INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

University of Alberta

MRObjects: An Object-Oriented Framework for Virtual Reality Applications

by

Yanping Liu 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science.**

Department of Computing Science

Edmonton, Alberta
Spring 1999

Abstract

Virtual Reality (VR) technology produces the sense of presence to allow people to experience environments that are not normally experienced. Because of this characteristic VR technology has been applied to various areas such as training, education, medicine, and design and prototyping and showed usefulness and potential benefits.

The development of VR applications, which includes modeling the virtual environment, handling various devices, providing proper user interaction techniques, and rendering the environment with adequate update rates, however, is very time-consuming and needs considerable effort. It is not efficient and practical to build the applications from scratch. High-level software support is crucial.

This thesis implements an object-oriented framework and toolkit, MRObjects, to provide high-level support for VR application development. As a framework MRObjects captures the general structure of most VR applications to relieve individual developers from regular routines. High-level geometric modeling, behavior modeling, and a rich set of interaction techniques that meet the requirements of most VR applications are the principal aspects that MRObjects focuses on. The work of this thesis concentrates on building the application framework and providing high-level geometric modeling support. One of the primary goals of MRObjects is to facilitate the creation of portable VR applications. This goal is achieved by a two-tier organization of the system which is composed of a platform-independent application interface and its underlying platform-dependent implementation. The version of MRObjects implemented in this thesis is for SGI workstations with both the GL and OpenGL supported.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Mark Green, for his continual technical and financial support, guidance, and encouragement throughout this thesis work.

My sincere appreciation is extended to Dr. Ben Watson and Dr. Robert Grant for being in my examination committee and providing valuable comments.

Special thanks go to Lloyd White for his continuous assistance coupled with enthusiasm and patients. His endeavor in making the Computer Graphics Lab a convenient and helpful working environment is highly appreciated.

Sincere thanks go to David Falkner for sharing his ideas and knowledge during the implementation of this thesis.

I would like to thank each of the members of the Computer Graphics Group, in particular Ping Yuan for providing information on the Athabasca Hall walk-through project, Hongzhi Wang for sharing his ideas in handling tracker device, and Pablo Figueroa for his enthusiasm in reading this thesis and providing very helpful suggestions.

Gratefulness is also extended to Dave Clyburn (Director, the Academic Support Centre/Effective Writing Resources) for reviewing this thesis and providing great editorial suggestions making this thesis a more polished one.

Finally, I would like to thank my parents, my husband, and my daughter for their understanding, encouragement, and lasting support.

Without all these support this work could never be done.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Virtual Reality (VR) is an advanced human-computer interface that presents users with a computer-generated environment that simulates the real world and allows users to enter this world and interact with it. An important aspect of VR technology is that it enables users to actively participate in a virtual world – to move around and manipulate the objects in it – and not just operate a computer, as with a general interactive computer graphics system. VR technology is very exciting and promising because it provides people with the opportunity to experience an environment that is not normally experienced.

Virtual reality has been around for years, but it is still a young and emerging technology. Hardware performance is not good enough to make the simulated environment seem real, and much research on VR technology needs to be done, including gaining a better understanding of human interface problems, and developing better software models for VR systems. Because of its potential benefits, for the past decade or so, virtual reality has attracted much interest in many different application areas, such as training, education, medicine, design testing, visualization, and entertainment.

Flight Simulators were the earliest and perhaps the most convincing examples of VR technology. They provide the most cost-effective method for training pilots and testing new aircraft designs. Operating flight simulators, the trainees can experience aircraft flying, develop new skills in handling aircraft under unusual operating con-

ditions, and discover the flight characteristics of new aircraft without endangering their lives and damaging valuable aircraft or other property. Traditional flight simulators use real cockpits and special equipment, with the outside view being computer-generated. More recently, virtual reality technology has been employed in flight simulators that train pilots in a lab setting with VR devices that aim at generating visual, auditory, and haptic realism.

A closely related area to training is education. The best way to learn is by experience, actually seeing, hearing, and doing. Virtual reality makes learning by experience possible. The virtual physics laboratory at the University of Houston [17] helps students to grasp important and difficult physics concepts. Students can control the laboratory environment as well as the physical properties of objects to visualize physical phenomena, and manipulate virtual objects to observe their behavior. Trajectories can be traced, and time can be frozen or run backwards so that students can observe time-based phenomena. Any field that deals with three dimensions or complex phenomena has a great potential for applying virtual reality.

In medical and health-related areas virtual reality technology is beginning to find its place. Virtual medical equipment and virtual patients give practitioners and students more opportunities for "hands-on" experience. High-risk and unusual surgical cases can be practiced any number of times without risking patients' lives. With virtual humans, medical students can study anatomical structures without any limitations on the availability of cadavers. Medical research at the cell and gene levels can be facilitated in a virtual environment through exploration and intervention.

While CAD/CAM frees designers from paper and pencil and provides more productive support and freedom in design, it still operates in two dimensions and requires the designers to translate their 3D ideas into a series of 2D operations and construct a 3D object from a 2D blueprint. Virtual reality has opened the door for designers to a 3D space and let them freely explore, analyze, and manipulate their designs, without the barriers of a 2D screen or paper. Virtual design systems can eliminate the costly construction of physical mock-ups and, more importantly, greatly speed up the prototyping process. Caterpillar Inc. has been working with the University

of Illinois, Urbana-Champaign, on a virtual environment for testing the viability of different tractor designs [3]. William Ribarsky, et al., at Georgia Technology Institute perceived some limitations of conventional CAD and animation systems in design and construction and tried to solve the problem by means of virtual reality [18].

Virtual architectural walk-throughs provide an "in place" experience for designers and also clients to evaluate architectural and interior designs before construction. Matsushita's kitchen design project [27] is a success in virtual reality applications. Customers are placed in a virtual kitchen which is assembled based on their preferences, and they can open a cupboard, look around, and decide whether they really like the kitchen before they buy it. People can not get the same information from looking at a picture as they can from actually being in the environment. Further, they may not only just visualize the design but also interact with the environment and change the design in real time.

Entertainment is a major source for virtual reality applications. The defense and aerospace industries invented VR, but the entertainment industry is introducing it to the rest of the world [21]. A three dimensional game in which the players can "actually participate" is much more fascinating than a two dimensional computer game. Examples are Autodesk's virtual racquetball and W industries's Virtuality [2].

Whatever the situations and requirements, the goal of the applications is always generating a virtual environment that simulates the real world as accurately as possible so that the users can have the feeling of actually "being there"; they can perceive it, experience it, understand it, interact with it, and perhaps even change it just as they can in real life. This goal is by no means easy to achieve. The fundamental work involved in building the virtual environments includes geometric and behavior modeling, input and output device handling, and real-time graphics rendering. For moving around in the virtual environments, various navigation metaphors must be provided. Adequate interaction techniques are important for users to interact with the virtual environments. Model segmentation which partitions the virtual worlds into smaller divisions may be necessary for fairly huge and complex environments. For distributing computation or multiuser cooperation in distributed virtual envi-

ronments, network support is necessary. It is impractical or almost impossible to do all these things from scratch in the development of a VR application. Even an application in the concept-showing stage requires a great amount of time and effort.

To facilitate VR application development, software support at various levels is necessary. Low-level device accessing software is essential. With this level of support, applications can be provided with some standard input/output interfaces for various VR devices, and the developers need not worry about the underlying hardware details. The virtual environments of real VR applications tend to be complicated. Support for modeling the geometric properties and behaviors of objects in the environments can greatly simplify the developers' work and speed up the development process. User interaction, networked simulation, and all other general technologies that are needed for VR application development should also be supported by various software and tools other than being explored by individual developers.

The objective of this thesis is to develop high-level support for VR application development. We will develop an object-oriented toolkit, MRObjects, which is built on top of the existing MR Toolkit [9], a set of software packages which provide lower-level support for the development of VR applications (a more detailed discussion of this toolkit is found in Section 2.3). MRObjects will provide VR application developers with an object-oriented interface, facilitating object geometric modeling and behavior modeling. It will also provide an application framework that helps the developers in constructing their applications. An important part of the system is to explore proper interaction schemes to meet various application requirements.

A problem in VR application development is that the application is generally platform-dependent. MRObjects is designed to be platform-independent. Normally, developers with MRObjects model their applications at a higher object level and don't need to conform to any special graphics packages or platforms. As a result, the applications are portable among different platforms. The high-level abstraction also makes it possible for people from application areas outside of computer science to do application development.

1.2 Outline of Thesis

In the next chapter, after a closer look is taken at some typical VR applications, the general requirements for VR applications are analyzed, followed by an overview of the previous work on software tools for supporting VR application development. In Chapter 3, we give an overview of MRObjects and discuss its system structure and design. Chapter 4 presents a more detailed description of MRObjects. Following the class hierarchy, we focus on the properties that are implemented in this thesis and the functionalities and public interfaces of all the classes, along with some implementation issues. In Chapter 5, we describe how MRObjects works. We use two real projects as examples to explain how to develop VR applications with MRObjects. Finally, in Chapter 6, we conclude and discuss the need for future work.

Chapter 2

Background

Virtual reality has been applied to a wide range of areas. In this chapter, we first look at some typical VR applications. We then try to generalize the scheme for developing these applications and their general requirements for both hardware and software. Finally, previous work on supporting VR application development is discussed and the work of this thesis is proposed.

2.1 VR Applications

2.1.1 Training

Training is probably the most promising area for VR applications. Training in some skills can be expensive, impractical, or even dangerous. VR can be used in various training programs to let the trainees gain the required skills and experience situations before they ever face real ones.

It is well understood that benefits can be obtained using a VR system to train pilots. Training applications in similarly complex tasks, such as training astronauts for manoeuvring during space walks [27] and training the officers of submarines [28], have also been investigated by VR researchers.

Researchers at the Naval Research Laboratory built a virtual environment [25] to train Navy Shipboard firefighters. With the virtual environment system, the firefighters can familiarize themselves with an unfamiliar part of the ship, practice firefighting procedures and test firefighting tactics and strategies without risking lives or property.

[16] reported an immersive virtual environment training model constructed for NASA's Hubble Space Telescope (HST) repair mission. This was the first large-scale implementation of Virtual Environment (VE) technology for training personnel for an actual space mission. The training system provides accurate and realistic models of the real objects in the mission environment, including the HST, the space shuttle cargo bay, and the maintenance/replacement hardware required for completing the major procedural steps associated with the planned extravehicular activities. Object behaviors and operations of objects were also portrayed in the virtual environment to the extent necessary to complete the planned goals.

In the training process, a primary mission goal or task was fulfilled by a series of specific interactions between the trainee and the virtual environment. The trainees were actively immersed in the environment through a head-mounted display. They could maneuver with six degrees of freedom by using two joysticks, one in each hand, and wore a VPL DataGlove on their right hand. Both the head-mounted display and the glove carried a Polhemus Fastrak sensor.

The developers used NASA-developed software, Solid Surface Modeler and Tree Display Manager, as supporting tools to model objects and determine their interrelationships. They used a SoundTool utility for audio feedback in the training.

More than 100 ground-support flight team members actively trained in the virtual environment. The training experience had a positive effect on the HST repair mission and has broadened and deepened NASA's interest in the use of VE technology as a training tool.

2.1.2 Education

Researchers at the Virtual Environment Technology Laboratory (VETL) have been working on building a virtual laboratory for science education [17]. In Project ScienceSpace, they have built three virtual environments [31]: NewtonWorld, MaxwellWorld, and PaulingWorld. In NewtonWorld, students can explore Newton's Laws of Motion as well as the conservation of both kinetic energy and linear momentum. Students interact with NewtonWorld using a "virtual hand" and a menu system. They

can control the mass and velocity of the objects (two balls) and watch their movement and collisions from several viewpoints. Multi-sensory cues – visual, auditory, and haptic – are employed to help students experience phenomena. MaxwellWorld has been designed to enable the examination of the nature of electrostatic forces and fields, to aid students in understanding the concept of electric flux, and to help them empirically "discover" Gauss's Law. Students can interactively place both positive and negative charges of various relative magnitudes into the world and view the force on a positive test charge, electric field lines, potentials, surfaces of equipotential, and lines of electric flux through surfaces. An electric field line can be "grabbed" by a student and relocated. PaulingWorld allows one to examine the structure of both small and large molecules from any viewpoint and in a number of single or mixed representations.

The virtual environments use a head-mounted display, a three-dimensional acoustic environment, and a hand gesture input system. They used software tools including NASA's Solid System Modeler and VRTool by LinCom to build their applications.

Don Allison, et al., at Georgia Institute of Technology developed an immersive virtual gorilla environment [1]. The purpose of building such an environment is to help middle school students learn about gorillas' interactions, vocalizations, social structures, and habitat. In this virtual gorilla exhibit, the students can take on the role of a juvenile gorilla, actually enter the gorilla habitat, walk round, and interact with other artificial gorillas in the environment. The virtual gorillas can show specific reactions to students' behaviors. All the reactions were carefully modeled from observations on gorilla real life that would occur in real scenarios. Through such interactions, students can learn about gorilla behaviors, interactions, and group hierarchies. The knowledge they obtain may not be derived through normal educational means, and their first-hand experience is almost impossible to get in the real world. Besides, the students enjoy the virtual experience.

The authors used their Simple Virtual Environment (SVE) toolkit as support software to build the system. Besides constructing the gorilla habitat model and gorilla geometry, great efforts were made to accurately simulate gorilla behavior. The

system used a head-mounted display to produce the sense of immersion and a hand-held joystick as an input device to control the navigation. Gorilla vocalization and audio feedback were employed in the system.

2.1.3 Visualization

Virtual reality can be an effective tool for visualization. Providing 3D space navigation and other appropriate controls, it can facilitate people's understanding of huge data sets, complex systems, and abstract concepts.

Researchers at the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago explored making virtual reality an effective tool for visualization with their virtual environment, the CAVE, in several applications [6]. One of the applications was to visualize a job execution on an Intel Touchstone DELTA parallel supercomputer with 128 processors. The goal of this application was to help scientists develop efficient algorithms for coordinating jobs on parallel computers. The application involved visualization of the 128 processor array, inter-processor message passing, and the processors' states (e.g., whether they are free to continue processing or idle, waiting for other processors to meet their request). Through the visualization, the scientists can observe and study message traffic, processor throughput, and processor utilization. This gives the scientists a better understanding of the job execution process and helps them determine the efficiency of different job execution algorithms.

Future work involves employing interactive methods to allow the scientists to control job execution, for example, stepping through the time segments of job execution.

Stytz, et al., at the Air Force Institute of Technology developed a solar system modeler [23], an immersive virtual environment that allows its users to visualize celestial objects in near-earth, deep-space, and interplanetary orbits. The size, complexity, and variety of the solar system make it difficult for people to comprehend the motion of heavenly bodies and other physical phenomena in the outer space. Virtual environments of the solar system that are based on orbital motion information and accurately portray each body's appearance will be an illuminating way for people to study planetary motions.

The Solar System Modeler provides a 3D GUI for immersive operation; it graphically models all celestial bodies throughout the solar system in 3D and accurately portrays their locations and orbital behavior. The system enables users to interact with satellites, planets, and moons and provides users with a variety of information and control facilities that assist the users in comprehending the state of the virtual environment.

The system assumes the use of a head-mounted display or a large screen surround display for visual output. A Polhemus 3Space Fastrak tracking system is used to control the user's viewpoint. A mouse and keyboard are used for the user's interactive control over the system. A mouse and keyboard are used as input devices simply because precise, adequate control could not be obtained from other 3D devices such as spaceballs, VPL Datagloves, or joysticks. The system uses Performer, ObjectSim, and their Object Manager network management facilities as supporting software.

2.1.4 Design and Prototyping

Virtual reality technology has brought a new paradigm to computer-aided design and prototyping. Virtual design and prototyping enable designers and engineers to design, visualize, and manipulate the end product and test its characteristics in a three dimensional space, eliminating the limitations of flat 2D displays. Design flaws are more likely to be found and avoided before manufacturing, and thus valuable marketing time can be saved.

For example, Boeing's virtual reality project planned to import aircraft CAD data to a VR environment representing a detailed model of the aircraft, allowing engineers in helmet-mounted displays to actively explore the aircraft long before any manufacturing begins [21].

In addition, researchers at the NCSA (National Center for Supercomputing Applications) in the US and GMD (Germany's National Research Center for Information Technology) developed a collaborative virtual prototyping system for vehicle design for Caterpillar Inc. [14]. The system supports collaborative design review and interactive redesign. The designers work either in the CAVE Automatic Virtual Environment

(CAVE) or with the Responsive Workbench. The input to the system includes head and hand tracking, with head tracking controlling the operator's viewpoint. The system provides designers with two schemes during the design process. They can either drive the virtual vehicle around the virtual world or fly around the vehicle using a 3D mouse, a wand, which has a tracker, three buttons, and a joystick to examine the design. The system supports cooperations among geographically distributed sites. Any number of engineers can participate in a design process with a shared virtual environment. With real-time video and audio transmissions being integrated into the system, the participants can exchange design information and communicate with each other naturally.

The system was built on an existing virtual prototyping system. It used the dynamic simulation package Dynasty for real-time design testing, Alias for environment modeling, and Sense8's WorldToolKit for rendering. It also made use of some other tools for graphics rendering, communication, audio and video transmission.

It is claimed that Caterpillar has used the virtual prototyping system in several projects at NCSA.

2.1.5 Medicine

The increasing usage of computers in Medicine has changed the way that health care is delivered [4]. On-line patient databases, remote consultation, digital radiography, and expert systems have become routine facilities. The introduction of VR technology to Medicine will make a even deeper change in medical services and revolutionize the way that doctors diagnose diseases, new surgeons practice surgical skills, and students get training.

Chua Gim Guan, et al., at Institute of Systems Science (ISS) at National University of Singapore developed a volume-based pre-operative surgical planning system, VIVIAN [11]. The system's interface allows the user to "reach in" to a 3D display where hand-eye coordination allows careful, dextrous work with 3D objects. The users use both hands to interact with the application with one hand attached to the patient's volume complex, turning and placing it as one would do with an object

held in real space and the other performs detailed manipulation, such as selecting an object and changing operating modes, through the surgical planning tools. VIVIAN features a user crop-and-clone facility to carve out volumetric objects from the MRI (Magnetic Resonance Imaging) volume to provide a clear view of the pathology and surrounding tissue. It also uses markers for measuring distances between points in a 3D space and drawing lines that guide approaches.

VIVIAN has been used by the neurosurgeons authoring the paper to plan operations on patients having brain tumors in various locations. The neurosurgeons believe that VIVIAN provides the most efficient and comprehensive way in which they understand the complexity of anatomical and pathological relationships surrounding the lesion. They all agreed that the pre-operative experience of planning the approach virtually remains in the neurosurgeon's mind and is supportive during the operative procedure.

The system is written with the ISS BrixMed C++/OpenGL software toolkit and runs on Silicon Graphics Onyx workstations, relying heavily on hardware-assisted texture mapping. It uses a special self-developed 3D interface, the Virtual Workbench. The input devices used are FASTRAK trackers from Polhemus.

Grigore Burdea and other researchers from Rutgers-The State University of New Jersey and the University of Medicine and Dentistry of New Jersey proposed using a virtual reality digital rectal examination (DRE) simulation to aid DRE training in detecting prostate cancer [5]. Current DRE training requires medical students to examine a large number of patients before attaining adequate experience. A simulation system could remove the difficulty of finding patients willing to allow medical students to train on them and other limitations.

Their prototype system consists of a PHANToM haptic interface which provides force feedback to the trainee's index finger, a motion restricting board, and an SGI workstation. OpenGL and GHOST haptic library are used for prostate modeling.

The system models four types of prostate, each type having randomized tumor locations. The trainee can practice any of the provided cases. While the trainee is palpating the prostate, the prostate model deforms and forces are fed back by the

PHANToM, corresponding to his/her finger's movement and location in the virtual prostate model. When being tested, the trainee has to diagnose the case presented to him/her without seeing the prostate on the screen. The trainee's diagnosis responses and actions can be recorded for later analysis.

It was believed that the system showed eventual usefulness as a teaching aid, provided realism in the modeling was increased. Better hardware that can provide accurate force feedback and position sampling is crucial for achieving a more realistic haptic simulation.

2.1.6 Discussion

A distinct characteristic of VR, and also a key benefit that can be obtained from VR applications, is the participation experience. A flight simulator provides the trainee the feeling of flying a real aircraft in a real situation; a medical student operating on a virtual human is able to actually practice surgery; a house-buyer can obtain a clear idea of what a house's interior design (room separation, furniture placement, light sources, etc.) is by entering a virtual house with certain designs; and being immersed in a virtual environment, a game player feels like playing with "real characters" rather than computers. Although VR applications are quite diverse and every application has its unique problems to address, all application systems attempt to produce a strong sense of presence to make the participants feel that they are actually operating in a real world and interacting with real objects. However, the current VR technology is far from mature in both hardware and software for achieving the sense of presence. The applications described here are mostly at the research stage. For industry and other practical uses there is still a long way to go to achieve what virtual reality promises. Yet the potential usefulness and benefits of VR technology have been perceived and have inspired the researchers in VR and other related fields to strive all the way to reach their goal.

2.2 VR Application Requirements

The central task of all VR applications is to build a responsive virtual environment to present the participants with some proposed reality. This task makes great demands on both hardware and software resources. From an application development point of view, all applications have to do the following:

- Modeling the geometric objects in the virtual scene
- Defining and simulating object behavior
- Handling multiple input devices
- Providing adequate user interaction
- Rendering the environment with adequate update rates

The essential work in creating a VR application is to design and build the virtual scene. This work normally includes geometric modeling and behavior modeling. Most VR applications have very large and complex scenes, and building them from the polygon-level is very time consuming. Applications need support for higher-level geometric modeling and environment design, providing adequate structure to organize the objects into the virtual world.

Objects in the virtual environment generally have some type of behaviors, either predefined motion or reaction to user interaction or other objects' effects. Accurately simulating the objects' behaviors is essential to the success of VR applications. Behavior simulation is fairly difficult work and thus must be supported by software packages. Although high-level objects' behaviors are application-dependent, some common behaviors exist among objects. For example, all physical objects have attributes such as mass and weight and follow some rules and constraints such as Newtonian laws. Applications in the same or similar areas tend to have many commonalities. Behaviors common to most applications, means for defining a behavior, and behavior-related collision detection algorithms are better supported by some general tools rather than being left to the applications.

Most VR applications need interactive abilities which rely on various input devices. Compared with conventional desktop interactive systems which use a keyboard and 2D mouse, VR systems use a much wider range of input devices, most of them being three dimensional devices with six degrees of freedom. These 3D devices have the potential of providing direct and intuitive ways of manipulating 3D objects. Multiple devices can be used at the same time. Head tracking determines where the user is looking and is used to determine the user's view. Hand-tracking can show what the user is pointing at and can be used to pick objects. Gesture and/or speech recognition systems can be used to demonstrate what the user would like to do (e.g., grab an object). But handling all sorts of input devices needs a considerable amount of work and can distract the developers from their real task. In addition, doing so requires that the developers have special knowledge of how to deal with the devices. It is obvious that providing a high-level interface for device access is very important to free the application developers from this tedious and hard work.

Interaction is a key property of VR applications. Since users of VR systems operate in a 3D virtual space, they need to use direct and intuitive ways to interact with the virtual world. A wider range of 3D devices provides more options and support for developing interaction techniques that are suited to the 3D world. However, it is not desirable to have application developers explore various interaction schemes. Potentially, a standard set of interaction tasks that are useful for most applications such as object pointing and selection, object grabbing, object dropping, setting system parameters and selecting commands, and changing the properties of objects exist. Navigation is a key aspect of interaction and is especially important in 3D space. Some navigation schemes such as walk-through and look around are common to most applications. Support for the aforementioned and some other generic interaction metaphors that let the users operate and interact with the virtual world is required and highly prefered.

To make the virtual environment responsive, rapid frame rates and fast response to interactive user control are crucial. With limited hardware performance, efforts on the software side are very important. Some mechanisms, such as culling and level-of-

detail switching for reducing the complexity of the virtual scene and multiprocessing for obtaining high system performance, have been explored to meet the real-time requirement of VR systems. These mechanisms are not application-specific and can be supported as underlying tools for VR applications.

Support for multi-sensory cues, such as audio and haptic, as well as visual effects which are necessary in achieving the sense of presence, are also desired.

The complexity and huge amount of work needed for VR application development make it very demanding on software support. The common problems that all VR applications face also make it possible to provide various levels of software tools for VR applications. The goal of these software tools and support is to remove as much of the burden as possible from application developers and let them concentrate on their application-specific work. Researchers in the VR and computer graphics community have made great efforts in achieving this goal. In the following section, previous work on software support for VR applications is reviewed and discussed.

2.3 Previous Work

Paul S. Strauss, Rikk Carey, et al. at Silicon Graphics Computer Systems developed an object-oriented 3D graphics toolkit [22] for interactive 3D graphics applications. The toolkit provides a general and extensible framework for representing 3D scenes. It defines a set of object-level geometries (e.g., Cone, Cylinder, and FaceSet) to facilitate object representation. Geometries, properties, operations, and actions are all implemented as objects called nodes. Applications build their 3D scenes with these nodes organized into scene graphs (scene database). The toolkit implements an event model to enable direct interaction with 3D objects. A user input event is captured at the application level and distributed through the scene graph until it reaches the node which will handle it. Surrogate objects are employed to give visual cues for different interactions. The toolkit also defines a means, node kits, to help applications create structured, consistent databases.

This toolkit provides support mainly in object representation and scene modeling.

It is desktop-oriented and employs only 2D input devices – a keyboard and mouse. 3D space interaction techniques were not explored.

IRIS Performer [19] is a toolkit for real-time 3D graphics applications. It provides support for gaining maximal performance from 3D graphics workstations with multiple CPUs and relieves application developers from these difficult issues. The toolkit combines a low-level library for high-performance rendering with a high-level library that implements pipelined, parallel traversals of a hierarchical scene graph. Developers can employ multiprocessing in their applications without worrying about work partition among processes, process synchronization, and data sharing and communication. The toolkit supports culling to the viewing frustum, level-of-detail switching, and intersection testing for achieving proper frame rates. Some typical graphics and database operations such as multiple views, morphing, picking and run-time profiling are also provided.

Performer especially focuses on performance issues based on SGI graphics workstations. It does not include direct support on I/O devices, and issues of object behavior modeling and interaction are not addressed.

Researchers at SunSoft Inc. developed their high-level framework, TBAG, for interactive and animated 3D graphics applications [7]. TBAG provides a set of high level graphical types (e.g., points, vectors, colors, transforms) and constants of these types to facilitate geometry modeling. It provides interactive animation support by the constrainable entity which encodes desired animation and interaction (e.g., mouse motion, 3D tracker data) with objects that have time-varying values (e.g., geometric position). Among constrainables relationships can be established by setting up constraints. With constrainables applications need not handle input device polling and events for updating the animation parameters. TBAG implemented a set of manipulators for common interaction paradigms (e.g., selection with buttons). It also provides support for developing distributed and collaborative applications. TBAG is mainly concerned with support for animation applications. Navigation schemes which are important for VR applications are not explored.

Larry F. Hodges, et al. at Graphics, Visualization, and Usability Center of

Georgia Institute of Technology have developed the Simple Virtual Environment (SVE) library that provides mechanisms and software tools for virtual reality applications [30]. The SVE system operates as a single process which resembles the standard event loop structure of user interface applications. It handles some typical VR devices such as trackers, gloves, and head mounted displays. 3D interaction widgets are supported for user interactions and control of the application. Animation is supported by executing application-defined animation callback functions after event handling and before graphics rendering. The system performs view culling and switching of object graphics representations for rendering. Functions for dealing with audio are provided.

The SVE system provides support for various aspects of VR applications such as user interactions, animations, rendering, and input device polling. However, the support is at a relatively low level. Basically, application developers have to program the logic they need using the functions provided. The SVE system is still under development.

Researchers of initially User Interface Group at the University of Virginia [26] and now Stage 3 Research Group at Carnegie Mellon University, built Alice [29], a 3D scripting and prototyping environment for interactive 3D graphics. Alice models the world as a hierarchical collection of objects. Each Alice object has a list of action routines (callbacks) to implement animation. The action routines can be executed either once per frame or time-based. Alice employs a multiple process architecture to separate simulation from rendering to accomplish proper rendering frame rate. The multi-process architecture is transparent to the programmers.

Alice was originally developed on UNIX platform and is now ported to Windows 95/98/NT as a desktop system. It is designed to be simple enough for novices to develop interesting 3D environments and to explore the new medium of interactive 3D graphics. Alice users compose a virtual environment with pre-made objects and control the objects' appearance and behavior by writing scripts with Python, a relatively easy scripting language. Alice supports many common 3D file formats including .DXF and .OBJ to assist importing object models from outside Alice.

Mark Green, Chris Shaw and other researchers at the University of Alberta have built a toolkit, the Minimal Reality (MR) Toolkit, for supporting the development of VR user interfaces and other forms of three dimensional user interfaces [9][20]. The MR Toolkit consists of a collection of packages and tools, each package handling one aspect of VR user interfaces. The MR Toolkit supports a fairly wide selection of typical VR input and output devices, such as 3D trackers, gloves, and head-mounted displays. Its Data Sharing Package provides a higher-level interface for data sharing between two processes, supporting the decoupling of computation and graphics rendering. Workspace Mapping handles coordinate transformations among different coordinate systems (e.g., the device coordinate system, room coordinate system, and virtual world coordinate system) and the programmers need to deal only with their virtual environment coordinate system. This removes some potential confusion and extra work. The MR Toolkit also provides a set of standard 2D and 3D interaction techniques, such as a 2D panel in 3D space which is controlled by 3D devices. Communications and interactions among multiple applications running at different sites are supported by the MR peer mechanism. This allows multiple users to share the same 3D environment. Real-time performance analysis tools and sound as signaling events are also included in the toolkit.

The MR Toolkit provides the facilities that are most needed by VR applications and thus can greatly ease the work of producing VR applications. However, it provides relatively low-level services for the construction of VR applications. High-level support, such as virtual environment design and geometric modeling, is not provided. Programmers developing VR applications with the MR Toolkit have to use a certain graphics package (e.g., GL or Starbase) to build the virtual world from the polygon primitive level, and the resulting application is generally not portable because of the graphics package's dependency on a particular platform. Also they still need to know about device initialization (at a high level), arranging separate processes for computation or stereo image display, and synchronizing data structures and display operations between processes. 3D interaction and navigation techniques are not fully explored.

2.4 Work of This Thesis

The MR Toolkit has been licensed by more than 450 sites all over the world. Its success in providing the generally required support for VR application development as well as its limitations has created the need for its extension and further development.

The work of this thesis is to build MRObjects, an object-oriented toolkit, that sits on top of the MR Toolkit and provides a higher-level interface for VR application development.

MRObjects is designed to be platform-independent. The same application source code should run on all the supported platforms, either a PC or a workstation. This application portability can be achieved by producing a different implementation for each supported platform under the portable interface. MRObjects provides high-level geometric modeling and obviates the need for using a particular graphics package. This not only simplifies the work of building the virtual environment, but also makes the applications more portable. MRObjects supports a variety of input and output devices and full 3D interaction techniques which are demanded by most VR applications; they are also the areas in which most of the existing 3D graphics toolkits fall short. Building on top of the MR Toolkit, most of the MRObjects' support for various VR devices is automatic. MRObjects will explore a rich set of natural 3D interaction and navigation techniques, giving application designers more freedom in constructing proper interaction schemes.

MRObjects also provides a framework for VR applications. With MRObjects, application developers can concentrate on their application-specific issues and need not worry about the details of interacting with the required input and output devices and things like initialization, configuration and simulation loops. All they need to do is to build their virtual environment and put the environment into execution, MRObjects takes care of everything else.

In the next chapter, we will discuss the basic concepts behind MRObjects and its system design.

Chapter 3

MRObjects – System Design

In this chapter, we presents the overall structure and system design of MRObjects. We discuss the classes that constitute MRObjects at a level that represents the ideas behind MRObjects. Details about the classes' functionalities, implementation, and user interface are described in Chapter 4.

3.1 Introduction

As discussed in Chapter 2, VR applications are highly demanding of system resources and need significant amounts of time and effort to create. In practice, the development of VR applications depends on various levels of software support.

The MR Toolkit, developed at the University of Alberta, consists of a set of software packages that provide the essential low-level services for the construction of VR applications. The programming interface is a set of procedures provided in the packages. With the MR Toolkit, developers still have to be knowledgeable about some system-related issues. The applications have to initialize and configure the MR Toolkit, arrange multiple processes for decoupling simulation and computation or stereo display, declare and calibrate the devices used, and control the simulation cycle. It is prefered that this work be done at a high-level so developers can concentrate on the application-specific issues.

Although VR applications are generally different from one another, they have a lot of common characteristics. With this observation, the MR Toolkit imposes some

typical architectures on the applications. Most MR applications require at least two programs: one, the master program, controls the applications, and the other, the slave or computation program, is responsible for either stereo display or computation. The flow of control of a typical MR application can be roughly outlined as follows:

- Configuring the application
  - Initializing the MR Toolkit
  - Assigning the role of the program
  - Starting the slave/computation program
  - Selecting the devices used
  - Defining the shared data structures
  - Setting up coordinate system (optional)
  - Calibrating some devices
- Running the simulation loop
  - Obtaining the input from user interaction
  - Updating the virtual scene based on the input (this step may be done in a separate process)
  - Rendering the virtual scene

Since the above steps are common to nearly every MR application, there is no reason for each individual programmer to repeat these steps in his or her program. In addition, the similarities shared by MR applications also demonstrate the general requirements of all the VR applications since the MR applications are just VR applications that use the MR Toolkit as their low-level service support.

Capturing the general characteristics and control structure of typical VR applications in a framework can greatly facilitate the development of the applications. With the framework provided, the developers should be able to concentrate on the application-specific issues and not be distracted by general routine operations and

other low-level service details. The goal of MRObjects is to build such a framework which provides high-level support for VR application development. Built on top of the MR Toolkit, MRObjects encapsulates all the low-level and routine operations in a set of related classes that construct the framework and provides the developers with a high-level object-oriented interface. With MRObjects, the developers need not worry about how to construct their applications and how to make use of the underlying support library to control their applications, the framework provides this support automatically. The programmers need to provide only application-unique information for setting up the applications and implement only the application-specific parts.

3.2 Overview

The application framework provided by MRObjects is based on a collection of related classes. Figure 3.1 shows the the main classes that form the architecture of MRObjects.

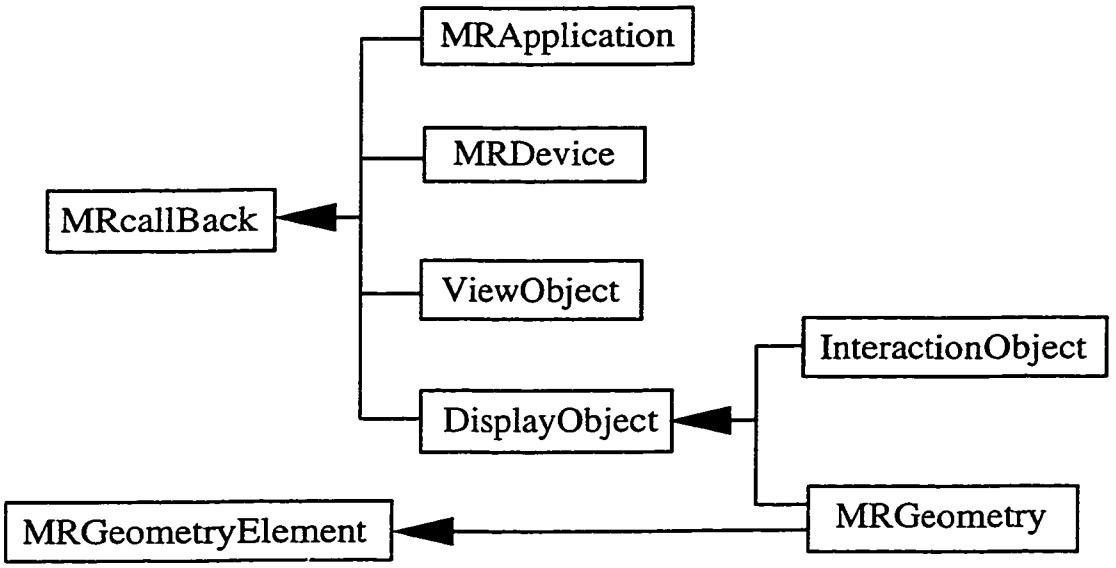

Figure 3.1: The MRObjects class architecture

Three key classes define the structure of the framework and also guide the construction of applications. They are the MRApplication class, the DisplayObject class, and the InteractionObject class.

The MRApplication class provides the interface with the MR Toolkit, handling the initialization of the application and interaction with the input and output devices. The DisplayObject class defines the basic protocol that is used by all the objects that are displayed on the screen. The InteractionObject class defines the basic protocol for all the objects that handle user interaction. All the other classes in MRObjects are built on top of or around these three classes, and simple applications can be constructed using only these three classes.

MRObjects provides object-level scene modeling support. A set of geometrical primitives, including physical shapes (e.g., spheres) and attributes (e.g., material, transformation, and light source), are defined in MRObjects. The representation of the primitives is a high-level notation that does not depend on any particular graphics package. For example, a sphere is described by its radius. The portable high-level geometrical representation is implemented through the MRGeometry class. The MRGeometry class provides the interface for defining geometrical objects in applications. The elements that define an MRGeometry object are modeled by the MRGeometryElement class. The MRGeometry class supports hierarchical geometrical models. Hierarchical objects can be constructed by adding an MRGeometry object to another MRGeometry object, and the depth of the hierarchy of an object is limited only by system resources (e.g., memory).

The MRGeometry objects are manipulatable entities. They have not only graphics representation but also associated behaviors. The basic and default behavior of an MRGeometry object is the output of its graphics. Other behaviors can be encoded based on the requirements of an application.

Building a complex virtual environment with high-level portable geometrical primitives instead of using a particular graphics package at the polygon-level not only improves the efficiency of the developers but also results in portable applications. Portability is one of the main aims of MRObjects.

Interaction techniques are important to VR applications. MRObjects considers various input devices, from high-cost 3D trackers to low-cost spaceballs and joysticks, as well as conventional keyboards and mice. It will provide a set of standard interac-

tion techniques that are required by many typical VR applications. With this level of support the developers need not consider handling various input devices, and in most cases neither do they need to spend their valuable time on implementing some general interaction metaphors. They simply choose what they need from the standard set.

The InteractionObject class acts as an interface to various interaction techniques. Any derived class of InteractionObject implements a particular interaction metaphor. Reactions to the user interaction and the interaction between objects in the application are implemented with a callback mechanism by the MRcallBack class. All the classes that need the callback mechanism are derived from this class.

Navigation is generally considered as a kind of interaction. In MRObjects, however, a separate class, ViewObject, is responsible for viewing and navigating. The ViewObject class provides a general interface. Its derived classes define the specific navigation schemes. Some considerations about why ViewObject is defined separately from the InteractionObject class is discussed in Section 3.3.3.1.

The MRDevice class defines the interface for various devices used in MRObjects.

Various aspects of MRObjects' system design are presented in the following sections. Some properties are not fully implemented in the work of this thesis. A detailed discussion on the classes defined in MRObjects, mainly concerned with the implementation issues, is presented in later chapters.

3.3 The Framework

MRObjects is designed as a high-level, platform-independent framework and also as a toolkit to facilitate VR application development. The main purpose is to free the application developers from low-level device handling and work such as system initialization, simulation control, rendering, and user interactions that are generally required by most applications, allowing the developers to build their applications in a more intuitive and productive way without the requirement of high expertise in system-specific knowledge.

The framework implemented by MRObjects is defined mainly by three core classes:

the MRApplication class, the DisplayObject class, and the InteractionObject class. These classes determine the way in which applications are built.

3.3.1 The Application Interface

The MRApplication class provides the main interface with applications. It handles device and system initialization, manages all the resources and objects, and maintains the state of the application and controls its execution.

For each application, there should be only one instance of the MRApplication class. The programmer needs only to declare the MRApplication object, and this object will take care of the execution of the application. Without any specification, the MRApplication object simply uses its default settings. However, applications tend to have some specific requirements. There are several ways that developers can customize their applications. They can provide command line arguments, or they can call the methods defined in the MRApplication class to configure their applications.

3.3.2 The Object Display Interface

The DisplayObject class defines the basic protocol for object display. It is an abstract class, and its subclasses will define the actual objects to be displayed. All the objects that have graphics output on the screen should be defined by classes that are derived from DisplayObject.

The protocol basically defines two operations: object update and object display. The update mechanism provides an interface for any modification applied to the objects. For example, some objects may change their internal graphics representation with rendering speed consideration in response to the user's viewing position and direction. User interaction and the objects' behavior can change an object's description parameters, such as a sphere's radius or a rotating object's rotation angle. The update mechanism gives these objects a chance to perform these modifications before they are drawn on the screen.

The basic behavior of an object is to display its graphics. DisplayObject provides two approaches to graphics output. The first approach is based on some predefined

subclasses of DisplayObject which provide platform-independent geometry modeling. The MRGeometry class, for example, is such a class which is designed as the interface for defining high-level 3D graphics objects. Applications use these portable 3D objects to model their geometry and need not worry about object rendering. The display of the objects is done automatically by the classes' built-in drawing functions. This approach embodies the ideas of MRObjects and produces portable applications.

The second approach for graphics output lets application developers explicitly define a drawing function in a class derived from DisplayObject. The derived class represents an application-specific graphical object to be displayed. The drawing function is defined using the native graphics package. This approach gives the programmers some flexibility to model their virtual environment. It also provides an easy way to retrofit an existing traditional application with a VR interface. The programmer can quickly prototype a VR application by wrapping the existing code in a DisplayObject object and integrating it into the MRObjects framework. In this way previous work could be saved instead of being simply ignored, which is common when upgrading an application to a new software environment. This approach, however, greatly restricts the portability of the resulting application. And the geometry modeling process is not very efficient. There is another problem as well. The graphical geometries described by the wrapped code can be treated only as a whole, although they may describe a complex scene which is composed of a collection of objects, and the user cannot interact with part of it. The programmer can of course break the code into parts and build them into a hierarchical structure, but this could introduce a lot of extra work. More importantly, doing so actually gives the work of geometry modeling that is supported by the system back to the programmers. In cases of developing new applications, the first portable approach is recommended.

3.3.3 The Interaction Interface

In MRObjects the InteractionObject class is designed as the basic interface for various interaction techniques. The InteractionObject itself is an abstract class and does not implement any interaction scheme. All the interaction schemes provided by MROb-

jects are defined by subclasses of the InteractionObject class. Basically, each subclass suggests one interaction metaphor.

In some previous systems (for example, JDCAD and Thred) the interaction techniques are closely related to the objects in the particular applications. This can greatly restrict the generality of the techniques and their reuse in other applications.

The idea behind MRObjects is to decouple various interaction techniques from particular situations and make them available to all applications. Ideally, a rich set of standard interaction techniques are provided at a high level and independent of the objects interacted with. In most cases the developers simply choose the techniques that are appropriate to their applications and need not worry about the implementation details. However, it is not possible to provide a complete collection of interaction techniques. With the hierarchical object-oriented design, it is easy to extend the standard collection at any point.

The InteractionObject class interacts with the MRApplication class and some other classes (e.g., the MRDevice class). The objects of InteractionObject (including all its derived classes) are managed by the MRApplication object, and an InteractionObject object may declare some MR devices that are used by the interaction technique. InteractionObject responds to new device values (generally as the result of user interactions) through callbacks.

Each InteractionObject class that defines an interaction scheme should define one or more callbacks that process the interaction and pass information to the other objects in the application that are affected by the interaction. These callbacks may be invoked at each step of the interaction or at the end of a successful interaction, depending upon the interaction technique implemented by the class. For example, when an object is dragged from one place to another, the callback for changing the object's position is called whenever the device's position value changes. However, while an item in a menu is selected, the callback for the menu item is called only after the selection is completed. The callback mechanism of MRObjects is discussed in a later section.

Since most InteractionObject objects display some type of information to the user,

this class is a derived class of DisplayObject and supports the DisplayObject protocol as well.

3.3.3.1 Navigation

The ViewObject class was originally designed only as a container for the user's current position and viewing direction as well as the viewing pyramid used to display the objects. Since navigation controls the user's viewpoint movement in a virtual environment, it is natural that the ViewObject class serves as the interface for various navigation techniques. This is the role that ViewObject plays. With this interface, MRObjects can provide some typical navigation metaphors by deriving classes from the ViewObject class. Users can choose a navigation metaphor by declaring an object of a proper ViewObject's subclass that implements the metaphor and pass a pointer of this object to the MRApplication object. There is only one ViewObject object defined in an application. Without explicit declaration, the system provides a default navigation metaphor which depends on the input devices employed in the application.

The ViewObject class is currently designed separately from the InteractionObject class. On one hand, since navigation techniques are part of the interaction techniques, conceptually the ViewObject class should be derived from the InteractionObject class. In addition, ViewObject may also have graphical output. For example, some applications may need a simulated person (avatar) representing the user himself or herself to be present in the scene. This requirement makes it necessary and natural to put the ViewObject class under the InteractionObject class.

On the other hand, since navigation techniques are used to control the user's viewpoint (not objects in the virtual scene) and generally there is only one viewpoint in an application for a single user, there should be only one ViewObject object active (multiple ViewObject objects can be declared, but only one instance is active at any moment. A way of switching among these objects may be defined). This makes the handling of the ViewObject objects different from other interaction objects.

This thesis follows the current design.

3.3.4 Application Structure with MRObjects

The general structure of an MRObjects application is as follows:

1. Declaring the MRApplication object
2. Defining the objects in the virtual world with geometry subclasses of DisplayObject
3. Running the program

This structure actually implies two phases in an MRObjects application. The first two steps comprise the first phase when the application is configured. This includes selecting the devices used in the application, determining the type of displays to be produced, and establishing the initial set of DisplayObject and InteractionObject objects in the application. Operations such as selecting devices or type of display are optional. If the application does not provide information for the selection, MRObjects provides defaults. Once the configuration is completed, the application moves to the second phase where the configured application runs.

The following code is the "Hello, World" equivalent example which draws a coordinate system with its three axes:

```
#include <MRObjects.h>
MRApplication *theMRApp;
main(int argc, char **argv)
{
    /* Step 1: Declare the MRApplication object */
    theMRApp = new MRApplication(''Axes'', &argc, argv);
    /* Step 2: Define the geometry for the axes */
    MRGeometry *axes = new MRGeometry();
    axes = axes
        + MRColour(1.0, 0.0, 0.0)
        + MRLine(0.0, 0.0, 0.0, 1.0, 0.0, 0.0)
        + MRColour(0.0, 1.0, 0.0)
```

```
            + MRLine(0.0, 0.0, 0.0, 0.0, 1.0, 0.0)
            + MRColour(0.0, 0.0, 1.0)
            + MRLine(0.0, 0.0, 0.0, 0.0, 0.0, 1.0)
            ;
        /* Step 3: Run the program */
        theMRApp->run();
}
```

At the start of the program an instance of MRApplication is created and assigned to the global variable, theMRApp. The application may provide some command line arguments for configuration. The MRGeometry class is a subclass of DisplayObject. It is the portable geometrical definition interface (discussed below) for modeling the geometrical objects in the application. MRColour and MRLine are functions that return geometrical elements (color and line, respectively) that compose the geometry.

Although far from practical, this example clearly shows the structure of an MRObjects application and the way to build it.

3.4 Portable Geometry Modeling

One of the main goals of MRObjects is to facilitate the creation of portable VR applications. The application developed with MRObjects on one platform, for example, the PC, should run instantly on any other platforms, such as high performance workstations, without modification.

3.4.1 The Idea

The goal of creating portable applications cannot be achieved by simply selecting an existing graphics package since it is unrealistic to expect a single graphics package to run on all platforms. High graphics performance is often achieved by special purpose graphics packages that are tuned to the special display architectures. There is no way that a single portable package will approach the performance of a specially-tuned platform-dependent package.

Given this reality, MRObjects takes an alternative approach which provides applications with a portable interface while achieving efficient graphics performance on all platforms. The key idea behind this approach is to distinguish between a geometrical and a graphical representation of an object and use the geometry compilation technique. A geometrical representation represents an object's inherent properties including its shape and attributes that determine the object's appearance. A graphical representation of an object is a set of graphics primitives that are used to render the object on a particular workstation in a particular context. The geometrical representation is a high-level, platform-independent notation, each object having only one such representation. The graphical representation, however, is platform-dependent, and an object can have various such representations depending on the context.

Application developers will use the high-level geometrical notation to describe the geometrical objects in their applications. A geometry compiler is then used to convert this geometrical description into graphics primitives using the native graphics package. The high-level notation avoids using low-level graphics primitives and allows the geometry compiler to convert the object into the most efficient set of primitives for the current platform. So an object has two representation models: the high-level geometrical representation for the developers and the low-level graphical primitives for object display.

When an object is displayed, MRObjects first determines whether the object has been compiled. If not, the geometry compiler is invoked to produce the native graphical representation. This compiled representation is stored in the object and used when this object is displayed. Once the object is compiled, the geometry compiler is not called unless the object is changed. Since native graphics primitives are used for displaying objects, the best possible performance can be achieved from the local graphics system.

A possible problem that may arise from this approach is the geometry compilation overhead. For static objects, the geometry compiler is executed only once to produce the native graphics primitives, and the overhead can be ignored. For dynamic objects, an incremental compiler could be used, so only the changed part of the object descrip-

tion needs to be recompiled. As long as the modifications are minor, this geometry compilation approach still results in a more efficient graphics system. Besides, most of the changes that occur in dynamic objects are transformation matrices, and for most graphics packages, their incremental compilation is quite fast and efficient. So there is a good chance that applications using MRObjects will run as fast as applications that use the native graphics package. At the least, MRObjects does not bring serious performance degradation to the applications. In addition, we gain in portability.

In this approach, the portion that deals with the geometrical representation is universal across all the MRObjects implementations, while a new geometry compiler must be produced for each graphics package, and possibly for each workstation. The finer the geometry compiler is tuned, the faster the graphics will run. Any improvement in the geometry compiler will benefit all applications.

3.4.2 The Portable Interface

The distinction between geometrical and graphical object representation helps MRObjects achieve both portability and good performance. This two-level geometry modeling scheme is implemented by the MRGeometry class, along with the MRGeometryElement class.

Derived from both the DisplayObject and MRGeometryElement classes, the MRGeometry class inherits the basic DisplayObject interface and represents a hierarchical geometrical model. The MRGeometry class maintains both the geometrical and graphical representations of objects. The high-level geometrical representation is a list of geometrical primitives, or geometry elements. A geometry element can be a physical shape, an attribute, a transformation, a light source, or another MRGeometry object. All the geometry elements are defined by the MRGeometryElement class and its derived classes. This is discussed further in the following subsection. A detailed description of all the geometry element classes is presented in Section 4.8.

The low-level graphical representation is a set of graphics primitives that result from geometry compilation. Whenever the geometrical representation is changed, MRGeometry calls the geometry compiler to get the new graphical representation.

The developers define the geometry of the objects in their applications by creating MRGeometry objects. Geometry elements are then added to the contents of an MRGeometry object using the '+' operator, which is overloaded in the MRGeometry class and acts as a basic insertion operator. The developers interact only with the high-level geometrical representation. The low-level graphical representation, however, is transparent to the applications. The developers may not even be aware of the existence of the underlying compilation process.

3.4.3 The Geometry Elements

The geometry elements are the building blocks for creating geometrical objects. All the geometry elements are modeled under the MRGeometryElement class, and each derived class of MRGeometryElement defines one kind of geometry element. The geometry elements provide the high-level notation for the geometrical shapes or attributes they represent. For example, a cone shape is defined by class MRConeObject with its base radius and height as the descriptive properties. It does not specify how the cone is to be displayed. Displaying the cone is determined by the geometry compiler.

The general geometry elements are not stand-alone entities and can only be used as elements in building an MRGeometry object. For example, a cone object has to be defined by creating an MRGeometry object with an MRConeObject element as its member. The exception is the MRGeometry itself, which is a special geometry element.

For the sake of applications, each geometry element class has at least one corresponding factory function that creates an instance of the class and returns a reference (or pointer) to the new instance. Generally, several versions of factory functions with different parameters are overloaded for the convenience of writing applications. A factory function call can be used directly as the operand of the '+' operator in defining the MRGeometry objects. Doing so makes it much easier to create geometry elements and build MRGeometry objects.

3.5 MRObjects Devices

User interaction is a basic characteristic of VR applications. The user of an VR application should be able to control his/her viewpoint to have different views of the virtual environment and control the behavior of the objects in the virtual world. User interactions are achieved through the use of various devices. MRObjects abstracts all the physical devices that are supported for user interaction into several types of logical devices based on their functionalities. The logical device types identified by MRObjects are value, trigger, and glove devices. These types are explained in Section 4.6.

Interaction techniques implemented in MRObjects are based on logical devices. They do not assume the use of any particular physical devices. Instead, only logical devices are requested for the suggested interaction techniques. For example, the pointing and selecting technique use a value device which produces both position and orientation data for pointing, and a trigger device which generates a boolean value for selecting. The value device can be a 3D tracker or a joystick, and the trigger device can be a keyboard or a mouse. Since a logical device represents the type of value that it can generate, it simplifies the interface between interaction techniques and devices and supports easy integrating of new physical devices.

A logical device can be implemented by different kinds of physical devices, and a physical device can be used for different types of logical devices. For example, a value device can be realized by a 3D tracker or a joystick, and a keyboard and mouse can be used as both value device and trigger device. A device mapping file is used to map a logical device to a physical device. Since this file is separate from the application source code, an application may choose different physical devices for a logical device from time to time with the program being unchanged.

The MRDevice class is designed to represent the abstraction of logical devices in MRObjects. It defines the common attributes and behaviors that all devices have and provides the basic interface for defining various logical devices. A device with particular characteristics is defined by a derived class from MRDevice.

An MRObjects device is a kind of resource of an application, and all the devices used in the application are managed by the MRApplication object.

3.6 The Callback Mechanism

Callbacks provide a way to associate an operation (defined in the callback) with some conditions or events (e.g., user input). When a particular condition occurs, the corresponding operation is performed. The callback mechanism is required by several classes in MRObjects. Instead of handling callbacks in each individual class, this mechanism is implemented by a separate class, MRcallBack. Most classes in MRObjects (see Figure 3.1) have MRcallBack as their base class so they are capable of supporting callbacks, and this capability is inherited by all their derived classes.

The basic protocol defined in the MRcallBack class includes registering a callback, removing a callback, and invoking callbacks. The callback itself has to be defined in the class which needs callbacks.

3.7 System Architecture

In this section we conclude the MRObjects system design with Figure 3.2 which shows all the main classes and their relationships in MRObjects. Each box represents one class with its main functionalities described in it. Note that the lines between classes do not show inheritance but collaboration relationships. From the figure we can see that the MRApplication class connects to almost all the other classes. It is the center of the system structure and the heart of all the applications. The figure also demonstrates how an application interacts with the MRObjects system. The Application box represents MRObjects applications. The bold solid lines between Application and MRApplication class and MRGeometry class mean that applications declare instances of these classes, while the bold dashed lines between Application and ViewObject and InteractionObject classes mean that the application may optionally declare objects of these classes. An application may also declare objects of other classes. For example, an application may derive classes from DisplayObject and

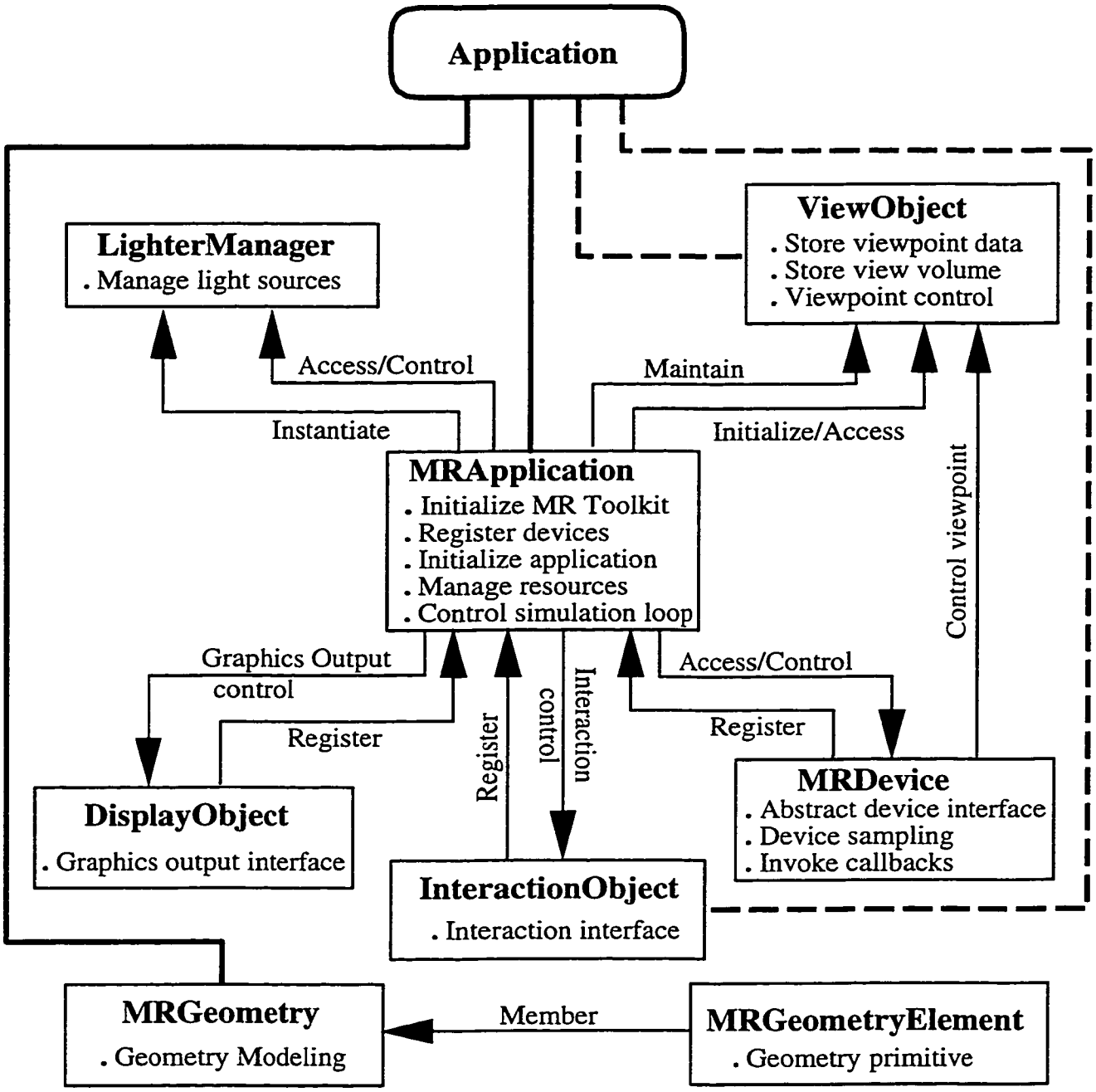

Figure 3.2: Class collaboration and application interface

declare objects of DisplayObject and the derived classes. Simple applications interact only with the MRApplication class and the MRGeometry class, the former being the application interface, while the latter is used to model the geometry in the application.

In this chapter, we have discussed some points concerning MRObjects system design. In Chapter 4, we will describe all the properties of MRObjects that have been implemented in this thesis and also some implementation issues.

Chapter 4

MRObjects – Implementation

An implementation of MRObjects can be divided into two parts: a portable part and a platform-dependent part. The portable part provides the portable interface for MRObjects applications. Ideally, it should remain the same across all types of platforms. The platform-dependent part, however, provides the underlying implementation based on particular computer systems. It has to be implemented for each platform that MRObjects works on. This thesis implements two versions of MRObjects on SGI workstations, using the GL graphics library and OpenGL, respectively. There are no differences between these versions with respect to the application development interface.

This chapter gives a complete image of the MRObjects class hierarchy implemented in this thesis. The class hierarchy is shown in Figure 4.1. Compared with Figure 3.1, Figure 4.1 contains more classes. The MRLightManager class is responsible for light source management, and it is under the control of the MRApplication class. MRPDevice is a subclass of MRDevice and defines a physical device. The ViewOrbitObject, ViewTrackObject, and ViewWalkObject classes are special cases of ViewObject, each implementing one navigation technique. The Pointer class represents one example of an interaction technique, which implements one style of object selection. A complete list of classes derived from MRGeometryElement implementing geometry primitives is given in a separate figure in the portable geometry section.

In the following sections, we examine all the classes in detail. Discussion is focused on the responsibilities and public interface of these classes, along with some

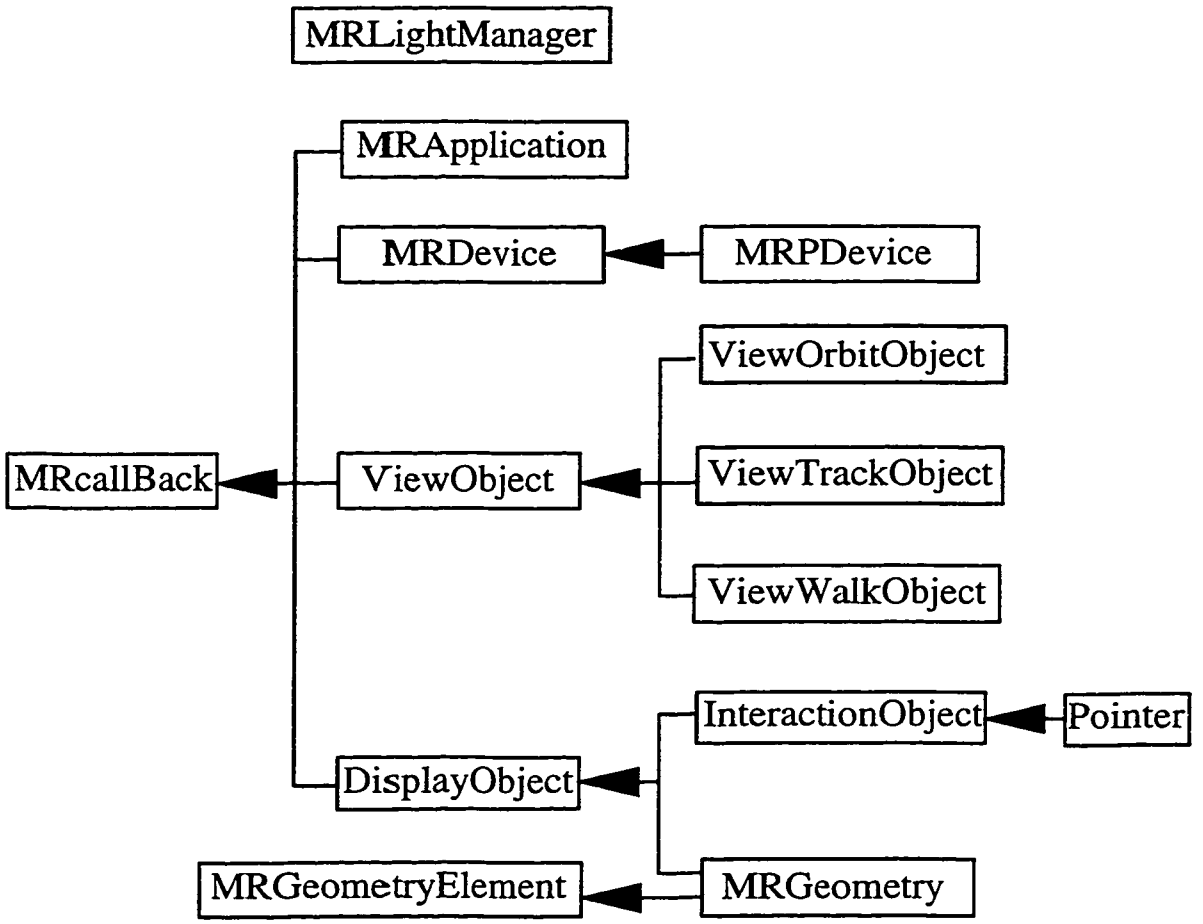

Figure 4.1: The implemented class hierarchy

implementation issues.

4.1 The MRApplication Class

The MRApplication class is the central class of the MRObjects system and its only instance is the central object of an application. The MRApplication object initializes the MR Toolkit, interacts with input and output devices, keeps track of the current state of all the objects in the application, controls the graphics output, passes user interaction on to the appropriate InteractionObject objects, and maintains global properties.

4.1.1 Data

The MRApplication class maintains some data structures for managing the resources, objects, and global properties of an application. The data structures include

- The Name of the application
- A list of DisplayObject objects
- A list of InteractionObject objects
- A list of MRObjects devices used in the application
- A set of pointers to global properties
- Parameters defining the display window
- Application state parameters

Each application should have a name. If the application does not give a name explicitly, the program name is used by default. The application name is used for identification and other purposes, such as finding a resource file.

All the DisplayObject and InteractionObject objects as well as all the devices used in the application are maintained by the MRApplication object.

There are a number of properties, such as material, texture, and lighting model, that affect the way in which geometries are rendered. These properties are potentially global; there is only one value for each property in effect at one time. MRApplication maintains a set of pointers that keep track of the current values for all these properties.

MRApplication also has some state parameters to record the state of an application. For example, there is a parameter to specify whether stereo display is used.

4.1.2 Behavior

The behaviors of the MRApplication class can be grouped into four categories.

4.1.2.1 Configuration

The following methods are defined for application configuration:

- Setting the display window size

  This function is used to set the size of the window on the screen for displaying the virtual environment.

- Registering MRObjects devices

  The devices used in the application must be registered with the MRApplication object. A device mapping file is used for specifying these devices. The file name is based on the name of the application. If the application does not provide such a file, the system uses its default one instead.

- Configuring the application

  This function decides the display type, sets graphics mode, and calls the underlying MR Toolkit procedures to configure the MR Toolkit and calibrate devices.

These configuration functions are mainly called in the constructor of MRApplication when creating the MRApplication object. Generally, the application does not need to call them explicitly. Some other MR routines for setting up the application are directly called in the constructor.

Configuring the application is the first step in an MRObjects application, and it is done after the MRApplication object is created. Only at this point can the application register objects. This means that all the geometry elements have to be defined or declared after the creation of the MRApplication object.

4.1.2.2 Object Management

After the configuration stage, the application can register objects and manage them during the execution. The following functions are designed for this purpose.

- Registering/unregistering the DisplayObject objects
- Registering/unregistering the InteractionObject objects
- Changing the state of specified DisplayObject
- Changing the state of specified InteractionObject

Object registration is done automatically when the object is created and unregistered when the object is destroyed.

Both DisplayObject and InteractionObject objects have a state attribute that controls how they participate in the application. The state of a DisplayObject object specifies whether the object is drawn or not, and the state of an InteractionObject object specifies whether the object is active or not. The state property of DisplayObject and InteractionObject is discussed in the sections on the corresponding classes. Note that the MRApplication object controls the state of these objects, not the objects themselves.

4.1.2.3 Global Property Maintenance

At present, MRApplication provides only the following two functions for controlling the lighting mode of the application:

- Switching the lighting model
- Customizing the lighting model

Some property maintenance tasks are not trivial, such as keeping track of the appropriate rendering property values for different objects in a virtual scene. In order to avoid overloading the MRApplication class, separate classes are designed to do the work. For example, we use a stack to maintain the properties which can have multiple values, and the property stack is managed by the MRStateObject class (discussed in Section 4.8.8).

4.1.2.4 Program Execution

- Checking for updates of the graphics objects
- Displaying DRAW objects and invoke ACTIVE InteractionObject objects
- Running the application

The first two functions are used to control the graphics output of the objects in the virtual scene. First any possible update of the objects is checked for. Then the objects are displayed. These two functions are called in the third function which starts the simulation loop. In the third run function, the devices are sampled, the callbacks associated with the sampled new values are invoked, the InteractionObject objects that are ACTIVE are invoked to process user input, and all the DisplayObject objects that are in a DRAW state are displayed. The run function should be called in the application once, and when it returns, the main() procedure should exit and the application ends.

4.1.3 Summary

The MRApplication class stores the main data structures of the application and implements the initialization, configuration, controlling, and most of the management tasks for the application. The minimum work of an application program related to this class is to create the MRApplication object for setting up the application and to call the run function for starting the execution of the virtual environment.

4.2 The DisplayObject Class

The DisplayObject class represents something that is displayed in the virtual environment. Anything in the application that needs graphical output should be defined as an object of this class or its derived classes. The DisplayObject class defines the protocol that is used by these objects.

The DisplayObject class is an abstract class. It has only two data members, one is the object's name, and the other is used to record the state of an object. A DisplayObject object is either in a DRAW state or an UNDRAW state, determining whether or not the object is displayed in the virtual environment. By default, the state of a DisplayObject is DRAW. A programmer may want to create a set of objects at the start of an application, but selectively display some of the objects at a particular time or in particular situations. Doing so allows the application to quickly switch the displayed objects without a delay caused by creating the objects during run time.

The DisplayObject class defines the following behaviors for its objects:

- Updating the object's graphics
- Drawing the object

These two behaviors are both defined as pure virtual functions in the DisplayObject class and must be overridden in all the classes that are derived from DisplayObject. If the programmers derive any class from DisplayObject, it is their responsibility to provide definitions for the update and draw methods.

Both of these functions are called once in each simulation cycle. The update method provides the opportunity to modify the graphics representation of an object in response to the user's current position and viewing direction before the object is drawn. The draw method implements object rendering, the basic and default behavior of a DisplayObject object.

More behaviors are defined in the derived classes and depend on the special characteristics of applications.

An application follows the DisplayObject's protocol to build its virtual world. However, it never creates any DisplayObject objects, it creates objects of its derived classes.

4.3 The InteractionObject Class

The InteractionObject class defines the basic protocol for objects that provide interaction techniques. Particular interaction techniques are defined by its derived classes. In this thesis only one subclass, the Pointer class, has been implemented to provide a pointing interaction technique. Most of the interaction techniques have to be explored in future work.

Besides the inherited properties and behaviors from the DisplayObject class, InteractionObject does not define any special characteristics at this moment except that the inherited state property has more options. Since, generally, an InteractionObject object also has graphics output, it can take either a DRAW or an UNDRAW state. In addition, the state of an InteractionObject object also has to specify either ACTIVE or INACTIVE to control how the object participates in the interaction. By default, an InteractionObject object is INACTIVE, which means that the object does not respond to any of the user's interactions, i.e., the interaction techniques associated with the object are not enabled. An INACTIVE InteractionObject object is normally in the UNDRAW state. If an INACTIVE InteractionObject object is in the DRAW state, it is drawn on the screen, but the user is not able to interact with it. This is used for interaction techniques that should always be visible, but may not always be in a state where they can be interacted with. The state of an InteractionObject object is expressed by both the draw property and the active property as bit combinations. There are three possibilities: INACTIVE and UNDRAW, INACTIVE and DRAW, and ACTIVE and DRAW.

4.3.1 The Pointer Class

The Pointer class provides a method of selecting objects with a pointing line. This interaction technique uses two devices. One is a pointing device, e.g., a 3D tracker, which is used to determine the pointing direction. The other is a trigger device, e.g., a button on the tracker, which is used for selecting the object. For object selection the pointing device has a pointing line as an echo to help in making the selection. The objects that intersect the line are potentially selected. An object is finally selected when the user presses the button. If more than one object intersects the pointing line, the nearest one is selected.

The Pointer class has to define callbacks for both pointing and selecting. The callback for pointing is responsible for determining the objects which are potentially selected. Also it should display the pointing line echo and feedback for the potentially selected objects. The selecting callback is called when the user selects the object by triggering the trigger device (for example, by pressing a button). This callback may register the selected object and do some manipulations on the object according to the purpose of the application (these two callbacks have not been implemented yet).

4.4 The ViewObject Class

The ViewObject class is a base class for implementing various navigation techniques. It contains information of the user's eye (also referred to as viewpoint), its current position and viewing direction, and the viewing pyramid used to display the object. The basic behavior of this class is to access the current viewpoint and parameters of the viewing pyramid. Other behaviors include initializing this ViewObject object, setting up the device, if any, for controlling the user's eye, and determining the new value of the eye.

For an application, there is only one ViewObject object defined. The initialization of the ViewObject object is done by the MRApplication object in its run function right before the simulation loop is started. The task of initialization involves setting the viewing volume and the initial value of the viewpoint.

The functions that set up the device and determine the new value of the eye are empty in the ViewObject class. They are overridden by ViewObject's derived classes which define particular navigation schemes.

Three derived classes have been defined.

4.4.1 The ViewTrackObject class

The ViewTrackObject class defines a navigation scheme in which the user's viewpoint is directly determined by an absolute device, e.g., a 3D tracker. The device's position and orientation are used directly to control the user's eye position and viewing direction. The problem with this scheme is that the physical space within which the user moves limits the virtual space in which the user navigates, and as a result this model may be used only for small virtual space navigation.

4.4.2 The ViewWalkObject class

This class implements a navigation model which simulates a walk-through or walk-around process. Relative devices, such as a keyboard and mouse or a joystick, are assumed to control the movement of the viewpoint. However, absolute devices, such as a 3D tracker, along with an additional control such as a button, can also be used for this navigation scheme. In this walk-through metaphor, there can be a variety of choices concerning how to control the speed of walking, when to walk and when to stop.

This scheme is suitable for architectural walk-throughs and any similarly sized virtual space navigation. For vast virtual environments, a similar scheme, called fly-through, with greater moving speed and probably choices for acceleration, can be implemented.

4.4.3 The ViewOrbitObject class

The ViewOrbitObject class defines a viewpoint control model that orbits around a certain object or a group of objects. The orbit basically follows the surface of a

sphere from left to right and top to bottom. The orbiting is carried out automatically without interactive control.

The properties that affect the orbiting behavior are as follows:

- Center of the orbit

  This is typically the center of the object(s) to be observed. By default, the center is assigned as the origin of the world coordinate system.

- Rotation axis

  It defines the axis about which the orbiting (rotating) is carried out. By default, the rotation occurs about the y-axis of the world coordinate system.

- Distance between the user and the orbit center

  This actually defines the radius of the orbit sphere.

- Parameters determining the start and end degrees of orbiting both along (e.g., beginning from the positive rotation axis and moving through the negative rotation axis) and around (in the rotation axis' perpendicular plane) the rotation axis, and the delta values for modifying the degrees in each step.

- Orbit behavior

  This property is intended to determine how the orbiting is carried out. It is not defined yet.

The main behavior of this class is to define the function that computes the current viewpoint based on the above properties.

This scheme is suitable for general observation of objects. A more desirable way of examining objects is perhaps user-controlled orbiting or direct object manipulation. User-controlled orbiting may be implemented later, while object manipulation is not part of navigation technique and is not considered here.

4.5 The MRcallBack Class

The MRcallBack class implements callbacks. Its data member is a list of callback lists, each callback list has a name and corresponds to a type of interaction. The public interface of this class is:

- Adding a callback to a particular callback list
- Removing a callback from a particular callback list

All the classes that are derived from the MRcallBack class inherit these two methods. These classes are responsible for defining their callbacks and then adding them to their appropriate callback list. Callbacks should be removed when they are no longer used.

MRcallBack also defines a protected method which is used to call all the callbacks on a callback list. The method is called through MRObjects devices when an interaction occurs. This part is defined in MRObjects and applications need not worry about it.

4.6 The MRDevice Class

The MRDevice class is an abstract class for defining all the devices used in MRObjects. Its derived classes will define a special type of device. The basic properties defined in MRDevice includes:

- Device name
- Device type
- Device value

Each device used in MRObjects has a predefined logical device name.

Device types determine what kind of values can be accessed from a device. MRObjects divides all the supported devices into three categories: value, glove, and trigger.

A value device corresponds to some form of tracker, and it produces position and orientation data. A glove device produces information on finger flexion. A trigger device corresponds to a button press or a hand gesture and has a boolean value. A device type is either value, glove, or trigger.

The device value property is a pointer to a structure which stores the actual device value.

The following behaviors are modeled in the MRDevice class:

- Sampling the device

  Reading the device's current value. This is a pure virtual function in the MRDevice class. All the derived classes have to define it according to the device it represents.

- Triggering the action that is associated with the device

  The newly sampled device value is used to invoke the callbacks that are registered with the interaction event that the device produces.

MRDevice functions as a server class which is used by other classes. Typically, interaction objects (objects of InteractionObject and its derived classes) have one or more member devices, which are used to implement user interactions, defined as MRDevice objects. The interaction objects are responsible for defining callbacks and registering the callbacks for the particular interaction events that are associated with its member devices. The detection of the interaction events and invocation of the callbacks, however, are handled by the MRDevice class and controlled by the MRApplication object.

4.6.1 The MRPDevice class

This class represents any physical device that is supported in MRObjects. It provides the same interface as MRDevice except that it overrides the sample and trigger functions based on the particular devices. MRPDevice is used to initialize all the devices that are requested by applications in their device mapping files during application initialization. Generally, applications do not declare any MRPDevice objects.

4.7 The MRGeometry Class

The MRGeometry class implements the portable geometry modeling interface . Applications declare objects of this class to define their virtual environments.

In this thesis, the geometry compiler has not been implemented, although similar functionality is provided. MRGeometry maintains only a list of geometry elements representing the geometrical representation of the object. The display of an MRGeometry object is implemented by its display member function, which in turn calls the display functions of the geometry elements that define the MRGeometry object. The display function of each geometry element is responsible for converting the geometrical representation of the element into the native graphics primitives and then rendering them.

For composing and editing MRGeometry objects MRGeometry defines a set of operations including

- Adding a geometry element to the geometry list
- Inserting a geometry element in the geometry list
- Removing a geometry element from the geometry list
- Replacing an element in the geometry list with another geometry element
- Obtaining the index of a geometry element in the geometry list

Since the addition operation is used heavily in building geometries, the '+' operator and '+=' operator are overloaded for this operation. They are defined both as member functions and friend functions of MRGeometry. Using the overloaded operators makes adding elements to the contents of an MRGeometry object easier and more natural. However, there are some limitations on using these operators. Generally, the first operand of the '+' operator should be a pointer or reference to an MRGeometry object, and the second is a pointer or reference to any of the geometry elements that can be added to an MRGeometry object. The result is always a pointer or reference

to an MRGeometry object. When the second operand is an MRGeometry object, hierarchical objects can be built up. It needs to be pointed out that the two operands can not be pointers at the same time.

An MRGeometry object is in the DRAW state when it is created and thus displayed automatically. If an MRGeometry object is built to be added to other objects as a child, then the object could be displayed twice, and probably with different properties (e.g., different positions, sizes, or orientations). To avoid this problem an MRGeometry object is set to UNDRAW whenever it is added to other objects as a child object. The same thing happens if one MRGeometry object is assigned to another. The one on the right side of the assignment becomes UNDRAW. If this is not desired, however, the programmer can always change the drawing state of an MRGeometry object manually.

MRGeometry has to define all the inherited methods, such as the draw method from DisplayObject and the save and load methods from MRGeometryElement, for its defined geometry. Basically, these methods are implemented by going through the geometry element list of MRGeometry, calling the corresponding methods defined in the elements.

4.7.1 Geometry Modification

In each rendering cycle, each DisplayObject object is first updated and then drawn. The MRGeometry class makes use of this inherited update function to implement geometry modification during run time.

In the next section we can see that each geometry element can have an application-defined behavior function that determines how the element changes during the simulation process. The update function of the MRGeometry class simply checks each of the geometry elements to see whether it has defined a behavior function. If so, the function is executed. In this way MRGeometry supports predefined dynamic scene changes, and the objects in the virtual environment can have predefined behaviors.

4.8 The Geometry Element Classes

We call all the classes that are derived from the MRGeometryElement class geometry element classes. The MRGeometryElement class is an abstract class that defines the basic interface for all the geometry element classes. Figure 4.2 shows the whole set of these classes.

Each geometry element class defines a type of high-level geometrical primitive. MRObjects provides a wide range of geometrical primitives. These primitives can be roughly divided into five groups: physical geometrical primitives, physical geometrical shapes, attribute elements, transformation elements, and light elements. Each group may have a local base class that abstracts the common characteristics for that group. All the groups are discussed in the following sections.

4.8.1 The MRGeometryElement Class

This class defines the general properties that all geometry elements have. They are

- Element name
- Element type
- Bounding box
- Dirty flag
- Pointer to behavior function
- Reference count number

Each geometry element can have a name. All the geometry elements have a type which is used to identify various geometry elements. Basically, each geometry element class defines one geometry element type. The bounding box is an important property which can support various computations, such as object selection and collision detection. The dirty flag is used to record whether the element has changed or not. It is set each time that the element is modified and cleared when the modification

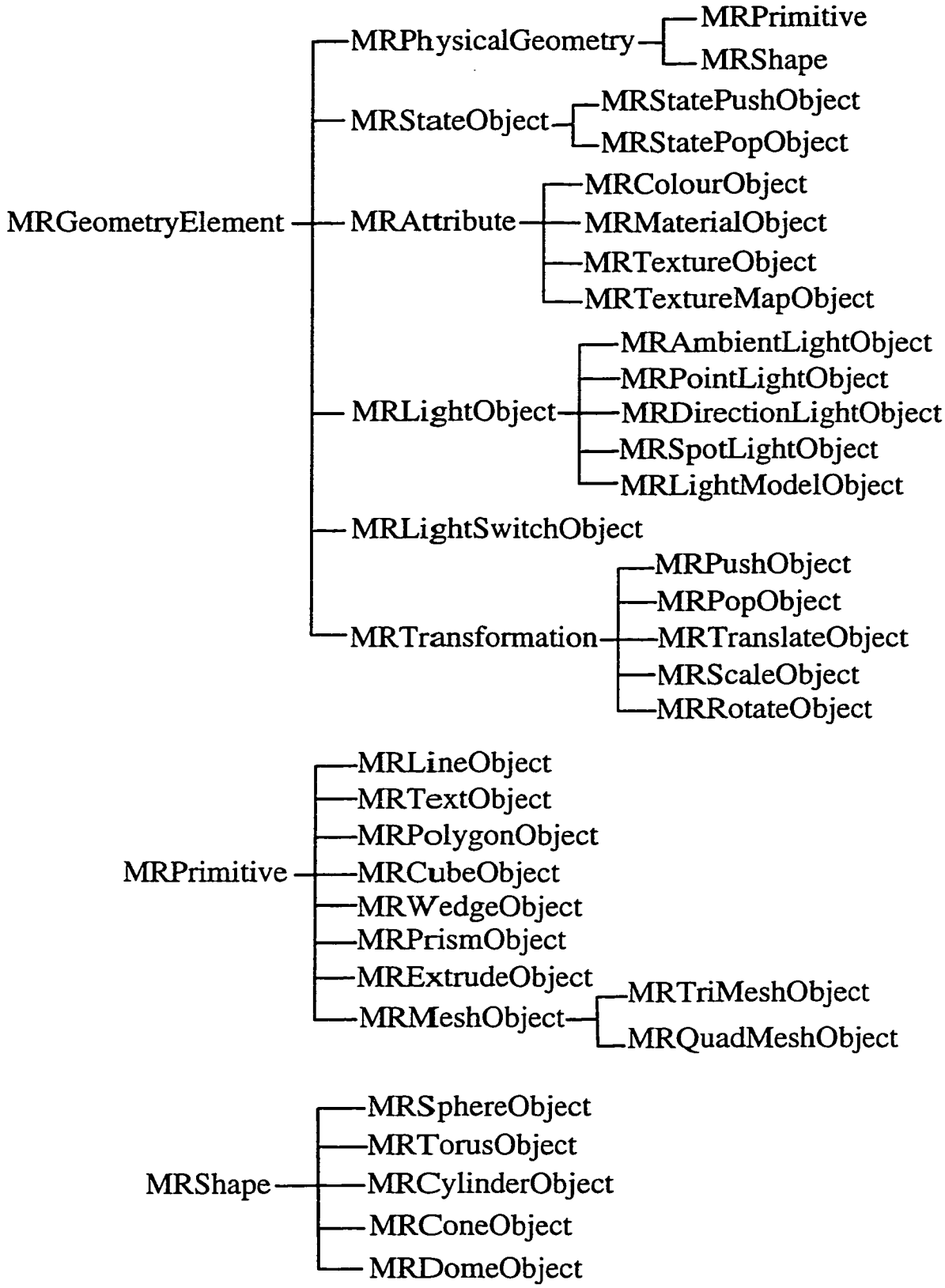

Figure 4.2: The Geometry Element classes

is handled. The purpose of this member is to assist with the efficient incremental compilation of the geometry objects.

MRGeometryElement supports predefined behavior for geometry elements. The behavior is defined by a function, called the behavior function. Generally, this function is provided and assigned to a pointer by the application. By default, the pointer to the behavior function is set to null.

The reference count number is designed for maintenance purposes. It records the number of places where the element is referenced. The element is removed only when this number is zero, meaning that no object refers to it.

The MRGeometryElement class defines five main behaviors for all geometry elements.

- Saving the contents of a geometry element to a file
- Loading the geometry element's definition, which has been previously saved by the save function, from a file
- Displaying the graphics of the geometry element
- Computing the bounding box of the geometry element
- Handling geometry element modifications

MRGeometryElement defines save and load as its basic behaviors so that the contents of MRGeometry objects can be saved to a file and also can be loaded from a file. In this way the definition of an application's virtual scene or just parts of it can be reused by different applications. ASCII files are used to represent the geometry element's definition so that the information can easily be shared between different hardware platforms.

The function for handling geometry element modifications checks the dirty flag to decide whether certain actions need to be taken. One primary action is to compute the bounding box. Whenever the geometry element is modified, its bounding box has to be recomputed.

These functions are empty in the MRGeometryElement class. Each derived class should define its own version of these functions.

4.8.1.1 Geometry Element Modification

An application-defined behavior function can be used to accomplish dynamic geometry modifications during run time. We consider two approaches for geometry element modification. One is to replace the geometry element with a new copy based on its new definition. The advantage of this method is that it supports geometry element sharing between geometries. Given that two objects share one geometry element at the beginning, if one object wants to modify the element during simulation time, it just makes a new copy of the element with the modification and leaves the original element untouched, so that the other object can still refer to the original element. This method, however, is not efficient since it repeatedly creates new copies of the changing geometry element.

The other approach, which is employed in this implementation, is to modify the geometry element definition directly. This approach is much more efficient than the previous one. The only concern is whether it affects geometry element sharing. In MRObjects a geometry definition is typically composed of three parts: physical geometry elements (e.g., a sphere), attribute elements (e.g., a red color), and transformation elements (e.g., a translation). Most geometry modifications can be fulfilled through various transformations. For example, instead of changing the radius of a sphere, we can use a scaling transformation. The position and orientation of a geometry element is determined by translation and rotation transformations. So in most cases, to modify a geometry, only the transformation elements need to be modified. Generally, these transformation elements are not shared by multiple objects. In conclusion, geometry element sharing is actually not a problem with this direct modification approach.

4.8.2 The Physical Geometry Elements

The MRPhysicalGeometry class abstracts all the classes that represent physical objects, e.g., a polygon, a cube, or a sphere. One behavior defined in this class is computing a geometry's bounding box. The class also defines a list of vertices that determine the bounding box. The vertex list must be calculated by each derived class, while the bounding box computation function can be inherited by all subclasses of MRPhysicalGeometry.

4.8.3 The Physical Geometrical Primitives

This group of classes represents geometrical primitives which have a fixed number of vertices and a fixed number of faces by definition. For example, the MRLineObject class defines a line with two vertices. The MRPrismObject class, a higher-level object, defines a prism with $2n$ vertices, where n represents the number of vertical faces that the prism has. Since this group of classes have a fixed number (normally not too big) of vertices or flat surfaces, they generally have constant descriptions for their objects and do not support level-of-detail. MRPrimitive is the base class for this group with the stated abstraction.

Most of the classes in this group correspond to the graphics primitives that are found in most graphics packages. The geometrical primitives defined under the MRPrimitive class are Line, Text, Polygon, Cube, Wedge, Prism, and Extrude. At present, triangular and quadrilateral Meshes are also classified in this geometrical primitive group since they have only one representation for the meshes. However, this may change in the future. An application may provide several meshes with different level of details for an object; or a simplified mesh (with less triangles or quadrilaterals) may be generated and used to represent the same object during run time when time is crucial. Either of these situations suggests that the Mesh element be placed in the geometrical shape group which is discussed in the next subsection.

Each primitive is defined by one class. All these classes define their save and load methods for writing to and reading from files and the display method for their

graphics output.

4.8.4 The Physical Geometrical Shapes

This group of classes defines some typical higher-level 3D shapes with smooth surfaces, for example, spheres. These shapes do not have corresponding graphical primitives in most 3D graphics packages. The surfaces of the shapes are approximated by triangular or quadrilateral meshes. The accuracy of the approximation depends on the number of triangles or quadrilaterals (polygons for simplicity) used to represent the surface. The greater the number of the polygons, the better the approximation, and the slower the rendering. So one property of this group of classes is the level of detail that controls how many polygons are used to approximate the shape. This property is defined in the MRShape class, the base class for this group.

In most cases the geometry compiler decides on the optimal number of polygons, and this number can change while the program is running. For a sphere, for example, a large number of polygons may be required if the sphere is close to the user and large in size, while a much smaller number can be used when the sphere is far away and small. In some cases the application designer may also want to control the number of polygons. This could occur when the designer is producing a special effect that depends upon the number of polygons in the approximation. MRShape provides the user with a public interface for doing this.

For higher efficiency, the vertex list used for computing the bounding box stores only the vertices that define the shape's self-defined bounding space instead of all the vertices that approximate the shape. For example, a cone is bounded within a pyramid, so only the five vertices of the pyramid are stored for computing the cone's bounding box. Sometimes, this can make the bounding box bigger than it really is. However, it may not be a problem in most cases, and we gain in efficiency.

The shapes defined in this group are the Sphere, the Torus, the Cylinder, the Cone, and the Dome.

4.8.5 The Attribute Elements

The attribute elements specify the properties of attributes that affect the way that objects are rendered. The attributes that are supported in this implementation include Colours, Materials, and Textures. The base class of this group of elements is MRAttribute.

The MRAttribute class has a state property to specify whether an attribute object is enabled or not, or whether it is valid or not. By default an attribute object is enabled. If an attribute object is disabled, it has no effect on its following geometries. An attribute object can be in the state of INVALID, this happens when errors occur when the object is created. The existence of invalid attribute objects is a waste of system resources (e.g., memory and time). Although the program can still execute, expected effect may not be achieved. This is an issue of error handling. Further investigation may be carried in future work. Each attribute class has a set of property members that define the attribute and a public interface for setting and accessing these properties. The basic behavior of an attribute object is to apply its properties to the system. When an attribute object is displayed, the defined attribute is applied to the following geometry objects until another attribute object of the same type is encountered.

4.8.5.1 Texture Mapping

Texture mapping applies an image onto an object's surface. Most graphics packages support texture mapping. The problem is how to map a texture defined in 2D texture space onto the surface of a 3D space object, i.e., to assign texture coordinates to the vertices of a geometric primitive. There is no general rule for doing this, and it really depends on the geometries and the requirements of the applications. In MRObjects the texture coordinates and their mapping to geometries' vertices are generally based on each individual physical geometry element. Most of the geometries defined under the MRPhysicalGeometry class have a predefined mapping from texture coordinates to the geometry vertices.

MRMeshObject also provides a way of accepting application-determined texture

coordinates. It has a data member recording the texture coordinates corresponding to all the vertices of a mesh object. An application can provide the texture coordinates when it defines a mesh object. Otherwise the texture coordinates are computed by one of the coordinate mapping algorithms defined in MRMeshObject.

MRTextureObject and MRTextureMapObject provide a higher-level interface for the application to specify the properties that describe how the texture is applied. The texture itself can be an array defined in the program or an image file. At present, GIF format images are supported.

4.8.6 The Transformation Elements

This group of classes defines geometrical transformations that are applied to elements with geometrical properties, e.g., size and position. The MRTransformation class is the base class of this group. It provides the general interface for all the transformation classes and acts as a holder for keeping some global data structures that are related to transformation operations. For example, it defines a static matrix stack for keeping track of the transformations applied to the geometrical objects. This matrix stack is used in bounding box computations.

The transformation classes include MRPushObject for pushing a transformation matrix onto the matrix stack, MRPopObject for popping a matrix off the matrix stack, MRTranslateObject for translation, MRScaleObject for scaling, and MRRotateObject for rotation.

4.8.7 The Lighting Elements

MRLightObject is the base class for defining various light sources. The light sources supported in MRObjects are Ambient light, Point light, Direction light, and Spot light.

All light sources have color and intensity properties, so these properties are defined in MRLightObject. MRLightObject also defines a state property to show whether the light source is switched on or off.

All graphics systems are limited in the number of light sources that can be supported at the same time. We call each supported light source a light target. MRObjects provides management for mapping the light objects defined by the application to the system's light targets. This work is done mainly by the MRLightManager class, which is discussed in the next section. MRLightObject has a data member to record the target index that the light object has been assigned. If the target index is INVALID, the light object is not associated with a system's light target so it is not enabled.

MRLightObject has a property-defining function for a light object to specify its properties and a target-setting function that associates the light object with a system's light target. The property-defining function is empty in MRLightObject and must be defined in each derived class.

The MRLightModelObject class defines the lighting model used in the application. A lighting model specifies the global ambient light that is not from any particular sources, whether the viewpoint is local to the scene or at an infinite distance, whether lighting is enabled for both the front and back faces of objects, and the parameters that determine how point light sources attenuate as the distance from them increases. To prevent a complete dark scene when no light sources are defined, the global ambient light intensity is set to a high value (e.g., greater than 70 % of the highest intensity supported by the system) by default.

The MRApplication object provides a default lighting model. The application, however, can customize its lighting model by changing the definition of the default lighting model or by simply declaring its own copy of the MRLightModelObject object. There is only one lighting model active at any time, and the MRApplication object maintains the current lighting model.

MRLightSwitchObject is used by the application to switch a light source during simulation. This is useful when the application wants to control the light sources in the scene. Note that this class is derived from the MRGeometryElement class and not from the MRLightObject class.

4.8.7.1 Defining Light Sources

MRObjects supports several types of light sources, such as directional light and point light. Applications define their light sources by creating objects of the appropriate subclasses of MRLightObject.

Generally, the light sources in the virtual scene have global effect. If a light is defined in the scene, it shines on all the geometries. So in most cases we define the light objects at the top level of the scene. Sometimes, however, the application might need local light sources, i.e., the light sources have effect only on a certain part of the whole scene. This is the case, for example, when some light sources do not shine on certain geometries or their effect can be ignored. In such situation, the light sources can actually be turned off for these geometries, resulting in simpler lighting computations. Applications can declare light sources locally with MRStatePushObject and MRStatePopObject (discussed next) isolating their effects or use MRLightSwitchObject to switch a light source on and off as desired to obtain local light source control.

4.8.8 The Property Management Elements

In Section 4.1, we mentioned about global property maintenance. Since it is not trivial, the MRStateObject class and its two subclasses, MRStatePushObject and MRStatePopObject, are designed for this task.

A property is defined by an object of the corresponding class. For example, the color property is defined by an MRColourObject object. For convenience, we call the objects that define properties property objects. To achieve different rendering effects for different parts of a virtual environment or isolating the effects of some property objects, push and pop operations are needed for saving the current property values, setting the local property values, and then restoring the original property values, as for transformations. MRStateObject defines a property stack to store the property objects that are pushed. MRStatePushObject and MRStatePopObject actually implement the push and pop operations for saving and restoring the current property values.

The application can give a parameter to the MRStatePushObject object to specify which properties are to be saved. If the parameter is not provided, then all the supported properties are saved. MRStatePopObject does not need such a parameter and it always knows what to restore.

The properties that are currently supported in MRObjects includes colors, materials, textures, light sources, and lighting models.

4.9 The MRLightManager class

An application can define a couple of light sources. Most graphics systems support a limited number of light source targets. A light source defined in an application has to be bounded with a system's light source target to take effect. The MRLightManager class is responsible for managing the application's light sources and associating them with the system's light source targets.

A light-switching method is defined in MRLightManager to do most of the management work. When an application creates an MRLightObject object, or when a light object is switched on, the light-switching function is called to allocate a system light source target. If there are light source targets available, the light object is associated with one target. If not, the light object does not have any effect on the application. When a light object is switched off or destroyed, if the object is bound to a system light target, the target is released by the light-switching function.

This is another example in which a non-trivial management task is undertaken by a separate class. There is only one MRLightManager object in an application and this object is under the control of the MRApplication object.

Chapter 5

Building MRObjects Applications

In Chapter 3, we showed the basic structure of an MRObjects application with a simple example. In this chapter, we discuss the development of VR applications using MRObjects in more detail. First, we show the application development model with MRObjects. Then we demonstrate the general steps in constructing an MRObjects application, identifying the parts provided by MRObjects, and the ones that must be provided by the application developers. We use two projects, conducted by the Computer Graphics Group at the University of Alberta, as examples to show how MRObjects can support the development of VR applications. One is the Athabasca Hall Walk-through project, which was done more than a year ago. The other is the 3D ATM Protocol Visualization project which is still being carried out. At the end of this chapter, we discuss the benefits that applications could obtain from the use of MRObjects.

5.1 Application Development Model

Figure 5.1 demonstrates the application development model using MRObjects. MRObjects sits on top of the MR Toolkit which provides the device handling and other low-level support. An MRObjects application never interacts directly with the MR Toolkit. Ideally, an MRObjects application is built only on top of MRObjects, using the portable programming interface provided by MRObjects. However, MRObjects may not meet the needs of all applications. An application may call the native graph-

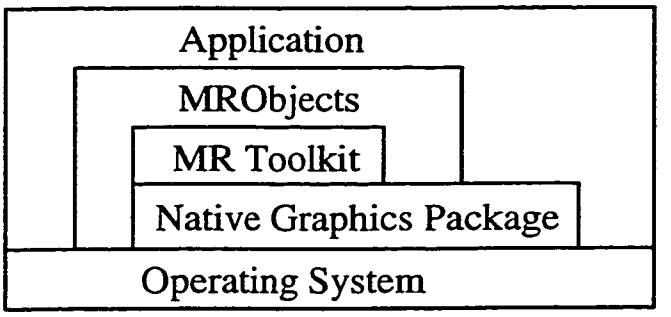

Figure 5.1: Application development model

ics package or other system libraries and facilities. In such case, the portability of the application depends on the portability of the graphics package and the system functions.

5.2 Building MRObjects Applications

Generally, as mentioned in Chapter 3, an MRObjects application can be built in three steps which are discussed as follows.

5.2.1 Declaring the MRApplication Object

The MRApplication class is the key class that is responsible for controlling the application. In the development of an MRObjects application, the first step in the main function is to declare an instance of the MRApplication class, the only one MRApplication object. Since all the objects and devices of an application are under the control of the MRApplication object, it has to be created before declaring any other objects. MRApplication has a set of default parameters for configuring the application. The developers, however, can customize their applications by providing command line parameters or by calling the public member methods of the MRApplication class. After the MRApplication object is created, for example, an application may choose a navigation metaphor by defining a ViewObject object of the appropriate kind and assign it to the application's only one ViewObject object by calling the setViewObject method of MRApplication. The application may also call the MRApplication's member functions to set the position and size of its display window.

5.2.2 Creating Virtual Scenes

Every application has its specific virtual scene to build. The application developers should first design and model the scene to determine how it is constructed. MRObjects provides support in creating a well-defined scene and rendering the scene. For defining a virtual scene, an application needs to declare objects of the DisplayObject class or its derived classes. The MRGeometry class, a subclass of DisplayObject, provides the portable interface for geometry modeling and is generally used in scene building.

If a virtual environment is composed of general geometries such as polygons, spheres, or extrudes, it is much easier to build the environment using the MRGeometry class than with general graphics packages. Generally, applications first declare objects of the MRGeometry class for their geometries to be defined. At this point, the MRGeometry objects have no elements in them. Then they use the overloaded '+' operator to add elements to the contents of the objects. The applications may also define behaviors for their geometries. The '+' operator of the MRGeometry class and the factory functions corresponding to the geometry elements make the scene-building process direct and easy.

When the scene is defined, the application developers have little else to do. They need not worry about how the scene is actually rendered because the display of the virtual environment is left to MRObjects, saving a considerable amount of development time.

However, it is not possible for MRGeometry and all the geometry element classes to provide a full set of geometries that meet the requirements of all applications. When it is not convenient to define a scene model using only the MRGeometry interface, an application may derive classes from the DisplayObject class to implement graphics definition and rendering following the basic protocol of the DisplayObject class. The native graphics package is likely used to define the display method for graphics rendering. As a result, the portability of the application is not guaranteed.

5.2.3 Running the Application

After an application has finished defining its virtual scene, it generally only needs to call the run method of the MRApplication object. At this point, the application enters its simulation loop. The control of the application is handled by MRObjects.

5.3 The Athabasca Hall Walk-through Example

The purpose of this project was to implement a virtual walk-through system for people to examine the structure and interior design of Athabasca Hall, to which the department of Computing Science is moving.

5.3.1 The Original Implementation

The project roughly consisted of the following work:

- Obtaining a proper scene model

  The original model for Athabasca Hall was a 2D physical plan which was represented by a number of DWG files from AutoCAD. These files were first converted to self-defined text files. Then a 3D model was designed based on the original 2D model.

- Preparing images for texture mapping
- Designing the rendering model
- Implementing scene culling and rendering
- Implementing navigation and viewpoint-scene collision detection algorithms

The project was originally done using the GL graphics package and the MR Toolkit on SGI workstations. It took two graduate students about six weeks to finish.

5.3.2 Implementation with MRObjects

The first three parts of the work are application-specific problems that need to be done by the developers no matter what software and platforms are used. MRObjects can provide support for the rest of the work. At present, culling and collision detection have not yet been implemented in MRObjects. However, they are generally required facilities and will be added in future development.

The scene in the project is relatively simple. It consists mainly of polygon and extrude geometries, which are supported by the geometry elements defined in MRObjects. In this case, it is quite easy to define the scene with MRObjects without much programming.

This example program consists of two major functions: a main() function and an object-creation function, the latter of which reads data from a data file and creates the corresponding geometrical objects. We use a flat object structure to represent the geometry of the scene. So only one MRGeometry object is defined. A pointer to this object is declared as a global variable so that the MRGeometry object can be referenced in both functions.

In the main() function, the MRApplication object is first created, followed by an MRGeometry object. Then the scene is defined simply by adding all the geometries to the contents of the MRGeometry object. Texture mapping is used to achieve realistic effect. An MRTextureMapObject object, defining the texture mapping parameters, and an MRTranslateObject object, translating the geometry's center to the origin of the world coordinate system, are first added to the MRGeometry object. Then the object-creation function is called for each data file to add all the geometries to the MRGeometry object. For applying different textures to different parts of the scene, an MRTextureObject object with appropriate texture definition is added to the MRGeometry object before the geometries that use the texture are added. When the scene definition is finished, the main() function calls the run method of the MRApplication object to start the simulation.

In this example, we also show how to define a behavior function for a geome-

try element object, an object of any derived class from MRGeometryElement. One problem in rendering the building is that the interior and exterior appearances of the outer walls of the building are different. This means that we cannot simply use one texture for all the walls. The original implementation used two different textures. When the user is inside the building, a texture for the interior-wall is used; when the user is outside the building, a texture representing the outside appearance of the walls is used. So the texture used depends on the run-time position of the user's viewpoint. With MRObjects this is implemented by defining a behavior function for the MRTextureObject object which applies the exterior-wall texture to the outer walls that follow. The behavior function tests whether the user's viewpoint is inside the building. If so, the texture object is enabled, and its defined exterior-wall texture is applied. Otherwise, the texture object is disabled, and the interior-wall texture, which was defined earlier in the MRGeometry object's contents, is used.

It only took several hours to implement the rendering of the building (except the stairs) with the support of MRObjects.

5.4 The 3D ATM Protocol Visualization Example

5.4.1 Background

ATM (Asynchronous Transfer Mode) is a layered protocol for high speed networks which support a multiplicity of services. The goal of the 3D ATM Protocol Visualization project is to investigate 3D visualization techniques which can provide effective error detection and performance analysis for ATM protocol data [13]. The main result of the first stage of this project, the result of a Master thesis, was a visualization prototype running on HP workstations using the Starbase graphics package and the MR Toolkit. This example is based on this prototype.

The ATM protocol data consist of several layers forming a protocol stack. The developers of the project designed a ring structure to represent the stream-line style data for each layer. Data on each layer are composed of a number of protocol data units and are displayed on an individual ring. Different 2D geometrical shapes and

colors are used to represent different protocol data units and different information. The scene representing the whole protocol stack is composed of several rings, and each ring has a lot of details on it.

In addition to the visualization model, the developers have spent a lot of time implementing a user-control interface, which enables users to change the parameters that control the data display, and a navigation scheme for users to navigate through the virtual 3D data space. They also have to render the scene using a low-level graphics package.

5.4.2 Implementation with MRObjects

For information visualization problems, there are no general rules to follow on how to represent a data set. Every visualization application seems to have its own unique problems.

MRObjects cannot reduce the effort of designing an appropriate model to visualize the ATM protocol data. However, MRObjects can support the developers in implementing their design. The user interactive control techniques and the navigation metaphor used in the prototype are not unique to this project. They could be provided by MRObjects. A rich set of interaction techniques will be employed in MRObjects so that the time spent on developing these techniques by individual developers can be saved.

MRObjects can also help in modeling the geometry in the project. But it is not as direct and easy as in the first walk-through example. The overall structure of the scene in this visualization project is simple, but the details are fairly complicated. It may not be appropriate to simply use the geometry element classes currently provided to define the scene. A possible approach is to define one or more subclasses of DisplayObject. The derived classes will both implement the inherited interface from the DisplayObject class and add application-specific features.

To quickly integrate the visualization project into the MRObjects framework, we simply derive a class from DisplayObject. This class is implemented as an interface between MRObjects and the original program of the project. The display function of

the class simply calls the draw function in the original Ring class for all the rings to be displayed. Of course, the original Starbase graphics calls have to be replaced. We tested this example on SGI workstations and used the GL graphics package.

5.5 Discussion

We used two actual projects as examples to show how to develop VR applications with MRObjects. Since the MRObjects' implementation for SGI platforms, which this thesis is based on, is not complete (in particular, it lacks support for various interaction techniques), our implementations for the two projects were not full-fledged with respect to their original implementations. We could not really show how much work could be saved using MRObjects compared with the original development. However, we clearly saw what work could be done by MRObjects, instead of the application developers and what effort could be saved with the support of MRObjects. In the first walk-through example, MRObjects greatly simplified the work of scene definition and rendering, and the resulting application is much easier to maintain and accommodate new requirements. The program implemented with MRObjects was easily ported to the newly opened VizRoom environment (at Computing Science Department of the University of Alberta) with little change. It would be much more difficult with the original implementation. With the second visualization example we demonstrated how to integrate an existing application into MRObjects.

It is certain that the work for developing these two projects would have been much easier with the help of MRObjects. More importantly, since the development of applications with MRObjects is at a higher object-level, the development process is more intuitive and less demanding on system-specific implementation details. The applications developed are more maintainable, flexible, and portable.

The two projects have quite different characteristics, demonstrating that MRObjects can provide general support for various kinds of applications.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

MRObjects is an object-oriented toolkit as well as a framework which provides high-level support for the development of VR applications.

The MRApplication, DisplayObject, and InteractionObject classes form the central part of the framework, defining the structure of the applications. The MRApplication class is responsible for the management of the application, the DisplayObject class provides the basic interface for objects with graphics output, and the InteractionObject class defines the basic interface for various interaction techniques.

One of the primary goals of MRObjects is to produce portable VR applications. This goal is achieved by the high-level, platform-independent graphics description interface which is implemented by the MRGeometry class, the MRGeometryElement class and all its derived classes. These classes are twofold: a portable application interface and an underlying platform-dependent implementation. The implementation in this thesis is for the SGI workstation platforms.

MRObjects supports a wide range of devices which are employed in various interaction techniques. The application developers need not handle these devices. They need only to specify what devices they would like to use and, typically, select the interaction techniques as well. MRObjects handles interacting with the devices.

The task of an MRObjects application is to create an MRApplication object, define objects of MRGeometry or subclasses of DisplayObject to build the virtual

environment, declare objects of subclasses of InteractionObject for desired interaction techniques, and then put the environment into execution.

The MRObjects implementation of this thesis is not complete. It can only support virtual environments with static scenes or objects with pre-defined simulation and limited interactions.

6.2 Future Work

The work of this thesis can be considered the first step in building the MRObjects system. There is still a lot of work that needs to be done.

6.2.1 Interaction Techniques

A main aim of MRObjects is to support full 3D interaction and provide VR application developers with a rich set of interaction techniques. Most of this work is left for future development.

At present, the main focuses of the interaction techniques are navigation and effective user control over the environment based on the requirements of the applications we are working on.

Navigation is always a major concern in a 3D virtual space. As a toolkit, MRObjects should develop various navigation techniques which serve different requirements as well as cognitive concerns. In addition to the navigation schemes implemented, more metaphors need to be developed. For example, navigation with respect to a specific target and constrained viewpoint movement in order to prevent the user from easily getting lost.

In all the navigation schemes, a way of recording the history of the viewpoint changes during the simulation seems useful in assisting the navigation. For example, if the user gets lost in a virtual environment, he/she may want to go back to a previously reached position and orientation to continue his/her navigation.

Menus may be used to provide effective and intuitive user control. A variety of menu techniques have been developed here in the Computer Graphics Group. The

MR Toolkit supports panels, and some other types have also been used in other systems. We may take advantage of the work already done in developing the user control interface for MRObjects.

Since MRObjects is a general-purpose package, an even wider range of interaction techniques needs to be developed to meet the general requirements of a wide range of applications. Direct object manipulation techniques, e.g., grabbing an object and placing it at or dropping it to another place and changing the properties of objects, are required in some VR applications and may be provided in MRObjects.

6.2.2 Behavior Support

For most VR applications, objects in the virtual environment tend to have certain behaviors. The behaviors can either be predefined animations or responses to user interaction. The present implementation supports only predefined behavior through user-defined functions. In order to support fully interactive VR systems in which users can actually interact with the virtual objects and objects may interact with each other, interaction response behavior should be explored and added to MRObjects. We need to investigate different scenarios from the application point of view to determine ways of supporting various kinds of behavior.

One consideration on assisting simple predefined motion or animation, such as a regular translation or rotation, is to define dynamic transformation elements. These elements can be derived from the normal transformation elements and provide regular delta changes in size, position, and orientation. For example, a dynamic rotation element can provide self-rotation animation, and the application needs to provide only a delta value for rotation angle, rather than a function for defining the proposed animation when specifying the dynamic rotation element. Another benefit is that the defined animation can be saved along with the definition of the whole geometry. This idea needs further consideration on what potential animation abilities and behavior should be provided by the dynamic transformation elements.

6.2.3 Collision Detection

As long as the objects in a virtual environment move, collision detection should be provided so that objects do not penetrate each other and they behave as in the real world. Collision detection is an important part of the future work for MRObjects.

Collision detection can be a very intensive computation in VR applications which are real time environments. With the present hardware performance any feasible algorithm has to use some kind of approximation to detect possible collisions. Collision detection based on bounding boxes is a reasonable scheme which greatly simplifies the computation and adequately meets the constraints of a real-time virtual reality system [8].

At present, a bounding box is defined as a basic property of the MRGeometryElement class for all geometry elements. It provides the basic support for a bounding box based collision detection algorithm.

6.2.4 Rendering Performance Improvement

Generally, high-level graphics support can increase the portability of the application but at the same time decreases the rendering speed. To ensure high performance as well as achieving portability, MRObjects will make use of the geometry compilation techniques that have been developing here at the Computer Graphics Group. Based on this approach, a geometry compiler is implemented to convert high-level geometrical object description into the most efficient set of display primitives for the underlying platform. For objects with static descriptions, the compiler is executed only once to produce the highly efficient graphics primitives. For objects whose descriptions change dynamically, an incremental compiler can be employed so that only the changed part of the object description needs to be recompiled.

Culling may also be used in MRObjects in the future to improve the rendering speed of the virtual world. The basic idea is to remove the objects outside the viewing frustum before rendering. This is especially useful when the virtual world is big, the number of objects is large, and the objects are complex, and, as a result, the time

that is spent on rendering invisible objects is far greater than the time spent on the culling computation. The application should be able to turn the culling on and off according to different situations.

The MR Toolkit is based on a decoupled simulation model in which multiple processes can be designed to fulfill the application's task collaboratively. MRObjects should take advantage of this functionality of MR and implement a multi-process structure for VR applications. One process, for example, performs stereo graphics output if the application uses stereo display, and one other process performs simulation computations. In this way, the rendering performance can be improved. In contrast to the MR Toolkit, the multi-process structure in MRObjects should be transparent to the application. The developers should not worry about how to define separate processes for computation and other sub-tasks of the applications.

6.2.5 Others

For easier virtual environment modeling, more geometrical elements need to be added to MRObjects, such as sweep objects, terrains, and 3D text.

For direct scene building and easy testing, an interactive scene editor is desirable. This task may be fulfilled by making use of the previously built 3D modeler – JDCAD+ [10][15] which allows users to design object geometry and behavior to construct interesting virtual environments without programming. The result can be saved to a file which can be loaded into an MRObjects application.

Some popular 3D object file formats, such as DXF and VRML, may be supported to facilitate the interaction of MRObjects with other 3D environment modeling systems and integrate available 3D models.

Support for multi-user and networked applications could be a big part of future work.

Bibliography

[1] Don Allison, Brian Wills, Doug Bowman, Jean Wineman, and Larry F.Hodges, *The Virtual Reality Gorilla Exhibit*, IEEE Computer Graphics and Application, November/December 1997, pp.30-38. .

[2] Steve Aukstakalnis, David Blatner, *Silicon Mirage–The Art and Science of Virtual Reality*, Chapter 9, Peachpit Press, INc., 1992.

[3] Duane K.Boman, *International Survey: Virtual-Environment Research*, IEEE Computer, June 1995, pp.57-65.

[4] Grigore Burdea and Philippe Coiffet, *Virtual Reality Technology*, Chapter 8, John Wiley & Sons, Inc., English edition 1994.

[5] Grigore Burdea, George Patounakis, Viorel Popescu, and Robert E.Weiss *Virtual Reality Training for the Diagnosis of Prostate Cancer*, 1998 IEEE Annual Virtual Reality International Symposium (VRAIS'98), pp.190-197.

[6] Carolina Cruz-Neira, Jason Leigh, et al., *Scientists in Wonderland : A Report on Visualization Applications in the CAVE Virtual Reality Environment*, IEEE 1993 Symposium on Research Frontiers in Virtual Reality, October 25-26, 1993, San Jose, California, pp.59-66.

[7] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi, *TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications*, SIGGRAPH'94, Computer Graphics Proceedings, pp.421-434.

[8] Annette B. Gottstein *Uses of Bounding Boxes In The Object Modeling Language*, MSc. Thesis, Department of Computing Science, University of Alberta.

[9] Mark Green and Lloyd White, *Minimal Reality Toolkit – Programmer's Manual*, Department of Computing Science, University of Alberta.

[10] Mark Green and Sean Halliday, *A Geometric Modeling and Animation System for Virtual Reality*, Communications of the ACM, Vol.39, No.5, May 1996, pp.46-53.

[11] Chua Gim Guan, Luis Serra, Ralf A.Kockro, Ng Hern, Wieslaw L.Nowinski, Chumpon Chan, *Volume-based Tumor Neurosurgery Planning in the Virtual Workbench*, 1998 IEEE Annual Virtual Reality International Symposium (VRAIS'98), pp.167-173.

[12] Larry F. Hodges, Benjamin A. Watson, G. Drew Kessler, Barbara O. Rothbaum, and Dan Opdyke *Virtually Conquering Fear of Flying*, IEEE Computer Graphics and Application, Vol. 16, No. 6, November 1996, pp.42-49.

[13] Lai Ling Kwan, *Visualization of ATM Network Data*, MSc. Thesis, Department of Computing Science, University of Alberta, 1998.

[14] Valerie D. Lehner and Thomas A. DeFanti, *Distributed Virtual Reality: Supporting Remote Collaboration in Vehicle Design*, IEEE Computer Graphics and Application, March-April, 1997, pp.13-17.

[15] Jiandong Liang and Mark Green, *Geometric Modeling Using Six Degrees of Freedom Input Devices*, 3rd International Conference on CAD and Computer Graphics Proceedings, pages 217-222, Beijing, China, August 1993.

[16] R.Bowen Loftin, Patrick J. Kenney, *Training the Hubble Space Telescope Flight Team*, IEEE Computer Graphics and Application, September 1995, pp.31-37.

[17] R.Bowen Loftin, Robin Benedetti, *Applying Virtual Reality in Education : A Prototypical Virtual Physical Laboratory* IEEE 1993 Symposium On Research Frontiers in Virtual Reality, pp.67-74.

[18] William Ribarsky, Jay Bolter, Augusto Op den Bosch, and Ron Van Teylingen, *Visualization and Analysis Using Virtual Reality*, IEEE Computer Graphics and Applications, January 1994, pp.10-12.

[19] John Rohlf and James Helman, *IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics*, SIGGRAPH'94, Computer Graphics Proceedings, pp.381-394.

[20] Chris Shaw, Jiandong Liang, Mark Green and Yunqi Sun, *The Decoupled Simulation Model for Virtual Reality Systems*, ACM SIGCHI Human Factors in Computing Systems, CHI'92 Proceedings, pp.321-328.

[21] Dave Sims, *New Realities in Aircraft Design and Manufacture*, IEEE Computer Graphics and Application, March 1994, pp.91.

[22] Paul S. Strauss, Rikk Carey, *An Object-Oriented 3D Graphics Toolkit*, Computer Graphics, 26, 2, July 1992, pp.341-349.

[23] Martin R.Stytz, John Vanderbrugh, and Sheila B.Banks, *The Solar System Modeler*, IEEE Computer Graphics and Application, Vol. 17, No. 5, September/October 1997, pp.47-57.

[24] Yunqi Sun, *A Hierarchical Model for Virtual Environments*, PhD thesis, Department of Computing Science, University of Alberta, 1995.

[25] David L. Tate, Linda Sibert, and Tony King, *Using Virtual Environments to Train Firefighters*, IEEE Computer Graphics and Application, Vol.17, No.6, November/December 1997, pp..

[26] UVa User Interface Group, University of Virginia, *Alice: Rapid Prototyping for Virtual Reality*, IEEE Computer Graphics and Applications, May 1995, pp.8-11.

[27] Edited by Kevin Warwick, John Gray and David Roberts, *Virtual Reality in Engineering*, Chapter 11, The Institution of Electrical Engineers, 1993.

[28] David Zeltzer, Nicholas J. Pioch, Walter A. Aviles, *Training the Officer of the Deck*, IEEE Computer Graphics and Applications, Vol. 15 No.6, November 1995, pp. 6-9.

[29] Web site: *http://www.alice.org/* .

[30] Web site: *http://www.cc.gatech.edu/guv/Virtual/SVE/* .

[31] Web site: *http://www.vetl.uh.edu/ScienceSpace/absvir.html* .