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.

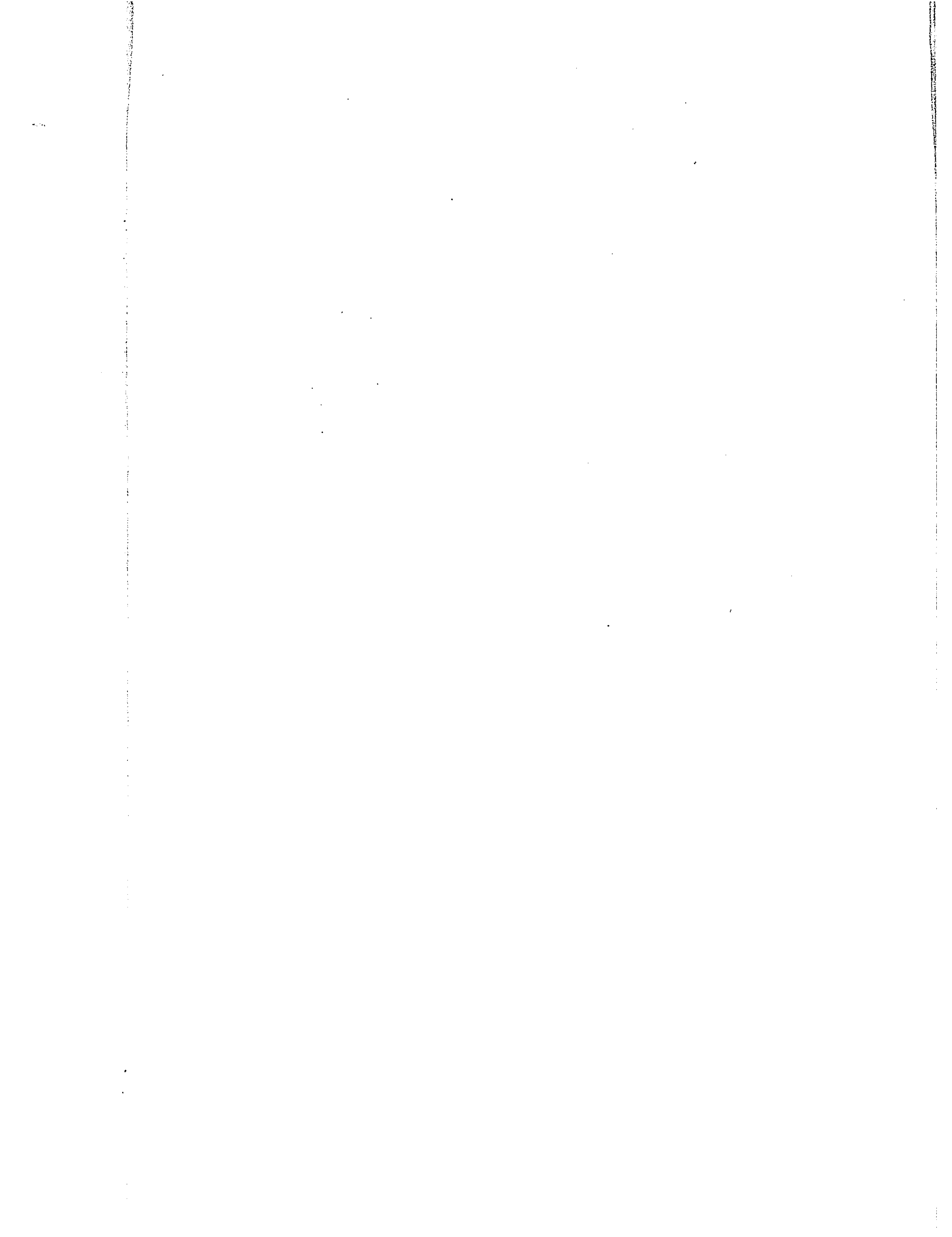
The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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.

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

UMI®



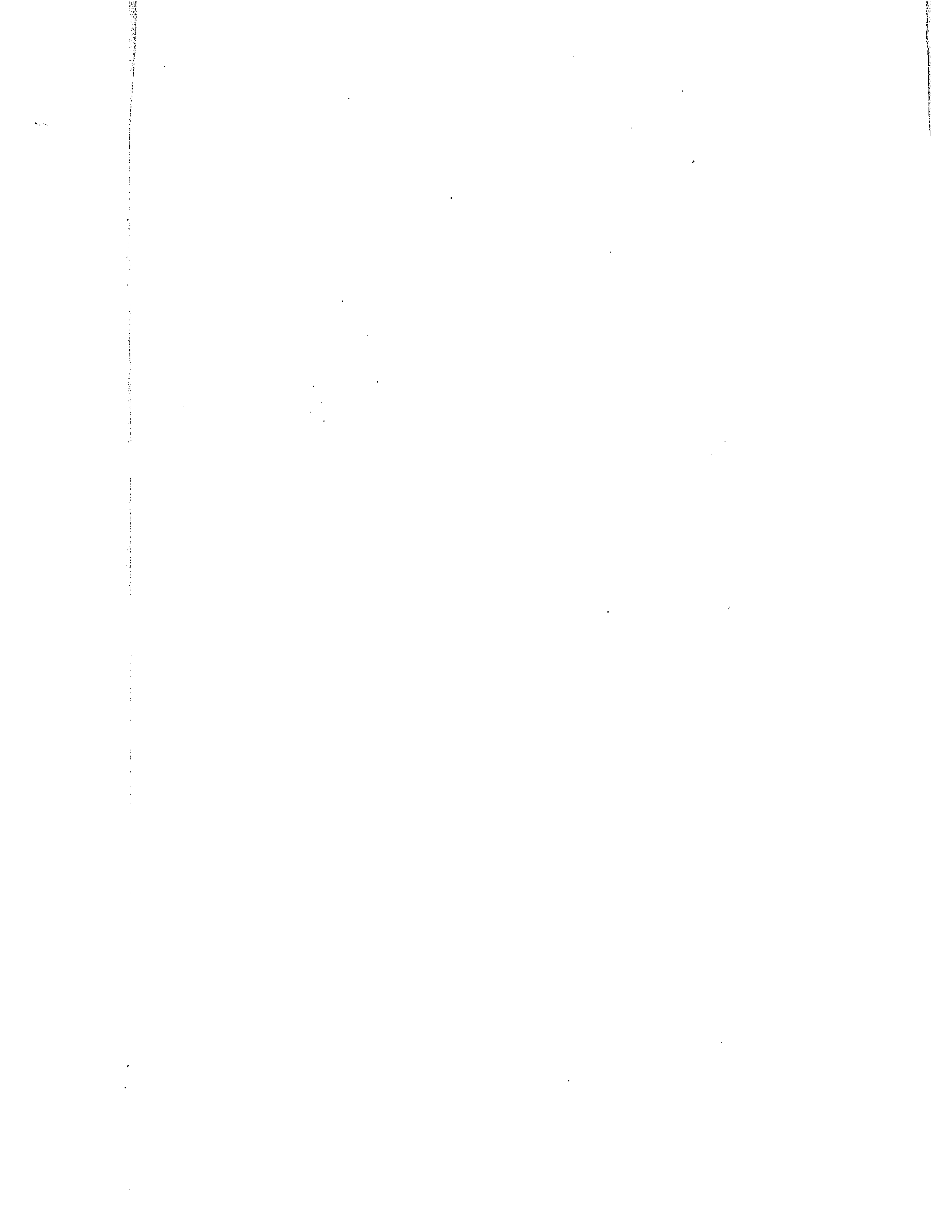
NOTE TO USERS

Page(s) not included in the original manuscript are unavailable from the author or university. The manuscript was microfilmed as received.

106-126

This reproduction is the best copy available.

UMI®



ATHES

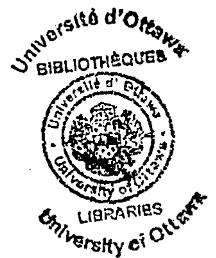
UNISON : A GENERIC FRAMEWORK FOR HAPTO-VISUAL APPLICATION DEVELOPMENT

by

Naim R. El-Far

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of

Master of Computer Science



Ottawa-Carleton Institute of Computer Science

School of Information Technology and Engineering

University of Ottawa

2005

© Naim R. El-Far, Ottawa, Canada, 2005

The author hereby grants the University of Ottawa permission to reproduce and distribute copies of this work in whole or in part.

UMI Number: EC52101

INFORMATION TO USERS

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 bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send 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.

UMI®

UMI Microform EC52101
Copyright 2007 by ProQuest LLC
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

Given the advancements in computer input and output via visual and auditory media, the next natural progression is to include the sense of touch via what is called haptic media. In this thesis, we review the evolution of auditory, visual, and haptic media since the 1940s, within the context of human-computer interaction, as a historical backdrop to our examination of current practices in hapto-visual application development. Assessing the fragmented, non-standardized development processes in use today to write hapto-visual applications, we try to find similarities and common denominators within the aforementioned processes, in order to come up with a generic framework that the application developer needs to be concerned only with, regardless of underlying technologies. In this work, we introduce Unison as a viable, extensible solution to standardize hapto-visual application development and detail its workings and implementation issues. We also present a scenario which has been used as a development tool for a hapto-visual e-commerce application, and conclude with suggestions for future work.

ACKNOWLEDGMENTS

First and foremost I wish to thank Dr. Nicolas D. Georganas for his supervision, support, and patience, without which this work would not have been possible. I also would like to thank Dr. Mojtaba Hosseini, Dr. Xiaojun Shen, and Mr. Francois Malric for their advice and assistance throughout my research career over the past two years here at the DISCOVER lab at the University of Ottawa.

I also would like to acknowledge Coach Denis Piché of the Ottawa GeeGees Football Team and Coach Louis Campbell of the Arkansas Razorbacks Football Team for pushing me to pursue both my graduate studies and my football career here at the University of Ottawa. To them, I say thank you.

DEDICATION

Heavenly Father, I thank you for the circumstances, good and bad, that have brought me here today.

To my mother, who has sacrificed her all to raise me, my brother, and my sister. I dedicate this, first and foremost, to you.

To the memory of my father, who in his death has taught me more than he ever could have in his life.

To my younger brother Fayez, who has set example after example for me over the past few years in fortitude and perseverance.

To my little sister Dalia, who is my never-ending well of genuine, unconditional love and support.

To Caro, who there is no one word to describe what she is to me, for teaching me more about myself than I knew was there.

TABLE OF CONTENTS

LIST OF FIGURES VII
LIST OF TABLES VIII
GLOSSARY IX

CHAPTER 1: INTRODUCTION 1

1.1. TECHNOLOGY IN OUR EVERYDAY LIFE 1
1.2. INTERACTION WITH COMPUTERS 1
1.3. CONTRIBUTIONS 2
1.4. ORGANIZATION 3
1.5. PUBLICATIONS ARISING FROM THIS THESIS 4

CHAPTER 2: THE EVOLUTION OF COMPUTER MEDIA IN HUMAN COMPUTER INTERACTION 6

2.1. INTRODUCTION 6
2.2. THE PUNCHED CARD 8
2.3. THE KEYBOARD 12
2.4. THE MOUSE 13
2.5. COMPUTER-GENERATED AUDIO 14
2.6. HAPTIC DEVICES 18

CHAPTER 3: AN OVERVIEW OF HAPTO-VISUAL APPLICATION DEVELOPMENT TODAY 22

3.1. DEFINITION 22
3.2. AN OVERVIEW OF THE CURRENT STATE OF HAPTIC DEVELOPMENT 23
3.3. CURRENT PRACTICES IN HAPTO VISUAL APPLICATION DEVELOPMENT 31
3.3.1. A SURVEY 31
3.3.2. ANALYSIS 41
3.3.3. CONCLUSION 44

CHAPTER 4: A HIGH-LEVEL LOOK AT UNISON 47

4.1. INTRODUCTION 47
4.2. UNISON'S INTEGRATION INTO AN HVAD ENVIRONMENT 51
4.3. UNISON'S EXTENSIBILITY 55

CHAPTER 5: THE DESIGN AND IMPLEMENTATION OF UNISON 57

5.1. INTRODUCTION 57
5.2. APPLICATION PROGRAMMING LANGUAGE 57
5.2.1. C AND C++ 58
5.1.2. JAVA 60
5.1.2. A COMPARISON BETWEEN C++ AND JAVA 62
5.1.3. PROGRAMMING UNISON 65
5.2. THE GRAPHIC COMPONENTS 66
5.2.1. VRML 66
5.2.2. JAVA 3D 67
5.2.3. OPENGL 68
5.2.4. XJ3D 69
5.2.5. CYBERGARAGE CYBERX3D LOADER FOR C++ 70
5.2.6. GRAPHIC COMPONENTS IN BUILDING UNISON 71
5.3. THE HAPTIC COMPONENTS 72
5.3.1. HAPTIC DEVICES STUDIED 72
5.3.2. HAPTIC APIS STUDIED 76
5.3.3. HAPTIC COMPONENTS IN UNISON 79
5.4. UNISON DATA STRUCTURES AND FUNCTIONS 81
5.5. NEGATIVE ASPECTS OF USING UNISON 87

CHAPTER 6: AN APPLICATION SCENARIO USING UNISON 90

6.1. INTRODUCTION 90
6.2. IMPLEMENTATION 93
6.3. USABILITY OF UNISON AS A DEVELOPMENT TOOL 96

CHAPTER 7: CONCLUSION AND FUTURE WORK 99

APPENDIX I: FORCE CONTROL PARADIGMS IN HAPTIC DEVICES 103

APPENDIX II: SOURCE CODE 105

BIBLIOGRAPHY 127

LIST OF FIGURES

Number *Page*

Figure 1: Immersion Haptic Workstation [37] 26
Figure 2: Program flow of a simple HapticMASTER application 32
Figure 3: Components and interfacing for application scenario in Section 3.3.1.2. 36
Figure 4: Components and interfacing for application scenario in Section 3.3.1.6. 41
Figure 5: A Look at some communications and interactions flowing within an HVA 44
Figure 6: Components of an HVA 46
Figure 7: Collection, grouping, standardization, and distribution of Unison's services 49
Figure 8: A basic diagram of Unison within an HVA 50
Figure 9: An example of Unison used in an HVAD project 55
Figure 10: OpenGL operation pipeline [57] 69
Figure 11: SensAble PHANTOM Desktop [63] 73
Figure 12: MPB Freedom 6S [42] 73
Figure 13: FCS HapticMASTER [36] 75
Figure 14: A conceptual overview of the major models in the Reachin API [47] 77
Figure 15: Typical application using the Ghost SDK [65] 79
Figure 16: Scene graph described by Table 5 84
Figure 17: Car showroom application components when developed with Unison 92
Figure 18: Screenshot from application in Section 3.3.1.3 showing haptic steering wheel 92
Figure 19: Screenshot from application in Section 3.3.1.3 showing jeep hood opening 93
Figure 20: Screenshot from application in Section 3.3.1.3 showing gear box 93
Figure 21: Execution mapping for the hood-lifting application between a Unison implementation and a HapticMASTER API implementation 95
Figure 22: A Full main.cpp File Written in Unison 97
Figure 23: Impedance control [72] 103
Figure 24: Admittance control [72] 104

LIST OF TABLES

Number *Page*

Table 1: Recent research and development efforts in hapto-visual hardware systems. 23
Table 2: Recent research and development efforts in hapto-visual software. 27
Table 3: A Table Comparing Haptic Devices Studied. 74
Table 4: How VRML and HapticMASTER API basic geometries are bound together in Unison 82
Table 5: A snapshot of an HVSG state in Unison. 83
Table 6: Impedance control vs. admittance control applications [72] 104

GLOSSARY

DISCOVER Lab. Distributed & Collaborative Virtual Environments Research Laboratory at the University of Ottawa, Ottawa, ON, Canada; the research lab where the author works.

DOF. Degrees-Of-Freedom.

GLUT. OpenGL Graphic Utility toolkit; a simple windowing API for OpenGL.

Graphic Scene. The part of the virtual environment that is rendered graphically and can be seen by the user on a computer screen, for example.

Haptic. Tactile.

Haptic Scene. The part of the virtual environment that is rendered haptically and can be felt by the user through a haptic device.

Hapto-Visual Synchronization. The necessary principle governing HVAs dictating that the user should feel force feedback through the haptic device at the same time he or she sees the virtual representation of the haptic device touch a haptic object.

HVA. Hapto-Visual Application; a computer application that engages the users senses of touch, sight and, optionally, hearing.

HVAD. Hapto-Visual Application Development; the steps necessary to create a HVA.

JNI. Java Native Interface; part of the Java Development Kit that affords programmers the ability to access platform-specific functions outside of the Java Virtual Machine.

OpenGL. The Open Graphic Library; an API with multiple bindings to different programming languages that is the common standard for graphic programming.

VRML. Virtual Reality Modeling Language; a language to describe 3D graphic scenes textually.

SDK. Software development kit.

X3D. An XML-based, 3D modeling language; the successor of VRML.

Xj3D. A VRML and X3D loader for Java

Chapter 1: Introduction

1.1. Technology in our Everyday Life

It is a great time to be a computer scientist! We are writing history and pioneering technology that will inevitably become as common place a few years down the road as the cell phone, the PDA, or the laptop are today. Every field from wireless networking, to the Internet, to technology integration, is seeing great steps taken on a frequent basis; thanks to wireless technology advancements, we can sit in airports worldwide and with our laptop be connected to anywhere on the face of this planet; the Internet allows us unprecedented access to the maybe the aggregate body of human knowledge in seconds; and the PDA has joined the cell phone, the laptop computer and the organizer, all in one. These are truly exciting times that are witnessing the aggressive integration of technology into our every day lives.

In doing so, technology must become more usable and more user-friendly; it must adapt to our needs rather than we adapt to its limitations. That is the inevitable conclusion that all technology manufacturers and innovators must realize before they achieve any prolonged and widespread success.

1.2. Interaction with Computers

In the spirit of the above argument, we look at our interaction with computers. We sit behind desks, stare at screens, and use keyboards and mice to interact with our artificial partner. It can speak to us via voice software and hardware, and we can speak back, and be understood albeit on a limited basis. Laptops sit on our laps and PDAs are in our hands but all we can do is feed them inputs through conventional means and, in return, see and hear

their output. The next progression is to be able to tactically interact with computers, and this is what the field of haptics is concerned with.

Haptics can be simply defined as the study of touch in human-computer interaction, bringing in a third dimension beyond vision and audition. The idea is simple: control motors in a physical device to stimulate the human skin and produce a haptic output. This can be done simply by vibrating a game controller in your hand when the game character falls off a cliff, or by impeding your hand's movement when manipulating a haptic device thereby making you think that you are pushing against a solid object when there really isn't one there.

1.3. Contributions

This work is mainly concerned with the practices and methods used today to develop hapto-visual applications (HVAs): applications that engage the user's sense of sight and touch (and optionally audition). At such an early stage of development[1], there are many different approaches and practices that groups and individuals around the world engage in, in developing hapto-visual applications. There really is no framework for assembly line development. And even more simply, there is no standard for haptic API programmers and hardware device manufacturers to follow, which might inevitably lead to the fragmentation and disorientation of both industry and academia members concerned with haptic research. Today, we are in the pre-MIDI[2], pre-OpenGL[3] ages of haptic development, waiting for a

[1] Haptic research has been documented in literature as far back as 1965 [7][24], however, it wasn't until the early 1990s that there were powerful enough computers and developed-enough graphic components that serious application development could take place.
[2] MIDI was the standard that united the computer audio industry and developers after the market became too fragmented for any one system to gain enough support and widespread use. See Chapter 2: Section 5.

consortium to give us a standard to follow and get all parties involved on the same frequency.

While we do not claim that this work is such a comprehensive standard, we do however present Unison: a generic framework for hapto-visual application development that is a first attempt at such effort to standardize hapto-visual application development. Unison is built on the simple principle of finding common denominators between different components, uniting these denominators in groups of generic services, which in turn are presented to the HVA developer as a standard, usable, and reusable interface.

In the process of conducting research to design and implement Unison, we have come across a wide array of technologies and tools available today which we survey in this work and present to the academic world as a time capsule containing the state of hapto-visual application development, as it stood at the beginning of the twenty-first century.

The above constitute our contributions to knowledge in this thesis.

1.4. Organization

In presenting Unison, we start in Chapter 2 by surveying the evolution of computer media in human computer interaction. This gives us a historical backdrop and provides strong motivation to push forward once reminded of the incredible leaps we have taken in the past few decades in the field of computer input/output media.

[3] What MIDI did for computer audio, OpenGL did for computer graphics; it provided a standard manufacturers and developers could refer to in developing their applications and hardware. See Chapter 5: Section 2.3.

Chapter 3 presents an overview of hapto-visual application development today. We examine the current state and different practices that individuals and groups around the world are pioneering, through a survey of a number of applications developed using different components and technologies. Our analysis and conclusions at the end of Chapter 3 lay ground to the premises Unison is built upon.

In chapter 4, we introduce Unison from a high-level perspective describing its structure, organization, and integration potential into hapto-visual application development processes.

Chapter 5 gets into the depths of Unison describing its design and implementation from a low-level standpoint. We survey technologies, tools, and components that were considered in programming Unison, and we present our choices backed with strong reasoning. We also introduce a prototype implementation of Unison and discuss the data structures and functions that make up a typical instance of the framework.

Chapter 6 shows how Unison applies to an actual application development scenario by reviewing implementation decisions in the context of an actual application, and finally, presenting some thoughts on Unison's usability value as a development tool in and of itself.

We finish with a recapitulation, conclusions, and possibilities of future work, all in Chapter 7.

1.5. Publications Arising from this Thesis

- Naim R. El-Far, Xiaojun Shen, and Nicolas D. Georganas, "Applying Unison, a Generic Framework for Hapto-Visual Application Development, to an E-Commerce

Application", Proc. IEEE Workshop on Haptic Audio Visual Environments and their Applications, Ottawa, Canada, October 2004.

- Naim R. El-Far, Xiaojun Shen, Mojtaba Hosseini and Nicolas. D. Georganas, "Unison: A Generic Framework for Hapto-Visual Synchronization and how it Operates over a Distributed Architecture", Proc. Biennial Symp. On Communic., Kingston, Canada, June 2004.

Chapter 2: The Evolution of Computer Media in Human Computer Interaction

2.1. Introduction

Any interaction with one's external environment must make use of our five senses (anomalous cognition aside[4]). These are our input media so to speak. On the other hand, we invoke our external environment with action, in the broadest sense of the word; as long as there is a receptor of, and hence a reactor to, our action, then we can invoke the environment around us through what we can think of as our own output media. These media can be anything from electric brain activity, in which case the receptor of our "action" would be an electroencephalograph's needle that moves, to just simply physically pushing an object around with one's hand, in which case the receptor would be the solid object's surface on which we push, thereby moving the object if we push with enough force. At the most basic level, the above described input and output media are how we interact with our environment.

Interaction with electronic devices - computers in specific -is no different. Today we can see, hear, and even feel generated output through computer screens, speakers, and haptic devices. We feed computers input through touch, motion, speech, and even eye movement [1]. As mentioned above, if we can make, or find, an appropriate receptor (hardware and software) for whatever action we take, we can interface the aforementioned action with a computer and use it as input; keyboards, touch screens, and haptic devices we invoke through touch; mice relay our hand movement; microphones our speech; and special tracking devices our eye movement.

[4] This is not to say that the author is a believer in the paranormal, but is definitely a believer in having an open-mind.

The relatively advanced stage of human-computer interaction (HCI) technology we are at today has, like most other things technological, evolved from very simple beginnings. Defining the starting point of HCI technology development is very much dependent on how one defines a computer and, by implication, its development timeline. The Merriam Webster Online dictionary defines the word as "one that computes; specifically: a programmable electronic device that can store, retrieve, and process data" [2]. The second part of the definition is clear and conventional, for our current time at least. The first, however, is of interest because, up until the mid 1940s, the word computer implied that a person was doing the computing, as opposed to the use of words like engine, calculator, and analyzer, which implied machine computing. The late 1930s and 1940s, in fact, witnessed the evolution of the use of the word "computer" thanks in part to the extensive funding, and therefore, the leaps and bounds that technology was taking in search for an edge during World War II. The ENIAC [3 - 5], Colossus [3 - 5], EDVAC [3 - 5], and the Mark II [3 - 5], were unequivocally called "computers" in the then-modern sense of the word.

It is hard to believe that, in only a few decades, we went from simply, yet revolutionarily, redefining the word computer to mean what it does today, to having computers be what they are today: indispensable and integral parts of our lives that we are highly dependant on.

In this chapter, and, relevantly, this work, we are not concerned with the advancements "inside the box", so to speak, but rather with how computer input and output media have evolved. We do not address the advancements in processors, memory and programming as

ends in and of themselves but rather as means of introducing the chain of evolution that brought us at a point today where our interaction with our technological companions involves three of our fives senses, namely: vision, audition and touch. In doing so, we survey the history behind computer input/output (I/O) media by examining the evolution of a few technologies chosen by the author mainly to help underline the advancements that we have made in the past few decades, as well as to provide a historical backdrop that would help put current work in the haptics field, the subject of this work, in perspective. We choose: punched cards, mice, computer generated audio and finally, but most relevantly, haptic devices, to be the media we examine the evolution of.

For the sake of having a time boundary, we emphasize developments from the mid 1940s onwards since that is when the word computer essentially stopped referring to a person and started referring to a machine.

2.2. The Punched Card

The punched card, the successor of punched tape, is an archaic, long-forgotten medium with very limited use in today's world. We have been reminded of them recently after the debacle that was the 2000 United States' presidential elections[5], but otherwise, one can hardly find them anywhere outside of a museum.

Despite their obsolescence nowadays, they were truly the first medium through which man communicated with machine (i.e. input) in a way that enabled some degree of automation.

[5] In the 2000 US elections, punched-card ballots in the state of Florida were the subject of much contention mainly because of voters not punching completely through the punched-card ballots creating errors in the count. George W. Bush won Florida and its 25 electoral college votes which were enough to win him the election despite losing the popular vote to Al Gore.

Later, punched cards also became a way of machines communicating the results of their computation (i.e. output).

The principle, on which punched cards are built, is simple; a standard, formatted blank card goes through a puncher that pierces it at specific spots each corresponding to predefined data, which thereby are stored on the card. The punched card is then stored, often in order somewhere within a stack of similar cards each with different punch patterns corresponding to different data, waiting to go through a punched card reader. Punched cards are read by passing them through a mechanism in the middle of a collection of electric circuits. By doing so, electric current flows only through the holes punched on the card closing the appropriate circuits corresponding to the appropriate punched patterns stored on the card, thereby decoding them.

Although mechanically operating devices such as looms, and later mechanical calculators, were done through predecessor punched tape and punched card technology between the late 18th and late 19th centuries, the Herman Hollerith patent that laid ground to I/O technology in use well into the 1970s, and even today with legacy systems, came about in 1884 [5]. The US legislature recognized the need to base their laws on data about the people that these laws affected and therefore the legislature mandated a general census every ten years. It took the better part of a decade to organize the data from the 1880 census into information that legislators could use, and with the rapidly increasing population, it became evident that the data from the 1890 census might not come out before it is time for the 1900 census. Hollerith, a US Census Bureau employee, built on the loom card-reader technology and the punched card was reinvented as an input medium in electromechanical

machines producing the 1890 census data in record time. Worthy of note here is that Herman Hollerith would go on to establish the Tabulating Machine Company in 1896, a direct predecessor of the International Business Machines Corporation (IBM) [5].

Advancements in punched cards technology came quickly as demand grew nationally and internationally. The dimensions of the cards have remained the same since Hollerith introduced the first 7 3/8" x 3 ¼" x 7/1000" card, which was probably so sized to fit in money boxes that were of the same width and length, but the number of columns and positions on the cards (density of the storage media) grew steadily; for the 1890 census, the cards had 20 columns with 10 punch positions in each column; in 1900, the cards had 24 columns also with 10 punch positions; and in 1910, the cards had 27 columns with 12 positions in each. In 1928, IBM introduced the patented rectangular hole; a space saving, more efficient shaped hole, in an 80 column-format card. The industry standard, however, still was the 45-column round hole card with 12 positions, so, as not to violate IBM's patent, Remington Rand, a major player in the pre-computer era, used a 6-bit code on the standard 45 column, round-hole card that logically equated that card to a 90 column one [6]. Beyond that, the punched card density did not change much, but their applications did.

By the late 1940s and early 1950s, card processing machines were being manufactured as standalone units used in business, accounting, and any other field that required considerable tabulation that justified the cost of Remington's and IBM's machines. The value of the punched card as an input and output media was evident and it carried over well to computers beginning with the Atanasoff-Berry Computer (ABC) [4], which was built between 1937 and 1942. Most other serious attempts at now famous computers that have

planted the seeds of modern day information technology involved the use of punched cards as I/O media. Such pioneering computers included: the Turing-complete Electronic Numerical Integrator And Computer (ENIAC) [5] of 1943; the British Colossus [5] of the second World War famous for breaking German ciphers; the Harvard Mark I [5] of 1944, also known as the IBM Automatic Sequence Controlled Calculator (ASCC); the binary Electronic Discrete Variable Automatic Computer (EDVAC) [5] of 1945 that was the first stored-program computer; and the 1947 Mark II [5] commissioned by the US Navy and made famous in trivia circles for having the first computer "bug" – literally a dead bug causing a relay failure during operation [7].

Punched cards continued to be a mainstay I/O media well into the 1970s, but their demise began to materialize with the rise of magnetic tapes and disks, which, although were part of the original ENIAC's storage facilities twenty years before, had just began to gain commercial momentum thanks to new innovation. So much, however, was already in punched card format that, for a time, there was the need for intermediate devices to translate slow processing punched cards into faster-access magnetic media. This kept the market for punched card technology alive until user demands for bigger timeshares (technology allowing access to a central computer via remote terminals allowing this computer to accommodate multiple users concurrently), faster feedback, and media that are more practical, got to a threshold. Timesharing technologies ushered in the effective end of punched card technology as a mainstay in human-computer interaction. Cathode-Ray Tube (CRT) terminals and teletype machines, as well as the evolution of magnetic tapes and disks, made the punched card outdated.

2.3. The Keyboard

There has not been as significant an evolution in keyboards since the 1940s as there have been in other computer input devices. For one, the QWERTY layout, still in use today, was patented in 1868 [8]. In addition, keys such as the shift and the tab as well as the numerals have been around since the time of the typewriter. Granted further functionality such as navigating the page as well as more accuracy in typing and formatting have come along as electric and then electronic typewriters were developed, but there was not much revolution when keyboards were directly interfaced with computers.

Early tele-typers from the 1940s printed to remote printers but did so directly through wire and not by way of computer, and even with the advent of computers from the 1940s onwards, punched cards (as was the case with ENIAC) and magnetic tape (as was the case with the BINAC [9]) remained an intermediary between keyboards and machines. This is not a shortcoming of keyboards per se, but of technology that only offered permanent storage devices, such as the punched cards and magnetic media, as buffers of keyboard input.

Timesharing research at the end of the 1950s brought about the idea of remote terminals and the need for quicker, direct input to computers. By the early 1970s, as terminal operation of timeshared systems came to be, keyboards had gained their biggest ground as computer input finally connecting immediately to the computer with no intermediary. Moreover, once the direct connection was made, subtle changes occurred in the architecture of the interface between the keyboard and computer having mostly to do with buffering the input, connection protocols and methods, as well as software and driver issues.

2.4. The Mouse

It is very difficult to imagine using a computer today without a mouse; user interfaces are simply built around the operator's ability to navigate content graphically through unrestricted, non-sequential access to visual areas on the screen, which is why most personal digital assistants (PDA) today have a stylus, the navigational equivalent of the mouse.

The advantages of what we now call mouse navigation were recognized by Douglas Engelbart of the Stanford Research Institute. In 1963, the first mouse, bulky, dodgy and least of all qualities, ergonomic, moved a point on a screen corresponding to the rotation of two gear wheels perpendicular to each other housed inside the mouse's body. Engelbart patented his invention on November 17, 1970 as an "X-Y Position Indicator for a Display System" [10].

Improvements on the mouse included Bill English of Xerox replacing the wheels of the mouse with a single ball that came in contact with perpendicular wheels in an attempt to lessen the bulkiness of the device [10]; the design of its modern day shape by the École Polytechnique Fédérale de Lausanne (EPFL) [10]; and the technology it used to detect movement. Optical mice work by either optical sensors detecting change in the surface underneath or by taking images of the surface underneath, process these images and accurately detect movement [10]. Most recently, laser mice, claiming to be up to twenty times more accurate than optical mice have been introduced to the market [10].

The evolution of the mouse not only centered on its physical design and technology but also functionality. Engelbart's mouse had one button, but later the standard for IBM PC-

compatible mice dictated three buttons. Mice differed before and still differ now in the number of buttons and scroll wheels that afford the user more functionality, depending on the operating system and other software in use.

2.5. Computer-Generated Audio

Giving this section the above title is a bit misleading. Perhaps we should have qualified it more by adding the word "purposefully" before "computer-generated" because computers have always produced noise, but up until the late 1950s, it was always deafening noise that came out of loud electromechanical components of printers, punchers and readers. The closest this indiscriminate racket ever came to something remotely pleasant was when engineers would, for fun, program a series of punch cards to instruct a printer to output certain lines at a certain frequency and even tempo, to produce systematic, melodic sounds [11].

Fun with printers aside, computer-generated sound is an integral part of HCI important in creating a more immersive interaction between user and computer. At first, and to some extent still, the goal of research in this field was to create and recreate music, a domain broad enough to include all types of sounds. But with the arrival of magnetic tape media and, with it, the technology to record real-life sounds, as well as the advancements in analog, digital and hybrid sound synthesizers, computer-generated sound proved to be a more diverse palate than our physical world itself, full with musical instruments, sounds of nature, and the like. And today, with the work done on speech recognition and recreation, as well as that done on improving high fidelity and high definition sound technology, our sense of hearing can be fully engaged and impressed by a computer.

There is a consensus in academia that Max Mathews of Bell Telephone Labs, New Jersey, was the first to generate sound from a computer [12]. In 1957, Mathews' Music I played a 17-second composition by one of its programmer's colleagues on an IBM 704. Eleven years later Music V, the last of the Music-N series developed firsthand by Max Mathews, offered simulated oscillators, mixers, amplifiers and other "unit generators" [13]. By the mid 1960s, the field had gained enough momentum, thanks mainly to Mathew's innovative work on the Music-N series, that specialized research centers were opening at MIT, Princeton, Stanford, and Columbia as well as internationally.

The appeal of computer-generated sound was great. Musicians were hired by research labs, most notably at the birthplace of it all, Bell Labs, where by 1963 the potential of "The Digital Computer as a Musical Instrument" [14] was argued and demonstrated often with the collaboration of musicians. The 1961 "Noise Study" [14] and the 1963 "The Dialogue" [14] were all musical compositions written with the help of mathematical and statistical functions that varied sound texture, producing music that can only be produced by a computer.

There came a point, however, where the momentum that computer music was gaining, quickly waned. The reason, pure and simple, was the wait, often measured in days and weeks, which the composer and programmer would have to endure before they could hear their machine rendition. Even with the advances in magnetic tape media, the Music V output, for example, still needed to be analogized and recorded on an audiotape before it could be heard. That was a lengthy process. Another factor that quickly put the brakes on the wide-spreading of computer music was the need for the composer to also be a

programmer and that, in and of itself, drastically cut the number of potential developers since programming was considered a very difficult and laborious endeavor.

The answer came with the birth of the synthesizer in 1964, a year that witnessed the independent invention of three types of synthesizers, two in the United States and one in Italy, all equipped with some combination of keys and envelope generators that invoked audio modules such as oscillators, noise generators, filters and amplifiers [15].

By addressing the problems of lengthy waiting and necessary programming knowledge associated with computer music, synthesizers branched out on an independent path from that of computers. The advantages that computers had over them were: a much bigger array of sounds that could be produced, precise control, better editing, and more stable storage. This, however, did not matter much because synthesizers took a life of their own even gaining parity with traditional music instruments in pop culture and bands. They got to a point where comparing them to computer music made as much sense as comparing an acoustic guitar with computer music; to the layman, it was like comparing apples to oranges. They simply were not in the same class [12].

The integrated circuits of the 1970s, however, brought digital synthesizers that could be more easily interfaced with the digital computers of the decade. This brought the synthesizer and the computer together in collaboration. Specialized computer-and-digital-synthesizer systems, such as the Synclavier [16], were introduced commercially. In parallel, research work aiming at improving the quality and function of a stand-alone computer working with a stand-alone digital synthesizer, was well underway with major manufacturers such as Kawai, Roland, and Yamaha leading the way.

The end of the 1970s saw a great increase in business-driven innovation in the field of computer-generated sound. As the market expanded, more producers came in to make more profit and, quickly, market shares split into unprofitable fragments between too many competitors with different, incompatible technologies. This fragmentation of the market hurt businesses and consumers alike and the need for standardization became evident. A consortium of manufacturers met in the early 1980s and agreed on a technical specification, they called the Musical Instrument Digital Interface (MIDI), to be the core of their technologies [17]. MIDI proved to be exactly what the consumers and businesses needed to spread computer sound technology, especially with the advent of more affordable and more "personal", micro-computers, and, in parallel, the dramatic fall in price and rise in functionality of the digital synthesizers of that era.

Computer sound generation from the 1980s onwards saw dramatic improvement in both hardware and software technology. Looking at the micro-computers – a somewhat outdated term now once used to refer to personal level, home and office computers as opposed to mini-computers and mainframes that possessed greater computational powers – the PC speaker and its Apple counterpart, ushered in computer-generated sound into our living rooms and offices around the mid to late 1980s. It could only produce monotones by playing 4-bit pulse-code modulations (PCM), using frequency modulation (FM) but thanks to software innovations, it could give the illusion of multiple tones [18]. The now defunct AdLib Corporation captured the IBM PC compatible user world with their namesake sound card [19] that could produce 8-bit mono sound and was a marked improvement over the PC speaker. Creative Labs' Sound Blaster series challenged AdLib with their own 8-bit mono sound card; the original 1989 Sound Blaster [20] that offered higher fidelity and more

features to the user. AdLib responded with their AdLib Gold [19], which failed against the SoundBlaster Pro in 1991 [20], and as a result, AdLib declared its bankruptcy and Creative Labs took over the world market with virtually no dispute [21].

Today, the top of the line Creative Labs' Audigy 4 Pro has its own 32-bit processor onboard, supports network connectivity, and supports industry standards that guarantee high-definition sound and interoperability such as DTS, Dolby and, of course, the old faithful MIDI standard [22].

2.6. Haptic Devices

First, we could see computer-generated output, and then we could hear it. The punched card, long before the invention of the cathode-ray tube (CRT), was essentially, a visual I/O medium; it could be read by anybody who knew the data format. As for audio output, bored engineers programmed noisy printers to play melodies producing computer music years before Music I ever played its first composition. Progress and innovation was underway in engaging both senses, and as computers became more sophisticated and powerful, they began to adjust to the needs of humans as opposed to us adjusting to theirs. This, by definition, is the main premise of the evolution of user interfaces, the progression of which logically and naturally opens doors on how to engage the three other senses in HCI. It is definitely not bizarre to think that one day all five senses will be immersed in a virtual environment, but in this section, we survey progress in haptics research and technology.

The first appearance of haptic research in literature is in 1965 when, in developing a robotic, artificial hand, Rajko Tomovic of the University of Belgrade experimented with pressure-responsive switches in the fingertips that were activated upon contact with an

object [23]. That same year, computer scientist Ivan Sutherland wrote "The Ultimate Display" [24], a concept paper that envisioned an immersive virtual environment in which a "computer can control the existence of matter. A chair displayed ... would be good enough to sit in".

The 1970s saw some early attempts at haptic development with the GROPE I and II projects at the University of North Carolina at Chapel Hill (UNC) [25]. The GROPE projects aimed at creating a haptic display for scientific visualization, and by the middle of the decade, GROPE II, running on an IBM System/360 Model 75 and with a 6 degree-of-freedom arm manipulator, was able to simulate resultant force from manipulating simple 3D objects, but nothing more sophisticated due to lack of computing power. The team estimated that they needed a computer one hundred times more powerful and so "mothballed" the project [25].

The 1980s saw serious haptics research especially as it applied to gaming. Atari Labs, in researching better gaming console technologies, improved on their joysticks and added force-feedback features for the first time in 1983 [26]. The 2D joystick could simulate springs, hard walls, and variant forces. A mechanical engineering student at MIT, later in 1988, wrote "The Design and Implementation of a Three-Degrees-of-Freedom Force Output Joystick", a thesis in which Massimo Russo described a 3D joystick with motors, brakes and force sensors [27]. Atari Games' 1 degree-of-freedom steering wheel, pedals and shifter boosted their game sales and by the end of the decade, research in gaming technologies was on the rise. The year 1988 also saw the introduction of the first tele-

operation application with force feedback. Warren Brodey's application [28] enabled an operator to control a remote robot's hand while "feeling" what the robot's hand felt.

The 1990s witnessed a dramatic leap in haptic applications mainly due to ever-more powerful computing resources. The GROPE project at UNC recommenced with its third installment [25] where it left off and was able for the first time to simulate the molecular structures it set out to simulate back in the late sixties and early seventies. Iwata and company, at the University of Tsukuba in Japan, drove hard at the opening of the decade developing several influential devices which were the basis of many future innovations and development standards, all before 1993 [29]. The Compact Master Manipulator [30] made the natural progression of providing force feedback to individual fingers; the Texture Key [30] was a 1 degree-of-freedom telegraph key that simulated surface contours; and the Pen [30] was just that, a 6 DOF haptic device capable of writing in 3D space as well as deforming virtual surfaces it came in contact with.

A 1996 thesis [31] by the co-founder of SensAble Technologies, then MIT student Thomas Massie, opened the door for the commercialization of haptics. In "Initial Haptic Explorations with the PHANTOM: Virtual Touch through Point Interaction", Massie introduced the PHANTOM device which, as the thesis title suggests, haptically interacted with virtual environments through a virtual representation of a pen point. The experience was to be the equivalent of a human exploring his or her physical environment with a pen, poking and prodding at objects. Research-oriented demand for the PHANTOM was great and as SensAble grew, so did the technology behind the PHANTOM line [32]. An important result of this success was the push to challenge the hardware available with

software innovations. In 1997, SensAble's Ghost SDK [32] was launched in the market to offer users access to the PHANTOM's hardware as well as later integrate graphic components directly with the virtual environment taking care of housekeeping level operations such as hapto-visual synchronization. In parallel, Immersion Corporation, a competitor of SensAble's, also found in 1993, launched DirectX FF API in partnership with Microsoft [33]. DirectX FF API gave developers access to force feedback (FF) devices for programming under Windows.

By the end of the 1990s and early 2000s, haptic applications were, if not commercially produced, at least researched, for use in fields such as medicine, training, tele-operation, manufacturing, design, and, of course, gaming.

And as we draw similarities between the evolution of other I/O media discussed in previous sections and that of the haptic media, we are confident in saying that the best is yet to come.

Chapter 3: An Overview of Hapto-Visual Application Development Today

3.1 Definition

We start out by defining hapto-visual applications (HVAs) as computer programs that, when used by a human, primarily engage his or her senses of touch (through haptic hardware) and sight (through visual hardware). We also include in this definition the optional engagement of the human sense of hearing (through audio hardware), but for the sake of brevity, and also for the relative ease of integrating audio to a strictly hapto-visual application[6], we say hapto-visual application instead of hapto-visual-auditory application. HVAs have both hardware and software components working in synchronicity to recreate a real-life interaction within a virtual environment. Typically, this is done by having the user physically interact with a haptic device, or collection of devices, while showing him, or her, the virtual environment, typically through a computer screen or a head-mounted display, and optionally having him, or her, hear it as well through headphones or speakers.

An example of an HVA can be as simple as Halo 2 [34], a video game, running on a Microsoft Xbox video game console equipped with the standard vibrating controller and displaying graphics on a TV screen. Should your character in Halo 2 be shot, for example, you would feel the controller vibrate in your hand differently than it would have if your character shot a weapon, fell, etc. Another example of an HVA, at the other end of the complexity spectrum, is the Reachin Laparoscopic Trainer [35] in which medical students

---

[6] With the exception of showing how audio could be integrated into the proposed framework in this work (discussed later in Chapter 4: Section 1), we make no attempt at accommodating audition at this stage of implementation of Unison, however, as will be shown in the aforementioned section, it is a relatively simple matter to include audition in the framework.

can be trained in abdominal endoscopy without risking the safety of a real-life patient. Students use haptic devices, optionally guided by a surgeon using his or her own device remotely, to operate in a virtual environment on a virtual patient.

3.2. An Overview of the Current State of Haptic Development

Today, industry and academia are breaking new grounds in haptic research and application development. Problems are being solved in the quest of more realistic, more powerful, and more useful products that are developed based on constantly improving technologies, development practices, and application scenarios.

Table 1 shows recent milestones reached in haptic hardware research and development, followed by Table 2, which highlights recent haptic software research, and development. Earlier works are reviewed in Chapter 2: Section 6.

Table 1: Recent research and development efforts in hapto-visual hardware systems.

Lab or Company	*Product*	*Notes*
FCS Control Systems	HapticMASTER	A 3 degrees-of-freedom haptic device that combines high position and force resolutions with an extended workspace and a maximum of 250 N of force output.
Immersion	Haptic Workstation	Immersive hapto-visual interface with goggles and haptic devices fitting over both hands with range of motion pivoting at the shoulder. The Haptic Workstation is made up of two Immersion CyberForce systems.
	CyberForce	A 6 degrees-of-freedom armature that communicates force feedback (both translational and rotational) to the arm and hand.

	CyberGrasp	A 1 degree-of-freedom exoskeleton device that fits over a hand and provides forced feedback to each finger individually.
	CyberTouch	A vibro-tactile feedback device that can fit over the fingers and the palm of a hand.
	CyberGlove	A sensor glove that measures position and bend of the hand and fingers.
	BMW iDrive	BMW-licensed software/hardware solution to navigate menus hapto-visually in BMW cars.
MPB Technologies	Freedom 6S	A 6 degrees-of-freedom haptic device with high position and force resolution. Stylus based.
SensAble	PHANTOM Omni	First haptic device cost-effective enough to be marketed directly to consumers. Stylus based.
	PHANTOM Desktop	High-fidelity haptic feedback through nominal position resolution of about 0.023 mm. Stylus based.
	PHANTOM Premium 3.0/6 DOF	Offers full arm movement pivoting at shoulder with six degrees of freedom in movement. Nominal position resolutions are: ~ 0.02 mm translational; 0.0023 degrees rotational (yaw and pitch); and 0.008 degrees rotation (roll). Stylus based.

Examining Table 1, while keeping in mind the revolutionary steps taken by the many I/O devices surveyed in Chapter 2 of this work, is very exciting. We are fortunate enough to witness the birth of a new and exciting mode of interaction with the computer and all the virtual environments that it is capable of producing. Relative to the evolution of computers

and their peripherals, and keeping in mind the growth rates of modern technology, it is fair to say that we are in the mid 1970s of haptic technology[7].

The FCS HapticMASTER [36], noted in Table 1, is a 3 degrees-of-freedom device manufactured by FCS Control Systems. The bulky-in-appearance robotic arm, alongside its box of a server unit, is of both surprising force and position resolutions. It recreates force with an only 0.01 N margin of error and can report the position of its end-effector to within 0.004 mm, virtually inconsequential to the human operator in most application scenarios. The HapticMASTER will be described in more detail later in Chapter 5: Section 3.1, but for the purposes of this section, we should note that the HapticMASTER can represent its physical end effector as any virtual haptic object of any size, the only constraints being those imposed by the device's workspace. This means that although the user is interacting with a physical end effector of any shape or design that can be fitted on the end of the robot's arm, the virtual representation of the end effector could be of any design and haptic properties. For example, if the HapticMASTER's end effector were to represent a virtual gear handle, it could be fitted with a physical gear handle end-effector making the HVA more usable, and more realistic.

Immersion Corporation's product line is quite impressive, and its Haptic Workstation [37], a truly immersive hapto-visual environment, is a sign of things to come. A video on Immersion's website [37] shows how the Haptic Workstation (comprised of 3D head-mounted display goggles, a head movement tracker, as well as two CyberForce [38] +

---

[7] The mid- to late-1970s is when personal computers began gaining momentum commercially making way for more widespread application development and the ensuing exponentially-growing involvement of computers in our daily lives. Haptic products at this time, we feel, are flirting with commercialization at the consumer level [73] which will inevitably lead the way to their wide spreading, analogous to the personal computer of the mid- to late-1970s.

CyberGrasp [39] + CyberGlove [40] setups, one for each hand) can afford a user the opportunity to virtually sit in a car, manipulate its dashboard controls with his or her hands and also turn the steering wheel. Immersion's Haptic Workstation is not only visually immersive, credit the 3D goggles, but it also allows the user control with his or her hands and fingers thanks to the, albeit bulky but certainly ambitious and realistic, CyberForce, CyberGrasp, and CybrGlove equipment. Figure 1 shows a Haptic Workstation setup.

Figure 1: Immersion Haptic Workstation [37]

Another product of Immersion, that marks a meaningful introduction of haptics into the consumer's world, is its BMW iDrive system [41]. Designed to allow the BMW driver full control of the car's facilities, everything from climate control to GPS navigation and Internet access, Immersion's round, force feedback dial that sits on the console next to the driver is sure to attract attention to the role haptics can play in similar application scenarios.

The MPB Freedom 6S [42] and the PHANTOM haptics product line [43] both implement the same physical human-haptic device interface paradigm, which also happens to be the

most common for small-scale commercial haptic devices; namely, the stylus. In this paradigm, interaction with the virtual haptic space is equivalent to interacting with one's physical space pushing and prodding at it with a pen; the haptic feedback you receive is that felt by the pen's tip. This description is not completely accurate but it is satisfactory for this survey's purposes.

Table 2: Recent research and development efforts in hapto-visual software.

Lab or Company	*Product*	*Category*	*Notes*
Center for Advanced Studies, Research and Development in Sardinia (CRS4)		Application (medical)	CRS4 has developed catheter insertion, mastoidectomy, and bone dissection applications among other HVAs.
Commonwealth Scientific and Industrial Research Organization (CSIRO)		Application (various)	Some of CSIRO main research activities are: The Haptic Workbench, a complete hapto-visual system for scientific and industrial visualization; Haptic body organs modeling; and haptic audio.
Distributed & Collaborative Virtual Environments Research (DISCOVER) Laboratory at the University of Ottawa		Application (industrial training)	An application to train technicians on how to replace faulty boards in an ATM Switch rack. The equipments used were a head-mounted display, a head motion tracker, a joystick (for avatar navigation) and the CyberForce, CyberGrasp, and CyberGlove setup.
		Application (e-commerce)	An application to interact haptically with a car in a showroom. See Chapter 6 for more details.
		Application (medical)	An application to simulate endoscopic tele-surgery.

FCS Control Systems	Haptic-MASTER	Application (various)	Robot-assisted rehabilitation for disabled patients (medical), training and simulation (aerospace), tele-operation (robotics).
HandShake VR	TiDeC	Application Support	The TiDeC Teleoperation Toolbox is one of the earliest commercial attempts in the field of haptics to (1) address haptic networking as a field in and of itself, and (2) address underlying problems that are emerging with current practices of hapto-visual application development over networks such latency, stability, and transparency.
Haptics Lab at the University of Southern California		Application (various)	Haptic robot tele-operation. Haptic rendering of seismic data. Haptic display of museum objects.
Immersion	Various SDKs	Application and Development	Specialized SDKs for programming and/or integrating haptics with various applications (e.g. Windows environment, games, medical applications, and automotive simulation).
Novint	e-Touch API	Development	A second-generation haptic API that supports several hardware platforms.
	e-Touch Sono	Application (medical)	Images taken with 3D ultrasound equipment can be digitized, given haptic properties and felt with a haptic device.
	Virtual Reality Dental Training System	Application (medical)	Hapto-visual training of dental students.
Reachin	GeoEditor	Application (oil industry)	A plug-in for the popular Earth Decision Sciences' GOCAD Earth modeling software, GeoEditor affords geologists working in the oil industry hapto-visual modeling.
	Laparoscopic	Application	Medical training through hapto-visual

	Trainer 2.0	(medicine)	guidance by an instructing surgeon. See section 2.1.
SensAble	OpenHaptics Toolkit	Development	A haptic toolkit, based on OpenGL, which targets OpenGL users affording them easy integration of haptic code with their OpenGL code.
	GHOST SDK 4.0	Development	A popular hapto-visual application development toolkit that takes SensAble from earlier first-generation versions of the SDK that only supported haptic development, to second-generation hapto-visual support.
Team GAMMA at the University of North Carolina, Chapel Hill	dAb	Application (artistic)	dAB allows the user to paint a virtual model using a haptic device.
	ArtNova	Application (artistic)	ArtNova allows the user to haptically model and apply textures to virtual 3D objects.

In examining Table 2 above, we begin with probably the most essential component of hapto-visual application development today: the haptic application programming interface (API). In order to access the facilities of a haptic device and communicate with it, developers need a haptic API provided either by the haptic device's manufacturer or by a third party. The most recent APIs today are of the second-generation; meaning that they not only give access to the haptic component of the virtual environment, but also the visual. APIs that only afford access to the haptic components are considered first-generation. Most software development toolkits (SDKs) and APIs developed today are second-generation, offering an array of features to the developer, aimed at speeding up the development process and providing higher-level programming capabilities. Chapter 5: Section 3.2 surveys haptic APIs in more detail.

From the survey in Table 2, we recognize that at this relatively early age of haptic applications development, the main areas of interest are medicine, training, simulation, design, robotic tele-operation, and a combination of the aforementioned. Furthermore, all applications surveyed above are of a very specific and limited set of operation parameters meaning that we are not yet at a stage where large hapto-visual virtual environments can be navigated fully. For example, medical training applications are limited to working with a small model of the area of interest, and Immersion's car simulator does not afford the user the ability to interact with anything other than the steering wheel, the dashboard, and the latter's components. These limitations are not software related in as much as they are confines imposed by haptic hardware range of motion and, more importantly, environment navigation issues. In emulating realistic haptic interaction, we note that our arm range of motion is maximally our arm length pivoting at our shoulders, and to physically interact with an object with our hands or arms, it has to be within this range of motion, if it is not, we can usually navigate our bodies to where it is. We carry this model to the virtual world where there exist some devices with enough range of motion to emulate our fullest pivoting at the shoulder – the Haptic Workstation for example – but the issue is navigating our virtual bodies to where an object of interest is within that range. At first glance, this might not seem as a very complex problem, and it is not as long as a developer uses conventional navigation devices such as the keyboard, a mouse, or a joystick, but to emulate walking in a haptic environment, that is a challenge (See Chapter 7).

3.3. Current Practices in Hapto Visual Application Development

3.3.1. A Survey

To better understand current practices in hapto-visual application development (HVAD), we survey example applications developed with various tools and components at a high level before going into the details of each component and tool in Chapter 5.

3.3.1.1. Feeling a Box Using the FCS HapticMASTER [44]

- Haptic Components: FCS HapticMASTER robot. HapticMASTER API v1.2.
- Graphic Components: OpenGL.
- Application Programming: C++. GLUT.

The HapticMASTER API – explained in more detail in Chapter 5: Section 3.2 – is built on top of a simple function that really is the essence of the admittance force paradigm (See Appendix I) that the HapticMASTER follows. This aforementioned function is `SetForceGetPostion()` which does just that; inputs a force vector to the HapticMASTER and, in return, after the HapticMASTER computes the resulting displacement, returns the end effector's position.

We look at a simple, "Hello World!" class example in order to get a feel for the programming of the HapticMASTER. In this scenario, a static box sits in the virtual environment rendered graphically by OpenGL on the host computer, and haptically by the HapticMASTER API on the server machine attached to the robotic arm of the HapticMASTER device. In addition, an end effector is rendered graphically on the screen and redrawn continually by a GLUT callback display function. The application affords the user control over the end effector's graphic representation on the screen by moving the

physical end effector attached to the robotic arm. Once the end effector collides with the haptic object, the user feels feedback. Worthy of note here is that collision detection is part of the API.

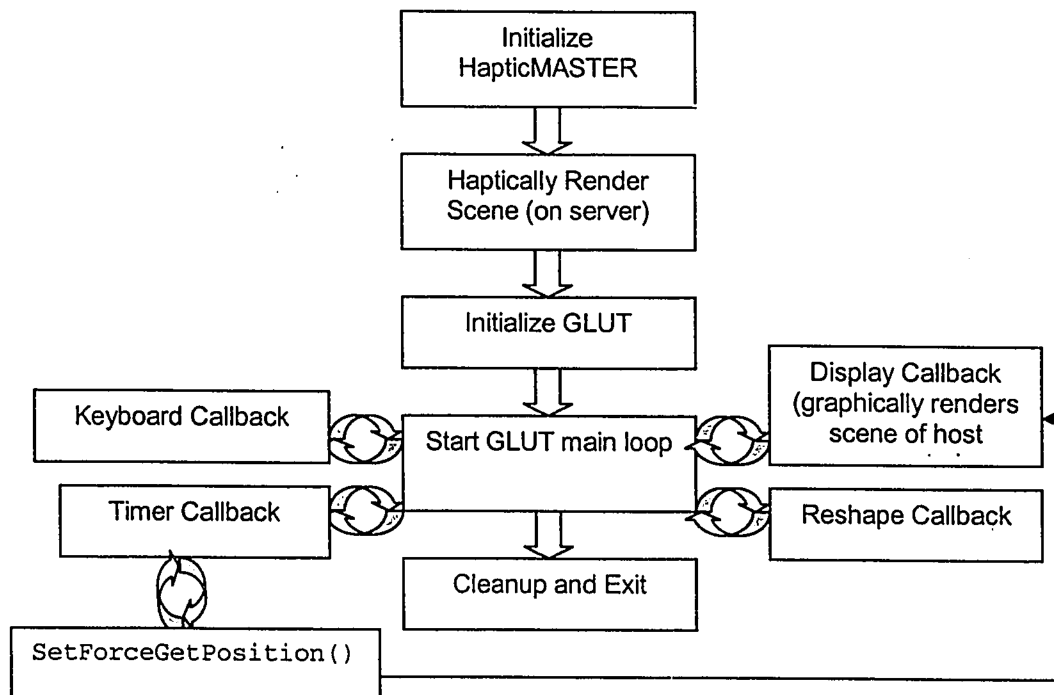


Figure 2: Program flow of a simple HapticMASTER application

Figure 2 shows the program's flow. The HapticMASTER connects to the host computer through 100/10 MBIT Ethernet, thus initializing it involves first finding the device at an IP address specified in a file created upon installation. Once the device is located and a communication channel is opened, the device's state is checked and, if it is not in the operational state (NORMAL_STATE), the device goes through necessary calibration and startup to enter the aforementioned state. Finally, the HapticMASTER's server unit's memory is cleared and initial values are set. The device is ready to run the program at this point.

Following initialization, the haptic scene is rendered in the server unit's memory after which GLUT creates the Windows interface for the user on the host computer's screen. GLUT's main loop runs with callback functions such as `glutReshapeFunc()`, which handles window resizing and `glutKeyboardFunc()`, which handles keyboard input. Most importantly though, `glutDisplayFunc()` and `glutTimerFun()`, which in turn calls the `TimerCB()` function, are ran to handle the graphic scene and the update of the set `Force` and got `Position` variables at a rate of 100 Hz.

After termination, gracefully done by user keyboard input, clean up operations take place on both the host machine and the server machine. The HapticMASTER's state is then changed from NORMAL_STATE to INITIALIZED_STATE, which locks the HapticMASTER's robotic arm until later reactivated.

3.3.1.2. Moving a Box Using the FCS HapticMASTER

- Haptic Components: FCS HapticMASTER robot. HapticMASTER API v1.2.
- Graphic Components: VRML v2.0, CyberGarage CyberX3DLoader, OpenGL.
- Application Programming: C++. GLUT.

In this example application, the author adds a dynamic features to the HapticMASTER operation, which is a non-trivial matter since the HapticMASTER's API does not explicitly provide for dynamic manipulation of objects; again, it is basically based on the `SetForceGetPostion()` function and that is how the developer is to add dynamic features to otherwise static, hapto-visual scenes.

We present two ways of achieving the above objective. The first allows for the end effector to move around the scene latching to "hot" points which can be thought of as physical handles. Once the end effector latches onto a hot spot, it can move the object in question around. To "let go" of the object, the user can either use one or a combination of keyboard and mouse inputs. Alternatively, the user can yank the end effector away from the object, thereby releasing it. These hot spots can be programmed in several ways including creating force fields, springs, or just basic permeable haptic objects.

Another way of moving objects around using the end effector is to couple it with the hapto-visual object at programming time meaning that the hapto-visual object in question has restricted values (for range of motion purposes) for position, orientation, etc., controlled by the position of the HapticMASTER's physical end effector.

To move a box, using the first method, we specify the way the free-roaming end-effector can interact with the box, and from what points it can interact with the box. For the purposes of this application, we set the interaction parameters with the box as follows: Once the end effector comes in contact with any point on the box, it can push or pull on it in any direction. This necessitates that every point on the box be a "hot" point; a point where the end effector would latch onto the box and manipulate it. We also create an invisible permeable bounding box (meaning only displayed in the haptic scene and not in the graphic scene) around the hapto-visual box that has dimensions slightly larger (~1cm) than the bounded box. We couple the positions of both the bounding and the bounded boxes together and program the keyboard callback function to look for enable/disable commands from the user that instruct the run-time process on when the user wants to latch

to the box or let go of it. Once the end effector enters the permeable bounding box, provided it is enabled, then it becomes trapped between both boxes and the display call back function will move both boxes and the end-effector's virtual representation around according to the user's manipulation of the end effector. The user can unlatch by yanking on the end effector, which would communicate force greater than the inside spring stiffness of the bounding box, thereby releasing the end effector from between both boxes. Alternatively, the user can disable the bounding box from the keyboard.

Locking the end effector at one point on the box as opposed to allowing it to move freely between both boxes is a matter of dynamically creating a stiff haptic object around the end effector that prevents it from leaving the latch point unless otherwise provided for with keyboard or mouse input for example.

Another aspect of this application scenario of great importance to us is the creation of the graphic scene. Using the CyberGarage CyberX3DLoader [45], the developer can parse and draw a VRML 2.0 [46] compliant file to the screen. Graphic rendering is done with OpenGL. Graphic scene graph handling is discussed in Chapter 5: Section 2, but for now Figure 3 shows a high level block diagram of the program's main components. As explained before, and as is the case with any HVA, at least three components must interact synchronously to create a hapto-visual environment. The driver in the middle is the C++ code that invokes CyberX3DLoader services to create a scene graph out of the VRML file which is then graphically rendered with OpenGL (through CyberX3DLoader) and displayed using GLUT's windowing services. Synchronization between the haptic and visual scene is done inside the main application, which also initializes the HapticMASTER

device and communicates back and forth with the haptic scene graph through the HapticMASTER API.

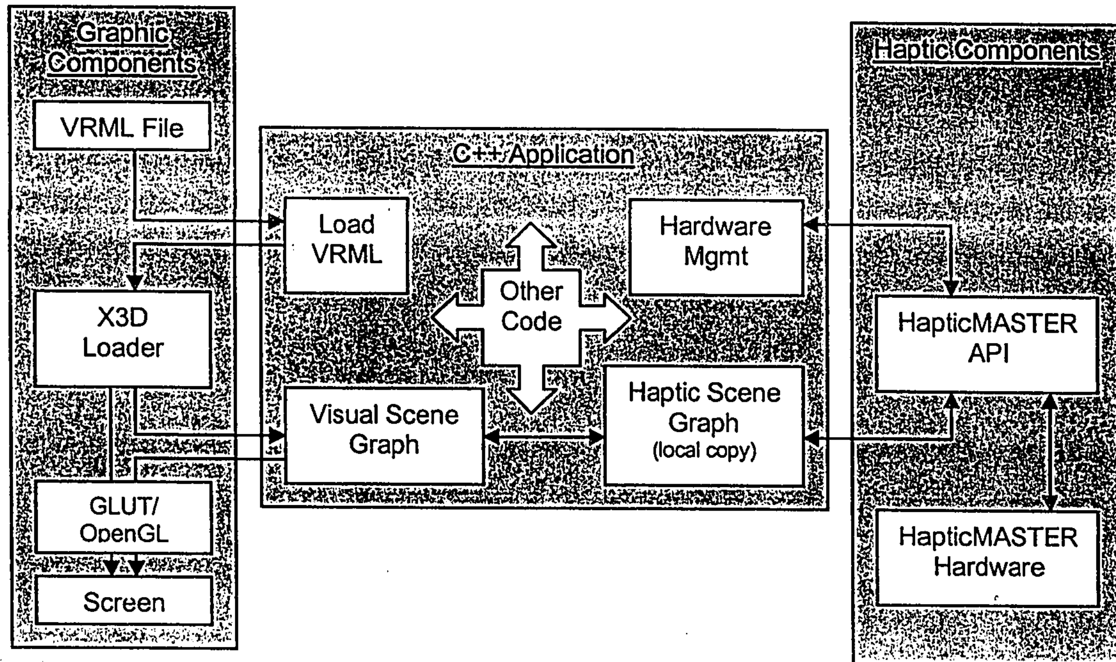


Figure 3: Components and interfacing for application scenario in Section 3.3.1.2.

3.3.1.3. Opening a Car's Hood, Turning its Steering Wheel, and Shifting its Gear Using the FCS HapticMASTER

- Haptic Components: FCS HapticMASTER robot. HapticMASTER API v1.2.
- Graphic Components: VRML v2.0, CyberGarage CyberX3DLoader, OpenGL.
- Application Programming: C++. GLUT.

Section 3.3.1.2 above explained how we can add dynamic manipulation using hot spots and bounding haptic objects to otherwise static hapto-visual scenes developed with the HapticMASTER API and loaded from VRML files. This section shows how dynamic features can be added through coupling position information of the physical end-effector

with position information of a hapto-visual object. This implementation still follows the components and interactions diagram in Figure 3.

The aforementioned dynamic feature is simple in concept but not in implementation, especially if the developer wants to recreate a true-to-reality model. Coupling the end-effector's physical position with the position of the hapto-visual object in the virtual environment essentially makes the end-effector a fixed handle on the object. Implementation wise, this is done by, instead of drawing the end-effector as a small round sphere, for instance, drawing it as the hapto-visual object in question setting the force vector to represent the gravity forces, for instance, acting on the box. Also, the developer has to define a fixed range of motion for the end-effector to simulate that in real life. The aforementioned two challenges are not trivial since, firstly, moving a fixed mass object in different speeds and directions implies different forces acting on the object, and secondly, calculating a real-to-life fixed range of motion is also an involved computation.

The above challenges are addressed in the implementation of a hapto-visual application using the HapticMASTER to lift a car's hood, turn its steering wheel, and shift through its gears. In the case of opening the car's hood, the end-effector is coupled with a virtual handle at the front center of the hood, therefore, when the haptic device is lifted, the hood also is, but in a virtual arc-like range of motion. Turning the steering wheel is done by coupling the end-effector with an arbitrary point on the virtual steering wheel and constricting the range of motion to within a hollow torus of real-life proportions. Finally, and most simply, the gear shift scenario predictably couples the end-effector with the virtual gear knob.

This particular application scenario, created by the author, is further discussed in Chapter 6.

3.3.1.4. Haptically Interacting with a Topographic 3D Model of Earth Using the SensAble PHANTOM Desktop – Using High-Level Python Scripting Support in Reachin API

- Haptic Components: SensAble PHANTOM Desktop. Reachin API v3.2. Reachin Display setup.
- Graphic Components: VRML. OpenGL.
- Application Programming: VRML. Python.

Reachin API [47] is probably the easiest API to get started developing with today. It offers the developer two venues to program; the rapid-application-development (RAD) and prototyping model discussed here, and the full fledged programming model discussed in the following section.

Using an executable, `reachinload.exe`, Reachin API allows the user to load hapto-visual scenes described in a modified VRML file that contains extra haptics-related nodes. Functionality for such a hapto-visual scene can be added through Python scripting, pointed to from within the modified VRML file and invoked by `reachinload.exe` at run-time.

The example discussed in this section is part of the demo sampler that comes with the Reachin API software. A modified VRML file describes a hapto-visual scene of a topographic, haptic model of the planet. The VRML file also points to a python script that adds the dynamic features to the application necessary for user interaction with the model. The user can freely rotate the model in place around different axes and feel the topography that makes up the planet using the PHANTOM Desktop device [63].

In examining the modified VRML file that describes the hapto-visual scene, we point out a few examples of Reachin API nodes: The `Display` node is the hierarchical parent node of all visual nodes in the scene which, if not explicitly programmed, Reachin would automatically add; the `Dynamic` node specifies the physical parameters (e.g. angular velocity and inertia) that the model abides to once is manipulated by the user; `Surface` nodes in all their types define the surface parameters of the haptic object such as friction, stiffness, etc.

3.3.1.5. Haptically Interacting with a Topographic 3D Model of Earth Using the SensAble PHANTOM Desktop – Using C++ Support in Reachin API

- Haptic Components: SensAble PHANTOM Desktop. Reachin API v3.2. Reachin Display setup.
- Graphic Components: VRML. OpenGL.
- Application Programming: VRML. C++.

As opposed to the RAD model above, Reachin API offers a full fledged and powerful development tool for HVAs using C++ and VRML.

The hapto-visual scene graph in Reachin API, whether Python implemented or C++ implemented, is the same. It follows more or less the VRML 2.0 model stripped of some irrelevant nodes (e.g. user interaction sensor nodes) and complimented by other Reachin-specific nodes (e.g. surfaces, haptic properties, etc). So when writing C++ code in Reachin API, the developer has an option of either loading a scene graph directly from a VRML file, as is done in this scenario, or creating a scene graph from source code based on VRML standards. The latter is less practical because it requires, among other things, recompilation

every time an object changes in the scene as well as plenty more code writing. The former is more widely followed because it separates the virtual environment's description (i.e. the VRML file) from the application code. Naturally, the developer needs full access to the objects of the virtual environment and the Reachin API provides that within its function libraries. Since the VRML file is loaded before it is continuously rendered haptically and graphically, modifications to the scene can be programmed from the C++ development environment.

Relevant to our discussion here is to mention that Reachin API is highly extendible thereby affording the developer customizable, higher-level haptic and graphic facilities, on top of those already built into the API that include hardware querying, collision detection, coordinate space transformation, interpolation, force modeling, high surface and texture customizability, and expandability of geometric objects.

3.3.1.6. Haptic Tele-Surgery Application Using MPB Freedom 6S [48]

- Haptic Components: MPB Freedom 6S. MPB Freedom 6S API.
- Graphic Components: VRML. Java3D. Xj3D.
- Application Programming: Java. JNI.

In examining this application, we see a wide shift to other development tools than the ones we have examined so far. This application has its graphics in VRML which are loaded into a Java/Java3D development environment using Xj3D, an X3D/VRML file loader for Java. Java 3D renders the graphics to the screen, and through Java Native Interface (JNI), the Java development environment can access the Freedom 6S API's C++ functions. The components and interactions in this scenario are shown in Figure 4.

This implementation has many performance bottlenecks. Via-JNI access to the Freedom 6S API is one, and Java3D rendering graphics from a via-Xj3D loaded VRML model is another. Furthermore, the excessive number of interfacing between components is challenging implementation wise especially when looking at the JNI interface. More basically, Java and C++ performances as visualization (and by extension, haptic) application programming languages is discussed in Chapter 5: Section 2.

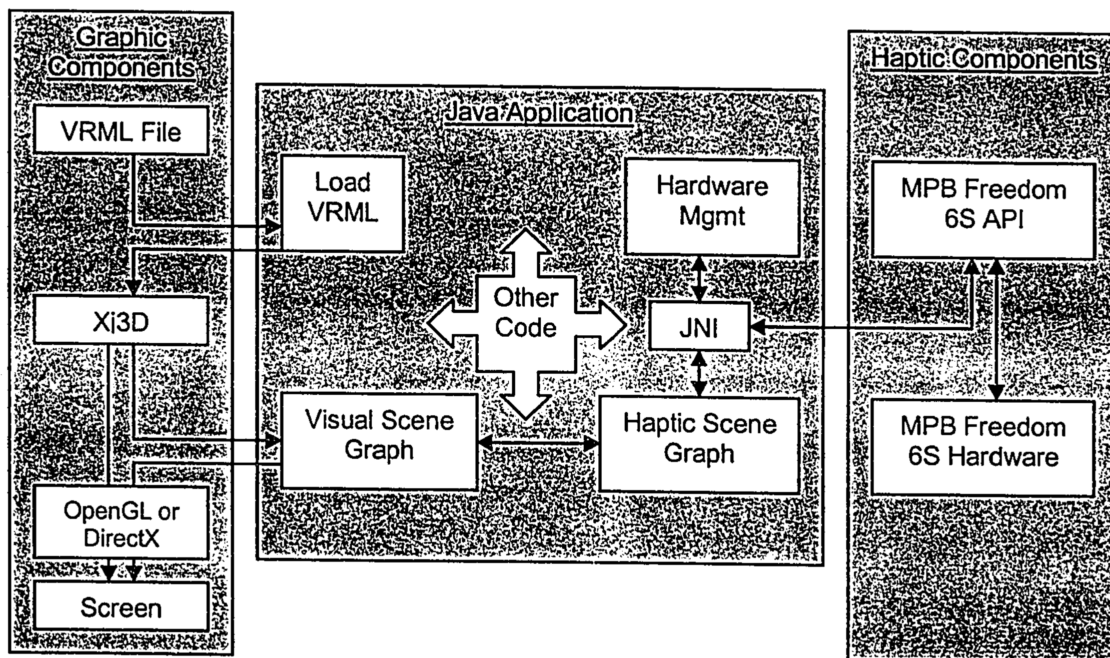


Figure 4: Components and interfacing for application scenario in Section 3.3.1.6.

3.3.2. Analysis

After reviewing the above examples, we can make some conclusions about the basic components of any hapto-visual application.

The haptic scene is that part of the hapto-visual scene graph that displays, or haptically renders, those objects that are assigned haptic properties while the graphic scene is that part

of the hapto-visual scene graph that displays, or graphically renders, those objects that are visible to the user's eye. In a virtual environment representing a living room, for instance, we can see the couches, the tables, the TV unit and the AC unit; those comprise the graphic scene. Haptically we can touch all objects but the AC unit which is out of the avatar's reach; those touchable objects comprise the haptic scene. Typically, both scenes will overlap for most application scenario purposes but they don't necessarily have to.

In the applications above, we have seen different ways of describing and interacting with a graphic scene graph from within the application code. Unmodified, VRML 2.0 compliant files can be loaded into the development environment using Xj3D for Java/Java3D and CyberX3DLoader for C++, as well as written with appropriate modification as is the case with Reachin API's C++ support. Modified VRML could also be loaded into Reachin API as a static scene, or a dynamic scene requiring Python scripting. Although more verbose and less commonly used, Reachin API supports from-scratch scene building within C++ based on the VRML model.

Loading and interacting with the haptic scene graph into the development environment is much more specialized and depends greatly on the haptic device and API in use. Whereas VRML, for instance, is a generic language to describe a graphic scene, haptic API-specific description methods are necessary to create the haptic scene. Reachin API bases its haptic modeling facility on VRML, simply adding exclusive nodes that describe everything from surface stiffness to interaction forces. The HapticMASTER API provides its own way of creating haptic objects from a combination of generic basic objects with properties. So does the Freedom 6S API.

Hapto-visual synchronization is another aspect that lies in the details of the APIs. Reachin, in that it uses one scene graph to combine both the haptic and graphic scenes, really does not need to synchronize both because they are rendered graphically and haptically together, albeit at a different rate, but sill unnoticeable to the human eye and touch, hence irrelevant. The HapticMASTER, on the other hand, leaves the synchronization up to the user. An independent thread loops until program exit, inputting force to the device and receiving position information from it, storing both globally for other threads to see, update and use. Synchronization is therefore done by reading shared global variables.

In a hapto-visual environment, not only can we see and touch the objects around us but we should also be able to interact with them in ways other than static touch. We should be able to move them around, push on buttons, open doors, close windows, etc. For that we need an event manager of sorts that governs dynamic interactions. From the survey in the previous section, we saw different ways of doing that, all highly dependent on the API and all subtly related to the way the haptic and graphic scenes are synchronized. The more hidden the synchronization process is from the developer; the easier it is to find functions to govern haptic interaction. For instance, Reachin API hides the synchronization completely from the developer by having both haptic and graphic scene graphs store in one multi-sensory scene graph, therefore haptically interacting with one graph, updates the other seamlessly. On the other end of the spectrum, the HapticMASTER API leaves the synchronization solely up to the user; therefore it is troublesome to write dynamic interaction functions.

3.3.3. Conclusion

Figure 5 shows some of the complexity associated with HVAD by diagramming common pieces of applications surveyed in Chapter 5. Not impossibly complicated as shown, but it is easy to see how the diagram can get more complex as other layers are added and more devices are added (e.g. Immersion Haptic Workstation). The core application has to interact with most every other component in the application, and in doing so, sometimes needs an interface such as a file loader or a native language translator. The haptic components of the diagram below also need to interact with the core application and the graphic components in order to achieve synchronicity. Windowing is an issue, as is rendering the graphic scene. Accessing the modeling files is an issue as is tackling a way to represent the haptic scene graph internally. In short, interfacing is an issue.

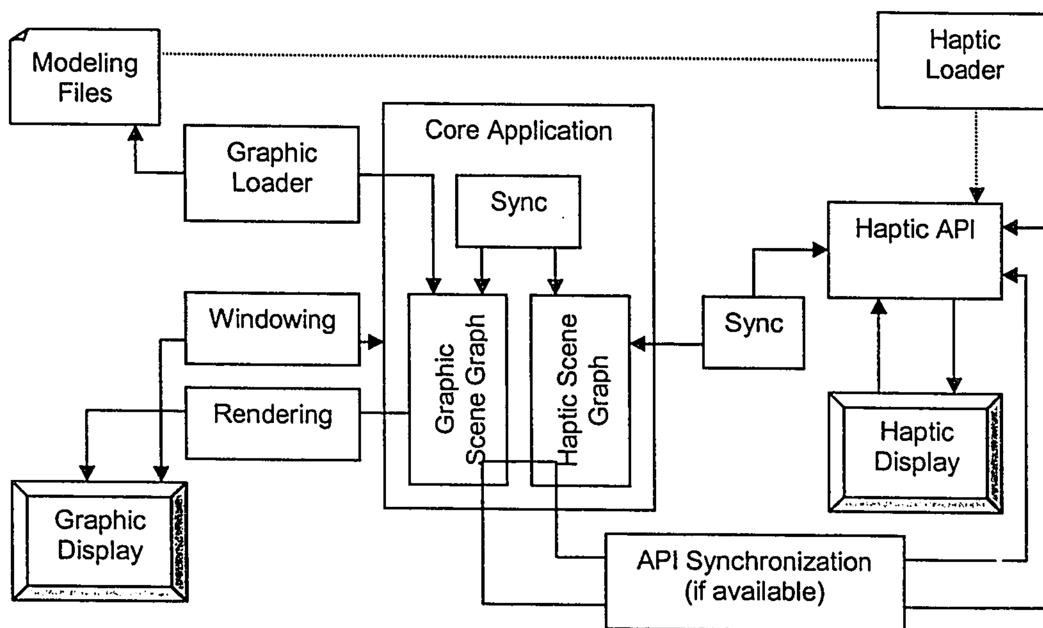


Figure 5: A Look at some communications and interactions flowing within an HVA

From our survey in Chapter 3, we have seen that practice has so far been to write custom code for each application developed with little consideration for or even possibility of reusability. If, for example, a project's graphics were in VRML, the application was coded in C++, the haptic device used was the PHANTOM Desktop and the haptic API used was Reachin, then whatever interfaces were written for that specific application could not be reused without major modification, if not complete rewriting, for a project that had its graphics in OpenGL, its application code in Java and used the HapticMASTER API. The problem is the specificity of the data structures that must be exchanged among the components of the hapto-visual application and the function calls that are afforded by the haptic API as well as the infrastructure the program flows over.

From the above discussion, and taking a wider angle look at the survey abovementioned, we can find similarities in the HVAD process that allow us to cleanly conceptually compartmentalize all parts of a hapto-visual application. We suggest the classification shown in Figure 6. At the bottom sits the core programming language (e.g. C++ or Java) which controls the flow of the program and coordinates most, if not all, communication among the other components. On top of it, is a layer, optional in part or whole, that interfaces the program code with API services, functions, and libraries. Xj3D is an example support software that loads VRML files into a Java/Java3D development, as is JNI that interfaces Java program code with C++-authored APIs. On top of the support layer sit the two other main component layers; the haptic and graphic. Within these components, A-Z services must be provided to create, load, render, update, save, get, and set most all objects that end up being drawn to the screen or to the haptic device.

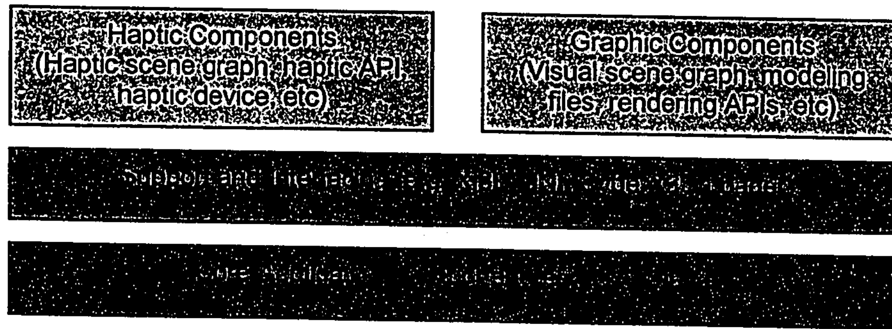


Figure 6: Components of an HVA

Chapter 4: A High-Level Look at Unison

4.1 Introduction

In addressing the complexities of highly customized HVAD processes discussed in Chapter 3: Section 3, we present Unison: a generic framework that serves to connect the core application software with the graphic and haptic components in order to create a structured and well-defined hapto-visual application. Unison's main premise is to provide generality and reusability by finding the greatest common denominators in HVA components. These common factors are grouped, or united, hence the name Unison, with those which are similar and shared, under generic service and function names, usable and reusable within a structured and ordered framework, with the purpose of creating a faster, better organized hapto-visual application development process.

To further explain the principle above, consider the following observations, made in the context of a hapto-visual application, which give rise to some later named services in Unison; most popular 3D modeling languages describe graphics in terms of nameable groupings, logical and object-oriented in the literal sense of the word, that have physical properties such as geometry, dimensions, and color. A collection of such groupings constitutes a graphic scene. In parallel, a typically overlapping group of the graphic scene objects can also have groupings but with haptic properties such as stiffness and elasticity. This subgroup constitutes the haptic scene. The core application naturally needs to be able to reference all objects in the graphic and haptic scenes to provide the functionality intended by the application.

These observations before mentioned are examples of observations global enough to provide a solid foundation for a framework of services that can truly be deemed generic and reusable. For example, if we were to have a function that would load a visual scene graph from a modeling file into an application development environment, then there is a great modularity and reusability value in having the same function be able to load graphs from different file formats, given, as discussed above, that these formats are similar – for example that they be text-based, meta-data, modeling files such as VRML or X3D files (discussed later). Unison does offer such a function that is intended to be able to load VRML or X3D files alike; `LoadGraphicSceneGraphFrom(filename)` can be called by a developer using Unison to load any modeling file supported by the framework into the development environment such that all nodes in the visual scene graph can be accessed, updated, or rendered to the screen. `LoadHaptoVisualSceneGraphFrom(filename)` is another Unison service that would similarly give the developer access to a complete hapto-visual scene from a modified VRML file, for instance, that would not only have the standard VRML nodes, but also haptic nodes (See Chapter 5: Section 3.2). By the same token, `SetGraphicProperty()` and `SetHapticProperty()` are two other example services that assign property values to property fields for, respectively, a graphic and haptic object.

Figure 7 shows the process which identifies common functions across different haptic APIs, groups them together, represents them as a service in Unison, and provides that service within Unison's plug-ins (later discussed in Chapter 4: Section 2).

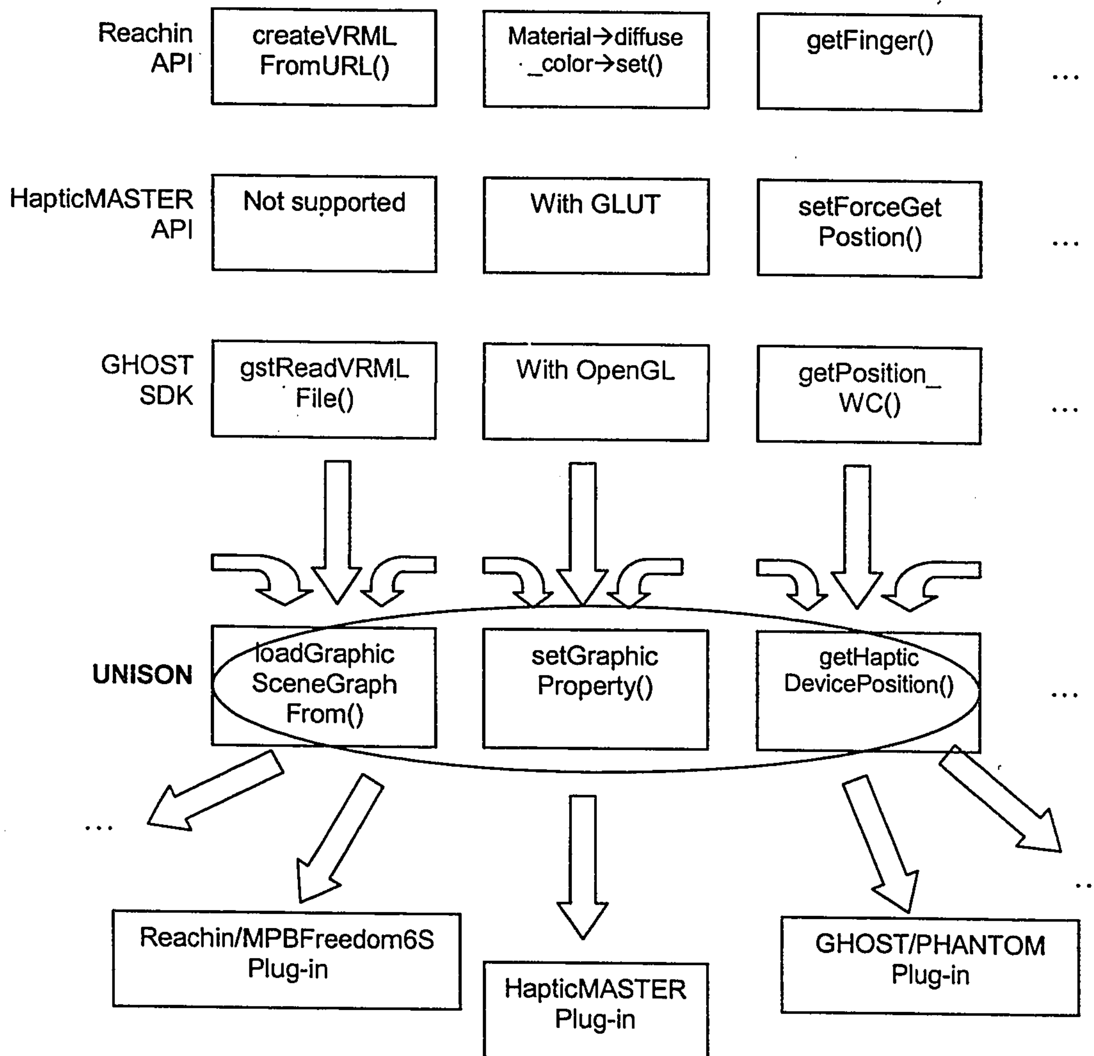


Figure 7: Collection, grouping, standardization, and distribution of Unison's services

The aforementioned services are examples of the essence of Unison's functionality; through finding common denominators in the hapto-visual application development process, Unison offers generic functions that, regardless of underlying technology, are called the same from a user's development environment, with the only difference being an overloaded parameter list.

Of note here is that there is no mention of audition, but this is not to say that it has not been considered. We have already defined an HVA as an application that optionally engages the

sense of hearing, in addition to the necessary engagements of the senses of sight and touch, so strictly speaking, an HVA does not require audio support. This, however, is not difficult to add for the very simple reason that event-triggered audio (e.g. sound resulting from collision) can be supported by invoking an audio file when the API, or Unison in future implementations, detects collision under specific parameters. For example, should collision be detected between two objects in the virtual world –maybe one of them would be the haptic device's virtual representation, but not necessarily so– a sound file could be played. Background sound is an even simpler matter; a thread can play a sound file in parallel to the application running. Future work in audio support is again referred to in Chapter 7.

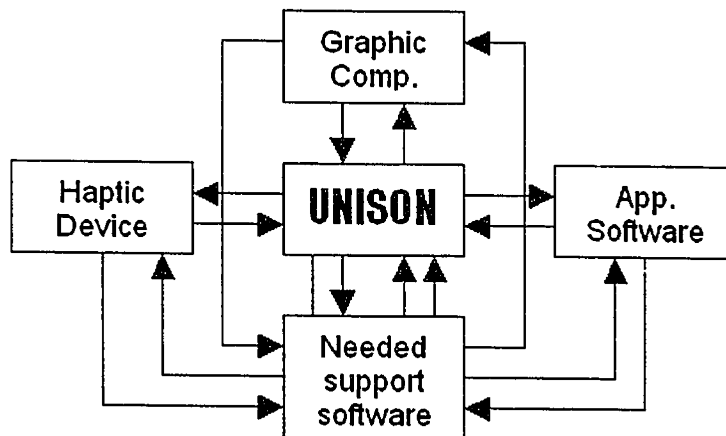


Figure 8: A basic diagram of Unison within an HVA

Figure 8 shows Unison's conceptual location within an HVA. Unison's services are called from the core application and tend to the application software's interfacing needs whether they are in: communicating with the graphic components, for example by loading a modeling file, updating the position of a graphic object in the scene graph, or initiating an animation sequence; or the haptic components, for example creating a haptic object,

changing an object's surface stiffness, or querying the haptic device for its current state; or support software, for example invoking a model-file loader or windowing APIs; or, finally, a combination of two or more of the three components mentioned, for example hapto-visual synchronization, and callback function programming. Worthy of note here is that the communication between the application software and needed support software, is strictly non-hapto-visual related; that is left solely to Unison. An example of the need for such communication between the application software and some support software, is that which arises should there be a need for network support, not provided for by Unison, hence the need for external support software.

4.2. Unison's Integration into an HVAD Environment

Unison, pure and simple, is a collection of services and functions grouped into header and library files to be included into a working HVAD project depending on the project's choice of haptic and graphic components.

As mentioned in the previous section, Unison groups different component-specific functions, which share a purpose, under one generic name that solely would concern the developer. So in effect, it is an abstraction layer hiding the specificities of lower-level programming, which are left to the author. In approaching the issue of how to organize the cogs and bolts of Unison –intricacies that are invisible to the developer, so that they could interface with, for example, the vast array of haptic APIs, different modeling file specifications, and haptic device hardware-level functionality– we saw best to follow a plug-in model, in which for every haptic or graphic component supported by Unison, a plug-in is provided that interfaces with the given component translating Unison's `set's`

into the component's syntax and organizing its `get's` in Unison's data structures. For example, Unison's plug-in for the FCS HapticMASTER populates a list of haptic objects in the haptic scene different than the PHANTOM Desktop plug-in would; both APIs have different basic geometries with different names and different access functions. However, the list being populated, an example of a Unison data structure, is the same.

It is important here to properly define a plug-in as it pertains to use in Unison, as opposed to its common use in programming elsewhere. Unison's plug-in model is loosely based, at this stage, on the definition that a plug-in simply is an addition to an existing piece of software that adds to its functionality primarily by allowing it to interface with third-party software – this is what we mean when we say that Unison uses a plug-in model to interface with different haptic and graphic components. In this sense of the word, a Unison HapticMASTER plug-in is similar to an Internet Explorer VRML plug-in; both plug-ins afford support for third-party software (the HapticMASTER API and VRML respectively) to the main application (Unison and Internet Explorer respectively). The mechanism by which this support is afforded, however, is very different between the two examples mentioned.

The intricacies of plug-in frameworks are beyond the scope of this work and vary among different implementations, however, we mention the following features typical to a plug-in framework such as that used in Netscape Browser [75], Eclipse [76], or the plug-in framework for C++ suggested by [77]: Late binding implies the creation of objects dynamically at run-time and later loading their methods when needed, so although the interface to call a function is visible, it is not loaded until invoked as is the case when using

JAR[8] and DLL[9] files; Service registration is another feature common to plug-in frameworks that calls for new plug-ins to reveal their functionality, through a predefined interface, to the main application allowing that main application, again through a predefined interface, to call methods within the new plug-in such as would be the case when a Flash[10] plug-in is installed in Netscape Browser and identifies itself as capable of viewing Flash files; Predefined interfacing, before mentioned, lays the basis for modularity in a plug-in framework by making sure that both the plug-in and the main application exchange data and control correctly in a predefined manner, and finally; Extension points in plug-in frameworks define locations where interconnectivity typically between different plug-ins could be established by calling extensions to the local plug-in as would be the case when having a plug-in require the services of another.

The above characteristics of typical plug-in models in use today are not all available in Unison at this stage (see Chapter 7). Unison is a step away from employing binding, that would be achieved by delivering its service implementation in DLL format for instance in a C++ programming environment. Service registration has neither been tackled nor is deemed necessary because Unison is not a real-time utility, but rather a development tool so there is no need, or place, for a plug-in to declare itself real-time to any application built using Unison; the only "service registration" is done by the user when he or she reviews the available services at development time. Predefined interfacing is a main premise of Unison, since that is how it derives its modularity, and is achieved primarily via overloading parameter lists while keeping service names generic. This approach does lower the

[8] Java ARchive (JAR) files are zipped files that ease the distribution of Java compiled classes.
[9] Dynamic Link Libraries (DLL) are executable function code modules invoked when necessary application code to which the DLL is visible.
[10] Macromedia Flash is a file format for graphic animation.

abstraction level that Unison proposes by dictating the use of specific parameter lists (see Chapter 5: Section 5), however that could overcome by better designing the interface (see Chapter 7). Finally, inclusion of extension points in Unison is deemed unnecessary under this current implementation because of the lowered abstraction level imposed by Unison's approach to plug-in interfacing; as it stands today, for a programmer to use Unison in his or her application, he or she must include the appropriate haptic and/or graphic Unison files that are organized as all-inclusive modules (i.e. include all necessary code that the API offers and is supported by Unison) meaning that the HapticMASTER and the Phantom 6S plug-ins would share a lot of the code in between them effectively overlapping in all but the API-specific functions. This leaves the need for the developer to be aware of the differences in required parameter lists between different plug-ins (which lowers the level of abstraction), but does away with extension point programming because of the all-inclusive nature of the plug-in files. It leaves no need to extend the plug-in and ask it to call anything but functions within itself.

Figure 9 shows the Unison plug-in model described above by presenting an example organization of an HVAD environment in which the core application is in C++, the graphic scene graph is loaded from a VRML file, and the FCS HapticMASTER and the HapticMASTER API v. 1.2 comprise the haptic components section.

Note that the developer is not concerned with the workings of any component below Unison. Unison's data structures, most importantly the synchronized hapto-visual scene

graph, look the same[11] to the developer regardless of Unison's plug-ins used. The developer only has to, in this example, specify the VRML file needed, and include necessary Unison files in his or her project.

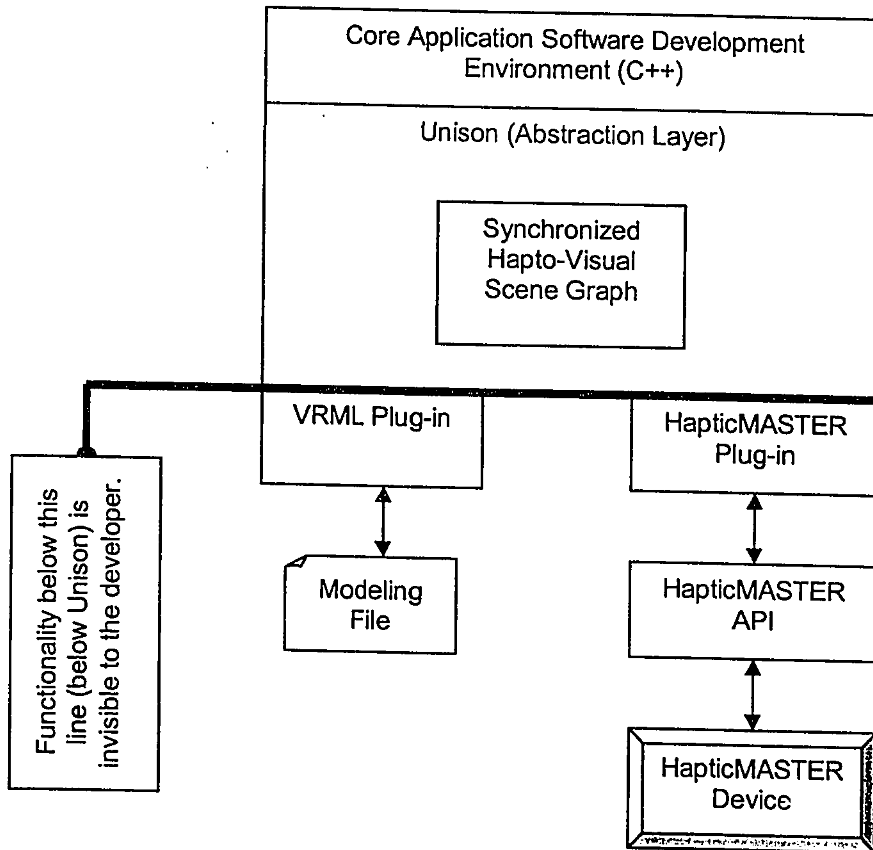


Figure 9: An example of Unison used in an HVAD project

4.3. Unison's Extensibility

At the time of writing this work, implementation has been completed for prototype Unison data structures to hold, and support holding, graphic and haptic scene graph information in one, synchronized multi-sensory scene graph. Also a prototype for a graphic component

[11] There is no escaping the need to have specific fields in Unison data structures unique to certain APIs, for instance. However, the abstraction model is not broken, because accessing these fields is still done with the same functions, albeit with an overloaded parameter lists.

plug-in has been implemented to load VRML files into a C++ development environment, and a prototype haptic component plug-in has been implemented to interface with the FCS HapticMASTER and the HapticMASTER API v. 1.2.

A list of functions supported as well as more details on these functions and on implementation specifics will be presented in Chapter 5.

The fact that Unison is built around a set of data structures visible to both Unison's services – and called from within them – as well as to the developer using Unison, extending the service set is a matter of interfacing with these data structures and perhaps adding to the them as the need arises. Support for other graphic and haptic components, available today or in the future, is simply a matter of programming to populate the lists that make up Unison. For example, in writing a plug-in to support the PHANTOM Desktop communicated with via the Reachin API, a developer who wishes to extend Unison has to simply find Reachin API functions that would return relevant information about haptic objects in the scene graph and put them in the multi-sensory scene graph data structure that is the cornerstone of Unison. Unison's developer should also provide a way to update a hapto-visual object in the scene graph, so a set method must be afforded to update position, orientation, size, or surface stiffness, for instance.

This modularity and structure in Unison's make-up allows for a high degree of extensibility, since, as things stand now, all hapto-visual applications have a haptic scene and a graphic scene that need to be synchronized, and updated and added to and deleted from, etc; all changes reflected in the rows of Unison's multi-sensory scene graph.

Chapter 5: The Design and Implementation of Unison

5.1. Introduction

After introducing Unison in the previous chapter, we take a closer look at its components in detail.

In order to service the haptic and visual components of an HVA, Unison must have haptic and visual component plug-ins as well as a core programming language of its own to write the code for the aforementioned plug-ins.

In this chapter, we start out by surveying available developments tools that have been considered for implementing Unison. We first look at possible programming languages to write Unison's code in. Next, we look at graphic programming tools and how they complement each other within the plug-in paradigm that Unison follows to support different underlying technologies. Finally, as we have done with the graphic component survey, we also look at haptic hardware and software explaining at the end how they fit, again, into the plug-in model that Unison uses to support different haptic components. Finally, we present some of Unison's functions and data structures implemented in prototypes functional at the time of writing this work.

5.2. Application Programming Language

The core hapto-visual application's programming language is a fundamental part of the HVAD process. Integration of all different parts of an HVA takes place within its lines of code; therefore it must be efficient, supported by the components, and of high performance in terms of loading time, memory management, data structure management, etc. The

aforementioned performance criteria carry over faithfully to the programming language chosen to write Unison's code in.

In looking at languages to use as HVAD tools, we considered C++ and Java; both high-level, third-generation, supported, powerful, and established programming languages in use today.

In surveying different haptic APIs, described later in this chapter, we found that C++ was almost always the language API developers chose, so we were inclined to use it right from the beginning. However, Java merited serious consideration also, primarily because of already available, open-source graphic components built on top of Java/Java3D on hand in the DISCOVER lab.

5.2.1. C and C++

In the early 1970s, and in developing applications of different levels for the UNIX system, Bell Labs' scientist Dennis Ritchie designed the C programming language to be an efficient tool that was of high-enough level to be developer friendly but low-enough level to afford power and access of the machine's facilities to the developer [49]. Over the course of the first years of its life, C was revised extensively before presented in "The C Programming Language", a book published in 1978 formally introducing C as a viable and robust tool to the world [50]. In gaining popularity, C reached a threshold where it needed to be standardized, and indeed it was in 1989 under the auspices of the American National Standards Institute (ANSI) [51]. The International Organization for Standardization (ISO) accepted ANSI-C in 1990 and standardized it under the popular name "C90". A revision followed in 1999.

C was a high point of the imperative language model of its time and bordered on the object-oriented principles that followed in C++. Technically referred to as a third-generation programming language (first-generation being machine code and second being assembly language), C was designed to be concise in syntax, efficient through affording developers access to low-level facilities of the machine, and well-structured in how it handled data. The downside of the power C offered is the extra burden it put on the developer to avoid programming pitfalls that higher-level languages take care of by not allowing the developer enough access to fall into to begin with. For imperative application purposes, C is the ultimate choice; operating systems, libraries, and APIs are written in C because of its power and efficiency. However, for higher level programming concepts, true object-oriented languages make more sense to use.

This brings us to C++, a language that can be thought of as a superset of C. Also developed in Bell Labs in the early 1980s, C++ addresses some of the issues of C while following a different language paradigm altogether; C++ is a truly object-oriented language, which means, briefly, that it is not concerned with procedure and steps but the objects these procedures and steps affect. C++ is backwards compatible, meaning that virtually all C code can be fit into C++ code with little or no change. This, by definition, implies that C++ also supports imperative programming.

For the reasons surveyed above, and due to C++'s backwards compatibility with C, the former has gained great popularity in application software development. And the more popularity the language gained, thanks to it proving its efficiency and value, the more

lower-level application areas began to provide APIs and libraries for it. ISO approved a C++ standard in 1997.

Of relevance to the purposes of this work, is the mechanism by which C++ code becomes machine code and instructs the computer on how to carry out an application – this is a main factor in the performance of an application. The idea is simple; a programmer abiding by the languages syntax and semantics writes code that logically performs a specific task. This code is run through a compiler which reformats, optimizes and interprets it, through several stages, into machine code. The machine code is finally run when the resulting executable is called. This whole process, generally referred to as compilation, takes place on a single platform (combination of hardware and software) and is not portable, meaning that if it the source code were to be compiled on a Windows machine, then the executable cannot be run on a UNIX machine. For the program to run on UNIX, it must be compiled on UNIX.

Compilation is efficient in that it optimizes the code to run as well as possible on a specific platform doing away with overhead code that would otherwise be necessary if cross-compatibility were an issue. Also, once an executable is created, it can be ran over and over without having to go back to the source code or any other intermediate compiler output for that matter. This efficiency and boost in performance comes with the price of platform-specificity.

5.1.2. Java

Motivated in part by the very same platform-specificity issue explained above, a Sun Microsystems team of researchers sought out to create a language that can be written once on any platform, and then readily ran on any other platform. In the early 1990s, James

Gosling and his team of researchers were looking for a programming language choice suitable for communication between consumer electronics. Frustrated with the lack of a true cross-platform language, they proposed Oak, later renamed Java to better reflect the "liveliness, animation, speed, (and) interactivity" of the company culture and product language [52], to be a programming language that one can "write once, run anywhere", a catch phrase that became closely associated with Java.

The main premise of Java's cross-platform operability is simple; a programmer writes code on a given platform and then this code is halfway "translated" into an intermediate form called bytecode. This bytecode is platform independent meaning that it is communicated the same to a UNIX machine as it is to a Windows machine. The difference is in the last step between bytecode and machine language, namely, the Java Virtual Machine, or JVM, which is machine dependent. JVM takes the cross-platform bytecode and interprets it on the go, during run-time, to native machine language.

The Internet began to gain stupendous momentum by the mid 1990s and the need for cross-platform light applications became evident; Apple machines had access to the same web resources as Windows and UNIX machines and they all needed to be provided the same content for. HTML enabled browsers to display static websites but offered no real interactivity, or activity for that matter. Java came to the rescue marketing its cross-platform compatibility. Netscape Navigator 2.0 came standard with the appropriate JVM and small, compact mini-application called Java applets, took Internet media one step further.

Cross-platform interoperability, however, is not without its drawbacks. Interpreting code on the go, as is the case with bytecode and JVMs, offers the great advantage of cross-operability as we have seen above, but the downfall is that it is slow and far from optimum. Compiled languages, such as C and C++, afford their compilers the opportunity to have an omniscient view of the code which makes for optimized code that could help compensate for programmer shortcomings. Furthermore, compiling on a specific platform ensures that the code is optimized for that platform, so although it won't run on a different system, it will run extremely well on the one in question. Developers who were considering Java for their applications began to ask themselves if the advantages that Java have over languages such as C++ were worth the performance hit they would take.

The Java community responded with Just-In-Time (JIT) compilation. In this source code to machine language paradigm, the bytecode is not interpreted during run-time but rather wholly or partially compiled just before, or when needed during run-time depending on the implementation. This attempt at lessening the performance hit suffered by Java interpretation has went through successive iterations to get to a point now where it rivals compiled languages in performance.

5.1.2. A Comparison between C++ and Java

As mentioned above, Java came as an "improvement" over C++ primarily in the area of cross-platform operability, but even with the more advanced JIT compilers, it sacrificed performance to help achieve that improvement, as will be discussed later. This is but one area in which these two object-oriented programming languages compare and contrast.

C++ has advantages over Java in that it provides easy access to low-level computer resources through operating system and hardware APIs. The fact that it is compiled and, with the advances in compiler optimization performance, it generally runs faster than Java interpreted, or even JIT-compiled, bytecode. C++ also offers the advantages, and disadvantages, of operator overloading, support for global declarations, pointer manipulation, and multiple-inheritance, among other features.

Java, on the other hand, is a more developer-friendly programming language thanks in part to its extensive collection of libraries directly in its core API. It offers automatic garbage collection that ensures efficiency in memory use as well as built-in security limitations that work to prevent loopholes and openings in code that can be maliciously exploited. The fact that its bytecode is independent of the JVM or JIT compiler translating it into machine code means that independent advances in JVM and JIT compiler technologies mean more efficient code regardless of when it was written and, of course, on which platform. We last note Java's built-in support for multithreading and distributed, networked programming.

Most of Java's advantages listed above, however, have a downside. Being a higher-level programming language than C++ and having so many checks and limitations to control things such as memory leaks and security integrity, Java takes power out of the hands of more advanced developers especially when there is a need to access operating system and hardware APIs as well as low-level resources. Also, Interpretation and JIT compilation do not match very well against compiled code in terms of performance and efficiency. Sun, as well as other Java SDK developers, has addressed the issue of slower performance of Java

applications in several ways but the consensus still is, even though JIT-compilers have markedly improved, C++ compiled code is, on average, still faster running [53].

Two recent studies compare the performance of Java against that of C++, as well as other languages. [53], written in 2002, compares the performance of a Java-based looping module to that of a C/C++-based one both used in large visual dataset animation. These loop modules; called JLoop and Loop respectively, are used to progress through large visual datasets, as well as roam, zoom, and probe them, within a Distributed Image Spreadsheet (DISS) visualization environment. [53] specifically studies the performance of JLoop and Loop in a DISS environment that processed Hurricane George imagery in the form of very large datasets. The actual processing was the evaluation of a hundred million quadratic polynomials with optimized and compiled C/C++ versus the same number of quadratic evaluations with Java 2 v1.4 optimized and JIT-compiled. [53] does not give more details about the testing environment except to say that the tests were done on a Windows P3 machine, a Linux machine, a SUN machine, an Alpha machine, and an SGI machine. The results concluded, averaging all six test platforms, that C/C++-based looping modules performed 2.0025 times better than Java based looping.

Another more comprehensive study was presented in [54]. Twenty-three standard application-design patterns as well as four components were all implemented in, among other languages, C++ and Java. Of interest to us is how optimized C++ (compiled with GNU 3.2) compared with optimized Java 2 (compiled with Sun JDK 1.4.0). For the four benchmarks the code was implemented to measure, namely a Boolean prime checker, an iterator, an object creator, and an object referencer, Java came in surprisingly as fast as

C++, however, if we take out the object creation benchmark, in which C++ performed five times slower than Java did, C++ came in 2.23 faster than Java. The authors, after implementing and testing the twenty-three design patterns in the study, conclude that, on average Java is only 1.15 times slower than C++.

5.1.3. Programming Unison

Looking at the above two studies, there is no doubt that JIT-compiled Java is catching up with C++, but it still, on average as we have seen, falls short. In carrying over to hapto-visual application development, we choose C++ over Java in programming Unison's services because of its demonstrated better performance. [53] showed that C++ was capable of processing large data sets in a visualization application almost twice as fast as Java did, and visualization relates well to the graphic, and by extension, haptic rendering that Unison is required to address. Another factor for choosing C++ over Java was that all haptic APIs studied first-hand (see Chapter 5: Section 3) were written in C++, therefore if we were to use Java then we self-impose the overhead that comes with using JNI, which for hapto-visual applications, is considerable as we have shown above in reviewing [53] and [54].

For windowing and GUI purposes, we use GLUT [55] because of the ease of its use, especially for the purposes of just displaying a scene to the screen and capturing user input from the mouse or keyboard, which is the extent that an application implemented with Unison reaches no further than. It also offers a variety of automatic callback functions as well as a timer function that greatly simplify multithreading in C++, also a feature needed to, among other things, carry out rendering loops in parallel. Finally, GLUT interfaces

nicely with OpenGL, on which it is built, and OpenGL as will be discussed in the following chapter, is an integral component of Unison's graphical rendering facility.

5.2. The Graphic Components

5.2.1. VRML

The Virtual Reality Modeling Language is simply a text-based, meta-data based modeling language to describe virtual, 3D, interactive environments. VRML was developed primarily for the World Wide Web as an efficient, portable, and compact language and has gone through several reviews, most markedly in 1995 when it was first launched, and in 1997 when first standardized by ISO under the common name VRML97, more formally known as VRML 2.0 [46].

VRML describes the virtual world by way of a hierarchical scene graph with nodes, branches and leaves representing semantics described in simple text syntax. Graphic objects in VRML are built from a variety of simpler components that can be either simple geometries or sets of points and vectors often created by third-party software especially in cases where objects are too complex to manually describe, and thereby build. VRML also supports image-based display of objects, as well as textures, lighting, backgrounds, fog, collision detection, sound, and animation. Interaction with VRML worlds can be either done with simple scripting using JavaScript or sensor-based interpolation which is built-in the VRML specification.

X3D, an open standard still under development, is VRML 2.0's successor, and a superset of it. It further formalizes the meta-data paradigm that VRML follows by strictly adhering

to XML standards and divides itself conceptually into profiles that developers can load for different needs of their applications.

5.2.2. Java 3D

Java 3D is a low-level, scene graph based API for Java that affords it 3D programming facilities. It basically provides for the creation of 3D geometries and interactions with these geometries regardless of underlying hardware and rendering considerations which are taken care of by a third-party API invisible to the Java 3D developer, sitting underneath the API. Java 3D can use OpenGL or DirectX for instance to render its scene graph described by the user or loaded into the development environment from a modeling file. Being a Java 2 API, Java 3D can run on most UNIX, Windows and Apple machines.

As discussed in Chapter 5: Section 2, Java does have one of the worse reputations when it comes to performance metrics. Java 3D will naturally add to whatever speed shortcomings Java has simply because it is more work done, but there are ways to better program and compile Java 3D code, most are built in the language specifically for performance concerns. Java 3D affords the user a degree of control in specifying, through what is referred to as capability bits, what the intentions for specific objects are, which in turn allows the compiler to better optimize the resultant code. Spatial bounding is another facility used to make sure that objects do not affect the environment outside of a specific bound lessening the factors that need to be taken into consideration when rendering the scene.

Java 3D is in a different class altogether than VRML. The former is a full fledged programming API that sits on top of Java and has considerably more power, inherent in

Java, than the latter. VRML, on the other hand, is a modeling language with minimalist programming capabilities afforded through scripting, which often is done in Java. But VRML does have the usability advantage that it is very easy to write in, as opposed to Java3D which requires elaborate adherence to Java syntax in order to write a given scene graph that otherwise could be described in a fraction of the time in VRML or, in even lesser time, with a graphic VRML authoring tool.

5.2.3. OpenGL

In the early 1990s, a need arose for a standard graphic library that would spare developers and hardware manufacturers the expenses and issues that came with non-standardized proprietary hardware/software graphic components. Silicon Graphics proposed a specification it later implemented based on its IRIS GL library, and OpenGL was born [56]. Ever since, the solution grew in popularity and is now supported by virtually every commercial graphic card and widely used by developers. The Open Graphic Library now has its own review board (OpenGL Architecture Review Board) that examines modifications and amendments that go into the language.

Besides the fact that is widely followed and used, OpenGL offers other advantages to the developer such as stability, scalability, and support. The fact that it has been around for so long and used so extensively ensures stability. OpenGL is highly scalable because of its architecture which can run on a supercomputer as well as on a PC. And since it is so widcly used, a developer community is available for support.

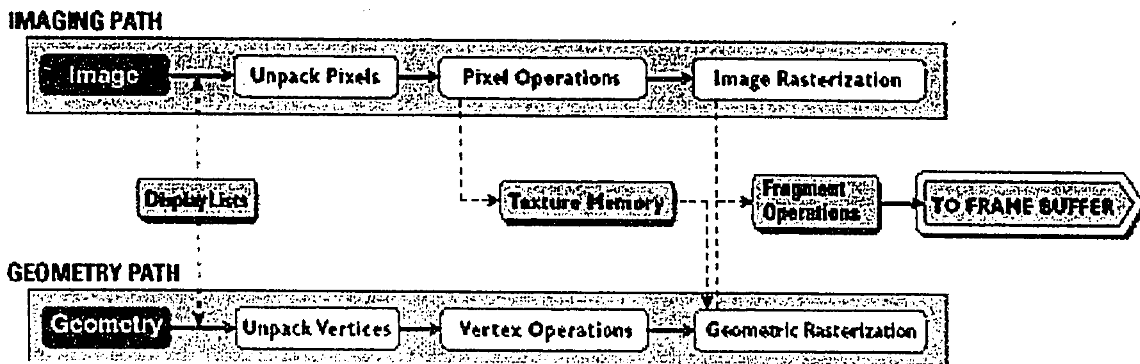


Figure 10: OpenGL operation pipeline [57]

OpenGL's capability as a complete graphic development tool should be credited to its operation pipeline shown in Figure 10. In rendering graphics to the screen, OpenGL has two paths: one image-based and one geometry-based. These two paths, or channels, of the pipeline can communicate with each other at various stages and work from the point they are called by a third-party API (such as Java3D) or invoked directly by the developer till they populate the frame buffers of the computer monitor. This A to Z pipelining is visible to the developer, who if programming directly in OpenGL, is afforded access to all aspects of the OpenGL state down to the single pixel. This makes OpenGL a very powerful API.

Worth mentioning here, and relevant to this work, is that OpenGL has direct bindings to languages such as C, C++, FORTRAN, Ada, and Java.

5.2.4. Xj3D

Xj3D is a toolkit developed completely with Java in open source and monitored by the Web3D Consortium, a non-profit organization that aims at spreading the use of the VRML97 specification as well as the in-progress X3D specification [58]. Xj3D, consequently, is able to create, load, manipulate, and save VRML and X3D scene graphs

from modeling files in an effort not only to add functionality to the aforementioned specifications, which is why they are being pushed by the Wed3D consortium in the first place, but also to use them as basis of bigger projects that require graphical modeling.

Briefly, we mention two other tools considered to load VRML files into a Java development environment: CyberGarage CyberX3D Loader for Java [59], highly functional and developer-friendly, and for the purposes of implementing Unison, we felt was as good as Xj3D, but it was not chosen because Xj3D had a bigger developer community which meant more support; and J3D-VRML97 [60], a Java.net community project based on Sun's VRML97 loader version 0.90.2 released in March 1999, which aims at updating the aforementioned loader to better meet Java's and Java 3D's updated requirements and facilities. J3D-VRML97 was not chosen because it was not, at the point where we were making implementation choices, as well developed as Xj3D.

5.2.5. CyberGarage CyberX3D Loader for C++

CyberGarage's CyberX3D and its loader [45] is a toolkit developed by a Japanese computer programmer, and alongside other VR toolkits, made freely available on his website [61]. CyberX3D which supports both Java and C++, albeit in different SDKs, affords the developer the ability to parse, load, and, with OpenGL, render scene graphs from X3D and VRML files.

CyberGarage CyberX3DLoader for C++ was the first VRML loader looked at, and it provided satisfactory functionality and ease of use that there was no need to look any

further, however, we mention OpenVRML [62], an active project part of the SourceForge.net[12] collection, because it was also briefly considered.

5.2.6. Graphic Components in Building Unison

As mentioned before, for Unison to function in the middle of an HVA, it must be able to communicate with all components making up an HVA. We have concluded in Chapter 5: Section 2 that although we can write Unison in Java, we chose C++ because of its better performance. Such is not the case with the graphic components that Unison needs to afford the developer support for, simply because Unison will not choose the developer's tools for him or her but rather provide a generic interface to develop an HVA regardless of the underlying technologies. This is after all one of its main premises. However, for implementation purposes, and for the sake of modularity, we presented a plug-in model for Unison such that each development tool would have its own plug-in, so if the user were to load model X3D files, an X3D Unison plug-in would be necessary, and so on.

In this section, we have surveyed VRML (and X3D), Java3D, OpenGL, Xj3D, and the CyberGarage CyberX3D Loader for C++, all as individual components that in combination make up an HVA's graphic component. For instance, a model file could be in VRML; using CyberX3D Loader we introduce it to a C++ development environment; and using OpenGL (via the CyberX3D Loader itself), render it graphically to the computer screen. For such a choice of underlying technologies, we have to implement Unison services that would call CyberX3D (and by association, VRML and OpenGL) functions. Unison's user

[12] SourceForge.net is a website that hosts a wide collection, allegedly the world's biggest [74], of open source code and applications on the Internet. It brings a community development paradigm to a wide array of projects from operating system to user-interface software development.

however, has to only provide the path to the VRML file and use Unison's services without having to worry about how the VRML file nodes are loaded into his or her development environment, or rendered to the screen.

At this stage of implementation, Unison is capable of loading VRML files into a C++ environment and rendering them to the computer screen via CyberX3D Loader – which is also used to load the VRML file – or directly using OpenGL/GLUT. Chapter 5: Section 4 lists some of Unison's functions.

5.3. The Haptic Components

5.3.1. Haptic Devices Studied

Table 3 shows a side-by-side comparison of the three haptic devices studied in the course of creating Unison. The PHANTOM desktop [63] and the MPB Freedom 6S [42] are similar in that they are operated the same way by a user; they both have a stylus for an end effector, both have 6 degrees-of-freedom positioning, and have relatively similar workspace volumes. The FCS HapticMASTER [36], on the other hand, is in a class of its own; it is the bulkier, more powerful, more "industrial-strength" machine with a much bigger range of motion and a different force control mechanism altogether (See Appendix I).

The differences described above and in Table 3 are important factors in determining which device is more suitable for which job. The PHANTOM Desktop's main limitation is a small workspace that measures only about 2,700 cm^3 (16 cm x 13 cm x 13 cm) which is comparable in size to about half a shoebox. Its maximal force output (impedance force

control) is roughly between 6.0 and 8.0 N, depending on what position the arm is in, and its position resolution is a miniscule 20 μm, comparable to the diameter of the thinnest human hair. These features and limitations of the PHANTOM Desktop make it most suitable for application scenarios with small, pliable haptic objects such as small-scale medical simulations and haptic design and modeling of small objects.

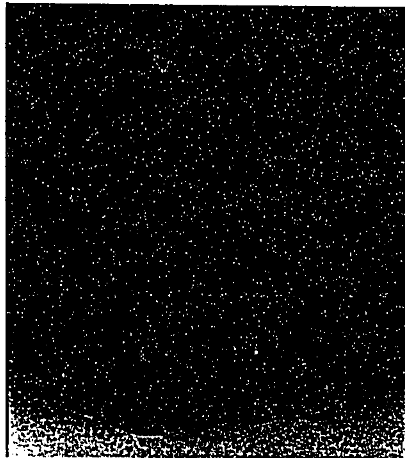


Figure 11: SensAble PHANTOM Desktop [63]

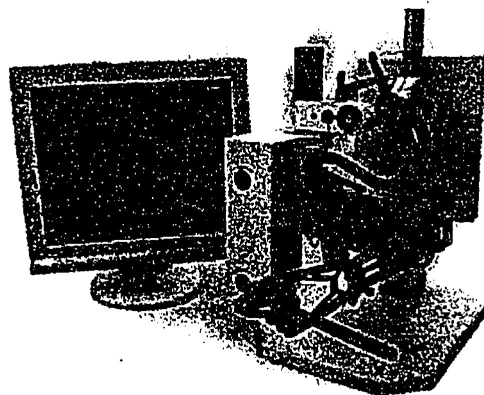


Figure 12: MPB Freedom 6S [42]

Table 3: A Table Comparing Haptic Devices Studied.

	PHANTOM Desktop	*MPB Freedom 6S*	*FCS HapticMASTER (v 2.1)*
Degrees of Freedom	6 DOF positioning, 3 DOF haptic feedback	6 DOF	3 DOF
End Effector	Stylus (exchangeable)	Stylus	Knob handle (exchangeable)
Workspace	~2,700 cm³	~13,000 cm³	~66,200 cm³
Position Resolution	20 μm	20 μm	< 4 μm
Force Resolution	N/A	1.5 mN (Translational), 0.1 mN.m (Pitch/Yaw), 0.5 mN.m (Roll)	< 10 mN
Maximal Force	Impedance force control: 7.9 N	Impedance force control: 2.5 N over 60 seconds or 0.6 N continuous (Translational), 160 mN.m over 20 seconds or 95 mN.m continuous (Pitch/Yaw), 60 mN.m over 20 seconds or 35 mN.m continuous (Roll)	Admittance force control: 250 N

The Freedom 6S distinguishes itself over the PHANTOM Desktop with a full 6 degrees-of-freedom in haptic feedback (translation, pitch/yaw, and roll). It also has a considerably larger workspace than the PHANTOM Desktop with just as resolute a position sensitivity. These features as well as impedance force control make the Freedom 6S a better choice for small volume haptic applications (e.g. medical, modeling) despite the lower maximal force

levels, which anyway should not be a huge deterrent in delicate object applications such as surgery.

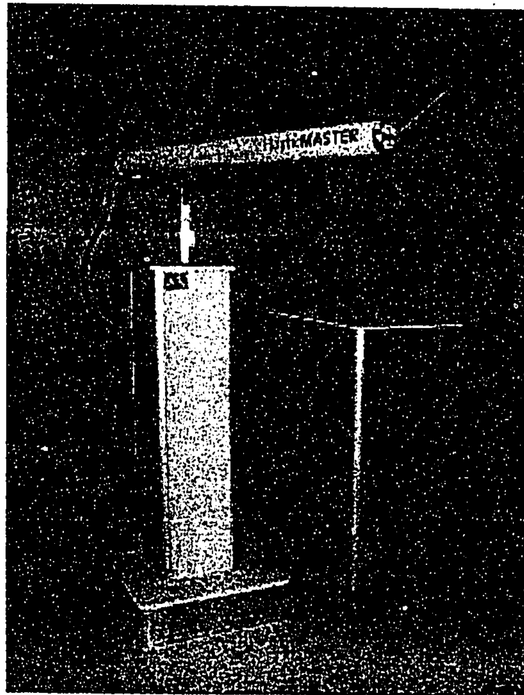


Figure 13: FCS HapticMASTER [36]

The HapticMASTER is in a different category than the previous two haptic devices. Not only does it have a much bigger workspace and a much higher maximal force level, but also, its force control paradigm, admittance control, makes it much more suitable for application scenarios that require simulated contact with solid, stiff, heavy objects. Its incredibly resolute force sensitivity (<10 mN), as well as the above features, make it ideal for real-life scale applications with roughly the range of motion of an arm in 3D space pivoting at the shoulder. Examples of such applications include injury and disability rehab, robotic tele-operation and vehicular control simulations. Worthy of note here is that the

HapticMASTER comes with a server that handles all haptic rendering leaving the graphic and/or auditory rendering to the host computer.

5.3.2. Haptic APIs Studied

The Haptic API is an integral part of any HVA; it is the link between the program code and the haptic device at the very least, as is the case with earlier first-generation APIs; and the program code and the complete hapto-visual-auditory scenes, as is the case with the more advanced, second-generation and higher APIs.

In this research, three major APIs are examined: Reachin API v 3.2 [47][64], GHOST API [65][66], and the HapticMASTER API v 1.2 [47].

The Reachin API is by far the most complete HVAD solution of the three and it is not inaccurate to say that it is an A-Z, all-inclusive development tool. Reachin API is built around two main concepts: a hierarchical, multi-sensory scene graph and an event-handling field network. The multi-sensory scene graph holds all the graphic and haptic objects in the virtual environment as they exist at any one given point in time. It is hierarchical with groups of nodes representing logical objects. The API's field network, on the other hand, holds all interaction rules that list which interaction affects which group in the scene graph. Conceptually, we can think of the scene graph as the stills that when played consecutively based on interactions subject to the field network, we get a dynamic hapto-visual environment.

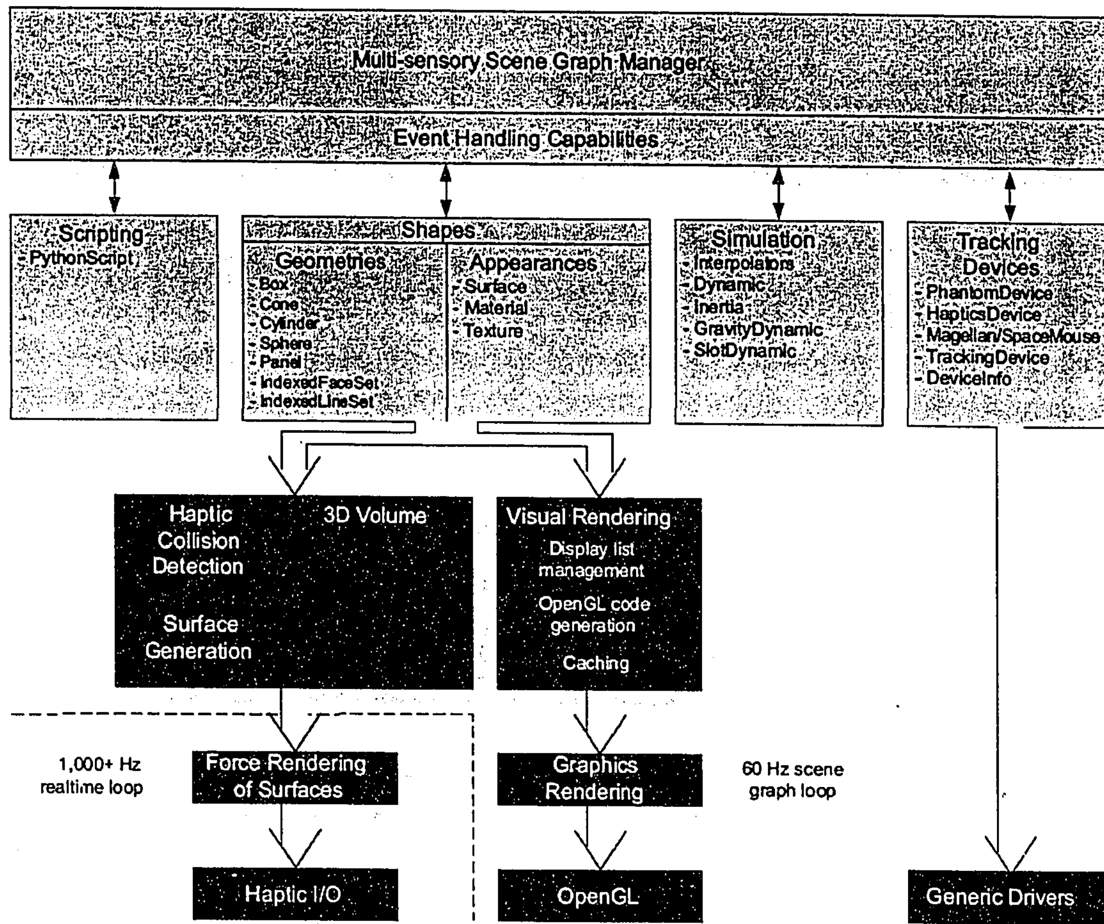


Figure 14: A conceptual overview of the major models in the Reachin API [47]

Figure 14 shows a modular view of the Reachin API. On top of the structure sits the scene graph manager that directly controls the multi-sensory scene graph but in accordance with the field network that has all the event handling capabilities (rules, facilities, etc). The scene graph is comprised of shapes that are native to the API (albeit based on VRML nodes). These shapes, once the API is instructed, are rendered haptically, at a rate greater than 1 kHz, and visually, at a rate of around 60 Hz. The graphic rendering is done by the API automatically calling the OpenGL API, while haptic rendering is communicated to the haptic device after object dimensions, surface generation, and collision detection are all computed. The API obviously has to communicate with the haptic device through

appropriate drivers but, within the same module, the API also offers integration capabilities with generic 3D tracking devices thanks to generic drivers that are part of Reachin API. As a side note, the API offers developers prototyping and rapid application development (RAD) by supporting PythonScript within VRML files. Python is an interpreted, very high-level programming language, and the Reachin API has nodes "PythonScript" that can invoke almost all API functions without the need to compile any code at all.

The HapticMASTER API studied was version 1.2. Released in 2003, it works exclusively with the FCS HapticMASTER and is characterized by it simplicity and power. The HapticMASTER API, programmed in C++, is a first-generation API in that it only involves itself with the haptic scene, but it is easily integrated with mostly generic OpenGL code, a feature that makes it comparably easy to use as some second-generation haptic APIs. What the HapticMASTER API lacks in high-level programming interface, the HapticMASTER device more than makes up for in hardware performance and power given to the developer (through the API). Unlike higher-level APIs that restrict the developer's direct control of the device, the HapticMASTER API is built over a single function that directly feeds the admittance control paradigm to which the device adheres. `SetForceGetPosition()` allows the developer to instruct the haptic device which vector force to recreate while querying it for the position of its end effector in the 3D Cartesian space of the virtual environment. The API documentation recommends that the thread constantly calling `SetForceGetPosition()` be invoked every 10 millisecond, as haptic rendering is done at a rate of 2.5 kHz. This rendering rate, 250% of the refresh rate the Reachin API affords, is affordable because of the dedicated server machine attached to the HapticMASTER's robotic arm.

The SensAble GHOST SDK v3.0, which includes the GHOST API, is an example of a first-generation API. Like Reachin API, GHOST is in C++ and follows the scene graph paradigm, only its scene graph is haptic only (meaning it only contains information about the haptic scene and not the graphic scene, which has to be addressed by the core application). Hapto-visual synchronization is done through call back functions as the visual part of the application runs completely inside the core application. As Figure 15 shows, the SDK does manage all aspects of the hapto-visual application interfacing cleanly with, for example, OpenGL or Open Inventor. A main strength of the GHOST SDK, however, that does distinguish it from other first-generation APIs, is its extensibility achieved through inheritable classes in C++.

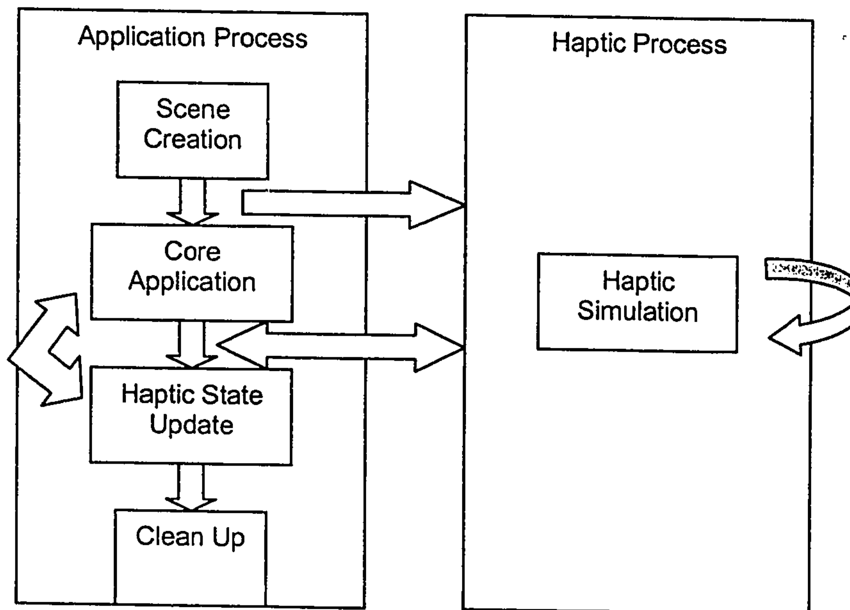


Figure 15: Typical application using the Ghost SDK [65]

5.3.3. Haptic Components in Unison

As was the case with the graphic components, haptic components are introduced to Unison as plug-ins, so if Unison's user wanted to code for the HapticMASTER in Unison, he or

she has to have the correct libraries for the code, namely the HapticMASTER plug-in that communicates with the HapticMASTER API.

In the previous sections, we have surveyed three haptic devices and three haptic APIs. Two of the three devices follow an impedance force control paradigm, while the HapticMASTER follows the admittance model (see Appendix I). This is not only a major difference in the hardware workings of the devices, but also the way the APIs communicate with the devices. In admittance control, the haptic device accepts user-input force, and after real-time internal computation, returns a position vector relaying where the end-effector in the virtual environment would be. Such is the case with the HapticMASTER and its API which are both built around the simple `SetForceGetPosition()` function. The MPB Freedom 6S and the PHANTOM Desktop operate differently, under the impedance control paradigm, accepting, as input, the position of the end-effector and, after internal computation, returning the force the end-effector would experience at that specific position in the hapto-visual environment. Their haptic-related functions work in the opposite way as those of devices like the HapticMASTER, although the APIs surveyed did a good job of hiding their force control paradigms by offering much higher level functions than the HapticMASTER's `SetForceGetPosition()`. For instance, you could still set the force at a specific position but only by implication, meaning that you could recreate force feedback experienced when colliding with a stiff surface by, in one way, designating the stiffness of the surface.

Overall, however, working with the higher level APIs of impedance control systems had both its advantages and disadvantages when the APIs' functions were being adapted to fit

Unison. For one, being so high-level meant that often invoking them was only a matter of writing a Unison function, such as `CreateConstantForce()`, that called a GHOST or Reachin API function that had the exact same parameter list and purpose but obviously a non-generic name, for example GHOST's `setConstantForceVector()`. The disadvantage was the sometimes laborious and strict access controls that lengthened the coding necessary to retrieve a specific field in a graphic node for instance, so instead of getting a pointer to a VRML node directly by only specifying its name, going through Reachin, you had to call a chain of functions to return that same handle.

5.4. Unison Data Structures and Functions

After reviewing all the components necessary to create a hapto-visual application, and also briefly describing how Unison is designed to accommodate them, we present in this section some Unison internal data structures and functions, all functional and implemented with the following tools, discussed in previous sections, chosen for the sake of testing and providing a prototype: We choose C++ as the core application language; GLUT to handle windowing and multithreading; CyberX3D Loader and VRML formats to load and describe visual scene graphs; and finally, the HapticMASTER and its API for the haptic components.

Table 4 shows the bindings between VRML shape nodes and HapticMASTER basic geometry shape nodes. Unison will refer to these geometries internally by the name given to them in VRML, so a VRML "box", which is a HapticMASTER "FcsBlock", is internally referred to as a "box". Note that the size of different geometries is dependent on the geometry. For instance, a box has three parameters for width, length, and height, however a cylinder only has two: radius and height, so to differentiate, we overload the

parameter list in C++. Cones, however, also have radius and height as size parameters, so when necessary, parameter list overloading is complemented with geometry type checks.

Table 4: How VRML and HapticMASTER API basic geometries are bound together in Unison

VRML Nodes	*Shared Properties* [13,14,15]		*HapticMASTER Basic Geometries*
Box	XYZ-Translation (through Transform Group). (X, Y, Z) floats for size.	X'Y'Z'-Center. (X', Y', Z') doubles for size.	CFcsBlock
Sphere	XYZ-Translation (through Transform Group). A float for radius.	X'Y'Z'-Center. A double for radius.	CFcsSphere
Cone	XYZ-Translation (through Transform Group). (Bottom radius, Height) floats.	X'Y'Z'-Center. (Radius, Height) doubles.	CFcsCone
Cylinder	XYZ-Translation (through Transform Group). (Radius, Height) floats.	X'Y'Z'-Center. (Radius, Height).	CFcsCylinder
IndexedFaceSet			Dependent[16]

We now describe Unison data structures, an absolute integral part of how Unison works. As mentioned before, Unison achieves standardization as a development tool by having standard data structures named the same and referenced the same regardless of underlying technologies. The main data structure that Unison provides is the synchronized hapto-visual

[13] Note transformation between different axes in different components.
[14] Transform Groups are VRML groupings that join geometries that make one object and apply translation, rotation and scaling to all of them in a hierarchy.
[15] Cylinders in both VRML and the HapticMASTER can be with or without one or both lids, but for the purposes of this work, we assume a closed cylinder.
[16] Any VRML file with a complex enough shape probably has IndexedFaceSet nodes to describe the shape. However, since there is no generic equivalent geometric shape in the HapticMASTER API, the user will have to approximate one of the basic shapes to overlay the IndexedFaceSet and create a rough haptic object.

scene graph (HVSG). Pure and simple, this is the heart of the HVA. This data structure, implemented as either a dynamic linked list, or more intuitively, as a 2D table, holds current information of all haptic and visual objects in the scene graph displayed via both the screen and the haptic device. Hapto-visual synchronicity is achieved implicitly by having each hapto-visual object in the graph rendered graphically and haptically according to the same position variables found in the HVSG with appropriate scaling and axes transformations[17]. Furthermore, all relevant fields and properties are represented in the HVSG. As the prototype implementation stands, and conceptualizing the HVSG as a table, Table 5 shows a possible state of the synchronized scene graph given the choice of components listed above. Table 5 describes the scene graph in Figure 16.

Table 5: A snapshot of an HVSG state in Unison.

Index	Internal-Ptr	External-Name	Geometry	Center	Size	Surface Stiffness
0	ptrBox_0	"Box"	BOX	(0,1.0,0)	(2.0,2.0, 2.0)	1500
1	ptrSphere_0	"Sphere"	SPHERE	(-3.0,1.0, 0)	(1.0)	1500
2	ptrCone_0	"Cone"	CONE	(3.0,1.0,0)	(1.0, 2.0)	1500
⋮	⋮	⋮	⋮	⋮	⋮	⋮

The HVSG described in Table 5 is a data structure that can either be implemented as a 2D array or, more functionally, as a linked list with nodes of types depending on the geometry of the object described. For the purposes of explaining the idea behind an HVSG, we follow the table model. The first column is an `Index` field that simply identifies each row uniquely for access purposes; it is the primary key of this table so to speak. The `InternalPtr` field holds the pointer to the entire record or linked list node. The naming of this pointer is done automatically depending on the geometry describe in the row as

[17] The 3D Cartesian plane and scaling in one component are not necessarily the same in another. VRML axes for instance point to X (increasing to the right of the screen), Y (increasing to the top of the screen) and Z (increasing out of the screen), while the HapticMASTER's X axes is increasing out of the screen, the Y is increasing to the right of the screen, and the Z is increasing to the top of the screen.

shown. The ExternalName field holds the DEF of the Transform Group[18], which allows for retrieval of the VRML node by name. The Geometry field describes the geometry of the object in question based on the conventions in Table 5. The Center field, with units in meters, corresponds to the translation of the Transform Group in VRML and simply as the center of the object in the HapticMASTER API. The two are not necessarily identical in values or units, but can be reconciled with a mathematical transformation. Handling the typically different, but sometimes similar parameters in the Size field, also in meters, has been discussed previously in this section and we only note here that it is done through parameter list overloading and checking the geometry if necessary. Finally, SurfaceStiffness, measured in N/m represents the ExtSpringStiffness variable the HapticMASTER API requires of the haptic objects it renders.

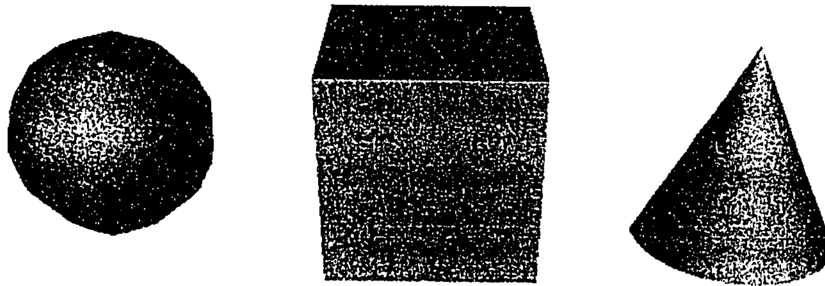


Figure 16: Scene graph described by Table 5

We now describe some of Unison's prototype functions, implemented and functional at this time as per the parameters set forth previously in this chapter.

[18] As Unison's implementation stands now, only DEFed Transform Group nodes will be loaded into the HVSG (DEF is an attribute of a node that gives it a unique name that, among other things, allows access to the node by referring to it by the aforementioned name). Expanding on this functionality to allow for other kinds of nodes to be loaded requires little modification in the code.

- `InitializeHapticDevice()`: This generic function takes a defined string corresponding to an integer, and based on a map in Unison, calls the necessary initialization functions that ready the hardware and open communication channels. The function returns a Boolean indicating success or failure.

- `LoadHaptoVisualSceneGraphFrom()`: This function takes a modified model file that has both haptic and visual information about a scene graph (e.g. modified VRML files in Reachin API), and populates the HVSG with the appropriate data. As the implementation stands now, both scenes are rendered.

- `LoadGraphicSceneGraphFrom()`: This function takes a VRML filename as a parameters and, using CyberX3DLoader functions, populates the HVSG accordingly. It also renders the scene to the screen.

- `CreateHapticObject()`: This function shows parameter list overloading at its best. For the parameters of the current implementation of Unison, a call to this function would include the following: an external name to be associated with the object; an internal pointer to reference it; geometry; overloaded size parameters; translation coordinates; and external surface stiffness. Called to a Unison HapticMASTER plug-in, this function will create a haptic object and enable it thereby rendering it to the haptic device.

- `CreateGraphicObject()`: As is the case with its haptic counterpart described above, this function creates a graphic object by adding it to the HVSG ensuring that it

will be rendered graphically next time the data structure is queried for graphic scene graph components.

- `DeleteHapticObject()`: Given the object's pointer, this function will delete it from the HVSG and clean up its pointers from the local host computer as well as from the HapticMASTER server.

- `DeleteGraphicObject()`: As is the case with the DeleteHapticObject() function, this service removes the graphic object from the HVSG.

- `SetGraphicProperty()`: Given a pointer to a graphic object, a field to update, and an overloaded update value, this function will set a graphic property in the HVSG.

- `SetHapticProperty()`: Given a pointer to a haptic object, a field to update, and an overloaded update value, this function will set a haptic property in the HVSG.

- `SetProperty()`: Since the HVSG holds both haptic and visual nodes, the mechanism of setting a property to one is the same as the other. The above two set functions ensure data integrity by only affording access to either haptic or graphic properties, but not both, as is the case with this function.

- `CreateConstantForce()`: This function instructs the haptic device to display a constant vector force, which would be the case in recreating gravity[19].

[19] In our implementation, force vectors recreated to simulate gravity can are constant downwards forces; they do not take into consideration acceleration due to gravity. We could afford to do that primarily because of the small workspace volume of the HapticMASTER.

5.5. Negative Aspects of Using Unison

Using Unison is not without its disadvantages, the biggest of which is the loss of the specificity afforded by custom haptic API/hardware combinations. This problem is common with the use of any generic alternative at a stage where there is no universally agreed upon standard. The loss of specificity is evident when Unison is compared with custom application development tools that are highly involved and broadly inclusive. Unison sits at the other end of the spectrum from the aforementioned tools (e.g. APIs or SDKs) by definition, since it is made up of the "concepts" that custom tools share amongst themselves.

To further explain this disadvantage, consider the following classes available in the Reachin API but not Unison: class `VibratingSurface`, class `PositionInterpolator`, and class `ConeInertia`. `VibratingSurface` describes two sine functions recreating horizontal and vertical vibration in a virtual surface. `PositionInterpolator` describes the "tweening" of a hapto-visual object between two spatial coordinates. And class `ConeInertia` describes the inertia of a solid cone in a hapto-visual scene. These classes do not have counterparts in Unison because, for instance, the HapticMASTER API does not support any of them. This is not to say that they are not programmable with the HapticMASTER API but just that such programming has to be done explicitly. Another example of the loss of specificity that comes with using a particular API as opposed to Unison can be show when contrasting the example in Chapter 3: Section 3.1.4 with its implementation in Unison; the aforementioned example is that of an HVA that uses `reachinload.exe` (part of the Reachin API) to load a VRML file with Python script. Since there is no counterpart to a tool like `reachinload.exe` in Unison –

nor in any other API mentioned in this work aside from Reachin API – instances where the former serves well (e.g. prototyping an HVA, rapid application development of an HVA) put Unison at a disadvantage because of the nature of using Unison, which involves full fledged application programming.

Another example of loss of specificity, this time pertaining to the hardware as opposed to the HVAD and its APIs, is the particularity of the haptic device's physical interface. The PHANTOM Desktop, a stylus-based haptic device, has a button on its stylus pen that can acts like a mouse button, the MPB Freedom6S, another stylus-based haptic device, does not. Neither does the HapticMASTER. So functions accepting input from that button (e.g. stored in Reachin's class `PhantomDevice → SFBool button`) are not supported in Unison because the stylus button is not a common feature to haptic devices.

Worthy of note here is that the problem of loss of specificity described above is not without a solution, namely more customized plug-ins for Unison, however the aforesaid solution can be taken to an extreme where Unison becomes nothing more than the haptic API but with different function names. Getting that specific obviously defeats the purpose of using Unison by violating its main premise of generality. It will take a consortium to come up with a standard handed down to haptic hardware and software developers or a de facto standard to appear through a haptic hardware/software manufacturer gaining larger and larger market shares, either way, we expect it to happen sooner or later.

Besides the problem discussed above, there really are not any more major drawbacks to using Unison simply because it is an abstraction layer, and a thin one at that (as evident by the functions described in the previous section which are often one call away from the

lower-level haptic or graphic API functions). Issues often thought of as potential drawbacks to using an API (which Unison can be categorized as for the sake of this argument) include performance, learning curve, and compatibility; being a thin abstraction layer ensures that Unison does not impose performance overhead on the HVA, so performance is not an issue; the learning curve of Unison is actually an advantage, not a disadvantage, not only because of the high readability of its function and data names, but also because as soon as a developer learns it once, he or she can use it repeatedly with different plug-ins; which brings us to the issue of compatibility that, also, is a non-issue because of Unison's plug-in model.

Chapter 6: An Application Scenario Using Unison

6.1. Introduction

After introducing Unison and talking about its different parts and components in some detail, we present an actual application scenario in which its services were used. In doing so, we not only demonstrate Unison's capabilities but also draw attention to the role that haptic applications will inevitably, we feel, start playing in the e-commerce world. Typical of an e-commerce shopper, one finds his or her way to an online store, and either browses or searches for a specific item looking at it, and if applicable as would be the case with a CD for instance, listening to it. The next progression would be to feel it.

In coming up with an e-commerce application scenario in which we could make use of the vast array of equipment available at the DISCOVER lab, we finally decided on a virtual car showroom scenario, in which a potential buyer could interact with the car in different ways to get a feel for it before an actual test drive or purchase.

Mainly due to range of motion constraints limiting smaller devices such as the MPB Freedom 6S and the PHANTOM Desktop, we chose the HapticMASTER as the haptic device in the application; it had a realistic enough range of motion, and also, was capable of recreating the considerable forces that are involved with interacting with a car (e.g. steering wheel, hood, door, etc). Naturally, that also meant using the proprietary HapticMASTER API.

As for the graphics, it was clear early on that if we were to showcase a graphically impressive model, we should survey available VRML files because creating a model from

scratch was more of a graphic designer's job than that of a computer programmer. There were several models available but the most detailed, and therefore most impressive, of which was that of a Jeep [67]. After some modification to the model, and in creating another for a showroom that would house the above mentioned car, all that was remaining was to decide on an application development language and thereby a VRML file loader.

We chose C++, the language the HapticMASTER API was written in, and decided on GLUT for windowing and operating-system-level features, most prominently, multi-threading. In choosing C++, we decided on the CyberGarage CyberX3D Loader for C++ and on OpenGL for graphic rendering (stand-alone if necessary, and via CyberX3D Loader).

Figure 17 shows how the components described above sit in Unison. The core, or main, application sits on top of Unison callings its services and functions, and able to access its synchronized hapto-visual scene graph. The VRML plug-in (built around the CyberX3D Loader in this implementation) and the HapticMASTER plug-in communicate with the VRML file (i.e. loading it) and the HapticMASTER device.

Since there is only a prototype of Unison implemented at the moment, we have chosen for the application to only simulate the turning of the jeep's steering wheel (Figure 18), lifting its hood (Figure 19), and manipulating its gear shift (Figure 20). We discuss the hood-lifting application below.

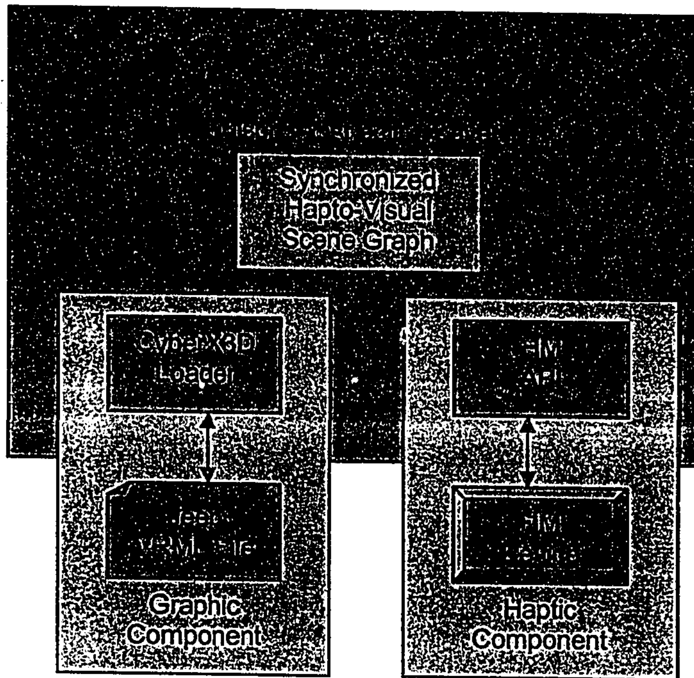


Figure 17: Car showroom application components when developed with Unison

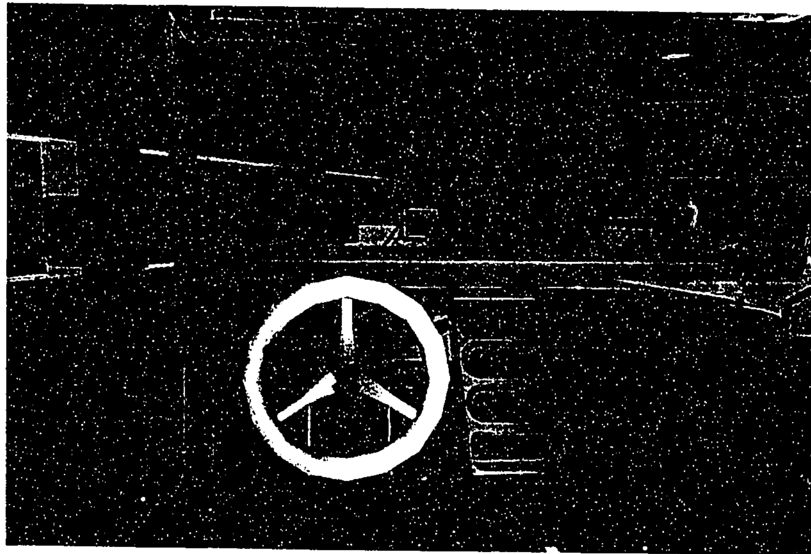


Figure 18: Screenshot from application in Section 3.3.1.3 showing haptic steering wheel

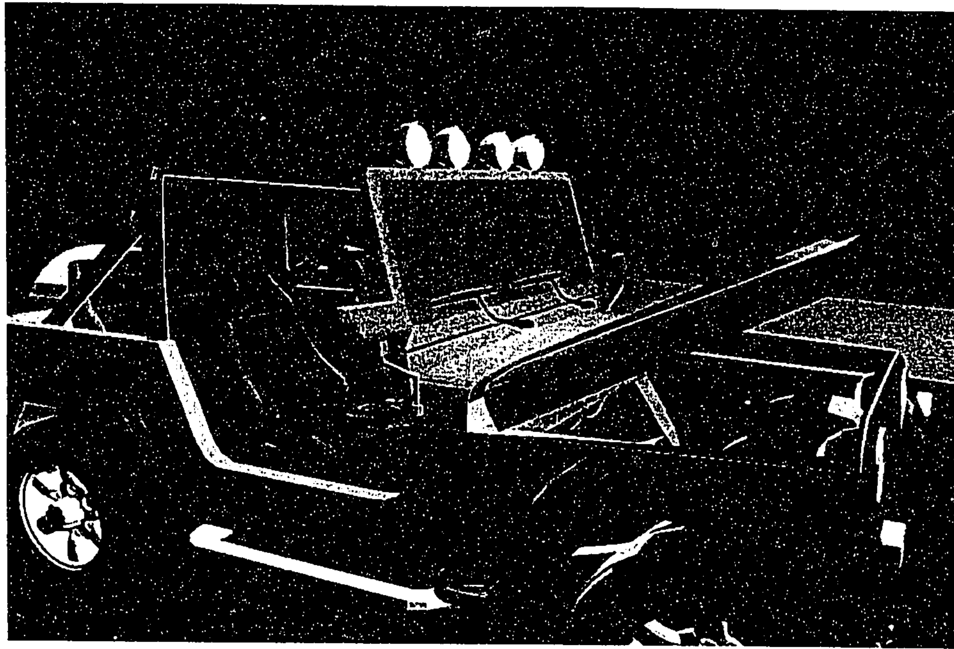


Figure 19: Screenshot from application in Section 3.3.1.3 showing jeep hood opening

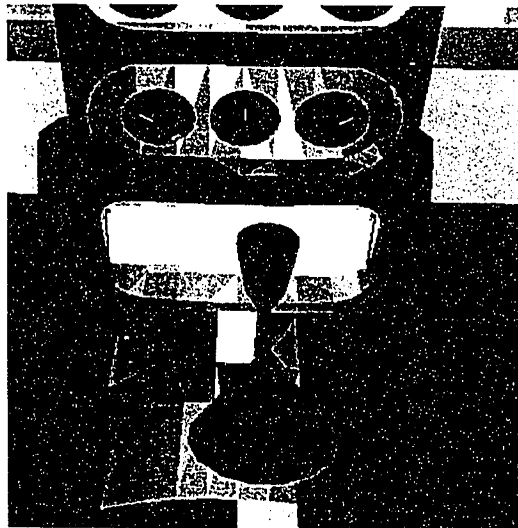


Figure 20: Screenshot from application in Section 3.3.1.3 showing gear box

6.2 Implementation

In Microsoft Visual C++ v 6.0, Unison HapticMASTER and VRML-loading header files are included in the project. A Unison class is instantiated with a VRML filename as a

constructor parameter within `main()`. This prompts Unison to populate its hapto-visual scene graph data structure with the `DEF` nodes[20] of the VRML file. Once the scene graph table is filled with the contents of the VRML file, the developer can access the "row" that holds the car hood data with a `get` function. Extra fields are provided to hold haptic properties in that row including: the representative haptic object, which in the case of a car hood would be a block object; size information, which in this case would have to be specified manually by the user[21]; and a force vector to represent gravity, since for the purposes of this example implementation, we follow the dynamic paradigm that couples the end-effector's position with that of the graphic object and constricts the end-effector's movement inside a hallow object to simulate a fixed handle on the car hood. The operation is simple; when the end-effector changes vertical position, the car hood graphically does so.

Figure 21 shows an execution mapping between some Unison functions of the hood-lifting application with their counterpart blocks of code should the aforesaid application be programmed directly with the HapticMASTER API (and the supporting tools described in Chapter 3: Section 3.1.3).

UNISON Implementation (with HapticMASTER plug-in)		**HapticMASTER API Implementation**
...		...
InitializeHapticDevice(FCS_HAPTICMASTER)	→	... ptrToHapticMASTER = ConnectToHapticMaster(...) ... ptrToHapticMASTER→SetRequestedState(FCS STATE_INITIALIZED) ... ptrToHapticMASTER→SetRequestedState(FCS

[20] With the current implementation of Unison, only DEF nodes are chosen to populate the hapto-visual scene graph. This, however, can be changed easily by calling a different function in CyberX3DLoader.
[21] The car hood in the VRML used in the example scenario is of type `indexedFaceSet` which does not have a size field per se; hence no automatic inference of the object size can be made given the current implementation of Unison.

		STATE_NORMAL) ... ptrToHapticMASTER→DeleteAll(); ... ptrToHapticMASTER→SetParameter(FCSPRM _INERTIA, 2.0); ...
...		...
CreateGraphicScene(VRML file)	→	... sceneGraph.load(VRML file) ... glutDisplayFunc(DrawSceneGraph) From VRML file ...
...		...
... carHood = getVRMLNodeByTypeAndName (SHAPENODE,"carHood") ... CreateHapticObject(..., BLOCK, ...)	→	... sceneGraph.findNode("CarHood"); ... (Traverse down to the shape node level) ... CreateFCSBlock(...)
...		...
glutTimerFunc(LiftingManager)	→	glutTimerFunc(LiftingManager)
...		...
DeleteAllHapticObjects()	→	ptrToHapticMASTER→DeleteAll()
...		...
return	→	return

Figure 21: Execution mapping for the hood-lifting application between a Unison implementation and a HapticMASTER API implementation

The application starts out by initializing the HapticMASTER, which involves connecting to it (via Ethernet) and calibrating the hardware by driving the robotic arm through its full range of motion. Once the hardware calibration is complete, the device's memory (on board the server computer it connects to) is cleared and an initial inertia parameter is set. This is necessary because of the admittance control paradigm the device implements (See Appendix I). All the above steps can be done by calling the Unison `InitializeHapticDevice()` function. The graphic scene is then loaded and drawn from the VRML file to a buffer. The Unison plug-in can accomplish this task by one simple function call, namely `CreateGraphicScene()`, which begins to populate the

synchronized hapto-visual scene graph data structure in Unison. Using the HapticMASTER API, the developer has to explicitly load the file, and write GLUT code to draw it to the screen when the GLUT main loop is invoked. After the graphic scene is in the synchronized hapto-visual scene graph and visible to the core application, Unison queries it for a reference to the shape node that represents the car hood. `getVRMLNodeByTypeAndName()` affords the user reference to any node in the graphic scene provided its type and name are known and adding a logical error check that makes sure data types are not confused (e.g. a shape node for a transform group node). Accomplishing the same with the HapticMASTER API implementation, and using CyberX3DLoader, requires the developer to traverse the scene graph looking for a node by that name, checking its type, and should it be a grouping node (e.g. a transform group node with a child shape node), search below it for the shape node. The developer then assigns a haptic geometric object to the graphic object counterpart (i.e. the car hood in this example), and writes the functionality of the application, namely lifting the car hood, in a parallel thread, call-back function controlled by GLUT (`liftingManager`). The application terminates the same in both implementations; with clearing the haptic objects as well as other housekeeping tasks like garbage collection, hardware state alterations, etc.

6.3. Usability of Unison as a Development Tool

Figure 21 in the previous section, and in discussing the implementation of some Unison functions, shows how concise and readable Unison functions are, and how much coding they save the developer. This, by definition, contributes to the argument of Unison's usability. And at this early stage of implementation, as we show below, Unison is truly of great value as a usable development tool.

Figure 22 is a complete rewriting in Unison of the same code used to program the HVA in Chapter 3: Section 3.1.1 (chosen here as an example for the sake of simplicity). What this program does, is it basically creates a haptic block in a virtual environment that is rendered graphically with OpenGL and displayed in a window using GLUT on a Windows platform. The haptic device in use is the HapticMASTER and its API version 1.2.

With the exception of `glutKeyboardFunc(Keyboard)` and the `Keyboard()` function body, which are part of a keyboard callback service supported by GLUT but not by the current implementation of Unison, all other code is grouped into generic services, making it more readable and the whole `main.cpp` file is cut by almost 95% of its original size from 368 lines of code to 19! Some of the improvement is because of default values that otherwise would have to be specified explicitly in an application but are provided in case no other values are in Unison, but all other improvement is due to generic, more readable, more concise, higher-level functionality grouping provided by Unison.

```
#include "Unison.h"
int main(int argc, char** argv)
{
    if (InitializeHapticDevice(FCS_HAPTICMASTER) == 0) {
        CreateHapticObject(HAPTIC_DEVICE_IN_USE, BLOCK, BlockCenter, BlockOrient, BlockSize, 5000, 5000);
    }
    InitializeGLUTWindowing(argc, argv);
    CreateGLUTWindow(800, 600, "FCS HapticMASTER Example 01 with UNISON");
    glutKeyboardFunc (Keyboard);
    CreateGraphicScene();
    DeleteHapticObject(HAPTIC_DEVICE_IN_USE, pBlock);
    return 0;
}
```

Figure 22: A Full main.cpp File Written in Unison

Note the readability of the code in Figure 22. `InitializeHapticDevice()` is a generic function that takes for a parameter a constant integer (defined here as

FCS_HAPTICMASTER), and upon successful return of the initialize function, creates a haptic object visible to the HAPTIC_DEVICE_IN_USE (FCS_HAPTICMASTER), of type BLOCK, with parameters center, orientation and size of the block. The final two parameters in the function represent, in grams, the stiffness of the outside and inside springs (walls) of the object. `InitializeGLUTWindowing()` passes any command line arguments to GLUT and then `CreateGLUTWindow()` displays a window of size 800 x 600 with the title specified. The `glutKeyboardFunc()` is a keyboard callback function, which in this case calls the `Keyboard()` function whenever the user hits a button on the keyboard. The `Keyboard()` function is defined in the `Unison.h` file here because it simply invokes the `DeleteHapticObject()` function whenever the user hits the `Esc` key, and this has become a convention for haptic applications we have implemented with Unison. `CreateGraphicScene()` creates and renders the graphic equivalent of the haptic block created before, and since it uses the same properties of the haptic block, both reside in the hapto-visual scene graph visible to the developer and all Unison services. `CreateSceneGraph()` also starts the GLUT main loop which runs all the call back functions, including the timer callback function, invisible to the user, which ensure hapto-visual synchronicity by referring to the position data of both end effector and graphic objects in the hapto-visual scene graph. Worthy of note is that the graphic rendering tool used here is OpenGL. Finally, collision detection is left to the HapticMASTER API, and to all other APIs for that matter; Unison makes no attempt at addressing collision detection.

Chapter 7: Conclusion and Future Work

We are at that point in the hapto-visual application development timeline not far from the threshold where a standardization effort would bring mass attention, development, and innovation to the field. Once there is a standard for devices and APIs to follow, all efforts would push in the same direction and what happened with computer audio and graphics will happen with haptic technology. We keep all this in perspective thanks to the historical backdrop we presented in Chapter 2. Over a century or so, we went from punched cards to PDAs, and over a few decades we went from the ABC to the Intel Pentium IV. The fact that technological innovation is advancing exponentially excites us.

In Chapter 3, we provided motivation for Unison by surveying current practices pointing out similarities that we can group together, generalize, and standardize, as an early attempt at gaining ground in standardizing the HVAD processes altogether.

Then we introduced Unison in Chapter 4, at a high level, and explained its premises and how it could be integrated into HVAD environments. We also discussed its extensibility due to the design model it followed, which enabled simple plug-in programming and addition to suffice in adding functionality.

In Chapter 5, we went into the specificities and details on Unison's design and implementation. We surveyed application development languages, compared and contrasted them before settling on C++ as the language of choice for writing Unison. We then examined graphic components and how a modeling file and its loader complement each other within the context of an application development process. CyberX3D Loader for C++ was chosen to handle VRML parsing, loading, and rendering for Unison's VRML

plug-in. Finally, we studied haptic devices and APIs distinguishing between them by purpose, function, and capabilities, primarily in range of motion and force recreation.

Chapter 5 also introduced Unison's inner workings, specifically its synchronized hapto-visual scene graph that holds all relevant information about all haptic and graphic objects in an environment, and also presented some implemented prototype functions that interacted directly with the haptic component, the graphic component, the core application, as well as a combination of two or more of the above.

Chapter 6 presented Unison as a viable development tool in an e-commerce application where a potential buyer could interact with a virtual car model. In this chapter we also discussed implementation specificities and glanced at Unison's usability, which for smaller less complex application, proved to yield a large saving in coding time and space.

Chapter 7 suggested three areas for future work, namely: texture recreation, physical body navigation, and of course, expanding on Unison's already available functions and libraries.

At this point in the lifetime of our haptic research efforts, we feel confident that standardization is the way to go and should be pursued in a consortium fashion to start the desirably vicious cycle that is more innovation fueling more spreading of the technology fueling lowered prices fueling more innovation, etc. Unison, an attempt at such standardization effort, is our contribution.

Much work remains to be done in the young and fertile field that is haptic research. In the course of preparing this work, we have come across a few areas of interest that we feel merit future follow up. From that list we point out: texture recreation, physical navigation

in virtual environments, and, for our work, further implementation of Unison services for different platforms and tool combinations, better adherence to common plug-in model frameworks, and support for audio.

Texture recreation is an essential component of a realistic haptic interaction. There exist applications today such as [68], [69], and [70], where high-fidelity recreation is achieved but only for *stiff* textures (e.g. those engraved in metal, wood, etc) using a stylus, for instance. The more natural way of haptically interacting with a virtual environment is through feeling texture with your fingertips. Special hardware is obviously necessary, and there are some attempts at this [71], however much work remains to be done.

Physical navigation in virtual environments is another interesting area of research. As applications stand today, we typically navigate from point A to point B in a virtual environment by using the keyboard, a mouse, or a joystick. However, researching methods of simulating walking in an environment would make the experience that much more interesting. This is more of a hardware manufacturing issue; however, the area is of value and interest.

Finally, and for the sake of carrying Unison forward, we shall be continuing to implement more services and functions for a wider range of haptic and graphic components. This work has only shown a VRML/HapticMASTER/C++ combination implementation and discussed the extensibility of Unison via plug-in additions. Work to support other graphic and haptic components remains to be done and changes to some infrastructure issues, such as the synchronized hapto-visual scene graph, need also to be completed to better align Unison with existing framework models (e.g. plug-in frameworks such as that of Eclipse) in hopes

of gaining credibility through adherence to established standards. Unison implementation in different programming languages should also be considered, whether it be done from the ground up, or through the use of wrappers and language interfaces. Support for audio in HVAs, discussed early on in the thesis, is also kept for future work; in search of a better, more realistic experience in a virtual environment, audio is as necessary as visual and haptic media.

The best is yet to come.

Appendix I: Force Control Paradigms in Haptic Devices

There are two force paradigms that control a haptic device: impedance control and admittance control.

In impedance control, the device reacts to the change of its end position by recreating force, so the model is "displacement in, force out". So if the device is moving in free air without coming into contact with a virtual haptic object, the feedback motors are completely disengaged. This however changes when collision is detected in which case the motors engage appropriately and the user feels feedback. This paradigm is show in Figure 23.

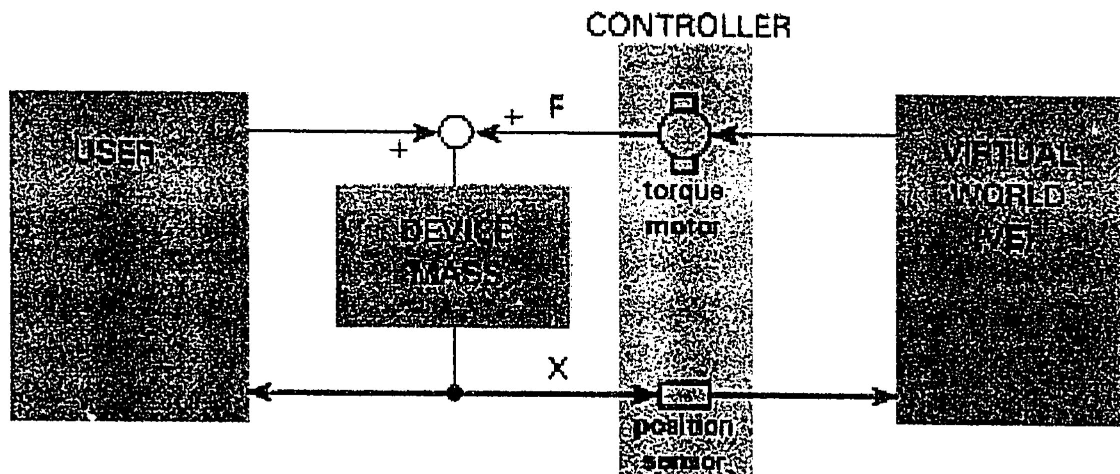


Figure 23: Impedance control [72]

In admittance control, the model is reversed. The device measures the force it experiences from the user and reacts by engaging its motors to move with certain velocity, acceleration, and of course transition; hence the paradigm is "force in, displacement out" as shown in Figure 24. Free air is an issue for this force model since the slightest application of force would result in considerable acceleration, however, much higher force can be recreated since when collision is detected with a stiff surface, for instance, whatever the change of force relayed to the device is, there will be no change in position.

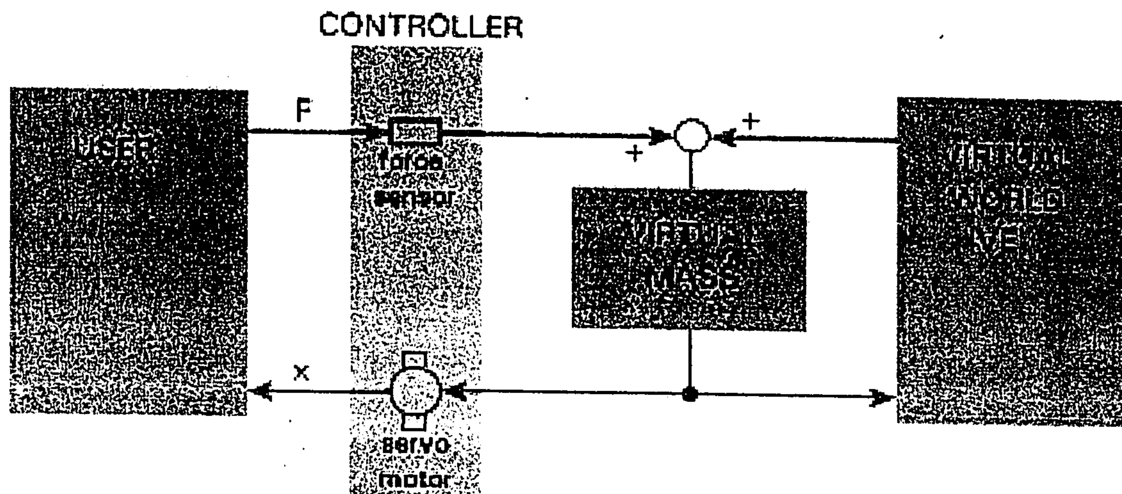


Figure 24: Admittance control [72]

The above observations of admittance and impedance control paradigms make each more suitable for a different set of scenarios as shown in Table 6.

Table 6: Impedance control vs. admittance control applications [72]

	Impedance	*Admittance*
Low mass	x	
Stable on physical surface	x	
Low costs	x	
Low friction		x
Stable on virtual surface		x
Can simulate added mass		x
Crisp master-slave control		x
Robust device		x

Appendix II: Source Code

Attached.

Bibliography

[1] EyeTech Digital Systems, "Quick Glance Eye-Gaze Tracking System", accessed on 09/12/2004, http://www.eyetechds.com/products.htm

[2] Meriam-Webster Online Dictionary, "Computer", accessed on 15/11/2004, http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=computer

[3] Caeruzzi, P. (2003). A History of Modern Computing. (2nd ed.). Cambridge, Massachusetts: The MIT Press.

[4] Rojas, R. and Hashagen, U., Ed. (2000). The First Computers - History and Architecture. Cambridge, Massachusetts: The MIT Press.

[5] Ifrah, G. (2001). The Universal History of Computing. New York, NY: John Wiley & Sons, Inc.

[6] Jones, D., Department of Computer Science at the University of Iowa, "Punched Cards: A brief illustrated technical history", accessed on 26/11/2204, last updated on 3/5/2004, http://www.cs.uiowa.edu/~jones/cards/history.html

[7] IEEE Computer Society, "Timeline of Computing History", accessed on 15/11/2004, last updated 1996, http://www.computer.org/computer/timeline/

[8] The Great Idea Finder, "Invention of the QWERTY Keyboard", accessed on 15/11/2004, last updated 21/10/2002, http://www.ideafinder.com/history/inventions/story098.htm

[9] Wikipedia, "BINAC", accessed on 22/11/2004, last updated 22/11/2004, http://en.wikipedia.org/wiki/BINAC

[10] Wikipedia, "Computer Mouse", accessed on 15/11/2004, last updated 1/11/2004, http://en.wikipedia.org/wiki/Computer_mouse

[11] Columbia University Academic Information Systems, "The IBM 1403 Printer", accessed on 15/11/2004, last updated 2002, http://www.columbia.edu/acis/history/1403.html

[12] Chadabe, J., Electronic Musician, "The Electronic Century Part III: Computers and Analog Synthesizers", accessed on 15/11/2004, last updated 1/4/2000, http://emusician.com/mag/emusic_electronic_century_part_2/

[13] Center for Computer Research in Music and Acoustics at Stanford University, "Historical View of Synthesizer Development", accessed on 15/11/2004, last updated 27/11/2003, http://ccrma.stanford.edu/~jos/kna/Historical_View_Synthesizer_Development.html

[14] Ballora, M., Music Department at Penn State University, "INART 55: History of Electroacoustic Music. Bell Labs in the 1960s", accessed on 17/11/2004, http://www.music.psu.edu/Faculty%20Pages/Ballora/INART55/bell_labs1960s.html

[15] The Encyclopedia of Computer Languages, Murdoch University, "Search results by year: 1964", accessed on 15/11/2004, http://hopl.murdoch.edu.au/findlanguages.prx?year=1964&which=byyear

[16] Obsolete.com, "The Synclavier", accessed on 15/11/2004, http://www.obsolete.com/120_years/machines/synclavier/

[17] Chadabe, J., Electronic Musician, "The Electronic Century Part IV: The Seeds of the Future", accessed on 15/11/2004. http://emusician.com/mag/emusic_electronic_century_part/index.html

[18] BrainyEncyclopedia.com, "PC Speaker", accessed on 18/11/2004, http://www.brainyencyclopedia.com/encyclopedia/p/pc/pc_speaker.html

[19] Wikipedia, "AdLib", accessed on 15/11/2004, last updated on 17/10/2004, http://en.wikipedia.org/wiki/AdLib

[20] Creative Labs, "Milestones", accessed on 18/11/2004, last updated 2004, http://us.creative.com/corporate/about/milestones/

[21] Creative Labs, "Corporate Fact Sheet", accessed on 7/12/2004, last updated 2004, http://us.creative.com/corporate/about/corpfact.asp

[22] Creative Labs, "Sound Blaster Audigy 4 PRO", accessed on 7/12/2004, http://us.creative.com/products/product.asp?category=1&subcategory=311&product=10853&nav=technicalSpecifications

[23] De Rossi, D., "Artificial tactile sensing and haptic perception", *Measurement Science and Technology*, vol. 2, pp. 1003 – 1016, Nov. 1991.

[24] Sutherland, I.E., "The ultimate display", in *Proceedings of IFIPS Congress*, vol. 2, pp 506 – 508, May 1965.

[25] Brooks, F.P. et al., "Project GROPE – Haptic Displays for Scientific Visualization", *Computer Graphics*, vol. 24, pp. 177 – 185, Aug. 1990.

[26] Will Ware's Personal Homepage, "Fun with Force Feedback", accessed on 15/11/2004, http://willware.net:8080/ff.html

[27] Russo, M., "The design and implementation of a three degree of freedom force output joystick", M.S. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1990.

[28] Open Book Society, "Warren Brodey – Uropatruljen", accessed on 16/11/2004, last updated 27/5/2001, http://www.openbooksociety.com/chaos/Norwegian/Venner/warren_brodey2.html

[29] VR Lab at the University of Tsukuba, "Overview", accessed on 7/12/2004, http://intron.kz.tsukuba.ac.jp/vrlab_web/sitemap/sitemap_e.html

[30] Minsky, M., "Computational Haptics: The Sandpaper System for Synthesizing Texture for a Force-Feedback Display", PhD dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[31] Massie,T., "Initial Haptic Explorations with the Phantom: Virtual Touch Through Point Interaction", M.S. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[32] SensAble Technologies, "Company: About us", accessed on 5/12/2004, last updated 2004, http://www.sensable.com/company/aboutus.asp

[33] Immersion Corporation, "Corporate – Background", accessed on 5/12/2004, last updated 2003, http://immr.client.shareholder.com/background.cfm

[34] Halo 2, "Halo 2", accessed on 1/12/2004, http://www.halo2.com/

[35] Reachin, "Business Areas", accessed on 11/11/2004, last updated 2003, http://www.reachin.se/businessareas/medical/

[36] FCS Robotics, "HapticMASTER", accessed on 11/11/2004, http://www.fcs-cs.com/robotics/products/hapticmaster

[37] Immersion Corporation, "Haptic Workstation", accessed on 11/11/2004, last updated 2004, http://www.immersion.com/3d/products/haptic_workstation.php

[38] Immersion Corporation, "CyberForce", accessed on 11/11/2004, last updated 2004, http://www.immersion.com/3d/products/cyber_force.php

[39] Immersion Corporation, "CyberGrasp", accessed on 11/11/2004, last updated 2004, http://www.immersion.com/3d/products/cyber_grasp.php

[40] Immersion Corporation, "CyberGlove", accessed on 11/11/2004, last updated 2004, http://www.immersion.com/3d/products/cyber_glove.php

[41] BMW World, "iDrive Technology", accessed on 1/12/2004, last updated 2004, http://www.bmwworld.com/technology/idrive.htm

[42] MPB Technologies Inc., "MPB Freedom 6S Hand Controller", accessed on 1/12/2004, last updated 2004, http://www.mpb-technologies.ca/mpbt/haptics/hand_controllers/freedom/freedom.html

[43] SensAble Technologies, "PHANTOM Devices", accessed on 1/12/2004, last updated 2004, http://www.sensable.com/products/phantom_ghost/phantom.asp

[44] Ruiter, B., *HapticMASTER API Programming Manual*, FCS Control Systems, 2003.

[45] CyberGarage, "CyberX3D for C++", accessed on 11/11/2004, http://www.cybergarage.org/vrml/cx3d/cx3dcc/index.html

[46] Web3D Consortium, "The Virtual Reality Modeling Language", accessed on 11/11/2004, http://www.web3d.org/x3d/specifications/vrml/ISO_IEC_14772-All/index.html

[47] Reachin Technologies AB Technical Staff, *Reachin API 3.1 Programmer's Guide*, Reachin Technologies AB, 2002.

[48] Zhou, J., Shen, X., Georganas, N., "Haptic Tele-Surgery Simulation", in *Proc. IEEE Workshop on Haptic Audio Visual Environments and their Applications*, Ottawa, ON, Canada, October 2004.

[49] Bell Labs, "C Reference Manual", May 1975, http://cm.bell-labs.com/cm/cs/who/dmr/cman.pdf

[50] Kernighan, B. and Ritchie, D. (1978). The C Programming Language. Prentice Hall, 1978.

[51] Kernighan, B. and Ritchie, D. (1988). The C Programming Language (2nd ed.). Prentice Hall, 1988.

[52] Codepedia, the Developers Encyclopedia, "Java FAQ – What is Java", accessed on 15/11/2004, last updated 22/12/2002, http://www.codepedia.com/1/JavaFAQ_WhatIsJava

[53] Sangappa,S., Palaniappan,K., and Tollerton,R. "Benchmarking Java against C/C++ for interactive scientific visualization" in *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, Poster Session, pp. 236 – 236, 2002.

[54] Billard, E., "Language-Dependent Performance of Design Patterns", in *ACM SIGSOFT Software Engineering Notes*, vol. 28, pp. 3 – 3, 2003.

[55] OpenGL.org, "GLUT – The OpenGL Utility Toolkit", accesses on 1/12/2004, last updated 2004, http://www.opengl.org/resources/libraries/glut.html

[56] Wikipedia, "OpenGL", accessed on 1/12/2004, last updated 11/11/2004, http://en.wikipedia.org/wiki/OpenGL

[57] OpenGL.org, "OpenGL Overview", accessed on 1/12/2004, last updated 2004, http://www.opengl.org/about/overview.html

[58] Web3D Consortium, "X3D Specifications", accessed on 1/12/2004, last updated 2004, http://www.web3d.org/x3d/specifications/x3d_specification.html

[59] CyberGarage, "CyberX3D for Java", accessed on 11/11/2004, http://www.cybergarage.org/vrml/cx3d/cx3djava/index.html

[60] Java.net, "j3d-vrml97: Java 3D VRML97 Loader", accessed on 1/12/2004, https://j3d-vrml97.dev.java.net/

[61] CyberGarage, "CyberGarage", accessed on 1/12/2004, http://www.cybergarage.org

[62] OpenVRML, "OpenVRML", accessed on 2/12/2004, http://www.openvrml.org/

[63] SensAble Technologies, "PHANTOM Desktop", accessed on 15/11/2004, http://www.sensable.com/products/phantom_ghost/phantom-desktop.asp

[64] Reachin Technologies AB, "Reachin API", accesses on 15/11/2004, http://www.reachin.se/products/reachinapi/

[65] SensAble Technologies Technical Staff, *GHOST SDK Version 3 Programmer's Guide*, SensAble Technologies, 1999.

[66] SensAble Technologies, "Products: PHANTOM and GHOST: GHOST Overview", accessed on 15/11/2004, http://www.sensable.com/products/phantom_ghost/ghost.asp

[67] DeVerno, M., "Virtual Car Dealership", Distributed and Collaborative Virtual Environments Research Laboratory, Ottawa, ON, Canada, Project Report 7/12/2003.

[68] Reachin API v. 3.2 demo, "Earth", Reachin Technologies AB, 2002.

[69] Reachin API v. 3.2 demo, "Surfaces", Reachin Technologies AB, 2002.

[70] Reachin API v. 3.2 demo, "Textures", Reachin Technologies AB, 2002.

[71] Northwestern University Laboratory for Intelligent Mechanical Systems, "Northwestern University Fingertip Haptics", accessed 1/12/2004, http://lims.mech.northwestern.edu/projects/fingertip/

[72] FCS Robotics, "Robotic Technology", accessed on 1/12/2004, http://www.fcs-cs.com/robotics/technology.

[73] Mittman, R., iHealthBeat – California Healthcare Foundation, "TECHNOLOGY FORESIGHT: Haptics – Reach Out and Touch Something", accessed on 1/12/2004, last updated 2/3/2004, http://ihealthbeat.org/index.cfm?action=lookupID&id=26512

[74] SourceForge.net, "SourceForge.net: A01, About SourceForge.net (en)", accessed on 1/12/2004, last updated 2004, http://sourceforge.net/docman/display_doc.php?docid=6025&group_id=1