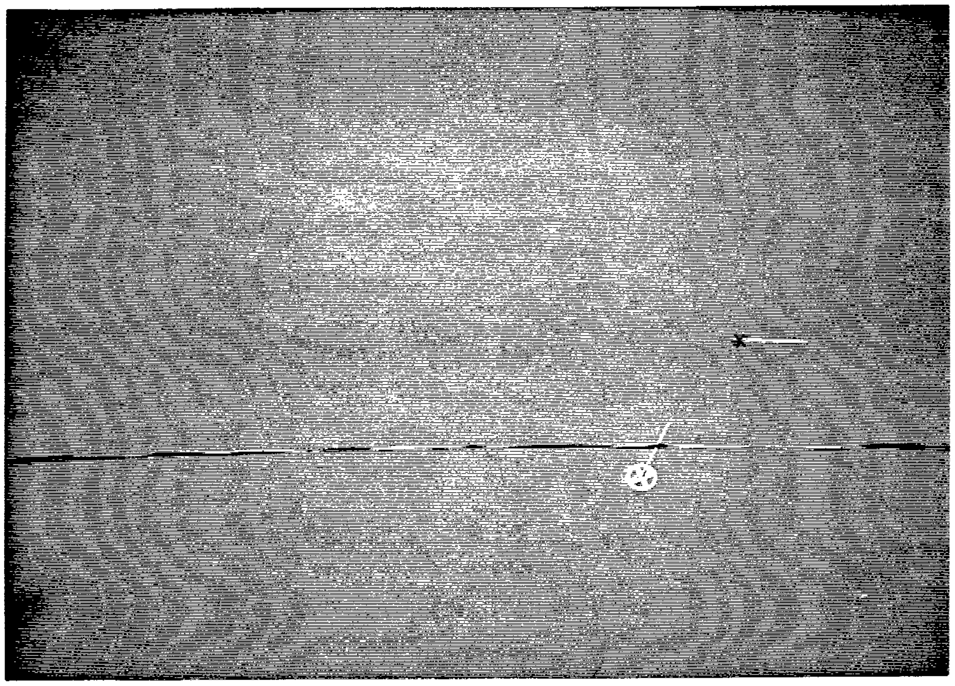


Figure 18 - Contents of Far bin.

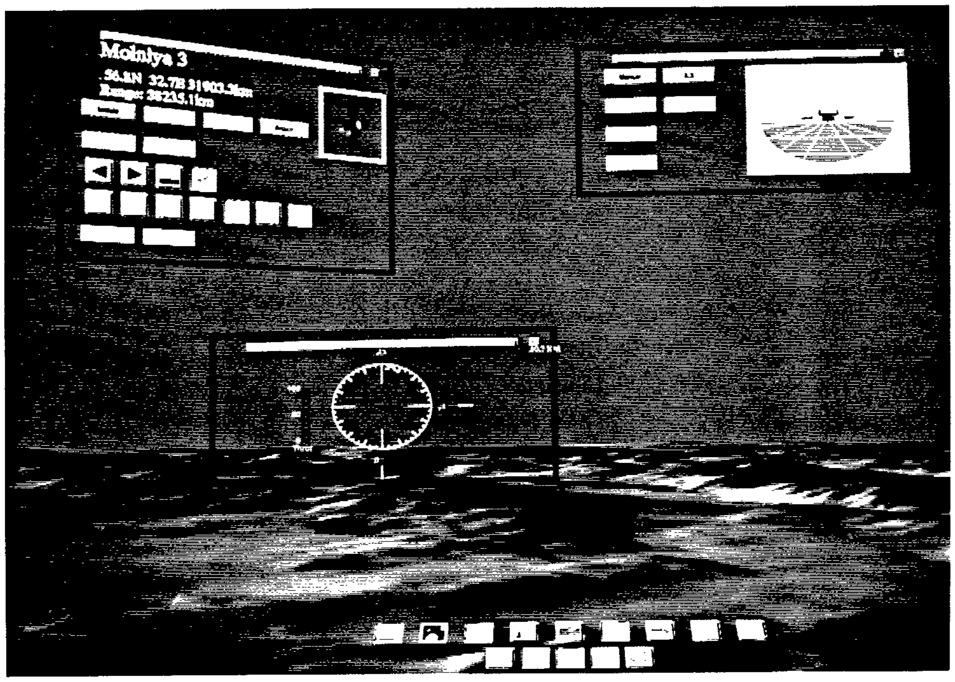


Figure 19 - Contents of Near bin.

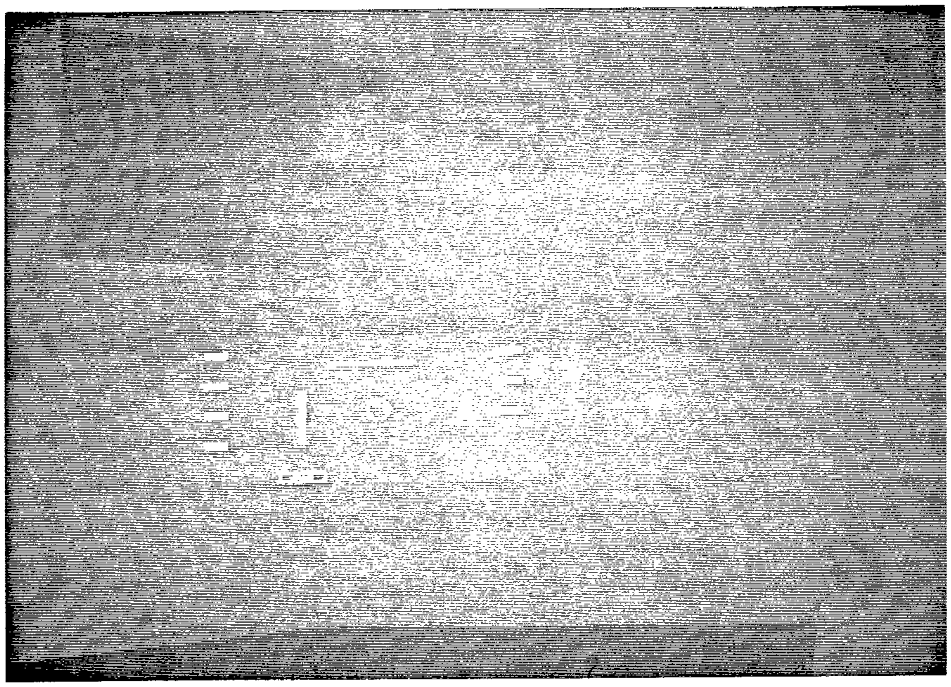


Figure 20 - Contents of Transparent-Near bin.

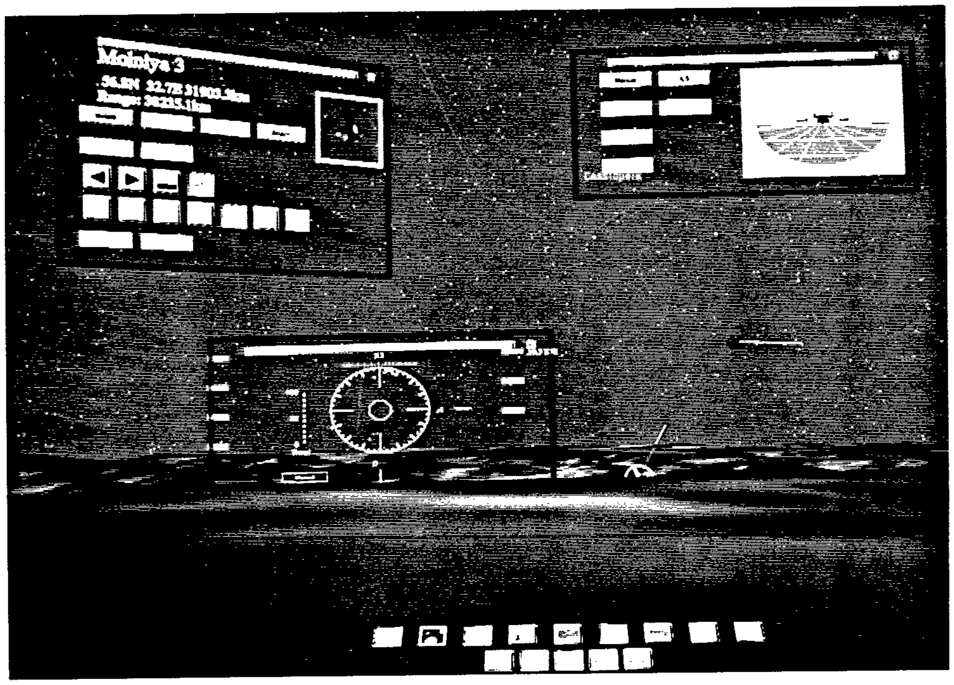


Figure 21 - Incorrect image without intermediate z-buffer clear.

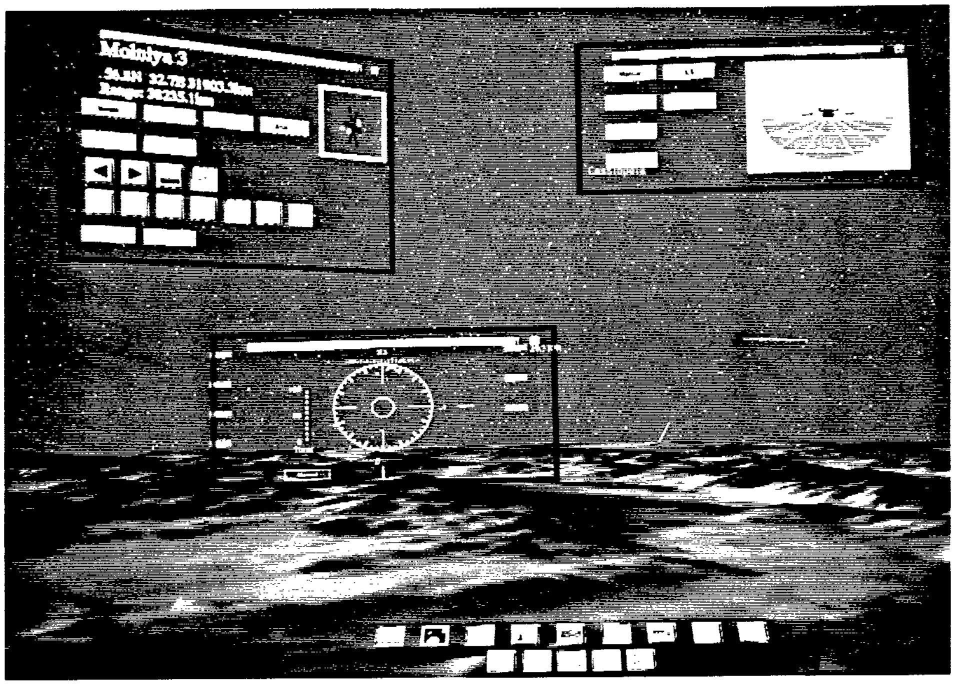


Figure 22 - Correctly rendered image.

6.2.1 Complications

Although the flimmer removal technique allowed the VSP to model an expansive region, it introduced several complications that had to be compensated for. The first involved the use of multiple levels of detail and the second complicated the use of instancing to reduce memory requirements. Each of the complications and the corresponding solutions are discussed below.

Performer pfLOD nodes can be used to automatically rendered different geometry based on the node's distance from the viewpoint. However, the flimmer removal technique complicated this process by dividing the position of far geometry by a scaling factor. The TwoLODObject corrected for this scaling factor and ensured correct rendering of its geometry.

Entities requiring more than two levels of detail used TwoLODObject to extend the number of levels available to the VSP. The extension was accomplished by first setting the TwoLODObject switch (the distance where the two primary levels were exchanged) to the near-far divider used for flimmer removal. Two separate models were built corresponding to geometry that would exist in the near and far bins. Additional LOD's within each model increased the overall levels of detail. To ensure that the far LOD's operated correctly, the switch values within the far models were divided by the scale factor of the far bin. Scaling the switch values was required because the scaling performed by the Renderer as part of flimmer removal did not affect the internal LOD switch values. In this manner, the TwoLODObject allowed the VSP to use arbitrarily complex LOD structures in spite of the complexity introduced by the flimmer removal technique.

The second complication involved the use of instancing. Instancing is a technique designed to reduce memory usage by taking advantage of replicated geometry in a scene. A single copy of the geometry can be loaded into memory and referenced from multiple places in the scene graph. A GeometryModel class was built to simplify the use of instancing in the VSP. This class assigned a identifier to all geometry that could be potentially instanced. Simulation components requested geometric models by their unique identifier and the GeometryModel automatically returned the instanced geometry or loaded the geometry if it had not previously been instanced.

The flimmer removal complications to instancing surfaced when geometry was instanced by two entities assigned to different flimmer bins. In this case, one of the entities had its geometry rendered at the incorrect time. The GeometryModel prevented this problem by maintaining two separate copies of the geometry corresponding to the near and far bins. As with SimObjects, the GeometryModel responded to the PutGeomInBin method by selecting the appropriate geometry and ensured that the SimObject's geometry was correctly rendered.

6.3 Modeling the environment

A vital aspect of an immersive VE is simulating the physical environment. For this research, two primary areas modeled were the terrain and the atmosphere.

6.3.1 Terrain

Although the VSP operated primarily in orbits above the Earth, during takeoff and landing significant portions of terrain were visible at a distance that required the simulation to render elevation and terrain features. Therefore, the Virtual Spaceplane used polygonal models of terrain patches based on Digital Terrain Elevation Data (DTED) Level 1 data. The National Imaging and Mapping Agency (previously the Defense Mapping Agency) prepares and distributes DTED Level 1 data. Each DTED cell covered a 1° by 1° area and was divided into 1201 profiles, each of which contained 1201 equally spaced elevation posts [DOD96].

The polygonal models were built using Coryphaeus' EasyT 4.0. EasyT had the capability to read in raw DTED files, create polygonal models based on a variety of user preferences (sampling rate of DTED posts, polygonalization algorithm, and level of detail setting) and save the resulting model in a Designer's Workbench file. Table 9 lists the parameters used to generate each cell of terrain. All of the patches used uniform edge sampling to ensure that adjacent cells of terrain fit together without gaps.

Table 9 - Terrain Generation Parameters.

Level of Detail	Polygonalization Algorithm	Number of Triangles	Switch Out Distance (km)	Edge Sampling Rate
0	Delaunay	7200	100	40
1	Delaunay	1800	300	40
2	Delaunay	200	400	40

Application of geographically correlated terrain textures obtained from Living Earth, Inc. improved the quality of the polygonal models. The Living Earth textures are based on high resolution satellite photographs which are processed to add color and correlate the textures to a specific geographic region. The VSP utilized two different texture resolutions to balance quality and performance (extensive use of texture mapping resulted in time consuming texture paging between hardware texture memory and main memory). Terrain immediately surrounding Edwards AFB used 3 arc second/pixel (≈160m/pixel) texture, while the remaining terrain patches (an 8° by 8° area centered at 35°N 118W°) used 1 km/pixel texture.

6.3.2 Earth's atmosphere

While many simulators operate within the Earth's atmosphere or in the voids of space, few must make the transition from one to the other. The VSP performed an approximation of the visual transition from clear blue sky to the star-filled black of space that occurred when the spaceplane passed from the earth's atmosphere to space and when day turned to night.

This research extended the monochrome 5000 star field existing in the Solar System Modeler to over 32,000 stars varying in size and color depending on the star's brightness and class, respectively. This star field was generated using a star catalog obtained from the Astronomical Data Center at NASA's Goddard Space Flight Center [ADC97]. The addition of lines representing the constellations further improved the star field. The constellation database was obtained from Chris Dolan, a doctoral candidate in astronomy at the University of Wisconsin [Dolan97]. Deriving the star and constellation data from different catalogs resulted in small errors between the line segments of the constellations and their corresponding stars. This error was deemed acceptable as a result of its small magnitude and the effort required to remove the errors.

The atmospheric and temporal transition was accomplished by calculating a sky coefficient every frame. This coefficient varied from 0.0 to 1.0 and indicated the degree that the sky appeared black and star-filled. A coefficient of 1.0 indicates a completely black sky and stars and constellations at their peak brightness; when the coefficient was 0.0 there were no visible stars or constellations and the sky was a light blue shade. The sky coefficient is composed of two factors, one for the current altitude and one for the local time of day. The altitude factor was determined by comparing the altitude of the viewpoint with two constant values, each signifying the top and bottom of an atmospheric transition layer. If the view was below the bottom threshold, above the top threshold, or in between, the VSP assumed the view was fully in the atmosphere, in space, or in the transition region, respectively. The local time of day factor is based on the angle created by extending vectors from the viewpoint to the center of the Earth and from the viewpoint to the Sun's position. Figure 23 illustrates the change in the sky coefficient as altitude and time of day change. Although this coefficient method does not accurately represent why the day/night transition is observed (the blue sky is caused by the scattering of light by the atmosphere - at high altitudes there is less atmosphere to scatter the light and at night there is less light to scatter - both contribute to darkening the sky), it sufficiently portrays the phenomenon to the user with very little computation.

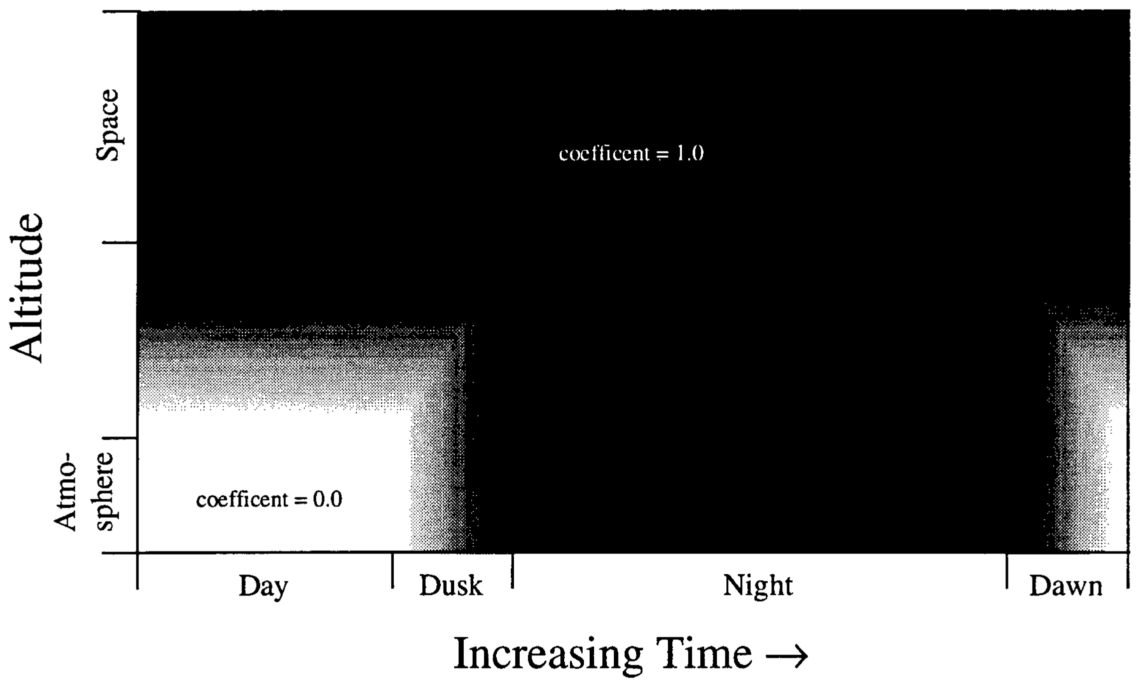


Figure 23 - Change in sky coefficient versus altitude and time.

Each of the components of the sky (the color, the stars, and the constellations) used the coefficient to vary an aspect of themselves that resulted in the overall transition from atmosphere to space. The sky color changed by modifying the color Performer used to clear the frame buffer each frame. The stars and constellations faded in and out by using anti-aliased points and lines, respectively; when the point size or line width approached zero, the objects disappeared.

6.4 Implementation Conclusion

This chapter described various implementation issues including jitter removal, flimmer removal, terrain and atmospheric modeling necessary to present an effective virtual environment. The flimmer removal technique logically increased the depth buffer by sorting the geometry in a scene into two bins depending on the geometry's distance from the viewpoint, uniformly scaling and translating the far geometry, and then rendering each bin separately with a depth buffer clear in between, resulting in a correctly rendered scene. Polygonal terrain models utilizing DTED data and geographically correlated textures and an approximation of the atmospheric change visible at day/night and atmosphere/space transitions increased the degree of immersion induced by the VSP. The next chapter presents the results and shows screen captures of the VSP.

7. RESULTS

The results chapter will mirror the structure of the requirements chapter by presenting a brief overview of the results of the entire VSP accompanied by more detailed results of topics relating directly to this research effort. An analysis of the implementation's success at realizing the original architectural and design goals and of the flimmer removal technique will conclude the chapter.

7.1 Completion of Requirements

This section will concentrate on the results of the research completed for this thesis with respect to the overall VSP requirements. Readers are encouraged to peruse the companion theses by Capt John Lewis [Lewis97] and Lt Troy Johnson [Johnson97] for detailed results on the remaining requirements.

7.1.1 Simulated Capabilities

Table 10 lists the capability requirements and corresponding method of completion.

Table 10 - Completion of Capability Requirements.

ID	Requirement	Resolution
	Flight Characteristics	
1.11	Maneuvering on runways	TaxiProp
1.12	Flight through the atmosphere	AeroProp
1.13	Maneuvering in space	AstroProp
1.14	Transition between flight regimes	OrbitEntryProp, common interfaces of PropModels
	Manual Operation	
1.21	Manually operate in the atmosphere	Input methods of AeroProp
1.22	Manually operate in space	Input methods of AstroProp
	Automatic Operation	
1.31	Automatically takeoff	TAKEOFF mode of Autopilot
1.32	Automatically fly specified routes	FLYROUTE mode, Route of Waypoints
1.33	Automatically enter orbit	ENTERORBIT mode of Autopilot
1.34	Automatically modify orbital parameters	HOHMANN and RENDEZVOUS modes
1.35	Automatically reenter the atmosphere	REENTER mode of Autopilot
1.36	Automatically land	LANDING mode of Autopilot

The architecture designed for the VSP succeeded in supporting the variety of propagation models used in the VE. The separation between PropModel and SimObject allowed development of a suite of propagation models paired with independently developed SimObjects to quickly and reliably populate the virtual environment with entities. A set of access methods in the PropModel superclass provided standard interfaces to access state information for the various PropModels. The separation of propagation and representation reduced the development difficulties associated with multi-regime entities such as the Gryphon and alleviated the need to create a propagation model that could operate in all flight regimes.

Although not implemented, the strong object oriented design and encapsulation should permit computationally intensive propagation models to fork off separate processes that communicate via the CODB. Decoupling these calculations from the primary simulation loop may improve performance by distributing computation and fidelity by increasing the update rate of models. State access methods (GetPosition, GetOrientation, GetVelocity, etc.) could be rewritten to access the most recent data from the CODB, and prevent other simulation components from feeling the impact of this restructuring (other than the increased performance).

The Autopilot class successfully performed a variety of basic maneuvering operations based on its state (TAKEOFF, FLYROUTE, LANDING, RENDEZVOUS, etc.), the current PropModel, and user specified inputs. The GetPropModelType method of PropModel, in conjunction with the Autopilot's state, provided the capability to automatically control the Gryphon across multiple flight regimes. For example, the TAKEOFF mode typically started in TaxiProp, but finished at a safe altitude above the ground in the AeroProp model. By knowing the PropModel type, the Autopilot reacted differently and provided appropriate inputs at different stages of a task. The Autopilot is not, nor was it intended to be, a fully functional, *intelligent* autopilot. However, more sophisticated control functionality could be added by changing the internal routines performed by the Autopilot.

7.1.2 Supported Missions

The required mission capabilities listed in Table 11 were successfully integrated into the VSP. The current implementation places restrictions and/or makes assumptions concerning parameters affecting the missions. The rendezvous mission does not include any time or fuel restrictions and does not provide the capability to physically dock with orbiting objects. The satellite deployment mission does not model the boost of a satellite to a higher orbit typical of many launches and only a single type of satellite can be launched. More information concerning mission support is given in Lt Johnson's thesis [Johnson97].

Table 11 - Completion of Mission Requirements.

ID	Requirement	Resolution
	Supported Missions	
2.1	Rendezvous with orbiting object	Auto RENDEZVOUS capability, Autopilot
2.2	Deployment of satellite	Payload capability of SimGryphon

7.1.3 User Interface

The architectural design succeeded in providing a framework for the development, implementation, modification and testing of a user-centered virtual interface. Initially, very little was known about the style, methods, and functionality of the interface, yet the architecture provided the flexibility to experiment and the firewalls to prevent changes from accelerating the system integrity towards chaos. The interface instantiations used to fulfill each interface requirement are listed in Table 12.

Table 12 - Completion of Interface Requirements.

ID	Requirement	Resolution
	Interaction Methods	
3.11	All functionality via three button mouse	Left button - geometry selection
		Middle button - head movement
		Right button - field of view
3.12	HMD with head tracking	N-Vision HMD and Ascension Bird™
	Configurable Cockpit	
3.21	Selectively display information	Minimizable panels
3.22	Modify location of information	Movable panels
	Displayed Information	
3.31	Gryphon state in atmosphere	Virtual HUD, Aero panel
3.32	Gryphon state during entry/reentry	Virtual HUD, Trajectory, Aero panels
3.33	Gryphon state in space	Virtual HUD, Orbit panel
3.34	State of consumables	Engineering and Payload panels
3.35	Target information	Target panel
3.36	Locating/acquiring targets	Target panel, selection of locators
3.37	System management and diagnosis	Engineering and Payload panels
3.38	Investigate hyper-text paradigms	Engineering panel
3.39	Minimize obstruction of view	Transparent panels
	Controlling the Gryphon	
3.41	No throttle and stick	Mouse interaction, Virtual HUD
3.42	Change state in the atmosphere	Virtual HUD, Autopilot
3.43	Change state in space	Virtual HUD, Target panel

The architecture succeeded in supporting the incremental development of the user interface by isolating user interaction from all components of the VSP, with the exception of the main Sim class. The Sim's TranslateInputs routine polled the various input devices and either directly caused changes in the VE (as in the case of keyboard input) or generated events processed independently of their source. Generally, the events were generated by using the mouse to select geometry in the virtual environment.

The conversion of user input to events allowed multiple input sources to reliably produce identical results and made the development of event processing independent from input processing. The event method could also provide interaction for an intelligent agent. In this case, the agent could produce events that cause changes in the interface and/or the environment on behalf of the user.

To simplify interaction development, event processing was dividing into two stages. The first stage occurred in the Sim and caused changes to entities and properties of the virtual environment. The second stage occurred in the Cockpit and caused changes to the virtual user interface. This staging was performed for two reasons. The Sim was the only class with access to all the entities and components of the VE. Second, the Cockpit was a dynamic, complex component that underwent continual, occasionally radical changes. The alternative, requiring a multitude of methods in the Cockpit to generate the necessary functionality, would have increased the complexity of the event processing.

Results on the actual interface design, development, and usability are given in Capt John Lewis' thesis [Lewis97].

7.1.4 Virtual Environment

The VSP succeeded in developing several aspects of the environment previously unexplored at the AFIT Virtual Environments Lab. The integration of geographically accurate terrain into a simulation based on a round earth coordinate system (WGS84) resulted in the ability to immerse the user in the atmosphere, in space, or anywhere in between. The visible atmospheric transition increased the presence of the user by making the environment change as expected. The environmental requirements summary is shown in Table 13.

Table 13 - Completion of Environmental Requirements.

ID	Requirement	Resolution
	Environment	
4.1	Convincing terrain near Edwards AFB	DTED based with LOD and textures
4.2	Model Earth, Sun, Moon	EarthProp, SunProp, MoonProp
4.3	Earth orbiting objects	GPS, DMSP, DSCSIII, TDRS, Molniya, Space station
4.4	Day/night, atmospheric/space transition	Calculation of sky coefficient, fading stars

Adding realistic terrain based on DTED data and geographically correlated textures dramatically improved the immersive effect of the VSP. Previous efforts using either finite flat planes or terrain created from someone's imagination immediately destroyed any sense of immersion. However, the combination used in the VSP was accurate enough to invite the user to accept the environment. The user could fly around and see the green mountains rise up around the desolate tan lake beds of the Mojave Desert where Edwards AFB is situated. Figure 24 and photo here

Figure 25 are screen captures from the VSP showing the terrain around Edwards AFB.

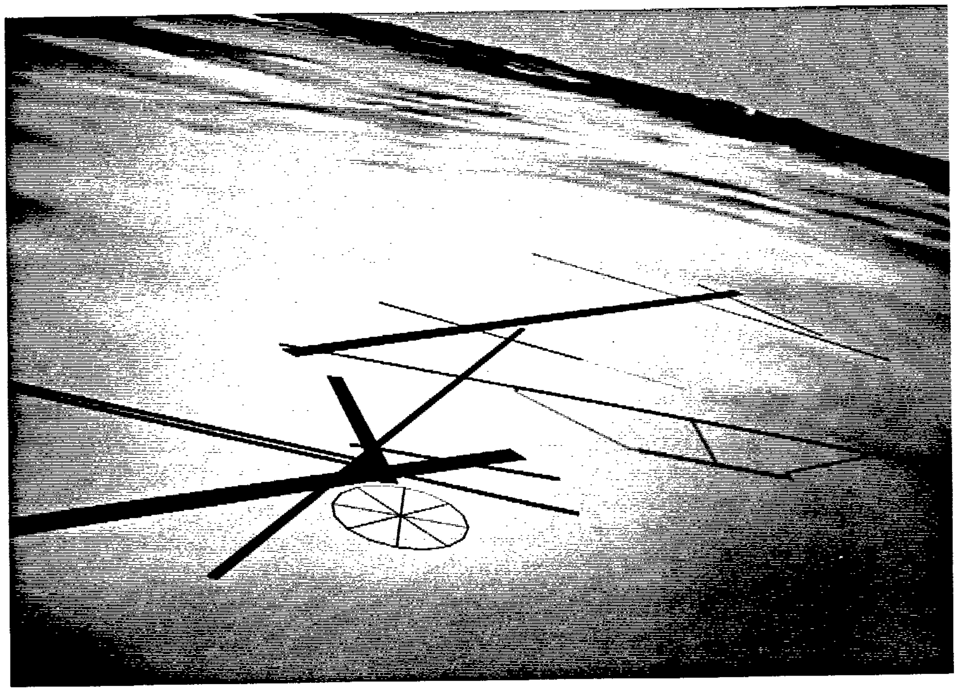


Figure 24 - Runway pattern of Edwards AFB.

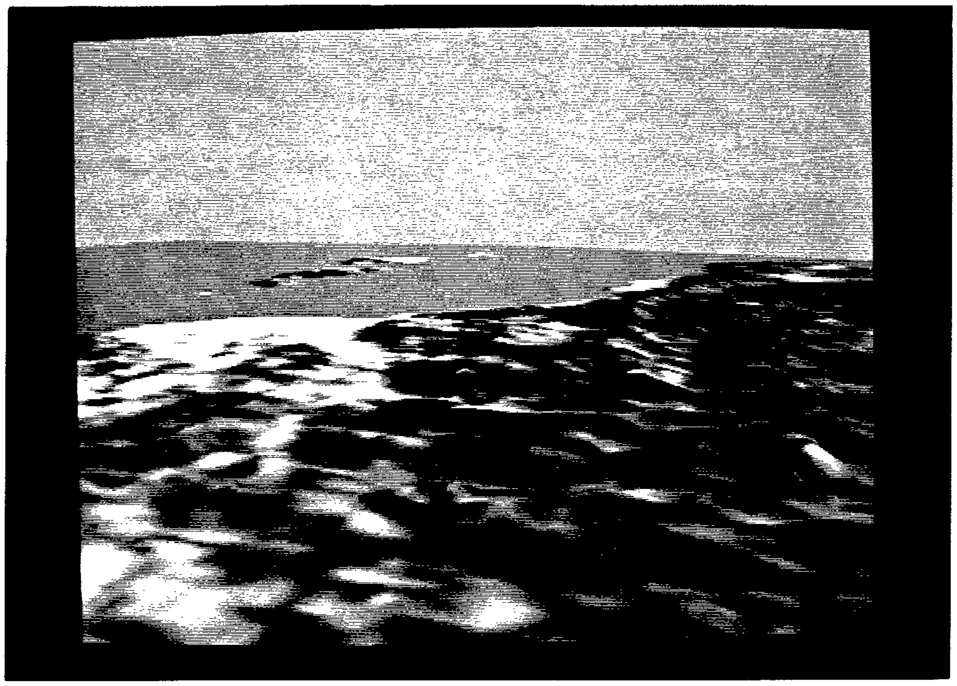


Figure 25 - Example terrain near Edwards AFB as seen from spaceplane.

The Sun and Moon also heightened the sense of immersion by improving the modeling of day/night illumination differences, moon phases, and even the seasonal change in sunrise and sunset times. Figure 26 is a view of the Earth from a high earth orbit showing the day/night terminator. Unfortunately, an element missing from the VSP is the casting of shadows. Addition of shadows would likely offer another dramatic increase in user immersion and automatically create eclipses and other illumination effects.

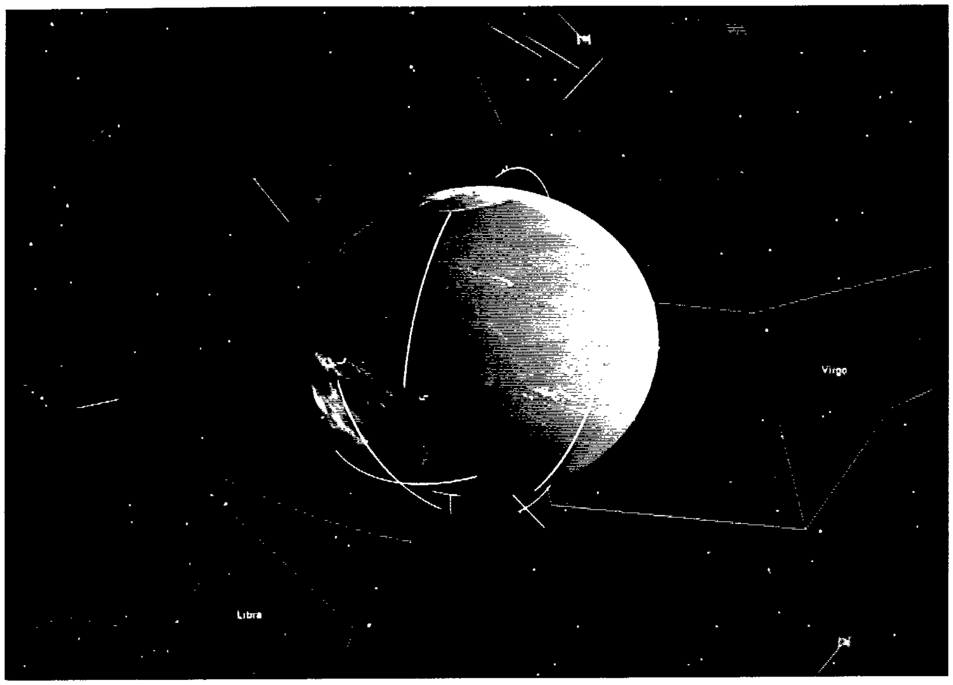


Figure 26 - View of Earth from high Earth orbit showing day/night terminator.

The VSP populated the VE with numerous satellite systems. Satellites from the Global Positioning System (25 satellites), Defense Meteorological Satellite Program (8), Defense Satellite Communication System (10), Tracking and Data Relay System (6), Molniya system (12) and a space station were added to the VE by creating polygonal models and using NORAD Two-Line Element (TLE) sets for orbit initialization. Figure 27 - Figure 32 show the satellite models in their respective orbits.

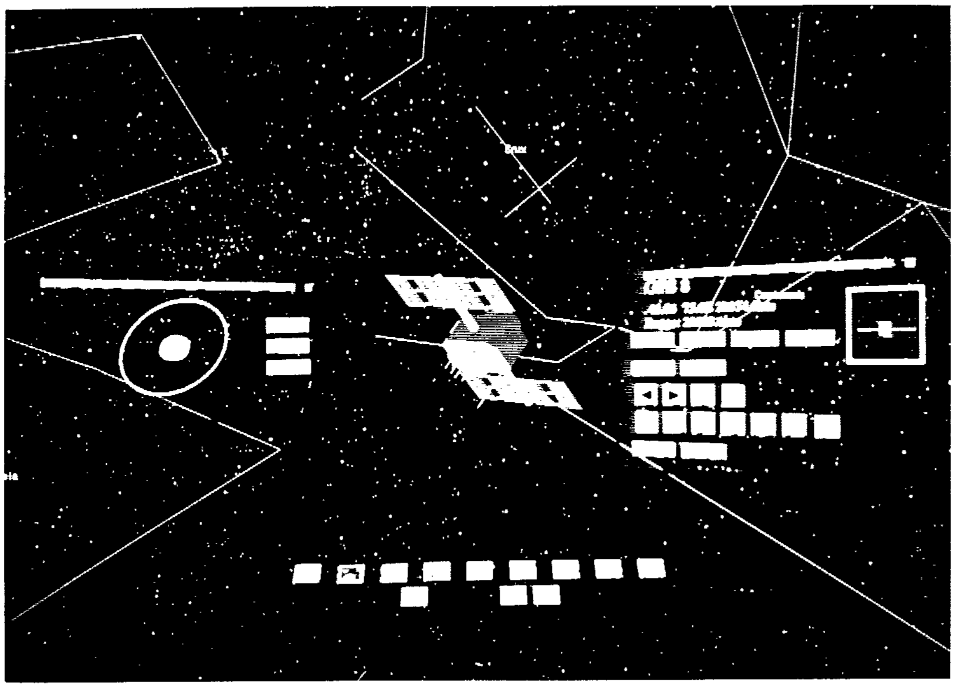


Figure 27 - Global Positioning System satellite.

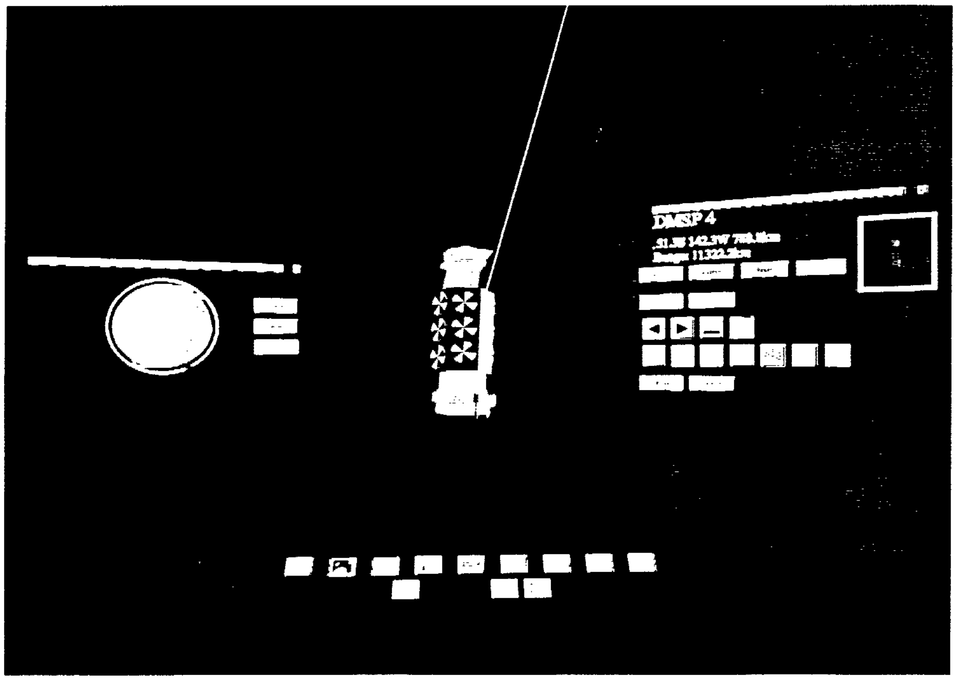
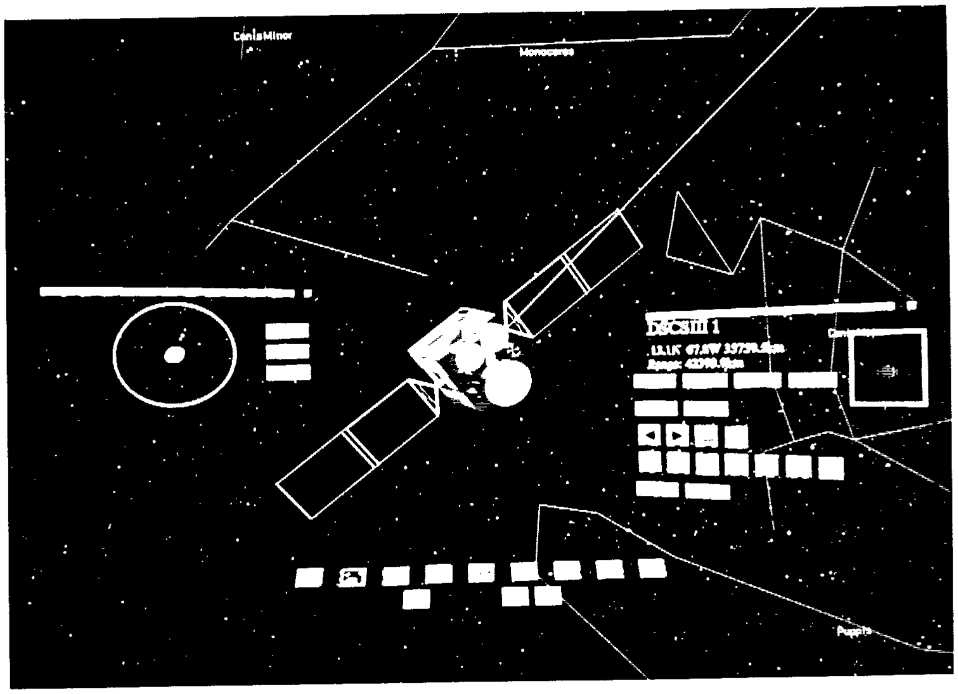
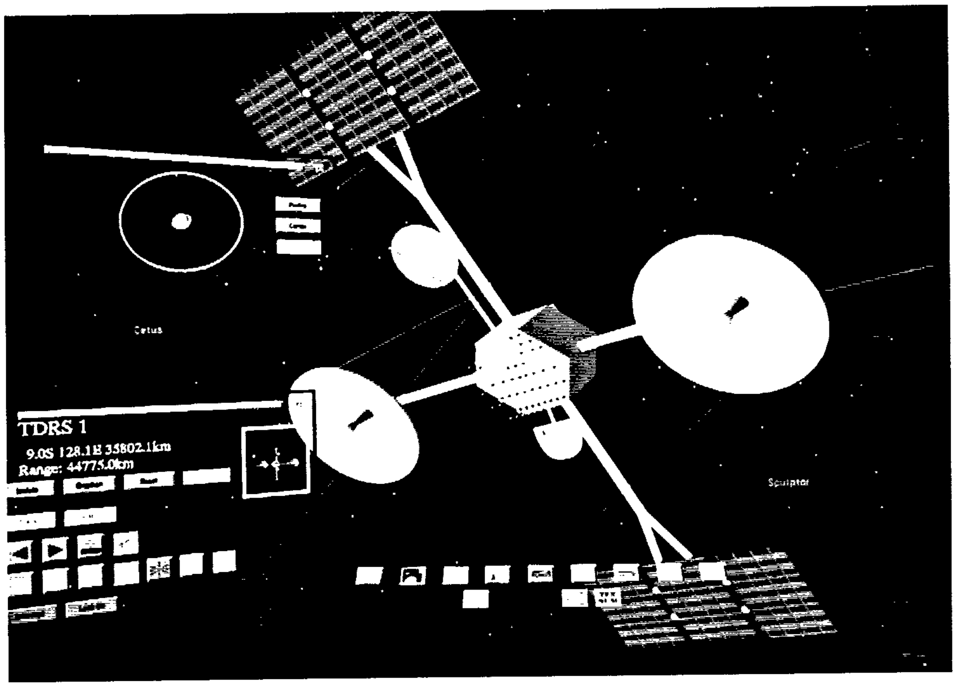


Figure 28 - Defense Meteorological Satellite System satellite.



DSCSIII 1
[illegible] 67.8W [illegible]
Range: [illegible]

Figure 29 - Defense Satellite Communication System satellite.



TDRS 1
9.0S 128.1E 35802.1km
Range: 44775.0km

Figure 30 - Tracking and Data Relay System satellite.

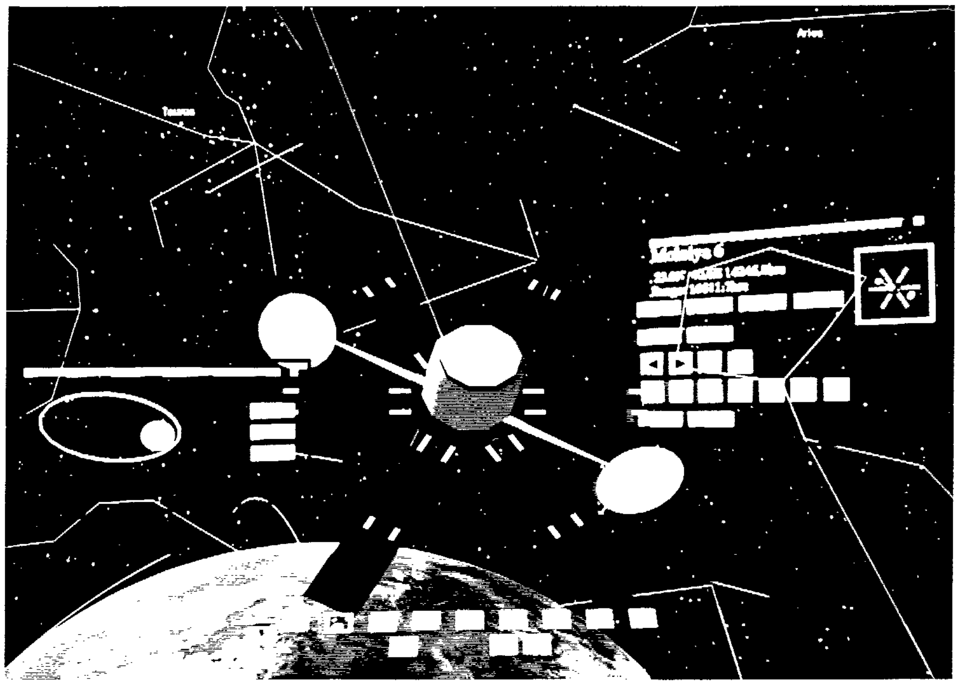


Figure 31 - Molniya satellite.

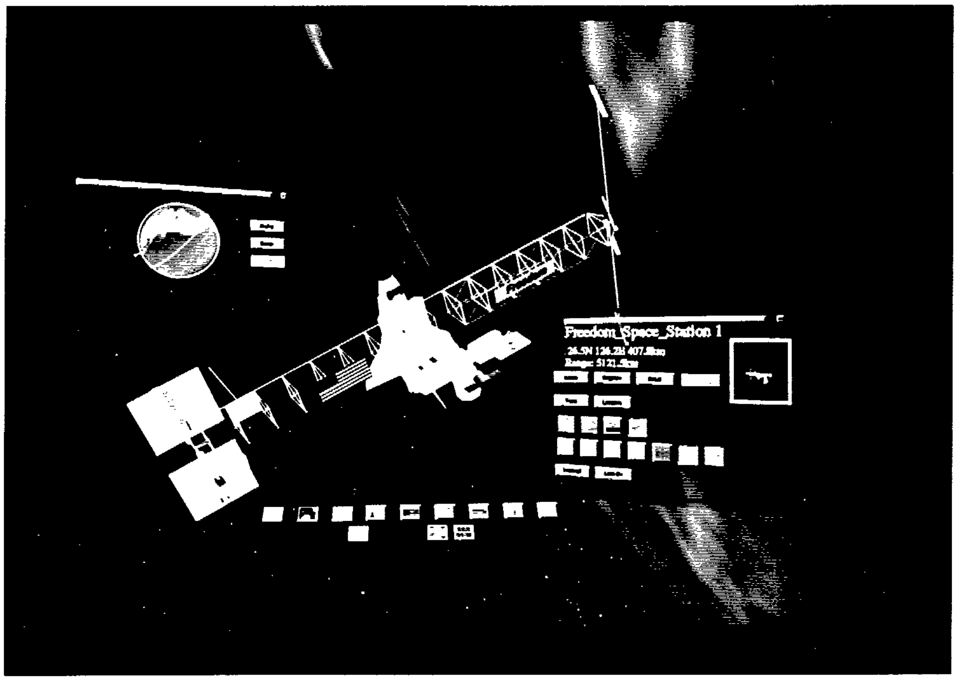


Figure 32 - Space Station.

The final environmental requirement related to the transition from the atmosphere to space. Changing the color of the sky and modeling the fade in and out of stars provided strong user cues about the current altitude and relative position of the Sun. The constellations provided useful navigation and orientation clues in the space environment.

7.1.5 Miscellaneous

Table 14 lists the Miscellaneous requirements for the VSP and how the implementation satisfied them.

Table 14 - Completion of Miscellaneous Requirements.

ID	Requirement	Resolution
	Miscellaneous	
5.1	Accept remote entities via DIS	DISEntityManager, DISEntity, DISEntityProp
5.2	Transmit Gryphon state via DIS	BroadcastSimObject in Sim
5.3	Mean of 15 frames per second	Average of 14.05 frames per second

The VSP accepted and recognized a subset of the DIS protocol packets. In particular, only entity state PDU's were recognized and used to simulate remote entities, and the Gryphon was the only locally simulated entity that had its state information transmitted. However, the BroadcastSimObject routine used to transmit state information of local entities accepts any SimObject as a parameter. Therefore, extension of requirement 5.2 to include other entities modeled in the VSP could be accomplished by simply calling BroadcastSimObject on additional SimObjects.

Meeting the minimum frame rate requirement was challenging and required constant attention and code optimization. The primary limiting factor for frame rate was the draw process, so any simplification or minimization of geometry that did not penalize the rendered quality of a scene was desirable. This geometric simplification was generally accomplished using specialized culling or LOD's.

A key example of the specialized culling was the 32,000 point star sphere. While the SGI Reality Engine efficiently rendered textured polygons, it was much less efficient at rendering the anti-aliased lines and points used for the star field. The efficiency was further reduced because only a couple hundred stars were visible at any instant (given a reasonable field of view). By grouping the stars into spatial regions using multiple pfGeoSets, the cull process drastically reduced the number of stars that were actually rendered, resulting in a shorter draw process and improved frame rate.

Several sample measurements of the VSP frame rate are given in Table 15. The first trial exercised the VSP in the atmospheric regime by performing an automatic takeoff, followed by a flyby of Edwards AFB and eventual landing. During this trial the atmospheric control system, virtual landing system, Navigation, and Aero Flight panels were all active with the field of view and head position at initial settings. The second trial was executed while in the space regime and consisted of performing an automatic rendezvous with DMSP satellite number two using deltaV option one. This trial lasted approximately ten minutes, during which the Navigation, Astro Flight, Target primary- and sub- panels were active with the field of view and head position once again at initial settings. During both trials all constellation lines, entity locators and trails were displayed.

Table 15 - Average Frame Rates of VSP trials.

Trial	Four 250 MHz R4400's Onyx Reality Engine^2 192 Mbytes main memory	Four 195 MHz R10000's Onyx^2 Infinite Reality 384 Mbytes main memory
Atmospheric	13.36 f/s	30.05 f/s
Orbital	14.737 f/s	29.94 f/s

The results show that the frame rate requirements were nearly met using the Reality Engine^2, an impressive feat considering the VSP contained over half a million primitives (triangles, lines, or points) and over 13 Mbytes of textures. The Infinite Reality system demonstrated its prowess by producing a consistent 30 frames per second.

7.2 Architectural Goals

The architectural goals specified in Chapter 4 were not explicit requirements of the VSP, but the completion and adherence to these goals directly impacted the success of the VSP. By acknowledging the development process, separating the responsibilities of simulation components, and supporting the user interface development, the architecture aided the management and resolution of the demanding, ever-changing requirements of the VSP.

7.2.1 Support Development Process

To support the development process, the architecture needed to allow multiple people to concurrently evolve the VSP without creating conflicts between components. The lack of dependencies between simulation components allowed the team members to progress independently on various aspects of the VSP without integration difficulties. Because the architectural integrity was actively maintained by the members and code integrations were conducted frequently, most integrations required less than an hour to complete and test.

The object oriented design allowed incremental growth of base class functionality to uniformly affect and extend the features and capabilities of the VSP beyond initial expectations. Examples include:

- Entity trails were added as an inherent attribute of SimObjects rather than the case by case approach used in the SSM.
- SimObjects included a method assigning a selection ID to the entity, allowing any SimObject to create an event when it was selected by the user.
- All PropModels had methods for accessing state information in a variety of standard coordinate systems. Standardization ensured that entities could interact (at least with regard to their state) regardless of the regime they operated within.
- The ability to track the viewpoint around a SimObject was added to the Renderer and required no other changes to the simulation.

7.2.2 Separation of Responsibilities

The purpose of separating the simulation responsibilities was to simplify the individual components, prevent modifications from causing changes to external components, and provide clear direction concerning where functionality should be implemented. Primary examples included rendering and entity management.

The isolation of rendering functions resulted in enormous dividends in time and effort. Development of the jitter and flimmer removal methods and resolution of all the coordinate systems used in the VSP was an evolutionary process requiring many iterations to complete. Performing all these tasks within the Renderer acted as a firewall to keep these iterations from affecting other components.

An exception to this goal was necessary for the flimmer removal, because the removal technique required sorting geometry into bins implemented using IRIS Performer calls on individual pfGeoSets. Because SimObjects maintain their local geometry (which could become arbitrarily complex), it was decided that they should account for any custom geometry configurations and assign their geometry to the appropriate bin via the PutGeomInBin method. However, SimObjects did not determine which bin they belonged in; that was the responsibility of the Renderer. SimObjects simply responded to the PutGeomInBin call.

Separating entity propagation and representation into the PropModel and SimObject classes also generated savings in time and effort by supporting code reuse. For example, TwoLODObject's were used for satellites, terrain patches, DIS entities, and the Gryphon even though the entities used completely different PropModels (in particular the AstroProp, FreeFlight, DISEntityProp, and multiple PropModels, respectively). The separation provided the means for the SimGryphon to use multiple PropModels without requiring replication of code or the development and testing of a single PropModel modeling a spaceplane's movement through multiple regimes.

7.2.3 Support User Interface Development

This goal was motivated by the observation that the style and method of both user interaction and the user interface were unknown when the architecture was designed. Therefore flexibility within an established design was needed. The VSP fulfilled this goal by separating the cause and effects of user interaction and providing specific locations to add user interaction. The cause and effect separation was accomplished by first converting user input into events and then processing the source independent events by making specific method calls for each event. The centralization of the process removed any ambiguity concerning where user interaction should be coded.

By making the top level Sim class collect and translate all user interaction into events, different input devices could reliably and consistently modify the virtual environment. This model also extends beyond user input and allows an intelligent agent to create events that perform activities normally attributed to the user.

7.3 Flimmer removal

Although development of a flimmer removal technique was not a direct requirement of this research, it became a vital element of the VSP and a key advancement resulting from this effort. The technique drastically reduced the flimmering problem caused by the immense region the Gryphon operated in and permitted the VSP to develop unhindered by concerns about the size and scale of the environment. It should be noted that the flimmer removal technique does not completely eliminate all flimmer; it only reduces the degree of flimmering in a specific scene.

The flimmer removal technique effectively increased the logical resolution of the system's depth buffer, without requiring changes in the underlying hardware. For example, in the VSP the distance from the near to far clipping planes was increased a thousand times without introducing additional immersion destroying flimmer. Without the flimmer removal technique, the z-buffer would have required ten additional bits per pixel to obtain an equivalent increase in depth buffer resolution ($2^{10} \approx 1000$). Given a screen resolution of 1280 by 1024, this increase equates to 1600 kilobytes of memory.

In comparison, the memory requirements of the flimmer removal technique increased with the number of entities in the virtual environment. Assuming the use of an 8-bit integer to store the bin assignment of each entity (yielding 256 bins, considerable more than the four bins used in the VSP), over 1.6 million entities could be simulated before the memory requirements of this technique surpassed the equivalent increase in the depth buffer.

There were several complications and disadvantages associated with the technique. Whereas the memory requirements were reduced (as compared to an increase using z-buffer only), the technique required additional processing to sort the geometry into the appropriate bins. Optimizations were performed that ensured that the additional sort processing was only completed when entities transitioned between the near and far bins. The current implementation is dependent on IRIS Performer functionality (specifically the independent rendering of each logical bin), so migration to other environments may require additional effort to replicate this functionality.

Complications arising from using multiple levels of detail were discussed in the Implementation chapter. The TwoLODObject resolved the level of detail problems created by the flimmer removal technique and the GeometryModel class allowed the use of instancing to reduce memory requirements. While the level of detail complication was resolved, a couple other potential problems with the current flimmer removal technique were not addressed, primarily because they did not impact the VSP.

The first relates to intersection testing. Performer contains functionality for performing intersection testing of line segments against the scene graph. Unfortunately, flimmer removal creates a discrepancy between entities' apparent and actual positions with regard to the scene graph (see Figure 33). The translation of far geometry completed by the flimmer removal technique may incorrectly occlude near geometry and result in erroneous intersection data. An intersection routine could compensate by detecting an intersection with far geometry and either ignoring the far intersection or continue the intersection process searching for near intersections that are actually closer.

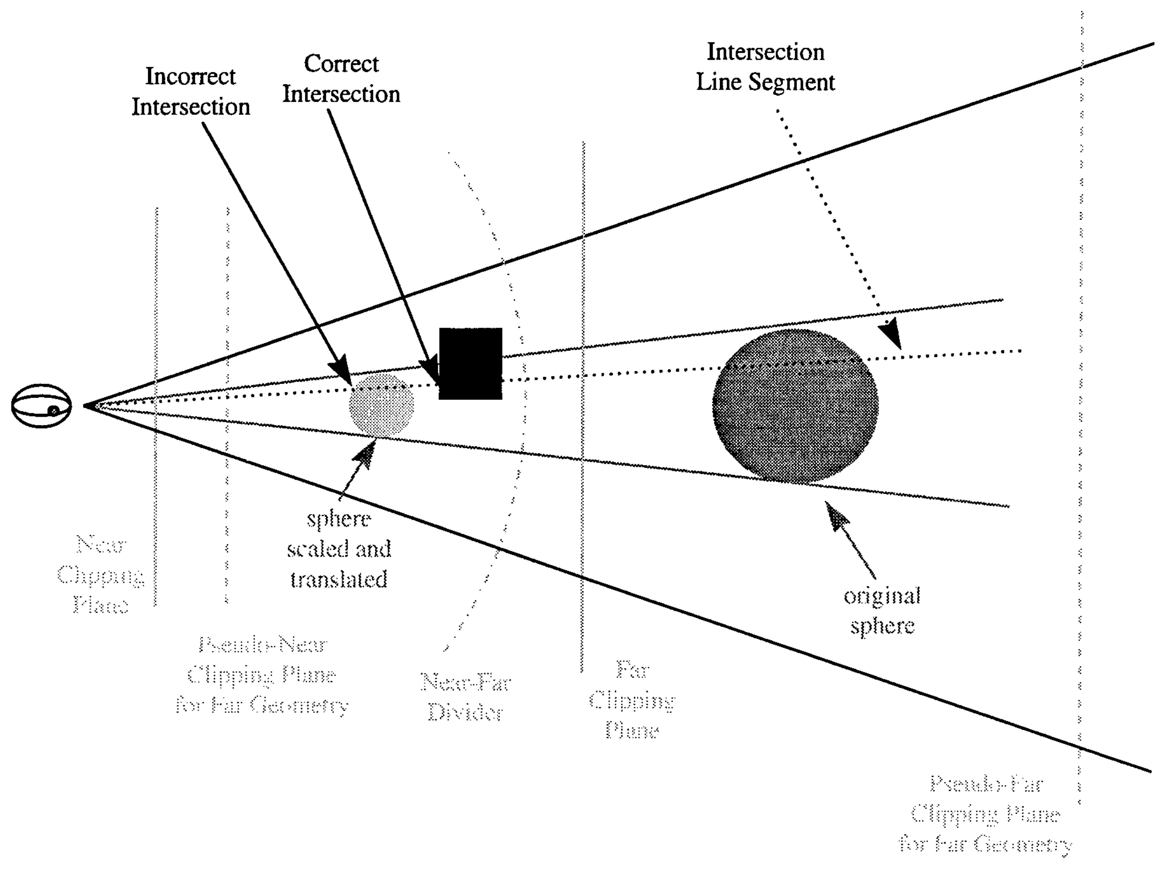

Incorrect Intersection
Correct Intersection
Intersection Line Segment
sphere scaled and translated
original sphere
Near Clipping Plane
Pseudo-Near Clipping Plane for Far Geometry
Near-Far Divider
Far Clipping Plane
Pseudo-Far Clipping Plane for Far Geometry

Figure 33 - Intersection complications introduced by flimmer removal.

The second problem is due to the finite distance between the near-far divider and the far clipping plane used by the flimmer removal technique. The problem surfaces when a very large object in the near region approaches the near-far divider. If the object's geometry extends beyond the far clipping plane, it will be partially clipped until it transitions to the far region, at which time the entire object will be visible (or possibly have different portions clipped against the far region's near clipping plane). For example, in the VSP, the near-far divider was set to 300 km and the far clipping plane (for near geometry) was at 480 km. Therefore, entities with geometry extending beyond 180 km from their origin could potentially be incorrectly rendered. Fortunately, the largest objects in the VSP were the 1° by 1° terrain cells whose corners only extended approximately 78 km from their origin. Larger objects such as the Earth did not apply since they were assigned to the permanent far bin.

Other than the complications discussed above, the flimmer removal technique resolved the rendering problems associated with the expansive region simulated in the VSP.

7.4 Results Conclusion

The discussion of flimmer removal and several resulting complications ends the Results chapter. In addition to flimmer removal, resolution of the initial VSP requirements and completion of architectural goals were addressed. The VSP succeeded in meeting the challenges presented by the requirements of the initial prototype and the architecture fulfilled it goals by providing a flexible framework for meeting the requirements using a rapid prototyping process. The final chapter will present conclusions of this research and propose areas for further investigation.

8. Conclusions and Future Work

The preceding chapter presented the results of the VSP with regard to the overall goals and requirements as well as the specific foci of this research effort. The final chapter of this document begins with a list of accomplishments attained during the development of the Virtual Spaceplane and a review of the thesis statement. The chapter closes by recommending several areas of research that would further improve the VSP's capability to model and investigate requirements for a military spaceplane.

8.1 Accomplishments

During its first year of development, the Virtual Spaceplane project succeeded in prototyping a functional virtual environment for a military spaceplane. An architecture specifically designed for rapid prototyping was used to implement a large, complex virtual environment featuring a unconventional user interface for controlling a military spaceplane through a variety of missions and operational regimes. In particular, the VSP:

- is based on a software architecture emphasizing support of the development process, separation of concerns, and user interface development
- utilizes the CODB to further support the development process and assist in the computational distribution of system components
- simulates a concept military spaceplane by dynamically switching between separate models targeted towards specific operation regimes of the vehicle

- includes a simple autopilot capable of directing the Gryphon towards an atmospheric or orbital objective
- supports rendezvous with orbiting objects and deployment of a satellite
- features a virtual user interface based on a management approach to controlling the spaceplane, rather than the traditional piloting approach.
- presents a convincing representation of terrain near Edwards AFB, the Earth, Sun, Moon, and stars, and the visible changes to the atmosphere during orbit entry/reentry.
- simulates the GPS, DMSP, DSCSIII, TDRS, and Molniya satellite systems and an orbiting space station.
- participates in large scale distributed virtual environments through a subset of the DIS suite of protocols.
- employs an original flimmer reduction technique for rendering the large spatial extent of the virtual environment.

These accomplishments demonstrate that the initial development of a virtual environment for a military spaceplane was completed and the groundwork has been laid for further investigation and research.

8.2 Thesis Statement Revisited

The specific purpose of this research effort was to develop the architecture and design for a military spaceplane virtual environment and to investigate several implementation issues relating to the large extent of the environment. The architecture's support of the rapid prototyping allowed a team of three researchers to concurrently investigate a variety of areas while preventing the concurrent changes from impeding progress on the project as a whole. An emphasis on separation of concerns and responsibilities of system components enabled the addition and modification of requirements and functionality throughout the VSP's development without requiring redesign and recoding effort of unrelated components. A centralized, two phase event model allowed an initially unknown, unconventional, and usable interface to evolve without being restricted by specific input mechanisms or changes in other components of the virtual environment.

8.3 Future Work

Despite the accomplishments and successes of the VSP project and of this particular research, continued effort in several areas should be pursued to extend the VSP beyond its current capabilities. These areas include further expansion of the distributed components of the VE, support for a greater variety of military spaceplane missions, modeling of the internal systems of a military spaceplane, continued development of the user interface, and extending the types of environmental phenomenon simulated by the VSP.

8.3.1 Distributed Virtual Environment

Although the VSP currently supports a portion of the DIS standards, further effort will increase its ability to participate in large scale distributed virtual environments. The VSP should concentrate on using the High Level Architecture (HLA) for this effort rather than the DIS standards. Investigating HLA topics complies with the recent Department of Defense mandate and takes advantage of the sophistication and features of HLA. Computational distribution can also be accomplished on a more local scale by forking off additional processes that execute the propagation models of local entities concurrently with the main application process.

8.3.2 Missions

A wider variety of missions will increase the VSP's value as a tool for investigating military spaceplane technology. These missions may include reconnaissance of earth or space based targets, pop-up missions for deployment of military assets, docking with orbiting stations, and investigating potential on-board weapon systems. These missions are described in the military spaceplane system requirement [PLSTD97] and technology roadmap [MSPICT97] documents as desired applications. Therefore, the VSP should implement these missions to improve its ability to investigate the capabilities of a MSP.

8.3.3 MSP Systems

The current VSP models a MSP's movement through the environment without considering the internal systems required to complete the operations. The following systems should be simulated to increase the fidelity of the MSP model; engines, propellant storage, thermal protection, life support, communication, and sensors.

8.3.4 Interface

Continued development of the user interface should accompany all other future work on the VSP. Interface development applies primarily to extension of the missions and MSP systems, but may also include completely new areas. For example, intelligent agents could assist the MSP commander in completing mission objectives by filtering vast amounts of sensor data to find relevant or critical information. Any continued interface effort should comply with the existing interface guidelines to maintain consistency and usability.

8.3.5 Environment

Integration of phenomenology and environmental factors will increase the degree of user immersion and realism of the VSP. These factors may include extending regions of the globe covered by terrain models, increasing the quality of these terrain models, and introducing atmospheric and space phenomenon such as Van Allen belts and weather that potentially affects the completion of missions.

8.4 Conclusion

The domains of space and virtual environments are challenging areas that will require tremendous amounts of effort to fully explore and exploit. The new Air Force vision of Global Engagement and the investigation of military spaceplane technology are two signs that the Air Force is starting a transition into becoming a space and air force. Virtual Environment technology holds the promise of providing a new media for exploring new concepts and interacting with the ever increasing power of modern computing systems. The Virtual Spaceplane project investigated some of the capabilities possible when these two domains are merged into a single system.

BIBLIOGRAPHY

Adams96	Adams, Terry A., Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype. MS Thesis, AFIT/GCS/ENG/96D-02, Air Force Institute of Technology, Air University, December 1996.
Akeley93	Akeley, K., "Reality Engine Graphics," Computer Graphics: SIGGRAPH 93 Conference Proceedings, ACM Press, New York, 1993: 109-116.
Appino92	Appino, Perry, A., J. Bryan Lewis, Lawrence Koved, Daniel T. Ling, David A. Rabenhorst, and Christopher F. Codella, "An Architecture for Virtual Worlds," Presence: Teleoperators and Virtual Environments, Vol. 1, No. 1, Winter 1992: 1-17.
ADC97	Astronomical Data Center, "1113A - General Catalog of 33342 Stars, 1950.0 (GC; Buss 1937; ADC 1992)." NASA Goddard Space Flight Center, *http://adc.gsfc.nasa.gov/adc-cgi/cat.pl?/1/1113A/* (Feb 1997).
Barrus96	Barrus, John, W., Richard C. Waters, and David B. Anderson, "Locales: Supporting Large Multiuser Virtual Environments," IEEE Computer Graphics and Applications, November 1996: 50-57.
Bate71	Bate, Roger R., Donald D. Mueller, and Jerry E. White, Fundamentals of Astrodynamics, New York: Dover Publications, 1971.
Blau92	Blau, Brian, Charles E. Hughes, J. Michael Moshell, and Curtis Lisle, "Networked Virtual Environments," Proceedings 1992 Symposium on Interactive 3D Graphics, ACM Press, New York, 1992: 157-160.
Brooks87	Brooks, Frederick P. Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," IEEE Software, April 1987.
Bryson92	Bryson, Steve and Creon Levit, "The Virtual Wind Tunnel," IEEE Computer Graphics and Applications, July 1992.
Bukowski97	Bukowski, Richard and Carlo Sèquin, "Interactive Simulation of Fire in Virtual Building Environments," Computer Graphics: SIGGRAPH 97 Conference Proceedings, ACM Press, New York, 1997: 35-44.
Cusumano97	Cusumano, Michael, A. and Richard W. Selby, "How Microsoft Builds Software," Communications of the ACM, Vol. 40, No. 6, June 1997: 53-61.
DAF96	Deptartment of the Air Force, Global Engagement: A Vision of the 21st Century Air Force. Washington: Air Force Directorate of Stategic Planning, Fall 1996.

DOD87	Department of Defense, World Geodetic System 1984 (WGS84), Its Definition and Relationships with Local Geodetic Systems. DMA TR 8350.2, Washington: Defense Mapping Agency, 1987.
DOD96	Department of Defense, Performance Specification Digital Terrain Elevation Data. MIL-PRF-89020A, Washington: National Imagery and Mapping Agency, 19 April 1996.
Dolan97	Dolan, Chris, "Stars and Constellations." University of Wisconsin, *http://www.astro.wisc.edu/~dolan/constellations/constellations.html* (May 1997).
Ellis91	Ellis, S. R., "Nature and Origins of Virtual Environments: A Bibliographical Essay," Computing Systems in Engineering, Vol. 2, No. 4, 1991: 321-347.
Erichsen93	Erichsen, Matthew N., Weapon System Sensor Integration for a DIS-Compatible Virtual Cockpit. MS Thesis, AFIT/GCS/ENG/93D-07, Air Force Institute of Technology, Air University, December 1993.
Garcia96	Garcia, Brian, W., Design and Prototype of the AFIT Virtual Emergency Room: A Distributed Virtual Environment for Emergency Medical Simulation. MS Thesis, AFIT/GCS/ENG/96D-07, Air Force Institute of Technology, Air University, December 1996.
Hitchner93	Hitchner, Lewis, E., "Virtual Planetary Exploration: A Very Large Virtual Environment," Implementing Virtual Reality: SIGGRAPH '93: 20th International Conference on Computer Graphics and Interactive Techniques, course notes, August 1993: 4.1-4.16.
Hong97	Hong, Lichen, Shigeru Muraki, Arie Kaufman, Dirk Bartz, and Taosong He, "Virtual Voyage: Interactive Navigation in the Human Colon," Computer Graphics: SIGGRAPH 97 Conference Proceedings, ACM Press, New York, 1997: 27-34.
IEEE93	IEEE, IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications, Piscataway, N.J., 1993.
IRIS95	IRIS Performer Programmer's Guide. Silicon Graphics, Inc., 1995.
Johnson97	Johnson, Troy, D., The Virtual Spaceplane: Integrating Multiple Motion Models and Hypertext in a Virtual Environment. MS Thesis, AFIT/GM/ENG/97D-01, Air Force Institute of Technology, Air University, December 1997.
Kayloe94	Kayloe, Jordan R., Easy-Sim: A Visual Simulation System Software Architecture with an Ada9X Application Framework. MS Thesis, AFIT/GCS/ENG/94D-11, Air Force Institute of Technology, Air University, December 1994.

Kestermann94 Kestermann, Jim B., Immersing the User in a Virtual Environment: The AFIT Information Pod Design and Implementation. MS Thesis, AFIT/GCS/ENG/94D-13, Air Force Institute of Technology, Air University, December 1994.

Kunz93 Kunz, Andrea A., A Virtual Environment for Satellite Modeling and Orbital Analysis in a Distributed Interactive Simulation. MS Thesis, AFIT/GCS/ENG/93D-14, Air Force Institute of Technology, Air University, December 1993.

Lewis97 Lewis, John, M., Requirements, Design, and Prototype of a Virtual User Interface for the AFIT Virtual Spaceplane. MS Thesis, AFIT/GM/ENG/97D-02, Air Force Institute of Technology, Air University, December 1997.

Macedonia94 Macedonia, Michael, R., Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz, "NSPNET: A Network Software Architecture for Large-Scale Virtual Environments," Presence: Teleoperators and Virtual Environments, Vol. 3, No. 4, Fall 1994: 265-287.

Macedonia97 Macedonia, Michael, R. and Michael J. Zyda, "A Taxonomy for Networked Virtual Environments," IEEE Multimedia, Vol. 4, No. 1, January-March 1997: 48-56.

Meeus91 Meeus, John, Astronomical Algorithms, Richmond, Virginia: Willmann-Bell, 1991.

MSPICT97 Military Spaceplane Integrated Concept Team, Technology Roadmap for a Military Spaceplane System. Version 1.1, Draft, 14 February 1997.

PLSTD97 Phillips Laboratory Space Technology Directorate, System Requirements for a Military Spaceplane. Version 1.0, Draft, 24 April 1997.

Rohlf94 Rohlf, J., J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," Computer Graphics: SIGGRAPH 94 Conference Proceedings, ACM Press, New York, 1994: 381-394.

Rumbaugh91 Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, Object-Oriented Modeling and Design, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1991.

Shaw92 Shaw, Chris, Jiandong Liang, Mark Green, and Yunqi Sun, "The Decoupled Simulation Model for Virtual Reality Systems," Human Factors in Computing System, Proceedings SIGCHI '92, ACM Press, New York, 1992: 321-328.

Snyder93 Snyder, Mark I., ObjectSim - A Reusable Object Oriented DIS Visual Simulation. MS Thesis, AFIT/GCS/ENG/93D-20, Air Force Institute of Technology, Air University, December 1993.

Stytz96 Stytz, Martin, R., "Distributed Virtual Environments," IEEE Computer Graphics and Applications, May-June 1996: 19-31.

Stytz97	Stytz, Martin, R., Terry Adams, Brian Garcia, Steven M. Sheasby, and Brian Zurita, "Rapid Prototyping for Distributed Virtual Environments," <u>IEEE Software</u>, September-October 1997: 83-92.
Williams96	Williams, Gary, <u>Solar System Modeler: A Distributed, Virtual Environment for Space Visualization and GPS Navigation</u>. MS Thesis, AFIT/GCS/ENG/96D-29, Air Force Institute of Technology, Air University, December 1996.
Wilson93	Wilson, Kirk G., <u>Synthetic Battle Bridge: Information Visualization and User Interface Design Applications in a Large Virtual Reality Environment</u>. MS Thesis, AFIT/GCS/ENG/93D-26, Air Force Institute of Technology, Air University, December 1993.
Zeltzer92	Zeltzer, David, "Autonomy, Interaction, and Presence," <u>Presence: Teleoperators and Virtual Environments</u>, Vol.1, No. 1, Winter 1992: 127-132.

VITA

Scott A. Rothermel was born on [redacted] in [redacted]. He graduated from South Dakota State University on May 4, 1996 with a Bachelor of Science in Computer Science, was ROTC Detachment 780's Distinguished Graduate, and was commissioned as a Second Lieutenant in the United States Air Force. On August 18, 1996 he accepted a commission into the Regular Air Force. Immediately following graduation, he entered the Air Force Institute of Technology in pursuit of a Master's degree in Computer Science. Upon completion of his Master's degree he is headed for Melbourne, Florida as the Lead Test Engineer for JSTARS software, communication, and data testing.

Forwarding Address:

[redacted]

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

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

4. TITLE AND SUBTITLE Architecture, Design, and Implementation of a Rapidly Prototyped Virtual Environment for a Military Spaceplane	5. FUNDING NUMBERS
6. AUTHOR(S) Scott A. Rothermel, Second Lieutenant, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P Street Wright-Patterson AFB, OH 45433-7126	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/97D-17
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr. Jerry Gibson Phillips Laboratory/VTS 3550 Aberdeen Ave SE Kirtland AFB, NM 87117	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)*

The new Global Engagement vision places increased emphasis on the Air Force's ability to control and exploit space. A military spaceplane combining reliable access to space, high operational tempos, and multi-mission capabilities is in conceptual stages of development. Virtual environment technology provides an opportunity to investigate system requirements and unconventional interface paradigms for this unique vehicle.

A virtual environment architecture and design based on support for a rapid prototyping development process, separation of concerns, and user interface development is presented. The rapid prototyping process allowed management of changing requirements via an evolutionary approach to implementation. Separation of the activites performed by the virtual environment into classes enabled high performance through computational distribution, prevented modifications from rippling through the system and impeding development, and promoted reuse of computation and geometric models. A technique was developed to reduce the flimmer induced by the large spatial extent of the virtual environment.

The architecture succeeded in providing a flexible framework for the AFIT Virtual Spaceplane. The Virtual Spaceplane is a large scale virtual environment within which an immersed user commands a military spaceplane through atmospheric and orbital regimes to complete several simulated missions via an unconventional virtual interface.

14. SUBJECT TERMS Simulation, Virtual Environments, Military Spaceplane	15. NUMBER OF PAGES 139
	16. PRICE CODE

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